Technical Report OSU-CISRC-6/99-TR16, June 1999

Criteria for Testing Exception-Handling Constructs in Java Programs

Saurabh Sinha and Mary Jean Harrold
Department of Computer and Information Science
The Ohio State University
395 Dreese Lab, 2015 Neil Avenue
Columbus, OH 43210 USA
{sinha,harrold}@cis.ohio-state.edu

Abstract

Exception-handling constructs provide a mechanism for raising exceptions, and a facility for designating protected code by attaching exception handlers to blocks of code. Despite the frequency of their occurrences, the behavior of exception-handling constructs is often the least understood and poorly tested part of a program. The presence of such constructs introduces new structural elements, such as control-flow paths, in a program. To adequately test such programs, these new structural elements must be considered for coverage during structural testing. In this paper, we describe a class of adequacy criteria that can be used to test the behavior of exception-handling constructs. We present a subsumption hierarchy of the criteria, and illustrate the relationship of the criteria to those found in traditional subsumption hierarchies. We describe techniques for generating the test requirements for the criteria using our control-flow representations. We also describe a methodology for applying the criteria to unit and integration testing of programs that contain exception-handling constructs.

Keywords: Structural testing, exception handling, unit testing, integration testing.

1 Introduction

Structural testing techniques [16] 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 [8]. *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.[1]

Control-flow-based structural testing criteria use a program's control-flow structure to guide the selection of test cases (e.g., [8, 9, 14]). *Data-flow-based* structural testing criteria use data-flow relationships in a program to guide the selection of test cases (e.g., [3, 6, 10, 15, 18]). Structural testing can be performed at several levels. Each level of testing has specific goals, and thus, has appropriate coverage criteria that are directed towards attaining those goals. Unit testing, for example, tests each individual module in isolation. Integration testing, on the other hand, tests the control and data interactions of the modules (e.g., [5, 7, 11]).

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

To evaluate the relative strengths of different test-selection criteria, the criteria are presented in a hierarchy that describes the subsumption relationship between the criteria [3, 16, 18]. Criterion A *subsumes* criterion B if and only if any test suite that satisfies A also satisfies B [3, 18]. Path testing is the strongest criterion for testing but is usually infeasible because programs with loops can have an infinite number of paths. Branch and statement testing, although frequently used, are the weakest criteria in these hierarchies, and may fail to reveal many faults in the program under test. Other control-flow-based and data-flow-based testing criteria fall between these two extremes.

Exception-handling constructs provide a mechanism for raising exceptions and a facility for designating protected code by attaching exception handlers to blocks of code. Several languages, such as Java, C++, and Ada, provide such constructs. A recent study of Java programs indicated that such constructs occur frequently [20]. Despite the frequency of their occurrences, the behavior of exception-handling constructs is often the least understood and poorly tested part of a program [19]. The presence of such constructs introduces new structural elements, such as control-flow paths in a program. To adequately test such programs, these new structural elements must be considered for coverage during structural testing.

Test requirements for exception-handling constructs are often stated informally, and lack rigor and discipline. For example, existing testing tools for Java programs provide coverage of all exception handlers in a program [22]. Other informally-stated criteria require all exceptions to be raised in the program under test. These criteria lack a systematic and structured approach to testing the behavior of exception-handling constructs. The criteria require simple coverage of statements that raise exceptions, and those that handle exceptions. The criteria do not require testing different types of exceptions that can be raised or handled at the same statement; nor do they require a systematic testing of the data and control interactions within and across modules that result from the presence of exception-handling constructs. These criteria therefore, suffer from the same weakness as statement coverage.

One reason for the absence of a structured and disciplined approach to testing the behavior of exception-handling constructs is the lack of a representation that depicts the behavior. In recent work [20], we described intraprocedural (within a single module) and interprocedural (across modules) representations for programs that contain exception-handling constructs. In this paper, we describe a class of adequacy criteria that can be used to rigorously test the behavior of exception-handling constructs. We first describe the exception-handling constructs in Java, and the control-flow representations for those constructs (Section 2). We then present a hierarchy of exception testing criteria, and illustrate the relationship of the criteria to those found in traditional data-flow testing criteria [3, 18] (Section 3). We also describe techniques for generating the testing requirements for the criteria using our control-flow representations, and describe a methodology for applying the criteria to unit and integration testing of programs that contain exception-handling constructs (Section 4). Finally, we discuss related work (Section 5), and present conclusions (Section 6).

2 Background

In this section, we provide a brief overview of exception-handling constructs in Java, our language model; details of the Java language can be found in Reference [4]. We also discuss our control-flow representations that account for exception-handling constructs.

class C2

```
  enter M1
 1  if <condition>
 2     throw new E3()
    try
 3     while <condition>
          try
 4a          call M2
 4b          return M2
 5        finally
 6           if <condition>
 7              <statement>
 8           if <condition>
 9              continue
10           if <condition>
11              throw new E3()
12     <statement>
13  catch( E2 e2 )
14     if <condition>
15        throw e2
16     print e2
17  <statement>
  exit M1
```

```
  enter M2
     E e1;
18   if <condition>
19      e1 = new E1()
20   else e1 = new E21()
21   if <condition>
22      <statement>
23   else <statement>
24   if <condition>
25      throw e1
26    print e1
  exit M2
```

class C1

```
    enter M
      try
27a     call M1
27b     return M1
28  catch( E e )
29     print e
30  <statement>
    exit M
```

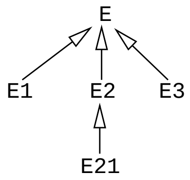

Figure 1: Pseudo-code for sample program that uses Java-like exception-handling constructs (above); hierarchy of exceptions used in the sample program (below).

2.1 Exception-Handling Constructs

In Java, an exception is an object: it is an instance of a class derived from class `java.lang.Throwable`, which is defined in the standard Java API. An exception can be raised at any point in the program, through a `throw` statement. The expression associated with the `throw` statement denotes the exception object; the expression may represent a variable (for example, `throw e`), a method call (for example, `throw m()`), or a new-instance expression (for example, `throw new E()`). 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`. A `try` block contains statements whose execution is monitored for exception occurrences. A `catch` block, which may be associated with each `try` block, is a sequence of `catch` clauses that specify *exception handlers*. Each `catch` clause specifies the type of exception it handles, and contains a block of code that is executed when an exception of that type is raised in the associated `try` block. A `catch` clause also specifies a *catch variable*: a variable that is initialized with the handled exception, and whose scope is limited to the block of code for that `catch` clause. A `try` statement can have a `finally` block. The code in the `finally` block 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 exception that may or may not be handled in the associated catch block, or by reaching a `break`, `continue`, or `return`

