Analysis and Testing of Programs with Exception-Handling Constructs

Saurabh Sinha and Mary Jean Harrold
Department of Computer and Information Science
The Ohio State University
395 Dreese Lab
Columbus, OH 43210 USA
TEL: (614) 292-2568 FAX: (614) 292-2911
{sinha,harrold}@cis.ohio-state.edu

Abstract

Analysis techniques, such as control-flow, data-flow, and control-dependence, are used for a variety of 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 impact of exception-handling constructs on several analysis techniques. The paper presents techniques to construct representations for programs with exception occurrences and exception-handling constructs, along with 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 and software engineering tasks.

Keywords: Exception handling, program analysis, program slicing, structural testing, regression 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., [8, 14, 16, 30]), 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] (e.g., [7]) and for interacting procedures (*interprocedural*) (e.g., [18]). Some of this research has addressed the problems of performing analyses for programs with transfers of control, such as `continue` and `goto` statements, that can affect the analyses at the intraprocedural level (e.g., [1]). Other research has addressed the problems of performing analyses for programs with transfers of control, such as **exit()** statements, that can affect the analyses at the interprocedural level [13]. To be applicable to programs 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.

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 applications. From each subject, we obtained a count of classes that contain `throw` statements. Table 1 summarizes the results of the study. For each subject, the table describes the subject, lists the number of classes examined, and lists classes that contain `throw` statements.

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

Subject		Number of	Classes
Name	Description	Classes	with `throw`
debug	Sun's Java debugger	45	14
jar	Sun's Java archive tool	8	3
java_cup	LALR parser generator for Java	35	17
javac	Sun's Java compiler	154	46
javadoc	Sun's HTML document generator	3	2
jdk-api	Sun's JDK API	712	169
AAS-Java	Aristotle Analysis System Java analyzer	302	46
jlex	Lexical-analyzer generator for Java	20	3
swing-api	Sun's Swing API	1588	142
Total		2867	442

As the table illustrates, 15.4% of the examined classes contain `throw` statements. Within a subject, the percentage of classes that contain `throw` statements ranges from 8.9% to 66.7%. This study supports our belief that the use of exception-handling constructs in practice is significant enough that it should be considered during various analyses.

Several researchers have considered the effects of exception-handling constructs on various types of analyses. One approach [5] performs control-flow analysis to compute points-to information in the presence of exceptions. Another approach [23] performs control-flow analysis to facilitate interprocedural path profiling, and handles code that contains exception-handling constructs. Other work [32] addresses the problem of computing exception propagation in Ada programs. None of these techniques, however, defines representations on which analyses, such as control-dependence computation, and software engineering tasks, such as regression test selection, can be performed.

To facilitate such analyses for software engineering tasks, we investigated the effects of exception-handling constructs on various types of analysis, 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 three types of analysis — control-flow, control-dependence, and program slicing — and we discuss the use of our analysis information in two applications

— structural testing and regression test selection. We discuss the effects of exception-handling constructs on the analyses, and then present new techniques for performing the analyses and constructing appropriate representations in the presence of exceptions. We restrict our discussion to representations for programs that contain Java-like exception-handling constructs; representations for constructs in other languages, such as C++, can be constructed similarly. 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 (e.g., [5, 20, 22]).

In this paper, we also present results of empirical studies that we performed using our program analysis system, Aristotle Analysis System for Java (AAS-Java) written in Java, that analyzes Java byte-code files.[3] These studies evaluate the precision of our representations and compare the results obtained using our techniques with those obtained using techniques that fail to account for exception-handling constructs. These results suggest that analyses and representations that do not consider such constructs can fail to identify important 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 these techniques, and some empirical studies pertaining to the techniques. Then, Section 5 discusses the use of our representations for two applications: structural testing and regression test selection. Finally, Section 6 gives our conclusions and presents future work.

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 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. 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 `try` statement can have a `finally` block. The `finally`-block code is always executed, regardless of the way in which control transfers out of the `try` block: by reaching the last statement in the try block, through an

[3]AAS-Java 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 [12].

```
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
   . . .
}
```

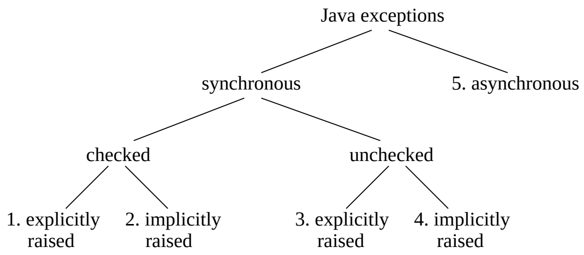


Figure 1: The syntax of exception-handling constructs in Java (left), and Java Exception types (right).

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 *non-resumable* 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 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, non-deterministic 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. For purposes of program analysis, we further classify a synchronous exception as explicitly-raised or implicitly-raised. 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. For example, a call to the API method `java.util.Stack.pop()` can raise a `StackEmptyException`; any expression that dereferences an object reference can cause a `NullPointerException`.

The techniques that we discuss in this paper are based on the assumption that the program point at which an exception is raised can be determined. Therefore, the techniques do not apply to asynchronous exceptions; a safe approximation of program points that may raise such exceptions would include all statements in the program. We also exclude implicitly-raised exceptions that are caused by the side-effect of an expression evaluation. 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. The techniques are applicable to all explicitly-raised exceptions; we discuss the implications of exceptions that are raised by library routines in Section 4.4.

3 Effects of Exception-Handling Constructs on Analyses

In this section, we first describe a program that we use to illustrate the concepts presented in the paper. We then discuss the impact of exception-handling constructs on four analysis techniques — control-flow analysis, data-flow analysis, control-dependence analysis, and program slicing — and further discuss the ramifications of exception-handling constructs on other applications.

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 the coins of value sufficient to cover the cost of the item, the machines 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 that are defined in the class `VendingMachine` to process the action. Method `insert()` (lines 5–9) 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, and returns the balance of the transaction. 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 the transaction.

We omit the details of methods, such as `Dispenser.value()` and `Dispenser.available()`, that are not relevant to our presentation; such methods do not raise any exceptions. For brevity, we also exclude the details of constructors of some of the classes and the initialization of constants such as `MAX_ATTEMPTS` and `MIN_SELECTION`.

[4] We adapted this example from one by Kung et al. that appeared in Reference [17].

```
public class VendingMachine {

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

  public VendingMachine() {
 1  totValue = 0;
 2  currValue = 0;
 3  currAttempts = 0;
 4  d = new Dispenser();
  }

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

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

  public void vend( int selection ) {
15  if ( currValue == 0 ) {
16    throw new ZeroValueException();
    }
    try {
17    d.dispense( currValue, selection );
18    int bal = d.value( selection );
19    totValue += currValue - bal;
20    currValue = bal;
21    returnCoins();
    }
22  catch( SelectionException s ) {
23    currAttempts++;
24    if ( currAttempts < MAX_ATTEMPTS ) {
25      showMsg( "Enter selection again" );
      }
      else {
26      currAttempts = 0;
27      throw s;
      }
    }
28  catch( ZeroValueException z ) {
    }
  }
}
```

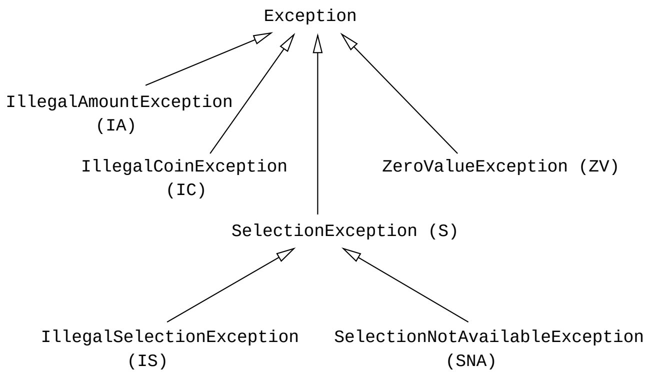
```
public class Dispenser {
  public void dispense( int currVal, int sel ) {
29  Exception e = null;
30  if ( sel < MIN_SELECTION || sel > MAX_SELECTION ) {
31    showMsg( "selection "+sel+" is invalid" );
32    e = new IllegalSelectionException();
    }
    else {
33    if ( !available( sel ) ) {
34      showMsg( "selection "+sel+" is unavailable" );
35      e = new SelectionNotAvailableException();
      }
      else {
36      int val = value( sel );
37      if ( currVal < val ) {
38        e = new IllegalAmountException( val-currVal );
        }
      }
    }
39  if ( e != null ) {
40    throw e;
    }
41  showMsg( "Take selection" );
  }
}
```

```
  public static void main() {
42  VendingMachine vm = new VendingMachine();
43  while ( true ) {
      try {
        try {
44        switch( action ) {
45          case INSERT: vm.insert( coin );
46          case VEND:   vm.vend( selection );
47          case RETURN: vm.returnCoins();
          }
        }
48      catch( SelectionException s ) {
49        showMsg( "Transaction aborted" );
50        vm.returnCoins();
        }
51      catch( IllegalCoinException i ) {
52        showMsg( "Illegal coin" );
53        vm.returnCoins();
        }
54      catch( IllegalAmountException i ) {
55        int val = i.getValue();
56        showMsg( "Enter more coins"+val );
        }
      }
57    catch( ZeroValueException z ) {
58      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).

3.2 Impact of Exceptions on Control-Flow Analysis

Control-flow analysis determines, for each program statement s, those statements in the program that could 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, utilize



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 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, a control-flow representation that incorporates exception-induced control flow is required.

The vending-machine program of Figure 2 exemplifies the complexity that the presence of exception-handling constructs can introduce into 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 have an impact on 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 [2]. Figure 4 depicts the CFGs for the methods in the vending-machine program. Each node is labeled by the line number of the source statement that it represents; some nodes represent several source statements. 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 exception-induced 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. In Section 4.1, we describe our approach to connecting such nodes in the CFG, and creating intraprocedural and interprocedural representations of the program.

3.3 Impact of Exceptions on Data-Flow Analysis

Data-flow analysis techniques compute, for different programs points, data-flow facts, such as definition-use pairs, reaching definitions, available expressions, and live variables, that hold at those points. Data-flow

[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.

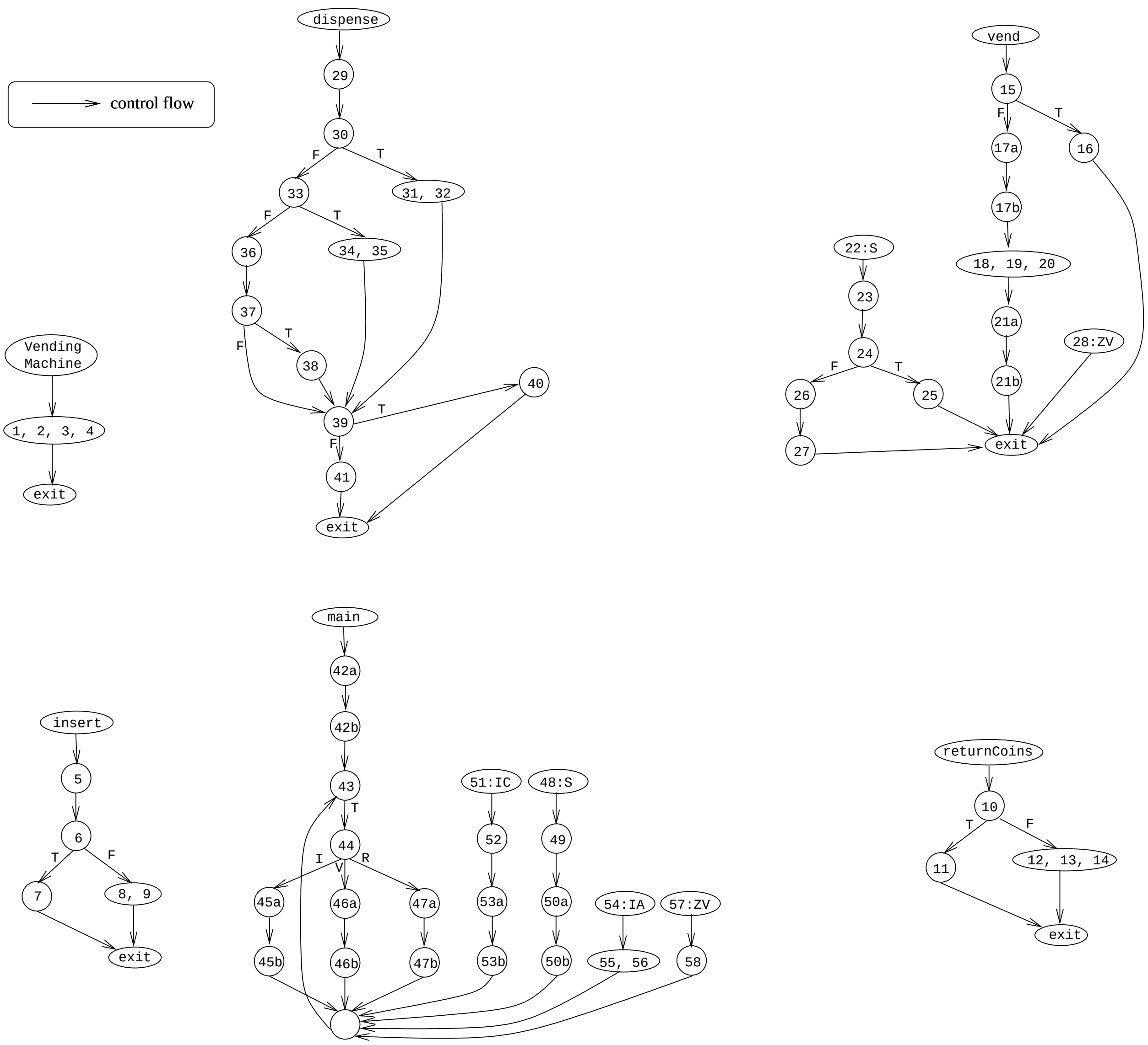


Figure 4: CFGs for methods of the vending-machine program.

information is used in activities, such as program slicing (e.g., [10, 15, 34]), data-flow testing (e.g., [8, 14, 21]), and compiler optimizations [2]. A typical data-flow problem is formulated as a set of data-flow equations whose solutions are iteratively propagated throughout the program using a CFG; the solution of the data-flow problem is the fixed-point solution of the equations. The presence of exception-handling constructs can cause an analysis solution to be *unsafe* in that the solution fails to be a conservative approximation of the true solution.[6] Unsafe analysis solutions are unacceptable in most applications of the analyses.

To illustrate, we consider the impact 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 program from d to u along which v is not redefined. Exception-handling constructs may cause a definition-

[6]Depending on the type of the data-flow problem, a conservative (safe) solution can be either a superset or a subset of the true solution [2].

use computation to be unsafe in two ways. First, a definition-use pair may not be detected because, in the CFG, there is no path from the definition to the use. In the vending-machine program of Figure 2, there is such a pair that includes statement 42, where vm is defined, and statement 53, where vm is used. However, in the CFG for `main()`, there is no path from node 42a that leads to node 53. Therefore, the definition-use computation would fail to detect this definition-use pair. 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. The results of other data-flow analyses, such as points-to analysis [5], can also be represented by a data-dependence graph. Clearly, the computation of correct data-flow information in the presence of exceptions depends on correctly representing the control flow for exception-handling constructs. The computation also requires that interprocedural data-flow analysis algorithms be modified to account for the additional interprocedural relations caused by exception-handling constructs. Our future work includes adapting existing interprocedural definition-use analysis to account for exception-handling constructs.

3.4 Impact of Exceptions on Control-Dependence Analysis

Control-dependence analysis [4, 7] 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 causes the node for s to be reached, whereas following the other edge may not cause that node to be reached. 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. For these analyses and tools to compute correct information, control-dependence analysis must consider the effects of exception-handling constructs.

The presence of statements that raise exceptions alters the control dependences in a program. For example, in method `insert()` in the vending-machine program of Figure 2, the `throw` statement in line 7 causes statement 8 to be control dependent on statement 6. This control dependence for statement 8 differs from the dependence that would be computed if line 7 were to contain an ordinary statement instead of a `throw` statement — statement 8 would then be control dependent on entry into `insert()`.

More significant than the impact of exceptions on individual methods is their impact on *interprocedural control dependences* — control-dependence relationships that hold across method or procedure boundaries [13]. One factor that affects interprocedural control dependences is the presence of *potentially non-returning call sites* (PNRCs) [13]: 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 programs. 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, such as statement 20, that follow the call site to be control dependent on conditional statements in the called methods (e.g., statement 20 is control dependent on statement 39 in `dispense()`).

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

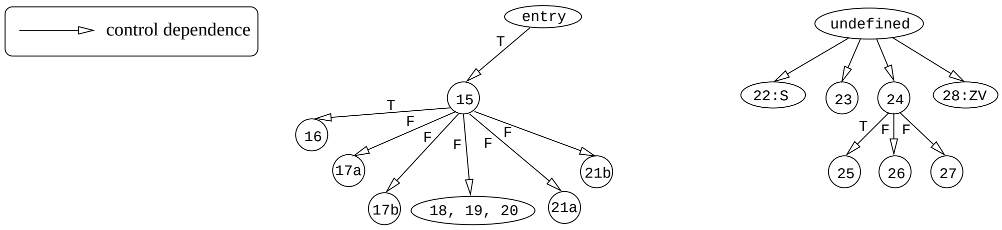


Figure 5: Control-dependence graph for method `vend()`.

have no intraprocedural control dependences. For example, the execution of statements 51–53, that 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 control-dependence graph. A *control-dependence graph* (CDG) [7] 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 on entry into 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 (unknown).

The problem of computing control dependences in the presence of exceptions can be solved by (1) constructing a correct control-flow representation, and (2) identifying PNRCs and accounting for their effects on control dependences. A correct control-flow representation accounts for changes to intraprocedural control dependences such as the one described in the first example. This representation does not, however, account for changes in intraprocedural and interprocedural control dependences that are caused by PNRCs: traditional control-dependence computation [4, 6, 7], when applied to a correct control-flow representation still fails to identify the control dependence of statement 20 on statement 39. Such changes in control dependences may in fact be significant. In Section 4.2, we present an approach that correctly computes control dependences that are caused by exception-handling constructs.

3.5 Impact of Exceptions on Program Slicing

A *program slice* for a program P, computed with respect to a *slicing criterion* $< p, V >$ where p is a program point and V is a set of variables referenced at p, includes statements in P that may influence the values of the variables in V at p [34].[7] There are two alternative approaches to computing slices: (1) using data-flow equations [10, 34] to compute the slices, or (2) constructing dependence graphs [15] and using graph-reachability on these graphs to compute the slices. None of the existing interprocedural slicing algorithms, however, accounts for the effects of exception-handling constructs on slicing. Thus, given a slicing criterion $< S, V >$, the slices computed by these algorithms may omit statements from the slice that affect the values of variables in V at program point S. The reason for this omission is that

[7] Slices can also be computed in the forward direction; in this case, the slice contains all statements in the program that may be affected by the value of variable v at program point p.

these algorithms use intraprocedural control-dependence information based on control-flow graphs for single methods or procedures (*intraprocedural control dependence*). Recent work [13] describes several ways in which this intraprocedural control-dependence computation inaccurately models the semantic dependences that exist between program statements.

One way in which intraprocedural control-dependence computation inaccurately models semantic dependence is seen in the effects of exception-handling constructs. For example, consider the `throw` statement in line 40 of the vending-machine program, which affects control dependences of statements in methods `vend()` and `main()`. If control dependence is computed intraprocedurally in the usual way, without considering the effects of the `throw` statement, statement 20, for example, will be control dependent on entry into `vend()`. On closer inspection, however, we see that statement 20 depends most immediately for its execution on statement 39, because of the `throw` in statement 40. Thus, according to the definition of semantic dependence [28], statement 20 is semantically dependent on statement 39, and therefore, statement 39 should be included in any slice that includes statement 20. However, because the control dependence information that existing interprocedural slicing algorithms use is computed without considering the effects of `throw` statements, the slice computed by existing algorithms does not include statement 39. In Section 4.3, we present a technique to compute safe slices in the presence of exception-handling constructs.