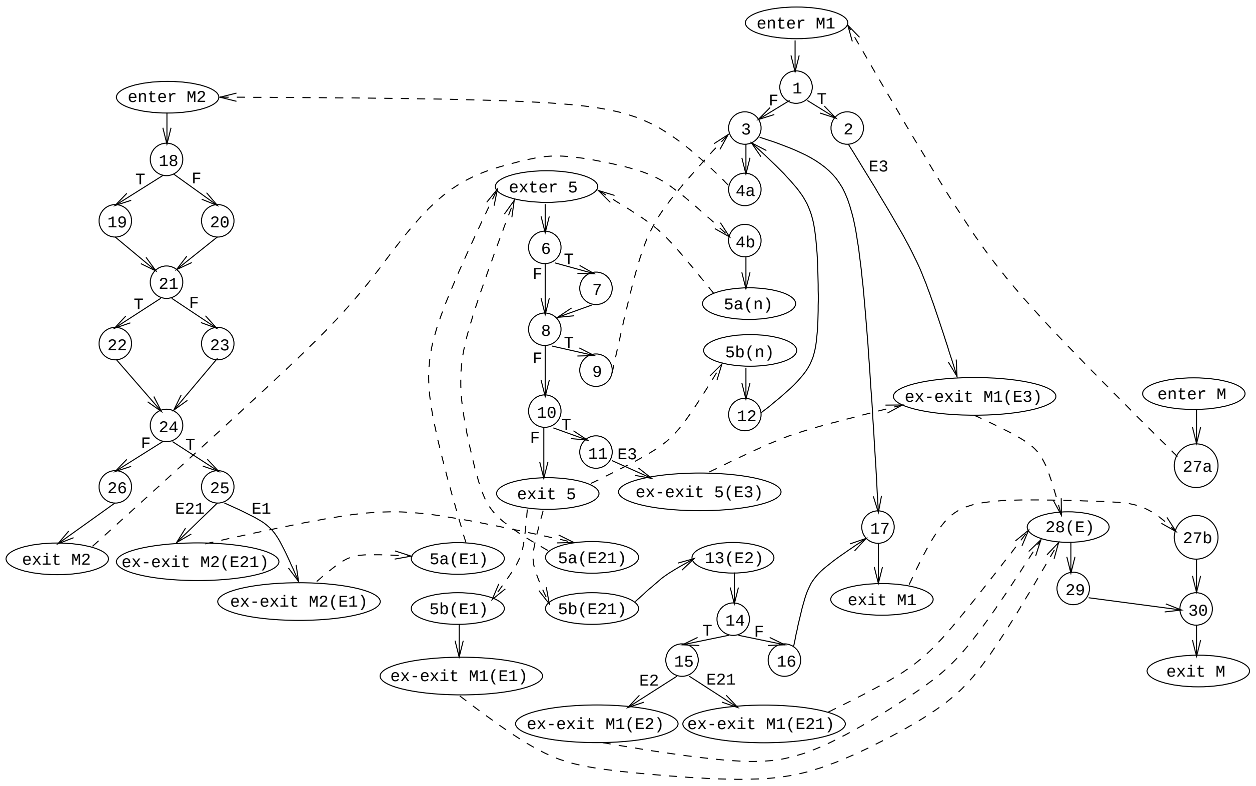

Figure 2: Interprocedural control-flow graph for the sample program.

statement.

Figure 1 shows the pseudo-code of an example program that uses a Java-like syntax to illustrate instances of a `try` statement; the inheritance hierarchy at the bottom of the figure illustrates the exception classes used by the program. For example, the expression of the `throw` statement in line 2 of the program is a new-instance expression; that statement raises an exception of type `E3`. The expression of the `throw` statement in line 25, however, is a variable, and that statement can raise an exception whose type is either `E1` or `E21`. Method `M` in class `C1` contains a `try–catch` statement; the `catch` clause of that statement handles exceptions of type `E` and specifies catch variable `e`.

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. For example, the exception raised in line 25 of method `M2` is not handled in that method, and is therefore propagated up to method `M1`, the caller of `M2`.

Exceptions in Java can be classified according to several 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. For example, in a multithreaded program, one thread can cause an exception to occur in another thread. A synchronous exception can be classified 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. The source of an implicitly raised exception, therefore, lies outside the application being analyzed.

Our graph construction techniques [20] 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. The techniques also do not apply to implicitly raised exceptions. The testing of these types of exceptions is beyond the scope of this paper; our current work includes investigating ways to extend our work to include them.

2.2 Control-Flow Representations

Before describing the control-flow representations, we introduce terminology to facilitate the discussion.

- A method or a block *directly raises* an exception if it lexically encloses a `throw` statement. A method or a block *indirectly raises* an exception if it lexically encloses a call site such that the called method propagates an exception.
- A method *directly propagates* an exception if it directly raises, but does not handle, an exception; similarly, a method *indirectly propagates* an exception if it indirectly raises, but does not handle, an exception.
- A `catch` clause is a *local handler* if it handles only those exceptions that are directly raised in the `try` block associated with that `catch` clause. A `catch` clause is an *interprocedural handler* if it handles only those exceptions that are indirectly raised in the `try` block associated with that `catch` clause. A `catch` clause is a *global handler* if it is both a local handler and an interprocedural handler.
- A *catch handler encloses a statement s* if s appears in the lexical scope of the `try` block. A *finally block encloses a statement s* if s appears in the lexical scope of the corresponding `try` block or a corresponding `catch` handler.

Our graph construction techniques construct intraprocedural and interprocedural representations. Figure 2 shows the control-flow graphs (CFGs) for methods from the sample program (CFG edges are shown as solid lines). We first illustrate the CFG construction using the sample program, and then describe the CFG more formally.

The CFG-construction algorithm [20] creates nodes in the CFG to represent exception-handling constructs, and creates edges to represent the control flow caused by the exception-handling constructs. To identify potential targets of a `throw` statement, the algorithm performs local type inferencing in each method to determine exception types that can be raised at `throw` statements in that method. The algorithm then creates out edges from a throw node for those exception types, and labels each edge with the corresponding exception type. For example, the `throw` statement in line 25 of the sample program can raise one of two types of exceptions; therefore, node 25 in the CFG for `M2` has two out edges and each edge is labeled with the exception to which it corresponds.

If the exception raised at a `throw` statement is handled in the same method, the algorithm connects the throw node to the catch node for the appropriate `catch` handler. To model propagation of exceptions out of a method, the technique creates exceptional-exit nodes: an *exceptional-exit node* is an exit point in the CFG for a method M, and has a type T associated with it, and represents the propagation of an exception of type T by M. For example, method `M2` propagates exception types `E1` and `E21`; therefore, the CFG for `M2` contains two exceptional-exit nodes (labeled "ex-exit M2(E1)" and "ex-exit M2(E21)") corresponding to those types; the in edge of one exceptional-exit node is labeled "E1", whereas the in edge of the other is labeled "E21".

In Figure 2, nodes that correspond to `catch` handlers are labeled by the statement number and the handler type. For example, the catch node that corresponds to the `catch` clause in line 13 of method `M2` is labeled "13(E2)".

The presence of `finally` blocks creates complicated control-flow paths. In Java, a `finally` block can execute in one of two contexts: a normal context or an exceptional context. A `finally` block executes in a *normal context* when (1) control reaches the end of a `try` block or a `catch` block, or (2) control leaves a `try` statement because of an unconditional transfer-of-control statement, such as `break`, `continue`, or `return`. A `finally` block executes in an *exceptional context* when control leaves a `try` statement because of an unhandled exception; the unhandled exception may have been raised directly or indirectly within the `try` statement. The context of execution of a `finally` block determines where control flows from that `finally` block: in a normal context, control flows to the statement that follows the `try` statement, or control flows to the target of an unconditional transfer statement; in an exceptional context, control flows to an enclosing `finally` block, an enclosing `catch` handler, or control exits the method with an unhandled exception. The CFG-construction algorithm treats each `finally` block as a separate method. The algorithm creates a separate CFG for each `finally` block, and inserts call nodes to those `finally` blocks for both contexts of execution. Figure 2 illustrates the CFG created for the `finally` block that appears in line 5 of the sample program; the entry and exit nodes of the CFG are labeled "entry 5" and "exit 5." The figure also contains three call and return nodes for calls to the `finally` block. The call node labeled "5a(n)" corresponds to the execution of the `finally` block in a normal context. The call nodes labeled "5a(E1)" and "5a(E21)" correspond to executions of the `finally` block in exceptional contexts. We now define a CFG formally.

Definition 1. Let M be a method. Let $F = \{F_1, F_2, \ldots, F_{|f|}\}$, $|f| \geq 0$, be the `finally` blocks in M, and $TS = \{TS_1, TS_2, \ldots, TS_{|f|}\}$ be the `try` statements that contain the `finally` blocks. A *control-flow graph* (CFG) $G = (N, E, EN, EX)$ for M is a directed graph.
N is the set of nodes in G.

- N contains one node for each executable statement in M excluding those that appear in the F_i.
- For a `catch` handler in M, N contains a catch node h.
- N contains two sets of distinguished nodes, EN and EX, that represent the entry and exit points of G, respectively. Nodes in EN have no predecessors in G: EN contains the entry node n_e of G, and the catch node for each interprocedural `catch` handler in M. Nodes in EX have no successors in G: EX contains the exit node n_x of G, and one exceptional-exit node ex_t for each exception of type t that is directly propagated by M.
- For a call site in M, N contains a call node c and a return node r.

- For a `finally` block F_i in M, N contains a set of call nodes $\{nc_{[i]}, nc_{[i,ut_1]}, nc_{[i,ut_2]}, \ldots, nc_{[i,ut_m]}\}$ and a set of return nodes $\{nr_{[i]}, nr_{[i,ut_1]}, nr_{[i,ut_2]}, \ldots, nr_{[i,ut_m]}\}$ for calls to F_i in normal contexts; $nc_{[i]}$ represents the call that occurs at the end of the `try` block and the `catch` handlers in TS_i; each $nc_{[i,j]}$ represents the call that occurs after an unconditional transfer statement that appears within the lexical scopes of the `try` block and the `catch` handlers in TS_i, and that causes control to be transferred outside TS_i; $m \geq 0$ is the number of such transfer statements.
- For a `finally` block F_i in M, N contains a set of call nodes $\{ec_{[i,t_1]}, ec_{[i,t_2]}, \ldots, ec_{[i,t_m]}\}$ and a set of return nodes $\{er_{[i,t_1]}, er_{[i,t_2]}, \ldots, er_{[i,t_m]}\}$ for calls to F_i in exceptional contexts; each $ec_{[i,t_j]}$ represents the call that occurs because an exception of type t_j is directly raised but not handled within the lexical scopes of the `try` block and the `catch` handlers in TS_i; $m \geq 0$ is the number of such distinct types of directly raised exceptions.

E is the set of edges in G.

- For a pair of nodes m and n ($m, n \in N$) such that there is a potential transfer of control from the statement represented by m to the statement represented by n, E contains an edge (m, n)
- For a pair of call node c and corresponding return node r, E contains an edge (c, r).
- For a `finally` block F_i in M, E contains a set of edges $\{(n, nc_{[i]}), (n_1, nc_{[i]}), \ldots, (n_m, nc_{[i]})\}$, where n represents the last statement in the `try` block in TS_i, and the n_i represent the last statements in the `catch` handlers in TS_i.
- For a `finally` block F_i in M, E contains an edge $(nr_{[i]}, n)$, where n represents the first statement that follows TS_i.
- For an unconditional transfer statement ut_j that appears within the lexical scopes of the `try` block and the `catch` handlers in TS_i, and that transfers control outside TS_i, E contains edges $(ut_j, nc_{[i,ut_j]})$ and $(nr_{[i,ut_j]}, n)$, where n represents the target of the unconditional transfer statement.
- For a throw node m and an exception type t that can be raised at the corresponding `throw` statement, E contains an edge (m, n) that is labeled t, and n is one of three types of nodes:
  1. If the raised exception is caught within M by `catch` handler H_i such that there is no `finally` block in M that encloses m and is enclosed by H_i, n is the catch node for `catch` handler H_i.
  2. If the raised exception is not caught within M and there is no `finally` block in M that encloses m, n is the exceptional-exit node ex_t.
  3. If there is a `finally` block F_j in M that encloses m, and if the raised exception is caught within M, F_j is enclosed by the `catch` handler, n is the finally-call node $ec_{[j,t]}$ that calls F_j in an exceptional context, and for each `finally` block F_k that also encloses m, F_j is enclosed by F_k.
- For a call to `finally` block F_i in an exceptional context for a directly raised exception of type t, E contains an edge $(er_{[i,t]}, n)$, where n is one of three types of nodes:
  1. If the raised exception is caught within M by `catch` handler H_i such that there is no `finally` block in M that encloses F_i and is enclosed by H_i, n is the catch node for `catch` handler H_i.
  2. If the raised exception is not caught within M and there is no `finally` block in M that encloses F_i, n is the exceptional-exit node ex_t.
  3. If there is a `finally` block F_j in M that encloses F_i, and if the raised exception is caught within

M, F_j is enclosed by the `catch` handler, n is the finally-call node $ec_{[j,t]}$ that calls F_j in an exceptional context, and for each `finally` block F_k that also encloses F_i, F_j is enclosed by F_k.

Each node in N that represents a predicate statement has exactly two successors;[2] each node that represents a `throw` statement has $n \geq 1$ successors; all other nodes in N except those in EX have exactly one successor. Each node in N is reachable from at least one node in EN, and at least one node in EX is reachable from each node in N. □

To create an interprocedural representation, the techniques construct an interprocedural control-flow graph (ICFG) by connecting CFGs using interprocedural edges. Figure 2 shows the interprocedural edges in the ICFG for the sample program as dashed lines. The ICFG-construction algorithm connects the CFGs at call sites using call and return edges: a call edge connects a call node to the entry node of the called method, and a return edge connects the exit node of the called method to the corresponding return node. To represent the interprocedural control flow caused by propagation of exceptions on the call stack, the algorithm creates exceptional-return edges: an *exception-return edge* is an interprocedural edge that connects an exceptional-exit node of the called method to a catch node, a finally-call node, or an exceptional-exit node in the calling method. For example, in Figure 2, exceptional-exit node ex-exit M2(E1) in the CFG for method `M2` is connected to finally-call node 5a(E1) by an exceptional-return edge. The algorithm also creates unconditional-transfer edges: an *unconditional-transfer edge* is an interprocedural edge that connects a node representing an unconditional transfer statement, such as `break` or `continue`, within a `finally` block (such that the destination of the transfer lies outside the `finally` block) to the destination of that transfer. For example, in Figure 2, node 9 is connected to node 3 by an unconditional-transfer edge. We now define an ICFG formally.

Definition 2. Let $\mathcal{P}$ be a program with methods $M_1, M_2, \ldots, M_n$, let $G_i = (N_i, E_i, EN_i, EX_i)$ be the CFG for method M_i, and let $F_i = \{F_{(i,1)}, F_{(i,2)}, \ldots, F_{(i,|f_i|)}\}$ be the `finally` blocks in method M_i. An *interprocedural control-flow graph* (ICFG) for $\mathcal{P}$, $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{EN}, \mathcal{EX})$, is a directed graph in which

- Each N_i is augmented with an exceptional-exit node ex_t for an exception of type t that is indirectly propagated by M_i.[3]
- For a `finally` block $F_{(i,j)}$ in M_i, N_i is augmented with a set call nodes $\{ec_{[(i,j),t_1]}, ec_{[(i,j),t_2]}, \ldots, ec_{[(i,j),t_m]}\}$; and a set of return nodes $\{er_{[(i,j),t_1]}, er_{[(i,j),t_2]}, \ldots, er_{[(i,j),t_m]}\}$ for calls to $F_{(i,j)}$ in exceptional contexts for indirectly raised exceptions;[4] each $ec_{[(i,j),t_k]}$ represents a call that occurs because an exception of type t_k is indirectly raised but not handled within the lexical scopes of the `try` block and the `catch` blocks in $TS_{(i,j)}$; $m \geq 0$ is the number of distinct types of indirectly raised exceptions.
- For a pair of call node $ec_{[(i,j),t]}$ and corresponding return node $er_{[(i,j),t]}$, E_i is augmented with the edge $(ec_{[(i,j),t]}, er_{[(i,j),t]})$.
- For a call to `finally` block $F_{(i,j)}$ in an exceptional context for an indirectly raised exception of type t, E_i is augmented with an edge $(er_{[(i,j),t]}, n)$, where n is one of three types of nodes:

  1. If the raised exception is caught within M_i by `catch` handler $H_{(i,j)}$ such that there is no `finally`

[2] A `switch` statement can be represented by a sequence of predicates such that each predicate has only two outcomes.

[3] If an exception of type t is also directly propagated by M_i, ex_t is added to N_i during the construction of G_i, and is not added again during the construction of $\mathcal{G}$.

[4] Call and return nodes for a call to `finally` block $F_{(i,j)}$ in an exceptional context for exception type t are created during the construction of $\mathcal{G}$ only if they were not created during the construction of G_i.

block in M_i that encloses $F_{(i,j)}$ and is enclosed by $H_{(i,j)}$, n is the catch node for `catch` handler $H_{(i,j)}$.

2. If the raised exception is not caught within M_i and there is no `finally` block in M_i that encloses $F_{(i,j)}$, n is the exceptional-exit node ex_t.
3. If there is a `finally` block $F_{(i,k)}$ in M_i that encloses $F_{(i,j)}$, and if the raised exception is caught within M_i, $F_{(i,k)}$ is enclosed by the `catch` handler, n is the finally-call node $ec_{[(i,k),t]}$ that calls $F_{(i,k)}$ in an exceptional context, and for each `finally` block $F_{(i,l)}$ that also encloses $F_{(i,j)}$, $F_{(i,k)}$ is enclosed by $F_{(i,l)}$.