4 Analysis Techniques to Accommodate Exception-Handling Constructs

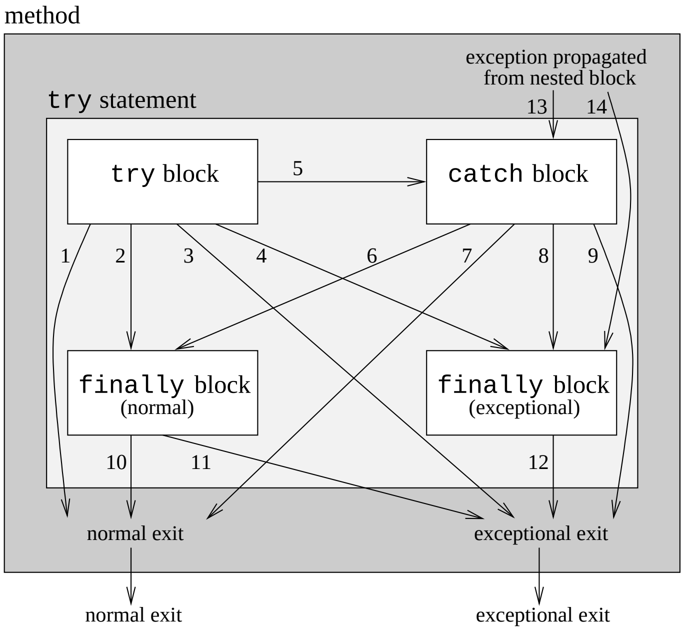
In this section, we describe analysis techniques for control-flow analysis, control-dependence analysis, and slicing that account for the effects of exception-handling constructs, and thus, can safely be applied to programs that contain such constructs.

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 a call site such that the called method propagates an exception. 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 of type T *handles* an exception of type E if E is propagated by the `try` block associated with that `catch` clause, and E is the same type as T or E is a subtype of T. A `catch` clause (or a `finally` block) *encloses* a program statement s if the `try` block associated with that `catch` clause (or `finally` block) lexically encloses s. A `catch` clause is a *local handler* if it only handles exceptions that are directly raised in the `try` block associated with that `catch` clause. A `catch` clause is an *interprocedural handler* if it only handles exceptions that are indirectly raised in the `try` block associated with that `catch` clause. A `catch` clause is a *global handler* if it is a local handler and an interprocedural handler.

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



1 try block raises no exception
2 try block raises no exception; finally block specified
3 try block raises exception; catch block does not handle exception; no finally block
4 try block raises exception; catch block does not handle exception; finally block specified
5 try block raises exception; catch block handles exception
6 catch block handles exception; finally block specified
7 catch block handles exception; no finally block
8 catch block handles exception, raises another exception; finally block specified
9 catch block handles exception; raises another exception no finally block
10 finally block raises no exception
11 finally block raises exception
12 finally block propagates previous exception, or raises another exception
13 nested block propagates exception; catch block handles exception
14 nested block propagates exception; catch block does not handle exception; finally block specified

Figure 6: Control flow in Java exception-handling constructs.

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.

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. In the following discussion, each type of path within a `try` statement is referenced by the edges that it traverses in the block.

Path 1 is taken if the `try` block raises no exception, and no `finally` block is specified.
Path 2,10 is taken if the `try` and `finally` blocks raise no exceptions.
Path 2,11 is taken if the `try` block completes normally, but the `finally` block raises an exception.
Path 3 is taken if the `try` block raises an exception, no handler in the `catch` block handles that exception, and no `finally` block is specified.
Path 4,12 (14,12) is taken if the `try` block raises an exception (or a nested block propagates an exception), no handler in the `catch` block handles that exception, but a `finally` block is specified.
Path 5,6,10 (13,6,10) is taken if the `try` block raises an exception (or a nested block propagates an exception), the `catch` block handles the exception, and the `finally` block raises no exception.

Path 5,6,11 (13,6,11) is taken if the `try` block raises an exception (or a nested block propagates an exception) that is handled in the `catch` block, and then the `finally` block raises another exception.

Path 5,7 (13,7) is taken if the `try` block raises an exception (or a nested block propagates an exception), the `catch` block handles the exception, and no `finally` block is specified.

Path 5,8,12 (13,8,12) is taken if the `try` block raises an exception (or a nested block propagates an exception), the `catch` block handles the exception but raises another exception, and the `finally` block raises no exception.

Path 5,9 (13,9) is taken if the `try` block raises an exception (or a nested block propagates an exception), the `catch` block handles the exception but raises another exception, and no `finally` block is specified.

In constructing the CFG for a method that contains exception-handling constructs, we connect the node corresponding to a `throw` statement to the local handlers that handle the raised exceptions. A throw-statement node may have multiple out-edges, one corresponding to each type of exception that can be raised at that `throw` statement. To distinguish these edges, we annotate 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 may raise one of three types of exceptions — `SelectionNotAvailableException` (`SNA`), `IllegalAmountException` (`IA`), or `IllegalSelectionException` (`IS`). Therefore, the node for that statement has three out-edges — one for each type of exception — that are labeled by the exception type. If a throw statement raises a single type of exception, the CFG node for that statement has a single out-edge that is also labeled by the exception type. For example, node 11 has a single out-edge labeled `ZV` because the corresponding `throw` statement only raises an exception of type `ZeroValueException (ZV)`.

To construct the CFG, those types of exceptions that can be raised at a `throw` statement must be available. Our CFG-construction algorithm performs type inferencing locally within a method to determine the exception types; we address this issue further in Section 4.1.3.

If the exception type corresponding to an edge is caught by a local handler, the target of an out-edge from a throw-statement node is a `catch` handler node. If, however, the exception is caught by an interprocedural or a global handler (that belongs in a separate method), that exception is propagated out of the method; the out-edge is not connected to a catch-statement node. To model the propagation of the exception out of the method, we connect the out-edge to an exceptional-exit node. An *exceptional-exit node* is an exit point in the CFG, and has a type T associated with it that represents the propagation of an exception of type T by the corresponding method.[8] The CFG for a method, therefore, has as many exceptional-exit nodes as the distinct type of exceptions that are directly propagated by the method. In Figure 7, the CFG for `vend()` has four exceptional-exit nodes because `vend()` can directly propagate four exception types — `SelectionException`, `IllegalSelectionException`, `SelectionNotAvailableException`, and `ZeroValueException`.[9] 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 `IllegalAmountException`, through the call to `dispense()`; the exceptional-exit node for the indirectly-propagated exception is created in the interprocedural representation (Section 4.1.2).

[8] In the discussion, we identify an exceptional-exit node by its associated type.

[9] Our type-inference algorithm (Section 4.1.3) determines that the `throw` statement in line 27 can potentially raise one of three types of exceptions: `SelectionException`, `IllegalSelectionException`, or `SelectionNotAvailableException`. Therefore, the CFG contains an exceptional-exit node for each of these types.

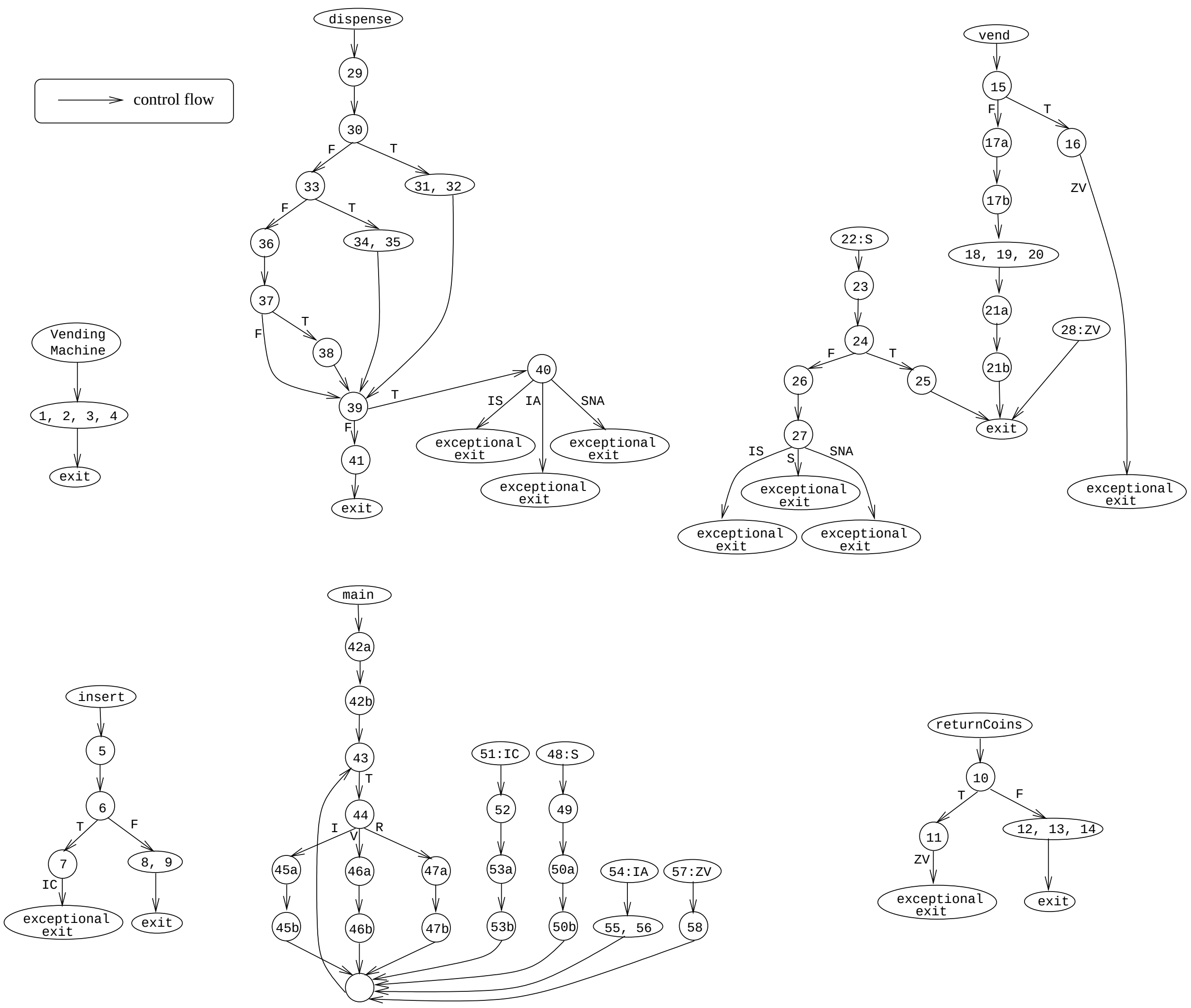


Figure 7: CFGs for methods of the vending-machine program constructed using our approach.

If exceptions are propagated out of a method, the CFG for that method contains exceptional-exit nodes, one for each exception type. We create such a representation with multiple exit points because distinct exception types that are propagated out of a method can be caught by distinct interprocedural or global handlers handlers. This representation explicitly shows types of exceptions that are propagated by a method, and therefore, facilitates program comprehension.

In Figure 7, nodes that correspond to `catch` handlers are labeled by the statement number and the handler type. In the CFG, the node for a local handler has in-edges for exceptions that are caught by that handler. A global handler catches directly-raised exceptions, and therefore, the node for such a handler also has in-edges in the CFG for those exceptions. 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.

The presence of `finally` blocks further complicates control flow in Java programs. In Java, a `finally` block can execute under two contexts: a normal context or an exceptional context. The two instances of `finally` blocks in Figure 6 illustrate the two contexts. A `finally` block executes under *normal context* when (1) control reaches the end of a `try` block or a `catch` block, or (2) when control leaves a `try` statement because of a transfer-of-control statement, such as `break`, `continue`, or `return`. A `finally` block executes under *exceptional context* when control leaves a `try` statement because of the propagation of an exception. The context of execution of a `finally` block determines where control flows from that `finally` block. A naive representation of this control-flow embeds the nodes for a `finally` block in the CFG for its containing method, and creates in-edges to the first node for the `finally` block for each execution of the block, and an out-edge from the last node for each in-edge. Such a representation might cause existing propagation algorithms for analysis to traverse paths that might enter the `finally` block along an edge that represents execution of that block in the normal context, but leave the `finally` block along an edge that represents the execution of that block in the exceptional context. To circumvent this problem, we treat each `finally` block as a separate method: we create a separate CFG for each `finally` block, and insert call nodes to those `finally` blocks appropriately.

The details of our CFG-construction algorithm can be found in Reference [33].

4.1.2 Interprocedural analysis