- $\mathcal{N} = \bigcup_{1 \le i \le n} N_i$.
- $\mathcal{EN}$ is the set of entry nodes of the methods in $\mathcal{P}$ such that for each $n_e \in \mathcal{EN}$, there is no call edge $(c, n_e) \in \mathcal{E}$; $\mathcal{EX}$ is the set of exit nodes of the methods in $\mathcal{P}$ such that for each $n_x \in \mathcal{EX}$, there is no return edge $(n_x, r) \in \mathcal{E}$.
- $\mathcal{E} = (\bigcup_{1 \le i \le n} E_i) - E_{(c,r)}$, where $E_{(c,r)}$ is the set of edges that connect call nodes to their corresponding return nodes.
- For a call site in a method M_i that calls a method M_j, $\mathcal{E}$ contains:
  - A call edge (c, n_e), where c is the call node in G_i and n_e is the entry node of G_j
  - A return edge (n_x, r), where n_x is the exit node of G_j and r is the return node in G_i
  - A set of exceptional-return edges $\{(ex_{t_1}, n_1), (ex_{t_2}, n_2), \ldots, (ex_{t_m}, n_m)\}$, $m \ge 0$, where each ex_{t_i} is the exceptional-exit node in G_j for exception type t_i that is propagated by M_j, and each n_i is one of three types of nodes in G_i:
    1. If the raised exception is caught within M_i by `catch` handler $H_{(i,j)}$ such that there is no `finally` block in M_i that encloses $F_{(i,j)}$ and is enclosed by $H_{(i,j)}$, n_i is the catch node for `catch` handler $H_{(i,j)}$.
    2. If the raised exception is not caught within M_i and there is no `finally` block in M_i that encloses $F_{(i,j)}$, n_i is the exceptional-exit node ex_{t_i}.
    3. If there is a `finally` block $F_{(i,k)}$ in M_i that encloses $F_{(i,j)}$, and if the raised exception is caught within M_i, $F_{(i,k)}$ is enclosed by the `catch` handler, n_i is the finally-call node $ec_{[(i,k),t_i]}$ that calls $F_{(i,k)}$ in an exceptional context, and for each `finally` block $F_{(i,l)}$ that also encloses $F_{(i,j)}$, $F_{(i,k)}$ is enclosed by $F_{(i,l)}$.
  - A set of unconditional-transfer edges $\{(ut_1, n_1), (ut_2, n_2), \ldots, (ut_m, n_m)\}$, $m \ge 0$, where each ut_i represents an unconditional transfer statement in M_j, and each n_i represents the target in M_i of that transfer.[5]

□

Let $\mathcal{P}$ be a program with methods $M_1, M_2, \ldots, M_n$, $n \ge 1$. A *flow graph* of $\mathcal{P}$ is a CFG for M_1 if $n = 1$, or an ICFG for $\mathcal{P}$ if $n > 1$.

A *path* in flow graph $G = (N, E, EN, EX)$ is a sequence of nodes $(n_1, n_2, \ldots, n_m)$, $m \ge 0$, such that for all $1 \le i < m$, $(n_i, n_{i+1}) \in E$. A *complete path* in flow graph $G = (N, E, EN, EX)$ is a path $(n_1, n_2, \ldots, n_m)$ in G, $m \ge 2$, such that $n_1 \in EN$ and $n_m \in EX$. A *realizable path* in an ICFG $\mathcal{G}$ is a path p in $\mathcal{G}$ such that each call edge in p is matched by its corresponding return, exceptional-return, or unconditional-transfer

[5]Unconditional-transfer edges are created only if G_j represents a `finally` block.

edge. A *simple path* in a flow graph G is a path in G in which all nodes, except possibly the first and the last, are distinct. A *loop-free path* in a flow graph G is a path in which all nodes are distinct.

3 Criteria for Testing Exception-Handling Constructs

In this section, we first present definitions for the exceptions testing criteria. We then introduce the class of criteria that test the behavior of exception-handling constructs.

3.1 Definitions for the Exception Testing Criteria

The *exception testing criteria* is a class of test selection criteria that, like the data-flow testing criteria [3, 18], require test cases to exercise certain paths based on data-flow relationships.

An *exception type* is a type whose instantiation can be raised at a `throw` statement and carries information from the point where the exception is raised to the point where that exception is handled. In Java, any class that is a subtype of `java.lang.Throwable` is an exception type. The sample program in Figure 1 uses five exception types: `E`, `E1`, `E2`, `E3`, and `E21`. An *exception object* is an instance of an exception type. For example, the sample program contains four exception objects — those that are instantiated in lines 2, 11, 19, and 20. In the discussion, we refer to these objects as $eobj_2$, $eobj_{11}$, $eobj_{19}$, and $eobj_{20}$. An *exception variable* is a program variable whose static (declared) type is an exception type. The sample program contains three exception variables — `e1` in method `M2`, catch variable `e2` in method `M1`, and catch variable `e` in method `M`. We associate a *temporary exception variable*, $evar_i$, with each `throw` statement i whose expression is a method call or a new-instance expression; $evar_i$ provides a handle on the exception object that is either created at statement i or returned by the method called at statement i. For example, the `throw` statement in line 2 has the temporary exception variable $evar_2$ associated with it. An exception object becomes an *active exception object* when it is raised at a `throw` statement. At any point in the execution of a program, there can be only one active exception object (that represents an unhandled exception at that point in the execution), although several exception objects may exist at that point. We define a unique program variable, $evar_{active}$, that keeps track of the active exception object: at any point in the execution of a program, the value of $evar_{active}$ is either undefined, if there is no unhandled exception at that point in the execution, or $eobj_k$, if $eobj_k$ is the active exception object at that point in the execution.

We associate definitions and uses of exception variables with nodes in a flow graph G (recall that G can be a CFG or an ICFG, depending on the level of testing being done). For each node i in G, an *exception definition set*, *e-def*(i), contains the set of exception variables that are defined at node i. A node i *defines* an exception variable if:

1. i assigns a value to an exception variable v — e-def(i) contains v
2. i is a catch node — e-def(i) contains the catch variable of the associated `catch` clause
3. i is a throw node such that the expression associated the corresponding `throw` statement is a method call or a new-instance expression — in this case, e-def(i) contains the temporary exception variable $evar_i$
4. i is a throw node — e-def(i) contains $evar_{active}$

For example, in the sample program, e-def(19) contains `e1` because statement 19 assigns a value to exception

Table 1: e-def and e-use sets for the sample program.

i	e-def(i)	e-use(i)
2	$evar_2$, $evar_{active}$	$evar_2$
11	$evar_{11}$, $evar_{active}$	$evar_{11}$
13	`e2`, $evar_{active}$	$evar_{active}$
15	$evar_{active}$	`e2`
16		`e2`
19	`e1`	
20	`e1`	
25	$evar_{active}$	`e1`
26		`e1`
28	`e`, $evar_{active}$	$evar_{active}$
29		`e`

variable `e1`. e-def(13) contains `e2` because node 13 is a catch node, and `e2` is the catch variable of the corresponding `catch` clause. e-def(2) contains a temporary exception variable $evar_2$ because node 2 represents a `throw` statement whose expression is a new-instance expression. e-def(25) contains $evar_{active}$ because node 25 is a throw node.

For each node i in G, an *exception use set*, *e-use*(i), contains the set of exception variables that are used at node i.[6] A node i uses an exception variable if:

1. i accesses the value of an exception variable v — e-use(i) contains v
2. i is a catch node — e-use(i) contains $evar_{active}$
3. i is a throw node such that the expression associated the corresponding `throw` statement is a method call or a new-instance expression — in this case, e-use(i) contains the temporary exception variable $evar_i$ that is defined at the same node.

For example, e-use(15) contains `e2` because node 15 uses the value of `e2`. e-use(13) contains $evar_{active}$ because 13 is a catch node. e-use(2) contains temporary exception variable $evar_2$ because node 2 represents a `throw` statement whose expression is a new-instance expression. Table 1 lists the e-def and e-use sets for the sample program.

For each node i in G, an *exception undefinition set*, *e-undef*(i), contains the set of exception variables that are undefined at node i. Each catch node undefines $evar_{active}$. An e-undef set is associated only with a catch node, and contains a single element, $evar_{active}$.

A *definition-clear path* (def-clear path) with respect to exception variable v is a path $(i, n_1, n_2, \ldots, n_m, j)$ in G, $m \geq 0$, such that there is no definition or undefinition of v in $n_1, n_2, \ldots, n_m$.[7]

To generate test requirements for exception-handling constructs, we identify, for a definition of exception variable v at node i, the set of nodes j that use the value assigned to v at i (a *definition-use set*). For example, node 20 defines `e1`, node 25 uses `e1`, and there exists a def-clear path with respect to `e1` from node 20 to node 25. Node 25, therefore, appears in the definition-use set of node 20 with respect to exception variable `e1`. The presence of exception-handling constructs induces a second type of definition-use set that crosses a `throw`–`catch` statement pair. For example, the definition of `e1` in node 20 is used in node 15, through the

[6] An *e-use(i)* can be classified as an *c-e-use(i)* or an *p-e-use(i)* based on whether node i uses an exception variable in a computation or in a predicate [3, 18]. For simplicity, we use *e-use(i)* to represent both of those types of uses.

[7] If G is an ICFG, we consider only realizable paths in G.

Table 2: e-du sets for the sample program.

$(v,\ i)$	e-du (v, i)	$(v \rightarrow w, i)$	e-du $(v \rightarrow w, i)$
$(evar_2,\ 2)$	2	$(evar_2 \rightarrow$ e, 2)	29
$(evar_{active},\ 2)$	28	$(evar_{11} \rightarrow$ e, 11)	29
$(evar_{11},\ 11)$	11	(e1→e2, 19)	15, 16
$(evar_{active},\ 11)$	28	(e1→e2, 20)	15, 16
(e2, 13)	15, 16	(e1→e, 19)	29
$(evar_{active},\ 15)$	28	(e1→e, 20)	29
(e1, 19)	25, 26	(e2→e, 13)	29
(e1, 20)	25, 26		
$(evar_{active},\ 25)$	13, 28		
(e, 28)	29		

mapping of `e1` to `e2` by the underlying exception-handling mechanism. Node 15, therefore, appears in the definition-use set of node 20 with respect to the mapping `e1`→`e2`. Identification of such definition-use sets enables a more thorough testing of exception-handling constructs.

To facilitate a formal definition of an exception definition-use set, we first define the following sets:

- For a throw node j and an exception variable v ($v \in$ e-use(j)), a *throw-variable definition set*, *tvar-def*(v, j), contains the set of nodes i such that $v \in$ e-def(i) and there exists a def-clear path with respect to v from i to j. tvar-def(v, j) stores nodes that contain definitions of v that reach the use of v at throw node j. For example, tvar-def(`e1`, 25) = {19, 20}.
- For a catch node i and the associated catch variable v, a *catch-variable use set*, *cvar-use*(v, i), contains the set of nodes j such that $v \in$ e-use(j) and there exists a def-clear path with respect to v from i to j. cvar-use(v, i) stores nodes that contain uses of v that are reachable from the definition of v at catch node i. For example, cvar-use(`e2`, 13) = {15, 16}.
- For an exception variable v and throw node i ($v \in$ e-use(i)), an *exception mapping set*, *e-map*(v, i), contains the set of pairs $<w, j>$ such that j is a catch node, w is the corresponding catch variable, and there exists a path $(i, n_1, n_2, \ldots, n_m, j)$, $m \geq 0$, in G. For example, e-map(`e1`, 25) = {<`e2`, 13>, <`e`, 28>}.

An *exception definition-use set*, *e-du*, contains, for each definition of an exception variable, all reachable uses of that definition, and is the union of the following two sets:

(1) *e-du*(v, i) (v is an exception variable) is the set of nodes j such that $v \in$ e-def(i), $v \in$ e-use(j), and there exists a def-clear path with respect to v from i to j.

(2) *e-du*($v \rightarrow w, i$) (v and w are exception variables) is the set of nodes j such that $i \in$ tvar-def(v, k), $j \in$ cvar-use(w, l), $<w, l> \in$ e-map(v, k), and there exists a def-clear path with respect to v from k to j.

Table 2 lists the e-du sets for the sample program.

An *exception definition-use association* (e-du association) is a triple (i, j, v) such that $j \in$ e-du(v, i), or a triple $(i, j, v \rightarrow w)$ such that $j \in$ e-du($v \rightarrow w, i$). For example, (19, 25, `e1`) and (20, 16, `e1`→`e2`) are two of the e-du associations in the sample program.

Apart from its normal state, an exception object has another state associated with it that indicates whether that exception object is the active exception object. A `throw` statement *activates* an exception object. A `catch` handler *deactivates* an exception object. An exception object can also be deactivated

Table 3: e-act and e-deact sets for the sample program.

i	e-act(i)	e-deact(i)
2	$eobj_2$	
9		$eobj_{19}$, $eobj_{20}$
11	$eobj_{11}$	$eobj_{19}$, $eobj_{20}$
13		$eobj_{20}$
15	$eobj_{20}$	
25	$eobj_{19}$, $eobj_{20}$	
28		$eobj_2$, $eobj_{11}$, $eobj_{19}$, $eobj_{20}$

Table 4: e-ad sets for the sample program.

($eobj_k$, i)	e-ad($eobj_k$, i)
($eobj_2$, 2)	28
($eobj_{11}$, 11)	28
($eobj_{19}$, 25)	9, 11, 28
($eobj_{20}$, 25)	9, 11, 13, 28

within a `finally` block when that block executes in the exceptional context, and (1) a `throw` statement is reached in the `finally` block (that statement deactivates the active exception object and activates a different exception object), (2) a `break` or `continue` statement is reached in the `finally` block that transfers control out of the `finally` block, or (3) a `return` statement is reached in the `finally` block.

To support activations and deactivations of exception objects, we associate these operations with nodes in G in a manner similar to definitions and uses: an activation is symmetric to a definition, whereas a deactivation is symmetric to a use. For each node i in G, an *exception activation set*, *e-act*(i), contains the set of exception objects that are activated at that node; similarly, an *exception deactivation set*, *e-deact*(i), contains the set of exception objects that are deactivated at node i. For example, $eobj_{20}$ appears in e-act(25) and e-deact(13) because node 25 activates $eobj_{20}$ and node 13 deactivates it. Table 3 lists the e-act and e-deact sets for the sample program.

The other definitions related to e-def and e-use sets extend to e-act and e-deact sets. Like the computation of uses that are rechable from a definition, we compute deactivations that are reachable from an activation. An activation of exception object $eobj_m$ deactivates any active exception object $eobj_n$. A deactivation of exception object $eobj_m$ deactivates an active exception object $eobj_m$. Therefore, for an activation of $eobj_m$ at node i to reach a deactivation of $eobj_m$ at node j, the path p from i to j must be (1) free of an activation of any $eobj_n$, and (2) free of a deactivation of $eobj_m$. However, because activation of any $eobj_n$ along p implies a deactivation of $eobj_m$, it is sufficient to impose only condition (2) on p to compute reachable deactivations.

A *deactivation-clear path* (deact-clear path) with respect to exception object $eobj_k$ is a sequence of nodes $(i, n_1, n_2, \ldots, n_m, j)$, $m \geq 0$, such that there is no deactivation of $eobj_k$ in $n_1, n_2, \ldots, n_m$. An *exception activation-deactivation set*, *e-ad*($eobj_k, i$), is the set of nodes j such that $eobj_k \in$ e-act(i), $eobj_k \in$ e-deact(j), and there exists a deact-clear path with respect to $eobj_k$ from i to j. Table 4 lists the e-ad sets for the sample program.

An *exception activation-deactivation association* (e-ad association) is a triple $(i, j, eobj_k)$ such that $j \in$ e-ad($eobj_k, i$). For example, (2, 28, $eobj_2$) is an e-ad association in the sample program.

A path $(i, n_1, n_2, \ldots, n_m, j)$, $m \geq 0$, is an *e-du-path with respect to exception variable* v (e-du-path(v)) if $v \in$ e-def(i), $v \in$ e-use(j), and (i, $n_1, n_2, \ldots, n_m, j$) is a def-clear simple path with respect to v. A path $(i, n_1, n_2, \ldots, n_m, j)$, $m \geq 2$, is an *e-du-path with respect to mapping* $v \rightarrow w$ (e-du-path($v \rightarrow w$)) if $i \in$ tvar-def(v, k), $j \in$ cvar-use(w, l), $<w, l> \in$ e-map(v, k), and there exists a def-clear simple path with respect to v from k to j. A path $(i, n_1, n_2, \ldots, n_m, j)$, $m \geq 0$, is an *e-ad-path* with respect to exception object $eobj_k$ if $eobj_k \in$ e-act(i), $eobj_k \in$ e-deact(j), and $(i, n_1, n_2, \ldots, n_m, j)$ is a deact-clear simple path with respect to $eobj_k$. An *association* is an e-du association, an e-du-path, an e-ad association, an e-ad-path, or a node.

A *path p covers an association* if p covers an e-du association, an e-du-path, an e-ad association, an e-ad-path, or a node.

- A *path p covers an e-du association* (i, j, v) if p contains a def-clear path with respect to v from i to j.
- A *path p covers an e-du-path q* if q is a subpath of p.
- A *path p covers an e-ad association* $(i, j, eobj_k)$ if p contains a deact-clear path with respect to $eobj_k$ from i to j.
- A *path p covers an e-ad-path q* if q is a subpath of p.
- A *path p covers a node n* if n appears in p.

A *set of paths covers an association* if some path from the set covers the association. A *test covers an association* if the path traversed by that test covers the association.

An *adequacy criterion C* is a function $C(P, T)$ that, given a program P and test suite T as inputs, evaluates to *true* if and only if the paths traversed by the tests in T cover the associations required by C. If $C(P, T)$ evaluates to *true*, the pair (P, T) *satisfies* the criterion C. Criterion C_1 *subsumes* criterion C_2 if and only if each program–test suite pair (P, T) that satisfies C_1 also satisfies C_2. Criterion C_1 *strictly subsumes* criterion C_2 if and only if C_1 subsumes C_2 and there exists a pair (P, T) that satisfies C_2 but does not satisfy C_1. Criterion C_1 is *equivalent* to criterion C_2 if and only if C_1 subsumes C_2 and C_2 subsumes C_1. Criteria C_1 and C_2 are *incomparable* if and only if neither criterion subsumes the other.