The propagation of exception from a called method to its caller creates interprocedural control-flow paths that must be represented in an interprocedural control-flow graph. An *interprocedural control-flow graph* (ICFG) for a program $\mathcal{P}$ consists of CFGs for every method and procedure P in $\mathcal{P}$. At each call site, the call node is connected to the entry node of the called method or procedure by a call edge, and the exit node of the called method or procedure is connected to the corresponding return node by a return edge.

Figure 8 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. If B completes normally, control returns to A through a normal exit from B. However, if B propagates an exception, control returns from B through an exceptional exit. Following an exceptional exit from B, control flows to one of three places: (1) if A's `try` block has an associated `catch` block where a handler catches the exception raised in B, control flows to that handler; (2) otherwise, if A's `try` block has a corresponding `finally` block, control flows to that `finally` block (the block executes in the exceptional context); (3) if neither of the above is true, method A also exits exceptionally, and the search for a handler continues in a caller of A.

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

Figure 9 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 the type of that exception) is connected to the appropriate catch node in the caller. For example, `insert()` propagates `IllegalCoinException` which is caught in statement 51 of `main()` (the

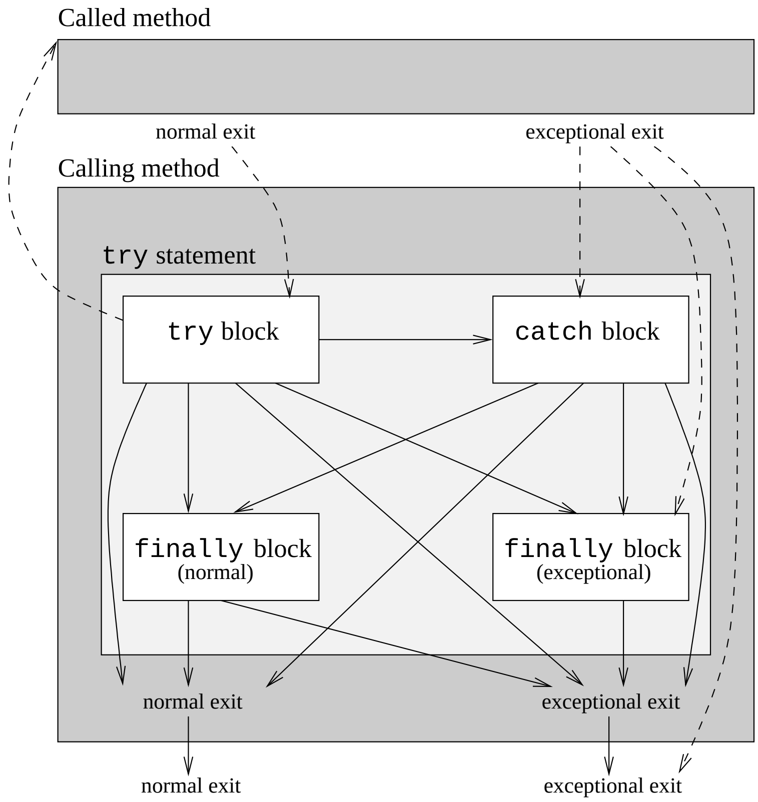


Figure 8: Interprocedural control flow in exception-handling constructs.

caller of `insert()`). Therefore, the exceptional-exit node in the CFG for `insert()` is connected, through an exceptional-return edge, to node 51 in the CFG for `main()`. A method may propagate an exception that is not caught in the immediate caller of that method, but by a method that lies further up in the call sequence. For example, `main()` calls `vend()`, and `vend()` calls `dispense()`. `dispense()` propagates an `IllegalAmountException`; `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 `IllegalAmountException`) 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 adds exceptional-exit nodes to the CFG of a method, one node for each exception type that is indirectly propagated by that method, and creates call, return, and exceptional-return edges for each call site in that method. The algorithm iteratively determines indirectly-propagated exception types for each method: if the algorithm creates a new exceptional-exit node for a method, it reprocesses all methods that call that method. An efficient implementation of the algorithm processes the nodes in a reverse topological order of strongly-connected nodes in the program's call multigraph,[10] which causes each non-recursive method to be processed only once; groups of recursive methods in a strongly-connected component, however, are processed iteratively until a fixed point is reached.

[10]A *call multigraph* for a program $\mathcal{P}$ contains a node N for each method in $\mathcal{P}$, 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.

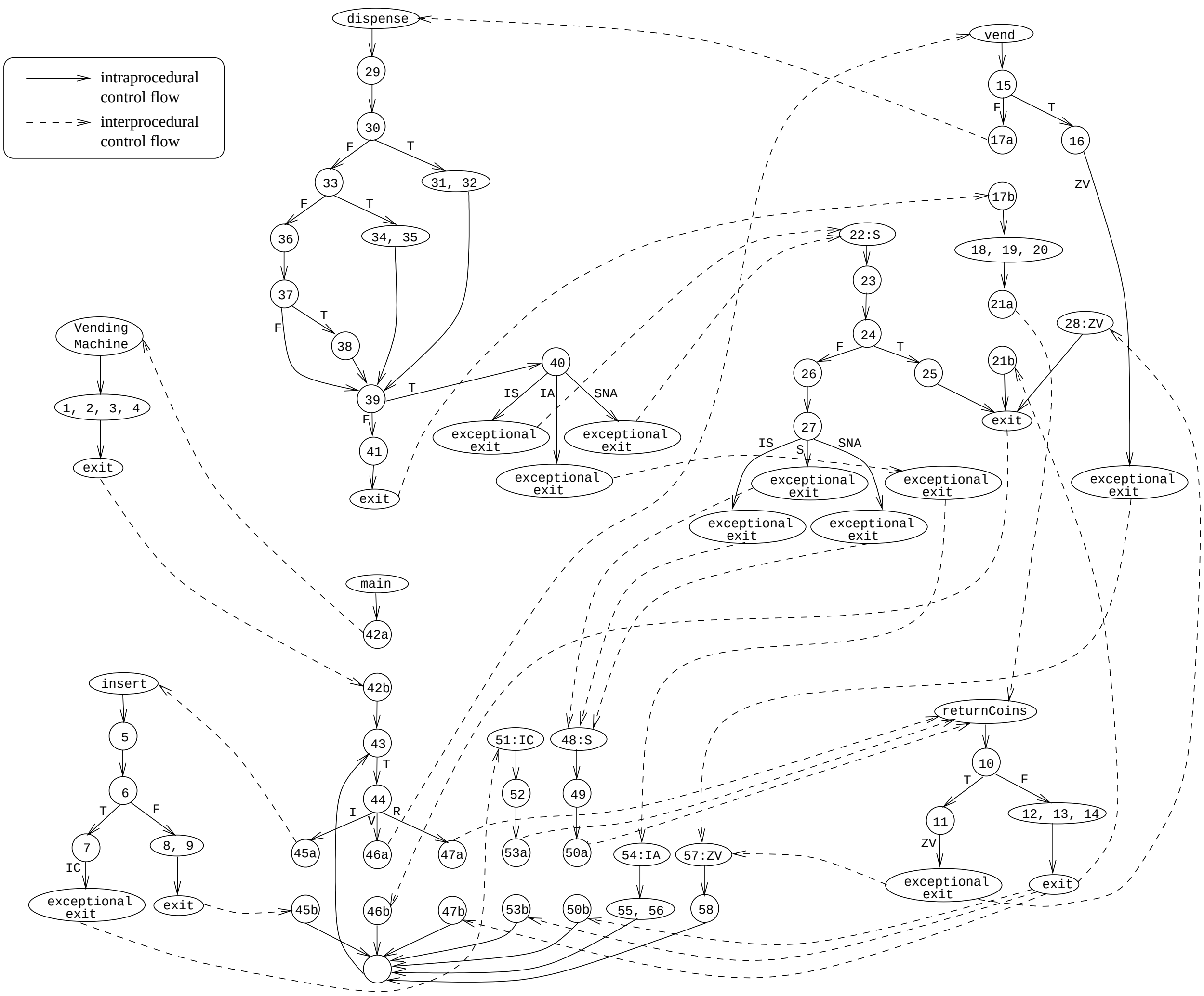


Figure 9: ICFG for the vending-machine program.

4.1.3 Precision of control-flow representations

Our CFG-construction algorithm requires information about exception types that can be raised at a `throw` statement. *Precise* type information at a `throw` statement includes only those types that can be reached at that `throw` statement along some execution path. The precision of the type information determines the extent to which infeasible[11] paths are introduced into our representations. In the CFG, each throw node has as many out-edges as the types of exceptions that can be raised at the corresponding `throw` statement — there is one out-edge for each exception type. Therefore, an imprecise (but safe) approximation of exception types results in additional edges emanating from throw nodes, such that paths starting along these edges are infeasible. For example, the path starting at the out-edge labeled `S` from node 27 in Figure 9 is infeasible because a `SelectionException` is not raised in the vending-machine program.

Type-inference algorithms (e.g., [26, 27]) attempt to determine the types for each expression in a program by solving type constraints, or propagating local type information through a program. Such techniques have

[11] A path is *infeasible* if there exists no input to the program that will cause the path to be executed.

traditionally been applied to optimization of dynamically-dispatched function calls. Recent work [5] uses context-sensitive 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 in which the exception type cannot be determined by an inspection of the `throw` statement. For example, a `throw` statement, such as the one in line 11 of the vending-machine program — whose expression is a new-instance expression — requires no type-inference analysis; the only type of exception that can be raised at that statement is `ZeroValueException`. The `throw` statement in line 40, however, requires type-inference analysis because it raises the exception object stored in a variable, and different exception objects are created and assigned to that variable along different paths to the `throw` statement. A `throw` statement may also raise an exception that is returned by a method called at that `throw` statement.

Table 2: Types of `throw` statement expressions.

Subject	`throw` statements	`throw` statement expressions: new-instance	variable	method-call
debug	61	56	5	0
jar	13	13	0	0
java_cup	30	29	1	0
javac	129	127	2	0
javadoc	5	5	0	0
jdk-api	703	683	20	0
AAS-Java	216	215	1	0
jlex	3	3	0	0
swing-api	352	336	4	12
Total	1512	1467	33	12

Our empirical evidence, based on the subjects listed in Table 1, indicates that in practice an overwhelming majority of `throw` statements mention a new-instance expression, and therefore, require no type-inference analysis. Table 2 lists the numbers and types of throw-statement expressions that appeared in our subjects. As the data in the table illustrates, 97% of the `throw` statements in our subjects required no type-inference analysis. Of the remaining 3%, two-thirds mentioned a variable. Only one of the subjects — `swing-api` — contained `throw` statements in which the exception object was returned by a method call.

Our approach to determining types for such `throw` statements avoids exhaustive propagation of type information through a program. Instead, while constructing the CFG for a method, we perform local type-inference analysis within that method, and safely approximate the exception types that can be raised at `throw` statements in that method. This local type-inference algorithm [33] performs a reverse data-flow analysis — starting at each `throw` statement — that searches for statements in which the types that reach this `throw` statement are defined. If the analysis reaches statements that define the types on all paths from the `throw` statement, it precisely identifies the exception types that (statically) reach the `throw` statement. For example, the type-inference algorithm traverses backwards on all paths from the `throw` statement in line 40, and precisely determines the types `IllegalSelectionException`, `SelectionNotAvailableException`, and `IllegalAmountException` for that `throw` statement.

If the analysis reaches the entry of the method, a `catch` handler, or a call site, it makes a conservative approximation of potential exception types. The algorithm can make a conservative approximation several times while analyzing the same `throw` statement because it traverses backwards along all paths starting at the `throw` statement. The conservative approximation can cause the algorithm to be imprecise. For example,

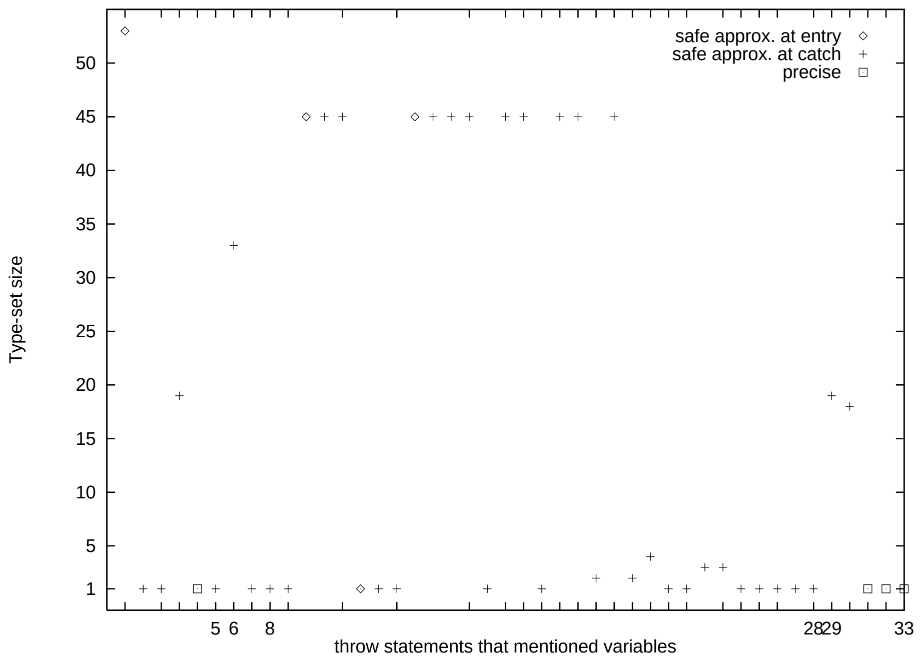


Figure 10: Sizes of type sets inferred by the type-inference algorithm for the cases where the `throw`-statement expressions were variables. The horizontal axis contains 33 calibrations, one for each `throw` statement that mentioned a variable. Data points for an indentation appear to the right of the preceding indentation and up to that indentation.

while traversing backwards on all paths from the `throw` statement in line 27, the analysis reaches the `catch` handler in line 22, and safely approximates the types `SelectionException`, `IllegalSelectionException`, and `SelectionNotAvailableException` for that `throw` statement. An inspection of the vending-machine program, however, reveals that the approximation is imprecise because `SelectionException` cannot be raised at that `throw` statement. The imprecision creates exceptional-exit node `S` in the CFG for `vend()` (Figure 7), although that node is not required.

Because the analysis makes a conservative approximation at the method boundary, it is safe even in the presence of aliasing. The algorithm is demand-driven because it computes information at the point at which the information is required, rather than exhaustively computing information for the entire program. The algorithm is flow-sensitive because it considers the control flow among statements in the method containing a `throw` statement, but is context-insensitive because it makes a conservative approximation at the method entry, a `catch` handler, or a call site.

To evaluate the efficacy of the type-inference algorithm, we performed a study, in which using the analysis results obtained by AAS-Java on our subject programs, we gathered empirical data about the sizes of the type sets that were inferred by the algorithm for each of the 33 `throw` statements that mentioned a variable. For each `throw` statement, we determined the type-set size when the type inferencing succeeded: the data-flow analysis terminated before reaching the entry node, a catch node, or a call node. We also determined the type-set sizes when the type inferencing failed, and made a safe approximation at either the entry node, a catch node, or a call node.

Figure 10 shows the type-set sizes for the 33 `throw` statements. The horizontal axis has 33 calibrations corresponding to the `throw` statements, and they are presented in the order in which the subjects are listed

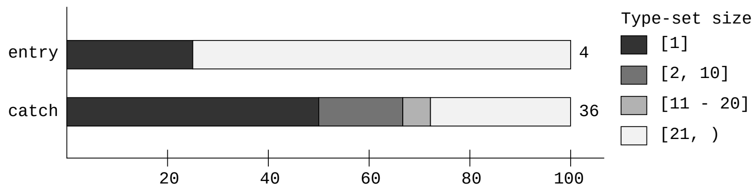


Figure 11: Type-set sizes for cases when the type-inference algorithm made conservative approximations at entry and catch nodes.