3.2 Exception Testing Criteria

We now present a class of exception testing criteria that can be used to guide the selection of test cases to test the behavior of exception-handling constructs. We describe the subsumption hierarchy of the exception testing criteria for a complete program P. We assume the following about P:

(1) All exceptions in P are raised explicitly;

(2) No exception is propagated out of P — any exception raised in P is handled within P;

(3) P contains no unreachable `catch` handler — each `catch` handler in P is reachable through an exception raised in P;

(4) Each exception variable in P is defined before it is used.

In the next section, we illustrate how the criteria can be used to test an incomplete system during unit and integration testing.

The *exception testing criteria*, like the data-flow testing criteria, require that operations on data elements be exercised along various paths in the programs. Because the goal of the exception testing criteria is to test the behavior of exception-handling constructs, the relevant data elements that are candidates for coverage are restricted to exception variables and exception objects only.

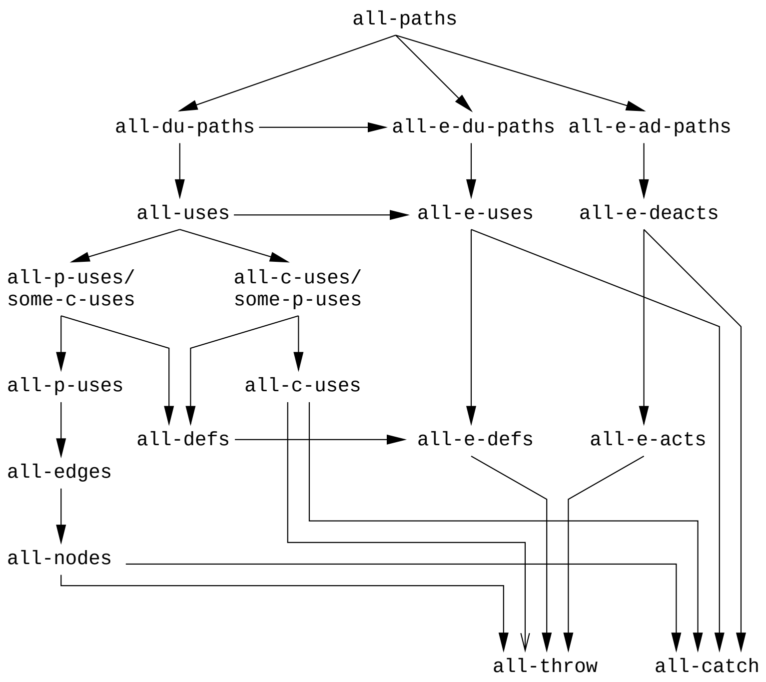

Figure 3: The subsumption hierarchies of the data-flow testing criteria (left), and the exception testing criteria (right).

Table 5: The exception testing criteria.

Criterion	Associations required
all-e-defs	$\forall i (\forall v \in$ e-def(i) (some $j \in$ (e-du$(v,i) \cup$ e-du$(v{\rightarrow}w,i))))$
all-e-uses	$\forall i (\forall v \in$ e-def(i) (all $j \in$ (e-du$(v,i) \cup$ e-du$(v{\rightarrow}w,i)))$
all-e-du-paths	$\forall i (\forall j \in$ e-du(v,i) (all e-du-paths wrt v from i to j)) $\forall i (\forall j \in$ e-du$(v{\rightarrow}w,i)$ (all e-du-paths wrt $v{\rightarrow}w$ from i to j))
all-e-acts	$\forall i (\forall\ eobj_k \in$ e-act(i) (some $j \in$ e-ad$(eobj_k, i)))$
all-e-deacts	$\forall i (\forall\ eobj_k \in$ e-act(i) (all $j \in$ e-ad$(eobj_k, i)))$
all-e-ad-paths	$\forall i (\forall j \in$ e-ad$(eobj_k, i)$ (all e-ad-paths wrt $eobj_k$ from i to j))

Like the data-flow testing criteria, which require the coverage of definitions and subsequent uses of variables, the exception testing criteria require the coverage of definitions and subsequent uses of exception variables. Unlike the data-flow testing criteria, however, the exception testing criteria provide coverage of an alternative, symmetric set of operations — activations and deactivations of exception objects. Exploiting the symmetry between the two sets of operations yields two parallel subclasses of criteria, one based on definitions and uses of exception variables, and the other based on activations and deactivations of exception objects.

Figure 3 presents the subsumption hierarchy of the exception testing criteria. Table 5 describes the

Table 6: Test requirements for the sample program generated by the application each exception testing criterion. Each e-du- and e-ad-path in the table is subscripted by the variable or the mapping for which the path is defined.

Criterion	Test requirement (Associations)
all-throw	2, 11, 15, 25
all-catch	13, 28
all-e-defs	(2, 2, $evar_2$), (2, 28, $evar_{active}$), (11, 11, $evar_{11}$), (11, 28, $evar_{active}$), (13, 15, e2), (15, 28, $evar_{active}$), (19, 26, e1), (20, 25, e1), (25, 13, $evar_{active}$), (28, 29, e)
all-e-uses	(2, 2, $evar_2$), (2, 29, $evar_2$→e), (2, 28, $evar_{active}$), (11, 11, $evar_{11}$), (11, 29, $evar_{11}$→e), (11, 28, $evar_{active}$), (13, 15, e2), (13, 16, e2), (13, 29, e2→e), (15, 28, $evar_{active}$), (19, 26, e1), (19, 25, e1), (19, 29, e1→e), (20, 25, e1), (20, 26, e1), (20, 15, e1→e2), (20, 16, e1→e2), (25, 13, $evar_{active}$), (25, 28, $evar_{active}$), (28, 29, e)
all-e-du-paths	$(2, 2)_{evar_2}$, $(2, \text{ex exit E3}, 28, 29)_{evar_2 \to e}$, $(2, \text{ex exit E3}, 28)_{evar_{active}}$, $(11, 11)_{evar_{11}}$, $(11, \text{ex exit E3}, \text{ex exit E3}, 28, 29)_{evar_{11} \to e}$, $(11, \text{ex exit E3}, \text{ex exit E3}, 28)_{evar_{active}}$, $(13, 14, 15)_{e2}$, $(13, 14, 16)_{e2}$, $(13, 14, 15, \text{ex exit E21}, 28, 29)_{e2 \to e}$, $(15, \text{ex exit E21}, 28)_{evar_{active}}$, $(19, 21, 22, 24, 26)_{e1}$, $(19, 21, 23, 24, 26)_{e1}$, $(19, 21, 22, 24, 25)_{e1}$, $(19, 21, 23, 24, 25)_{e1}$, $(19, 21, 22, 24, 25, \text{ex exit E1}, 5a, 5, 6, 8, 10, \text{exit}, 5b, \text{ex exit E1}, 28, 29)_{e1 \to e}$, $(19, 21, 22, 24, 25, \text{ex exit E1}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, \text{ex exit E1}, 28, 29)_{e1 \to e}$, $(19, 21, 23, 24, 25, \text{ex exit E1}, 5a, 5, 6, 8, 10, \text{exit}, 5b, \text{ex exit E1}, 28, 29)_{e1 \to e}$, $(19, 21, 23, 24, 25, \text{ex exit E1}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, \text{ex exit E1}, 28, 29)_{e1 \to e}$, $(20, 21, 22, 24, 25)_{e1}$, $(20, 21, 23, 24, 25)_{e1}$, $(20, 21, 22, 24, 26)_{e1}$, $(20, 21, 23, 24, 26)_{e1}$, $(20, 21, 22, 24, 25, \text{ex exit E21}, 5a, 5, 6, 8, 10, \text{exit}, 5b, 13, 14, 15)_{e1 \to e2}$, $(20, 21, 22, 24, 25, \text{ex exit E21}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, 13, 14, 15)_{e1 \to e2}$, $(20, 21, 23, 24, 25, \text{ex exit E21}, 5a, 5, 6, 8, 10, \text{exit}, 5b, 13, 14, 15)_{e1 \to e2}$, $(20, 21, 23, 24, 25, \text{ex exit E21}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, 13, 14, 15)_{e1 \to e2}$, $(20, 21, 22, 24, 25, \text{ex exit E21}, 5a, 5, 6, 8, 10, \text{exit}, 5b, 13, 14, 16)_{e1 \to e2}$, $(20, 21, 22, 24, 25, \text{ex exit E21}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, 13, 14, 16)_{e1 \to e2}$, $(20, 21, 23, 24, 25, \text{ex exit E21}, 5a, 5, 6, 8, 10, \text{exit}, 5b, 13, 14, 16)_{e1 \to e2}$, $(20, 21, 23, 24, 25, \text{ex exit E21}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, 13, 14, 16)_{e1 \to e2}$, $(25, \text{ex exit E21}, 5a, 5, 6, 8, 10, \text{exit}, 5b, 13)_{evar_{active}}$, $(25, \text{ex exit E21}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, 13)_{evar_{active}}$, $(25, \text{ex exit E1}, 5a, 5, 6, 8, 10, \text{exit}, 5b, \text{ex exit E1}, 28)_{evar_{active}}$, $(25, \text{ex exit E1}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, \text{ex exit E1}, 28)_{evar_{active}}$, $(28, 29)_e$
all-e-acts	(2, 28, $eobj_2$), (11, 28, $eobj_{11}$), (15, 28, $eobj_{20}$), (25, 11, $eobj_{20}$), (25, 28, $eobj_{19}$)
all-e-deacts	(2, 28, $eobj_2$), (11, 28, $eobj_{11}$), (15, 28, $eobj_{20}$), (25, 13, $eobj_{20}$), (25, 9, $eobj_{20}$), (25, 11, $eobj_{20}$), (25, 28, $eobj_{19}$), (25, 9, $eobj_{19}$), (25, 11, $eobj_{19}$)
all-e-ad-paths	$(2, \text{ex exit E3}, 28)_{eobj_2}$, $(11, \text{ex exit E3}, \text{ex exit E3}, 28)_{eobj_{11}}$, $(15, \text{ex exit E21}, 28)_{eobj_{20}}$, $(25, \text{ex exit E21}, 5a, 5, 6, 8, 10, \text{exit}, 5b, 13)_{eobj_{20}}$, $(25, \text{ex exit E21}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, 13)_{eobj_{20}}$, $(25, \text{ex exit E21}, 5a, 5, 6, 8, 9)_{eobj_{20}}$, $(25, \text{ex exit E21}, 5a, 5, 6, 7, 8, 9)_{eobj_{20}}$, $(25, \text{ex exit E21}, 5a, 5, 6, 8, 10, 11)_{eobj_{20}}$, $(25, \text{ex exit E21}, 5a, 5, 6, 7, 8, 10, 11)_{eobj_{20}}$, $(25, \text{ex exit E1}, 5a, 5, 6, 8, 10, \text{exit}, 5b, \text{ex exit E1}, 28)_{eobj_{19}}$, $(25, \text{ex exit E1}, 5a, 5, 6, 7, 8, 10, \text{exit}, 5b, \text{ex exit E1}, 28)_{eobj_{19}}$, $(25, \text{ex exit E1}, 5a, 5, 6, 8, 9)_{eobj_{19}}$, $(25, \text{ex exit E1}, 5a, 5, 6, 7, 8, 9)_{eobj_{19}}$, $(25, \text{ex exit E1}, 5a, 5, 6, 8, 10, 11)_{eobj_{19}}$, $(25, \text{ex exit E1}, 5a, 5, 6, 7, 8, 10, 11)_{eobj_{19}}$

associations required by each criterion in the hierarchy.

The first three criteria in Table 5 are based on definitions and uses of exception variables and are similar to the corresponding data-flow testing criteria [3, 18]. The *all-e-defs* criterion requires the coverage of each definition of each exception variable to some reachable use. The *all-e-uses* criterion requires the coverage of each definition of each exception variable to all reachable uses. The stronger *all-e-du-paths* criterion imposes a stricter requirement of covering all paths from each definition of an exception variable to all reachable uses.

The next three criteria are stated in terms of activations and deactivations of exception objects. The *all-e-acts* criterion requires the coverage of each exception activation to some reachable deactivation. The *all-e-deacts* criterion requires the coverage of each exception activation to all reachable deactivations. The *all-e-ad-paths* criterion requires the coverage of all paths from each activation of an exception object to all reachable deactivations.

Table 6 lists the test requirements for the sample program that are generated by the application of each exception testing criterion to the sample program. A *test requirement* for a program P with respect to criterion C is a set of associations that must be covered by tests in test suite T in order for (P, T) to satisfy C. Some criteria, such as all-e-defs and all-e-acts, have several alternative test requirements; Table 6 lists only one of those alternatives. For example, replacing the association (13, 15, `e2`) with the association (13, 16, `e2`) for all-e-defs yields a different test requirement that can also be used to satisfy all-e-defs.

Figure 3 shows the relationship of the existing criteria for testing exception-handling constructs — all-throw and all-catch — to the exception testing criteria. All-e-uses and all-e-deacts subsume all-catch, and all-e-defs and all-e-acts subsume all-throw.

Figure 3 also illustrates the relationship of the data-flow testing criteria to the exception testing criteria. For example, the figure shows that the all-uses data-flow criteria subsumes the all-e-uses exception testing criteria.

3.3 Proof of Correctness of the Subsumption Hierarchy

We now demonstrate the correctness the hierarchy shown in Figure 3: we prove each subsumption relationship in the hierarchy, and we illustrate that each pair of criteria that is not shown to be related in the hierarchy is incomparable. The subsumption proofs for the data-flow testing criteria are presented elsewhere [2, 18], and we do not restate them here. Likewise, in illustrating the incomparability of pairs of criteria, we exclude those pairs that comprise only data-flow testing criteria — for example, the incomparability of all-p-uses and all-defs; these are substantiated in Reference [18].

Table 7 lists the strict subsumption relations that must be proved to establish the correctness of the hierarchy shown in Figure 3. Each proof that criterion C_1 strictly subsumes criterion C_2 has the following necessary and sufficient conditions: (a) C_1 subsumes C_2 (denoted $C_1 \longrightarrow C_2$ in Table 7), and (b) C_2 does not subsume C_1 (denoted $C_1 \not\longleftarrow C_2$ in Table 7). Table 7 lists the necessary and sufficient conditions for each proof of strict subsumption, and enumerates the subparts of Theorem 1 (stated later in this section) that outline the proofs. The table also reduces the number of non-subsumption relations that must be demonstrated by identifying implications among the relations. For example, the relation "all-e-du-paths does not subsume all-paths" that is used to prove that all-paths strictly subsumes all-e-du-paths, can be inferred from another non-subsumption relation — "all-e-du-paths does not subsume all-nodes," listed as I7(b) in Table 9 — and therefore, need not be demonstrated. Lemma 1 proves the inference rule that is used to derive implications among non-subsumption relations.

Lemma 1: Let `predecessor`(C) be a function that, given a criterion C, returns a set of criteria that contains C and all predecessors of C in the subsumption hierarchy. Let `successor`(C) be a function that, given a criterion C, returns a set of criteria that contains C and all successors of C in the subsumption hierarchy. If criterion C_1 does not subsume criterion C_2, then for all $C_{1_{succ}} \in$ `successor`(C_1) and all $C_{2_{pred}} \in$ `predecessor`(C_2), $C_{1_{succ}}$ does not subsume $C_{2_{pred}}$.

Proof: Let C_1 not subsume C_2. Let $C_{1_{succ}}$ be any element in `successor`(C_1), and $C_{2_{pred}}$ be any element in `predecessor`(C_2). Suppose that $C_{1_{succ}}$ subsumes $C_{2_{pred}}$. By the definition of `successor`, C_1 subsumes $C_{1_{succ}}$, and by the definition of function `predecessor`, $C_{2_{pred}}$ subsumes C_2. Then by the transitivity of subsumption, C_1 subsumes C_2, which leads to a contradiction. Therefore, $C_{1_{succ}}$ does not subsume $C_{2_{pred}}$.

Table 7: Subsumption relations that must be proved to establish the correctness of the hierarchy shown in Figure 3, the subparts of Theorem 1 that prove each subsumption, and implications among the proofs identified using Lemma 1 ($\longrightarrow$ and $\not\longrightarrow$ denote the relations "subsumes" and "does not subsume," respectively).