in Table 2. Thus, the first five calibrations represent the `throw` statements from `debug`, the sixth calibration represents the `throw` statement from `java_cup`, and so on. There are three sets of data points in the figure. One set of data points correspond to the cases where type inferencing succeeded, and therefore, type sets were precise. The other two sets of data points correspond to the cases where type inferencing failed by reaching the entry node and a catch node, respectively. For the `throw` statements that were analyzed, type inferencing never failed by reaching a call node.

Some of the calibrations on the horizontal axis have several data points: for a given calibration, all data points that appear to the right of the preceding calibration and up to that calibration correspond to that calibration. For example, the second calibration has two data points, and the tenth calibration has 3 data points. A calibration has more than one data point if the type-inference algorithm made conservative approximations several times. Thus, while analyzing the `throw` statement that corresponds to the second calibration, the type-inference algorithm made conservative approximations at two catch nodes.

Of the 33 `throw` statements for which the type-inference algorithm was invoked, it succeeded for four statements; the type sets inferred by the algorithm contained a single element in each of these cases. For 29 `throw` statements, the type-inference algorithm made a conservation approximation at the entry or a catch node. However, for 13 of these statements, the type sets contained a single element, and therefore, type inferencing was precise. For another four statements, the type sets contained less than five elements. For the remaining 12 `throw` statements, the type-set sizes ranged between 18 and 53. The imprecision of the type sets in such cases causes several infeasible branches in the CFGs.

Over all its invocations, the type-inference algorithm reached an entry node four times, and a catch node 36 times. Figure 11 summarizes the type-set sizes for these cases; it shows the percentages of type sets whose sizes were in various ranges. Thus, 50% of the times that type-inference reached a catch node, the conservative approximation resulted in a single-element type set.

We performed no type inferencing for the 12 `throw` statements in `swing-api` where the exception objects were returned by methods. For these cases, on average, the type sets contained 6 elements. In future work, we will extend the type-inference algorithm to perform analysis in the called methods.

The empirical results support our belief that, in practice, exhaustive type-inference algorithms may not be required to determine exception types. Our approach to type inferencing worked well for over 98% of the `throw` statements that appeared in the subjects. In 12 cases, the type sets contained between 18 and 53 elements, and were therefore, very imprecise. Such imprecision in type information can cause several infeasible edges in the CFGs that are constructed using that information.

Although our approach to type inferencing fails for a very small percentage of `throw` statements, the imprecision in these cases generates very large type sets. There are several possibilities for improving the precision of the type sets. For example, the type-inference algorithm can be extended at entry and catch nodes to perform analysis in other methods. This extension requires an iterative approach in which the ICFG construction and type inferencing are interleaved. Another optimization that may be applicable is to examine object-creation sites in the program, and utilize that information to refine the type sets. That approach may be unsafe, however, because exception objects may be created in library calls rather than in the application itself. A third approach is to propagate information about `catch` handlers that enclose call sites throughout the program using the call multigraph, and use this information to create exceptional-exit nodes. Another potential approach is to solicit user input for the cases where the imprecision of the type sets exceeds some threshold value. In future work, we will investigate the feasibility of each of these approaches.

4.2 Control-Dependence Analysis

In Section 3.3, we saw that exception-handling constructs affect the control dependence relations intraprocedurally and interprocedurally by causing potentially non-returning call sites (PNRCs). To compute correct control dependences, the control-dependence computation must account for these effects. The control-dependence computation proceeds in two phases [13]: Phase 1 identifies PNRCs that are caused by `throw` statements, and uses this information to compute partial control dependences; Phase 2 uses partial control dependences to propagate control-dependence information across methods and procedures 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 determines the set of return sites for each call node. The *return-site set* for a call node contains all nodes in the CFG of that method to which control can return following the call. If the return-site set for a call node contains only the corresponding return node, the call site represented by that call node is definitely returning. If the return-site set contains more than one node, the corresponding call site is a PNRC. For example, the return-site set for call node 17a contains node 17b, node 22, and exceptional-exit node `IA`; following the call to `vend()`, control may return to either of these nodes. Similarly, the return-site set for call node 21a contains nodes 21b and 28.

The PNRC-identification algorithm performs reachability analysis on a call multigraph to identify PNRCs that are caused by exception-handling constructs or halt statements.[12] For a PNRC that is caused by a halt statement, the algorithm adds a super-exit node (explained below) for the method containing the call site to the return-site set for that call site. To identify a PNRC caused by a halt statement, the algorithm first identifies methods that contain halt statements, and then through reverse graph-reachability analysis on the call multigraph, identifies call sites from which those halt statements are reachable. For a PNRC caused by exception-handling constructs, the algorithm adds to the return-site set of the corresponding call site each node where control can potentially return following the call. To identify such a PNRC, algorithm first initializes information about directly-propagated exception types for each method, and then iteratively computes indirectly-propagated exception types for all methods. This part of the algorithm is similar to the ICFG-construction algorithm (Section 4.1.2). If an ICFG is already constructed, the information that is gathered by this part of the algorithm can be directly extracted from the ICFG: at a PNRC, the target

[12] The standard API method `java.lang.System.exit()` is the halt statement in Java.

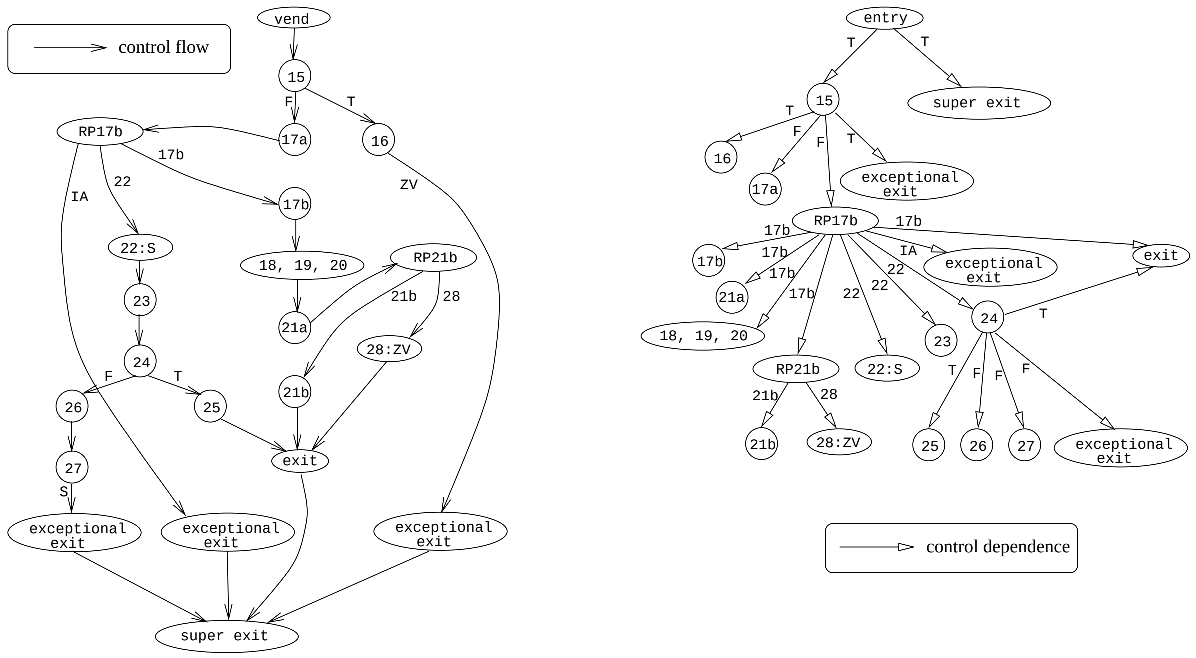


Figure 12: The ACFG for method `vend()` of the vending-machine program (left), and the CDG constructed from the ACFG (right).

of exceptional-return edges from the called methods are the nodes that are added to the return-site set for that PNRC.[13]

After the return-site set for each call site has been computed, 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 return-site set for the corresponding PNRC. An ACFG also contains a unique node, *super exit*, that represents all exits from M.

Figure 12 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 return-site set 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 out-edges to each node that appears in the return-site set 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 12 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

[13] The call-multigraph-based PNRC algorithm can suffer imprecision due to statically-unreachable code because the algorithm is flow-insensitive. A more precise, and thus 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, we do not expect the imprecision caused by statically-unreachable code to be significant.

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 3: 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)			

Table 3 lists the intraprocedural control dependences and 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 9 nodes, whereas they are the same for the remaining 6 nodes.

Partial control dependences have several useful applications: they can be used for computing slices [10], for computing procedure-level control dependences, and for computing interprocedural control dependences.

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 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.

The Phase 2 algorithm iteratively 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 only processes nodes, such as call, entry, return, exit, and catch, that are either sources or targets of interprocedural edges in the ICFG. After computing the fixed-point solution, the algorithm adjusts the control dependences of the remaining nodes.

For example, during the fixed point computation, the algorithm propagates the control dependence (39, F) from the exit node of the CFG for `dispense()` to return node 17b in the CFG for `vend()` along the return edge that connects the two nodes in the ICFG. After the fixed-point computation, the algorithm adjusts the control dependence of nodes 18, 19, and 20. For further example, the algorithm propagates the control dependence (40, SNA) from node 40 to node 22 along the exceptional-return edge that connects the two nodes; the algorithm later adjusts the control dependence of node 23, once the fixed-point solution has been computed. Table 4 lists the interprocedural control dependences for nodes in the CFG for method `vend()`.

4.2.3 Impact of exceptions on partial control dependences

To determine the extent to which the presence of exception-handling constructs affect control dependences, we gathered preliminary data about statements that had different intraprocedural and partial control depen-

Table 4: 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)		

dences. For each subject, we built CFGs and ACFGs for the methods in that subject using AAS-Java. 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 the CFG (excluding entry and exit nodes), we determined whether that node had different control dependences in the two CDGs.

Table 5: Effect of exception-handling constructs on partial control dependences.

Subject	Statement nodes	Nodes with affected partial control dependences
debug	11916	5270
jar	2064	276
java_cup	10331	1632
javac	28737	11897
javadoc	4573	1240
jdk-api	109860	51815
AAS-Java	26503	6123
jlex	6149	1972
swing-api	244188	139583

Table 5 presents the results of the study. It illustrates that the control dependences of a significant number of the nodes was affected: on average, control dependences of nearly half of the nodes in the subjects were affected. The percentage of affected nodes ranged from 13.4% for `jar` to 57.2% for `swing-api`.

These results are preliminary in that they do not indicate the actual differences in control dependences and neither do they indicate the impact of the differences in the control dependences on other analysis techniques, such a slicing, that use control dependences.

4.3 Program Slicing

Through their effects on interprocedural control dependence, exception-handling constructs can have significant impact on slice computation. Alternative techniques for slicing use either data-flow equations [34], or construct dependence graphs and perform graph-reachability on them [25, 7, 15] to compute slices. Reference [10] extends the data-flow-based slicing technique to compute slices in the presence of halt statements. Using the control-flow representations described in this paper, that technique can also compute slices in the presence of exception-handling constructs. In this paper, we describe extensions to the dependence-graph-based slicing technique to accommodate exception-handling constructs.

A *system-dependence graph* (SDG) [15] is a collection of *procedure-dependence graphs* (PDG) [7] — one for each procedure — in which nodes are statements or predicate expressions. *Flow-dependence* edges represent flow of data between statements or expressions; *control-dependence* edges represent control conditions on which the execution of a statement or expression depends. Each PDG contains an *entry* node that represents entry into the procedure. To model parameter passing,[14] an SDG associates formal parameter nodes with each procedure-entry node: a *formal-in* node for each formal parameter of the procedure; a *formal-out* node for each formal parameter that may be modified [19] by the procedure. An SDG associates a *call* node and a set of actual parameter nodes with each call site in a procedure: an *actual-in* node for each actual parameter at the call site; an *actual-out* node for each actual parameter that may be modified by the called procedure.

An SDG connects PDGs at call sites. A *call* edge connects a call node to the entry node of the called procedure's PDG. Parameter-in and parameter-out edges represent parameter passing: *parameter-in* edges connect actual-in and formal-in nodes, and *parameter-out* edges connect formal-out and actual-out nodes. Horwitz, Reps, and Binkley [15] compute interprocedural slices by solving a graph-reachability problem on an SDG. To restrict the computation of interprocedural slicing to paths that correspond to legal call/return sequences, an SDG uses summary edges to explicitly represent the transitive flow of dependence across call sites caused by data dependences, control dependences, or both. A *summary edge* connects an actual-in node and an actual-out node if the value associated with the actual-in node may affect the value associated with the actual-out node.

The initial work on SDG-based slicing described the SDG for imperative-language programs only. Recent work [11, 22] has extended the SDG to represent object-oriented programs. One of those extensions creates additional parameter nodes at a call site that sends a message to an object — one parameter node for each instance variable (of the receiver of the message) that is referenced or modified by the method invoked in response to the message.

Figure 13 shows the partial SDG for the vending-machine program that is constructed using the technique presented in Reference [15]. The figure shows the complete PDGs for `VendingMachine()`, `vend()`, and `dispense()`, and the partial PDG for `main()`; to simplify the presentation, the figure omits the PDGs for other methods. The figure represents call sites with a single node, because the traditional SDG does not distinguish call and return nodes. Call node 17 contains actual parameter nodes for *currValue* (shown as *cVal*) and *selection* (shown as *sel*) that are passed as parameters to the called method. Call node 46 contains parameter nodes for those instance variables of class `VendingMachine` that are referenced or modified by the member method `vend()`. The catch nodes 22 and 28 have no control dependence in-edges because intraprocedurally the control dependences of those nodes are undefined.

The SDG-based interprocedural slicing algorithm consists of two phases [15]. The first phase traverses backwards from the node in the SDG that represents the slicing criterion along all edges except parameter-out edges, and marks those nodes that are reached. The second phase traverses backwards from all nodes marked during the first phase along all edges except call and parameter-in edges, and marks reached nodes. The slice is the union of the marked nodes.

The shaded nodes in Figure 13 represent the nodes that are included in the slice for the criterion $< 20, currValue >$; the lighter nodes are included in the first phase, and the darker nodes are included in the second phase of the slicing algorithm. The slice fails to include statement 39 and its transitive dependences even though the slicing criterion is semantically dependent on statement 39.

Slicing on the traditional SDG fails to correctly identify these dependences because the traditional SDG is based on intraprocedural control dependences. Moreover, because the traditional SDG does not accom-

[14] Global variables are treated as parameters.

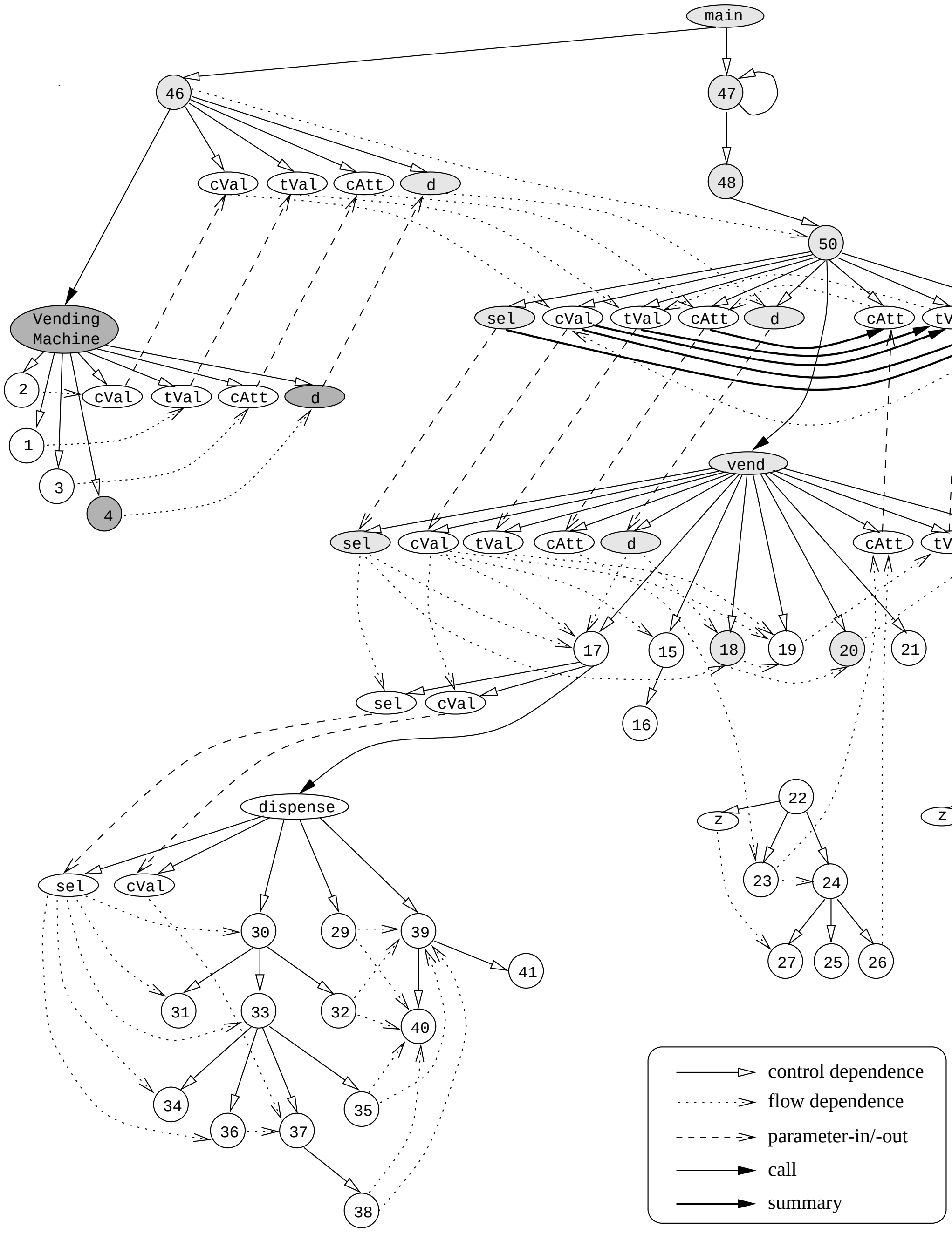


Figure 13: Partial SDG for the vending-machine program. The shaded nodes are included in the slice computed for criterion $< 20, \mathit{currValue} >$.

modate exception-handling constructs, it does not permit a slice computation that is initiated at a catch node. Figure 13 illustrates that a slice computation that begins at nodes 22 or 28 terminates immediately because those nodes have no in-edges.

To overcome its inadequacies, we extend the traditional SDG in several ways. The extended SDG represents the effects on control dependences caused by PNRCs, and accommodates exception-handling constructs.

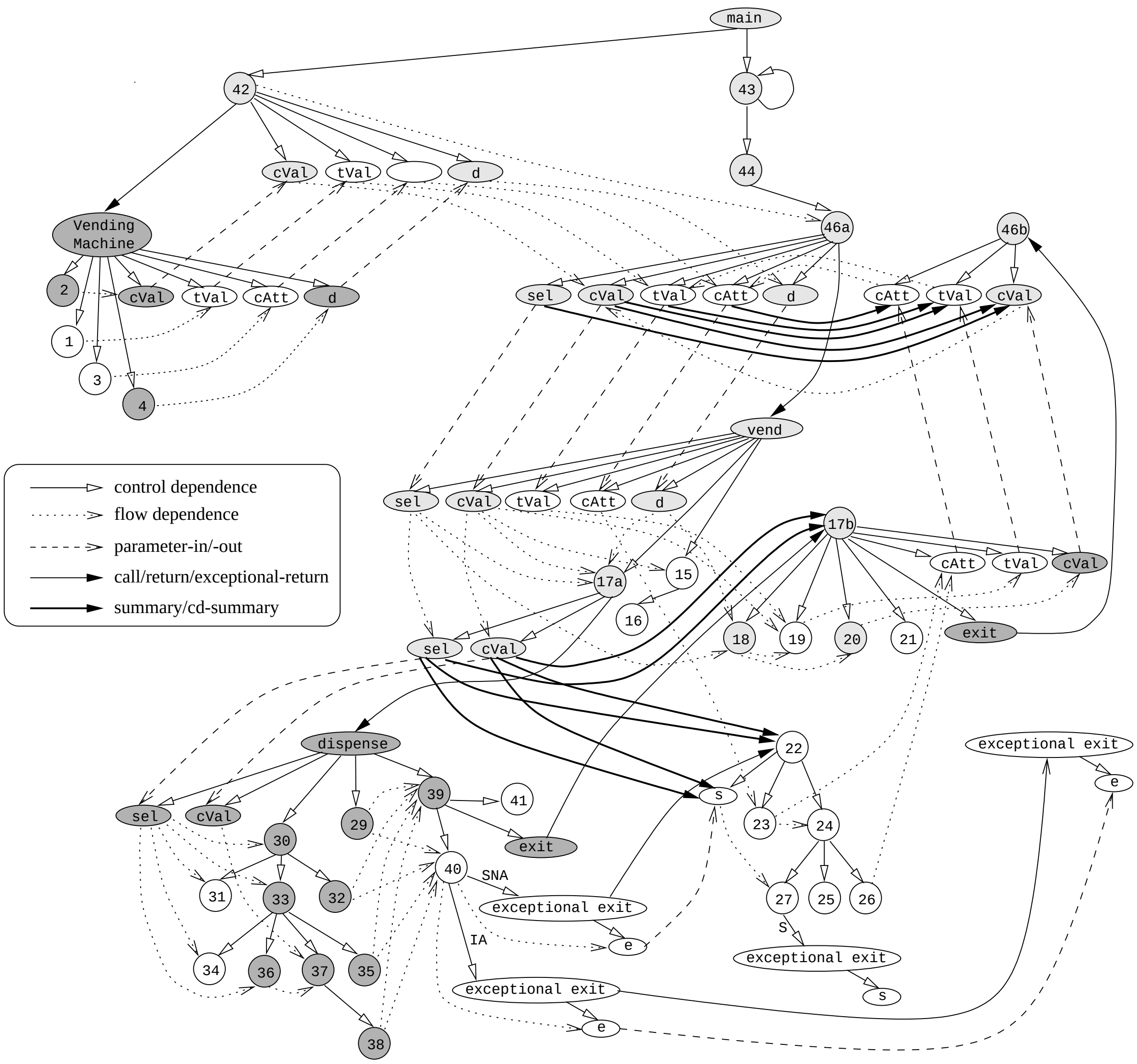


Figure 14: Partial extended SDG for the vending-machine program. The shaded nodes are included in the slice computed for criterion $< 20, currValue >$.

First, the extended SDG uses both call and return nodes to represent each PNRC, and entry and exit nodes to represent each method that can be reached from a PNRC. The extended SDG also contains return edges that, similar to call edges in the traditional SDG, connect exit nodes in called methods to the associated return nodes in calling methods. Two nodes are associated with a PNRC to represent the differences in control dependences of the call and return sites; two nodes are associated with procedures called from PNRCs for a similar reason. Previous approaches require only call and entry nodes, and thus, only call edges, because they assume that all called methods return to their call sites.

Figure 14 shows the partial extended SDG for the vending-machine program. Because the call site in line 46 is a PNRC, it is represented by call and return nodes in the extended SDG. At that call site, a call edge connects call node 46a to the entry node of `vend()`, and a return edge connects the exit node of `vend()` to the corresponding return node 46b.

Second, in the extended SDG, actual-in parameter nodes are associated with the call node, and actual-out parameter nodes with the return node; similarly, formal-in parameter nodes are associated with the entry node, and formal-out nodes with the exit node. The assignments represented by actual-out and formal-out nodes occur only if control reaches the corresponding return or exit node; associating the actual- and formal-out nodes with return and exit nodes, therefore, correctly represents dependences for those parameter nodes in the extended SDG. Previous approaches associate both actual-in and actual-out nodes with call nodes, and both formal-in and formal-out nodes with entry nodes.

For example, the actual-in nodes for the call site in line 46 are associated with the call node 46a, whereas the actual-out nodes are associated with the return node 46b.

Third, in addition to the summary edges that represent transitive data and control dependences for variables that are modified by a method, the extended SDG contains summary edges from actual-in node to return node; these edges represent transitive data and control dependences that influence return from a called method. To distinguish these types of summary edges, we call the former (summary edges in the traditional SDG) *df-summary* edges because they affect the data flow in the calling method, and the latter *cd-summary* edges because they affect the control dependences in the calling method. The cd-summary edges represent the effects on control dependence caused by PNRCs; they enable computation of interprocedural slices with minor changes to the interprocedural slicing algorithm.

For example, in Figure 14, the summary edge from the actual-in node for *currValue* at node 17a to node 17b is a cd-summary edge, and indicates that return of control from the call site at line 17 depends on the value of *currValue*.

Fourth, to accommodate exception-handling constructs, the extended SDG treats a `catch` node like a return node, and an exceptional-exit node like an exit node. Thus, an exceptional-exit node is connected to a catch node or another exceptional-exit node through an exceptional-return edge. For example, the exceptional-exit node `SNA` in the PDG for `dispense()` is connected to catch node 22. To represent the interprocedural data dependences that are introduced by an exception object, the extended SDG associates a formal-out parameter (for the exception object) with each exceptional-exit node, and an actual-out node (for the catch parameter) with each catch node for an interprocedural or a global handler. In the PDG for `dispense()`, the exceptional-exit nodes have a formal-out parameter for the exception object e. Catch node 22 has an actual-out node corresponding to the parameter s of that `catch` handler. A parameter-out edge connects these two parameter nodes to model the data dependence caused by the exception object.[15]

To ensure that interprocedural slices that include nodes from a `catch` handler are computed along legal call/return sequences, the extended SDG creates summary edges at catch nodes: df-summary edges connect actual-in nodes at a call node to actual-out nodes at a catch node, and cd-summary edges connect actual-in nodes at a call node to a catch node. Thus, catch node 22 has cd-summary edges incident on it that emanate from actual-in nodes *cVal* and *sel* at call node 17a. These edges indicate that the values of *cVal* and *sel* determine whether control flows to catch node 22 following the call. The df-summary edges incident on the actual-out node s at node 22 represent the transitive control and data dependences that determine the value of s. Using these summary edges, a slice computation that includes nodes from a `catch` handler correctly

[15]For clarity in Figure 14, we omit the exceptional-exit node `IS` from the PDG for `dispense()`. We also omit catch node 28, and summary edges incident on exceptional-exit node `IA` in the PDG for `vend()`; these would be represented similar to how catch node 22 is represented.

accounts for legal call/return sequences. For example, a slice computation that begins at the actual-out node `cAtt` at node 17b, includes nodes 26, 23, 22 in the slice during the first phase; on reaching node 22, the computation avoids descending into method `vend()` by following the cd-summary edges incident on node 22, and continues slicing at the actual-in nodes at node 17a. During the second phase, the slice computation enters the PDG for `dispense()` along the exceptional-return edge incident on node 22, and adds nodes from that PDG into the slice.

4.4 Analysis of Library Routines

The preceding discussions apply to application programs in which exceptions are raised explicitly. An application program can, however, raise exceptions implicitly through calls to library routines. The results presented in the paper are not based on analyses of those calls.

There are several orthogonal issues that need to be addressed in the analysis of library routines. The first issue concerns which library calls are considered for additional analysis. A safe approach that considers all library calls for analysis may be inordinately expensive. An alternative approach is based on exception types that are actually caught in the program being analyzed. This approach considers, for additional analysis, only those library calls that are enclosed by `catch` handlers. A program may in fact contain several `catch` handlers that handle only exceptions raised by library routines. Such a `catch` handler would have no in-edges even in the ICFG because it is reachable only along paths where a library routine raises an exception.

To determine the occurrences of such `catch` handlers, we constructed the ICFG for each subject, and identified those catch nodes that that has no in-edges in the ICFG. Table 6 shows that such `catch` handlers occur frequently in our subjects: on average, 47% of the `catch` handlers are such that they are not reachable in the ICFG.[16]

Table 6: Catch handlers that handle exceptions raised in library routines.

Subject	Number of `catch` handlers	Catch nodes with no in-edges in the ICFG	
		Total	Percent
debug	72	46	63.9
jar	14	10	71.4
java_cup	8	3	37.5
javac	128	31	24.2
javadoc	25	7	28.0
jdk-api	236	101	42.8
AAS-Java	105	81	77.1
jlex	2	1	50.0
swing-api	397	185	46.6
Total	987	465	47.1

The second issue in the analysis of library routines concerns how the exception types that can potentially be raised by a library routine are determined. An inspection of the library routine's signature would suffice to determine the potential checked exceptions that the routine can raise. Such an inspection may not, however, reveal unchecked exceptions that can be raised by that routine because a Java method is not obligated to declare in its signature any unchecked exceptions that it can raise. In such cases, execution profiles of the program obtained on test cases can be used to identify library routines that can raise exceptions. This

[16]Such a `catch` handler may represent dead code if the corresponding exceptions are not raised in either the application or the library routines.

approach requires modifications to the Java Virtual Machine, and its success depends on the thoroughness of the test cases.

The final issue in the analysis of library routines addresses what information is saved about those library routines that have been determined to potentially raise exceptions. At the simplest level, the information could simply record the set of exception types that can be raised. At successively higher costs that would facilitate more precise analysis, the information could also record the parameters of the library routine that determine whether exceptions are raised, and constraints on the values of those parameters that cause the exceptions to be raised.

These are issues that we will consider in the future to extend the analyses to incorporate library routines.

5 Applications of the Representations and Analyses

The representations that we presented in the previous sections can be used for many different testing and maintenance tasks. This section discusses briefly two such tasks: structural testing and regression test selection. For each task, we discuss the implications of exception-handling constructs for the task.

5.1 Structural Testing and Coverage

Structural testing techniques [24] use a program's structure to guide the development of test cases. For example, in branch testing, test cases are developed by considering inputs that cause certain branches in the program under test to be executed; similarly, path testing considers test cases that execute certain paths in the program. *Structural coverage* techniques give a measure of how well the structural elements of the program are executed by a given test suite. For example, the branch-coverage measure of a test suite is the percentage of branches in the program that are executed by the test cases in the test suite.

Structural testing is typically performed at several levels. Each level of testing has specific goals, and thus, appropriate coverage criteria that are directed towards attaining those goals. For example, unit testing tests each individual module in isolation. Integration testing, on the other hand, tests the interactions of modules that are caused by control and data coupling.

Testing requirements for exception-handling constructs are often stated informally, and in terms of coverage of `throw` statements and `catch` blocks only. For example, existing testing tools for Java programs, such as JavaScope from SunTest,[17] provide `catch` block coverage which requires the execution of all `catch` blocks; the tool, therefore, provides for an *all-catch* criterion. Other informally-stated criteria [3] require all exceptions to be raised, which can be re-stated as an *all-throw* criterion.

These criteria, however, emphasize simple coverage of a `throw` statement or a `catch` block; they do not consider the types of exceptions that can be raised at a `throw` statement or the types of exceptions that can be handled by a `catch` block. For example, during the unit testing of `vend()`, there are three alternatives for testing the `throw` statement in line 40: (1) execute the statement with a single test case; (2) execute the statement with one test case for each type of exception that can be raised at that statement; (3) execute the statement with tests for a subset of exception types that can be raised at that statement. Because potential exception types are explicitly represented in our CFGs, such a testing methodology can easily be implemented using our representations. Similarly, the unit testing of `dispense()` can select a suitable

[17] See `http://www.sun.com/suntest/products/JavaScope/index.html` for information on JavaScope.

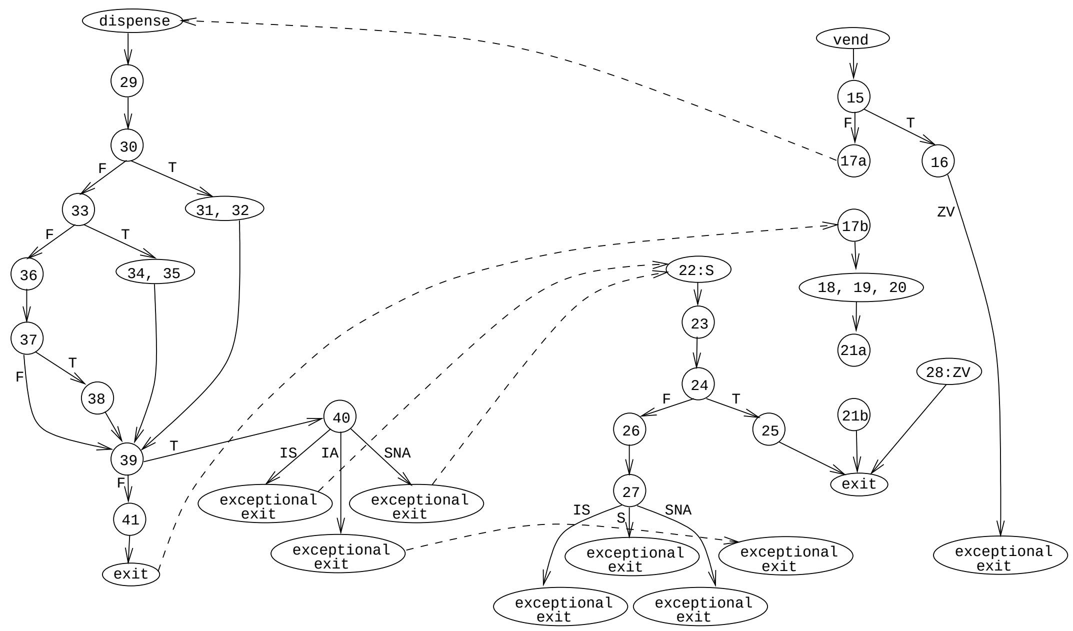


Figure 15: Portion of the ICFG that is constructed to determine the interactions of methods `vend()` and `dispense()` for integrating them.

alternative for testing the `catch` blocks by considering the types of exceptions that can be handled by those blocks.

To effectively test programs during integration testing, more rigorous criteria, that test the interactions of `throw` and `catch` statements, are required. For example, during the integration of methods `vend()` and `dispense()`, the all-throw and all-catch criteria require the executions of statements 40, 22, and 28. One drawback of this testing is that data coupling between `vend()` and `dispense()` caused by the exception object is not adequately tested: although each of the definitions of the exception object in statements 32 and 35 in `dispense()` is used at the `catch` statement 22 in `vend()`, the criteria require only one of those definitions to be exercised. Moreover, a testing criterion to test the integration of `vend()` and `dispense()` should exclude the coverage of `catch` statement 28 because that statement does not contribute to the interactions between `vend()` and `dispense()`. The interactions between `vend()` and `dispense()` can be analyzed by constructing the ICFG for the program; Figure 15 shows the partial ICFG that illustrates the interactions of `vend()` and `dispense()`.

Without performing analysis that identifies exception propagation and interactions between modules that are caused by exceptions, effective and adequate testing of exception-handling constructs is not attainable. Our current work includes development of a rigorous and structured methodology for testing exception-handling constructs at different levels.

5.2 Regression Test Selection

Regression testing is an expensive testing procedure that attempts to validate modified software and ensure that new errors are not introduced into previously tested code. One method for reducing the cost of regression testing is to save the test suites that are developed for a product, and reuse them to re-validate the product after it is modified. A *retest-all* approach re-runs all such test cases, but may consume excessive time and

```
     public void vend( int selection ) {
15     if ( currValue == 0 ) {
16       throw new ZeroValueException();
       }
       try {
17       d.dispense( currValue, selection );
18       int bal = d.value( selection );
19       totValue += currValue - bal;
20       currValue = bal;
21       returnCoins();
       }
22a    catch( IllegalSelectionException s ) {
22b      showMsg( "Transaction aborted" );
22c      returnCoins();
       }
22     catch( SelectionException s ) {
23       currAttempts++;
24       if ( currAttempts < MAX_ATTEMPTS ) {
25         showMsg( "Enter selection again" );
         }
         else {
26         currAttempts = 0;
27         throw s;
         }
       }
28     catch( ZeroValueException z ) {
       }
     }
```

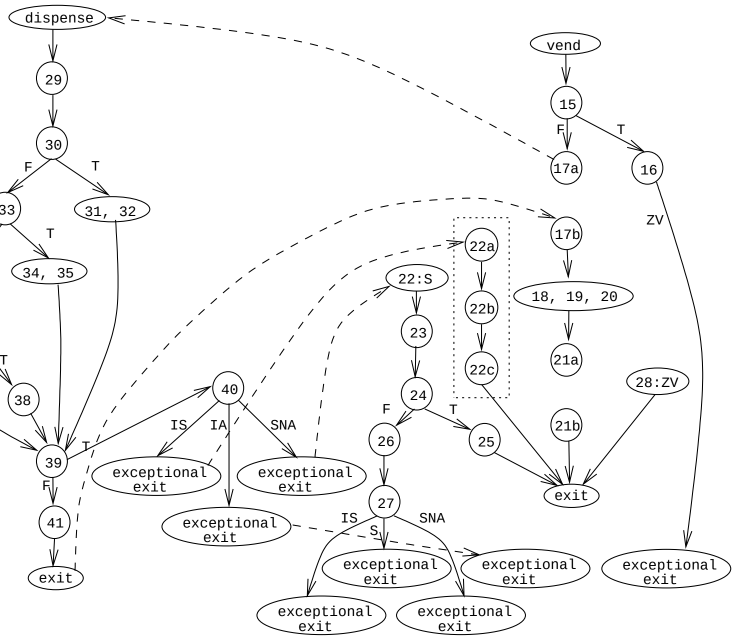


Figure 16: Modified version of `vend()` with a new `catch` handler (left), and the partial ICFG for the modified program that connects `vend()` and `dispense()` (right).

---

resources. *Regression test selection techniques*, in contrast, attempt to reduce the cost of regression testing by selecting a subset of the test suite that was used during development, and using that subset to test the modified program.

Rothermel and Harrold developed a regression test selection technique [30] that uses a program's CFG. Their technique first creates the CFG of the original program. Then, it uses this CFG to insert probes into the program so that, when the instrumented program is executed with a test case, it reports those edges that were executed. After this instrumented version of the program is executed with all test cases in the test suite, the technique creates a test history that associates with each edge, those test cases that executed it. When the program is modified, the technique creates the CFG for the modified program. The technique then walks the CFGs for the original and modified programs in a depth-first manner until, upon considering like-labeled edges in these CFGs, it finds that the text associated with the sink of the edge in the original program differs from the text associated with the sink of the edge in the modified program. At this point, the technique selects all test cases that, according to the test history, are associated with that edge.

Although this technique has been applied to object-oriented programs [29, 31], it does not consider the effects of exception-handling constructs on the test selection. Consider a modification to the vending-machine program so that in the modified version, an illegal selection causes the transaction to be aborted immediately, and does not allow the user to re-enter a selection. The modification is implemented by adding a new `catch` handler to method `vend()` that catches exceptions of type `IllegalSelectionException` and aborts the transaction. Figure 16 shows the modified code for `vend()`, and the portion of the ICFG that is affected by the modification to `vend()` (the dotted box in the figure shows the changed code). Because of the addition of the new `catch` handler, the exceptional exit node `IS` in `dispense()` is connected to node 22a, rather than to node 22 as it was in the ICFG of the original program (Figure 9).

Assuming that a test suite exists for the original version of the vending-machine program, the only tests from that suite that should be selected to test this modification are those that traversed the exceptional-return edge emanating from exceptional-exit node `IS` in `dispense()` in the original ICFG. Selecting tests that traverse any other edges, such as the edge (40, exceptional-exit `IS`), causes the test selection to be imprecise. Therefore, without constructing our interprocedural representation of the program, a precise test selection for the modification described above is not possible.

In its current version, the technique described in Reference [30] associates tests only with edges in the CFG; therefore, the technique does not consider tests associated with interprocedural edges in the ICFG. The technique also needs to be adapted so that it functions on multiple-entry, multiple-exit CFGs that result in the presence of exception-handling constructs. Our current work includes extending that technique, and investigating other issues related to regression test selection for programs that contain exception-handling constructs.

6 Conclusions

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 explicitly show exception types that can be raised at `throw` statements, and exception types that are propagated across methods. The representations can, therefore, provide valuable aid in understanding the behavior of exception-handling code in large software systems.

We discussed the impact of exception-handling constructs on analysis techniques, such as control-dependence and program slicing. The preliminary results that we gathered (Table 5) indicate that control dependences of a significant number of statements are affected by the presence of exception-handling constructs. Control dependences that are computed for such statements by traditional techniques can omit dependences and include incorrect dependences. Incorrect control dependences affect the computation of slices. Further experimentation with control-dependence computation and program slicing will reveal the extent to which those techniques are affected by exception-handling constructs.

Our studies provide several insights into the usage patterns of exception-handling constructs in Java programs:

- Type inferencing to determine exception types at `throw` statements is not required for a majority of the `throw` statements. In 97% of the `throw` statements in our subjects (Table 2), the exception object is instantiated at the `throw` statement.
- In cases where a `throw` statement mentions a variable, the exception object is rarely passed into the method as a parameter; the object is more often caught at a `catch` handler in that method and is then rethrown.
- Exceptions are frequently propagated out of the methods in which they are raised.
- A large percentage of the `catch` handlers that appear in the programs handle exceptions that are raised in library routines.

These insights help guide the development of a practical approach to analyzing exception-handling constructs, and improving on the techniques that we have developed. Our type-inferencing approach performs local, flow-sensitive, and demand analysis to determine exception types. The technique computed imprecise results for one-third of the `throw` statements on which it was invoked. The impact of this imprecision on the CFG construction needs to be investigated further. We outlined several extensions for improving the precision of our type-inference algorithm, and approaches to analyzing library routines. Experimentation is also required with a more diverse set of experimental subjects than we have considered so far to see whether our results generalize.

We discussed the implications of exception-handling constructs for structural testing and regression test selection. By introducing new structural elements and control-flow paths in programs, the presence of exception-handling constructs imposes additional requirements on test data generation, coverage identification, and test selection. Our current work includes an investigation of these issues, and development of a rigorous and disciplined testing methodology for programs that contain exception-handling constructs.

7 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 AAS-Java, 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 intraprocedural control-flow analysis technique and representation. Gerald Baumgartner, Donglin Liang, and Gregg Rothermel made many helpful comments on previous versions of the paper.

References

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

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

[3] E. V. Berard. Issues in the testing of object-oriented software. Available at http://www.toa.com/pub/net_articles/oo_testing_article.txt.

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

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

[6] 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.*, 13(4):450–90, October 1991.

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

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

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

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

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

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

[13] 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*, pages 11–20, March 1998.

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

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Prog. Lang. and Sys.*, 12(1):26–60, Jan. 1990.

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

[17] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen.

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

[19] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. pages 56–67, June 1993.

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

[21] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. on Softw. Eng.*, (3):347–354, May 1983.

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

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

[24] S. Ntafos. A comparison of some structural testing strategies. *IEEE Transaction on Software Engineering*, 14(6):868–874, June 1988.

[25] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proc. of the ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. on Practical Softw. Dev. Env.*, pages 177–184, Mar. 1984.

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

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

[28] A. Podgurski and L.A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–79, September 1990.

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

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

[31] G. Rothermel, M. J. Harrold, and Dedhia J. Regression test selection for C++ software. Technical Report Technical Report 99-60-01, Oregon State University, 1999.

[32] C. F. Schaefer and G. N. Bundy. Static analysis of exception handling in Ada. *Software—Practice and Experience*, 23(10):1157–1174, Oct. 1993.

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

[34] M. Weiser. Program slicing. *IEEE Trans. on Softw. Eng.*, 10(4):352–357, July 1984.