Theorem 1 subpart		Criterion	Strictly subsumes	Necessary and sufficient conditions	Implied by
S1	(a)	all-paths	all-e-du-paths	all-paths $\longrightarrow$ all-e-du-paths	
	(b)			all-e-du-paths $\not\longrightarrow$ all-paths	I7(b)
S2	(a)	all-e-du-paths	all-e-uses	all-e-du-paths $\longrightarrow$ all-e-uses	
	(b)			all-e-uses $\not\longrightarrow$ all-e-du-paths	I5(a)
S3	(a)	all-e-uses	all-e-defs	all-e-uses $\longrightarrow$ all-e-defs	
	(b)			all-e-defs $\not\longrightarrow$ all-e-uses	I2(a)
S4	(a)	all-e-uses	all-catch	all-e-uses $\longrightarrow$ all-catch	
	(b)			all-catch $\not\longrightarrow$ all-e-uses	I2(b)
S5	(a)	all-e-defs	all-throw	all-e-defs $\longrightarrow$ all-throw	
	(b)			all-throw $\not\longrightarrow$ all-e-defs	I1(b)
S6	(a)	all-paths	all-e-ad-paths	all-paths $\longrightarrow$ all-e-ad-paths	
	(b)			all-e-ad-paths $\not\longrightarrow$ all-paths	I4(b)
S7	(a)	all-e-ad-paths	all-e-deacts	all-e-ad-paths $\longrightarrow$ all-e-deacts	
	(b)			all-e-deacts $\not\longrightarrow$ all-e-ad-paths	
S8	(a)	all-e-deacts	all-e-acts	all-e-deacts $\longrightarrow$ all-e-acts	
	(b)			all-e-acts $\not\longrightarrow$ all-e-deacts	I3(a)
S9	(a)	all-e-deacts	all-catch	all-e-deacts $\longrightarrow$ all-catch	
	(b)			all-catch $\not\longrightarrow$ all-e-deacts	I2(b)
S10	(a)	all-e-acts	all-throw	all-e-acts $\longrightarrow$ all-throw	
	(b)			all-throw $\not\longrightarrow$ all-e-acts	
S11	(a)	all-c-uses	all-catch	all-c-uses $\longrightarrow$ all-catch	
	(b)			all-catch $\not\longrightarrow$ all-c-uses	I8(b)
S12	(a)	all-c-uses	all-throw	all-c-uses $\longrightarrow$ all-throw	
	(b)			all-throw $\not\longrightarrow$ all-c-uses	I8(b)
S13	(a)	all-du-paths	all-e-du-paths	all-du-paths $\longrightarrow$ all-e-du-paths	
	(b)			all-e-du-paths $\not\longrightarrow$ all-du-paths	I7(b)
S14	(a)	all-uses	all-e-uses	all-uses $\longrightarrow$ all-e-uses	
	(b)			all-e-uses $\not\longrightarrow$ all-uses	I7(b)
S15	(a)	all-defs	all-e-defs	all-defs $\longrightarrow$ all-e-defs	
	(b)			all-e-defs $\not\longrightarrow$ all-defs	I10(b)
S16	(a)	all-nodes	all-catch	all-nodes $\longrightarrow$ all-catch	
	(b)			all-catch $\not\longrightarrow$ all-nodes	I7(b)
S16	(a)	all-nodes	all-throw	all-nodes $\longrightarrow$ all-throw	
	(c)			all-throw $\not\longrightarrow$ all-nodes	I7(b)

□

Table 8 lists each pair of criteria that must be shown to be incomparable to establish the correctness of the hierarchy shown in Figure 3. Table 8 is an exhaustive enumeration of the pairs of incomparable criteria, and illustrates the completeness of Theorem 1 by listing the subparts of Theorem 1 that prove each incomparability relation. Each proof that criterion C_1 is incomparable to criterion C_2 has two necessary and sufficient conditions: (a) C_1 does not subsume C_2, and (b) C_2 does not subsume C_1. Because there are 59 incomparability relations listed in Table 8, 118 proofs of non-subsumption are required to prove the relations. However, each incomparability relation need not be proved explicitly. Table 9, like Table 7, reduces the number of non-subsumption relations that must be proved by identifying implications among the relations. The table further reduces the number of proofs by grouping together pairs of incomparable criteria such that incomparability of each pair in the group need not be proved. For example, the entry

Table 8: Incomparability relations that must be proved to establish the correctness of the hierarchy shown in Figure 3, and the subparts of Theorem 1 that prove each incomparability.

Criterion	Incomparable with	Theorem subpart	Criterion	Incomparable with	Theorem subpart
all-e-du-paths	all-e-ad-paths	I1	all-edges	all-e-du-paths	I7
	all-e-deacts	I1		all-e-uses	I7
	all-e-acts	I1		all-e-defs	I7
all-e-uses	all-e-ad-paths	I1		all-e-ad-paths	I4
	all-e-deacts	I1		all-e-deacts	I4
	all-e-acts	I1		all-e-acts	I4
all-e-defs	all-e-ad-paths	I1	all-nodes	all-e-du-paths	I7
	all-e-deacts	I1		all-e-uses	I7
	all-e-acts	I1		all-e-defs	I7
	all-catch	I2		all-e-ad-paths	I4
all-catch	all-e-acts	I3		all-e-deacts	I4
	all-throw	I2		all-e-acts	I4
all-du-paths	all-e-ad-paths	I4	all-c-uses/ some-p-uses	all-e-du-paths	I8
	all-e-deacts	I4		all-e-uses	I8
	all-e-acts	I4		all-e-ad-paths	I9
all-uses	all-e-du-paths	I5		all-e-deacts	I9
	all-e-ad-paths	I4		all-e-acts	I9
	all-e-deacts	I4	all-defs	all-e-du-paths	I10
	all-e-acts	I4		all-e-uses	I10
all-p-uses/ some-c-uses	all-e-du-paths	I6		all-e-ad-paths	I11
	all-e-uses	I6		all-e-deacts	I11
	all-e-ad-paths	I4		all-e-acts	I11
	all-e-deacts	I4		all-catch	I10
	all-e-acts	I4	all-c-uses	all-e-du-paths	I8
all-p-uses	all-e-du-paths	I7		all-e-uses	I8
	all-e-uses	I7		all-e-defs	I12
	all-e-defs	I7		all-e-ad-paths	I9
	all-e-ad-paths	I4		all-e-deacts	I9
	all-e-deacts	I4		all-e-acts	I9
	all-e-acts	I4			

for I1 in the table groups together nine pairs of incomparable criteria, which require proofs for eighteen non-subsumption relations. However, the proofs of two non-subsumption relations imply each of the other sixteen proofs. Lemma 2 proves the inference rule that is used to group together criteria in Table 9.

Lemma 2: Let $C^1 = \{C^1_1, C^1_2, \ldots, C^1_m\}$ and $C^2 = \{C^2_1, C^2_2, \ldots, C^2_n\}$ be two sets of criterion such that for all $1 \leq i < m$, C^1_i strictly subsumes C^1_{i+1}, and for all $1 \leq j < n$, C^2_j strictly subsumes C^2_{j+1}. If C^1_1 does not subsume C^2_n, and C^2_1 does not subsume C^1_m, then for all $1 \leq i \leq m$ and all $1 \leq j \leq n$, C^1_i is incomparable to C^2_j.

Proof: Let C^1_1 not subsume C^2_n, and let C^2_1 not subsume C^1_m. By the definitions of functions `predecessor` and `successor`, and by the definitions of sets C^1 and C^2, for all $1 \leq i \leq m$, $C^1_i \in \texttt{successor}(C^1_1)$ and $C^1_i \in \texttt{predecessor}(C^1_m)$, and for all $1 \leq j \leq n$, $C^2_j \in \texttt{successor}(C^2_1)$ and $C^2_j \in \texttt{predecessor}(C^2_n)$. Then by Lemma 1, for all $1 \leq i \leq m$ and all $1 \leq j \leq n$, C^1_i does not subsume C^2_j, and C^2_j does not subsume C^1_i. Therefore, each C^1_i is incomparable to each C^2_j. □

Lemma 3: Let $p = (i, m_1, m_2, \ldots, m_k, j)$ be a path in a flow graph G such that i defines a variable v, and p is def-clear with respect to v. Then there exists a path $q = (i, n_1, n_2, \ldots, n_l, j)$ in G such that q is a loop-free path or a simple path, and q is def-clear with respect to v.

Table 9: A reduction of the incomparability relations listed in Table 8 using Lemmas 1 and 2, and the subparts of Theorem 1 that prove each incomparability ($\longrightarrow$ and $\not\longrightarrow$ denote the relations "subsumes" and "does not subsume," respectively).

Theorem 1 subpart		Criteria	Incomparable to	Minimal necessary and sufficient conditions	Implied by
I1	(a)	all-e-du-paths, all-e-uses, all-e-defs	all-e-ad-paths, all-e-deacts, all-e-acts	all-e-du-paths $\not\longrightarrow$ all-e-acts	I4(a)
	(b)			all-e-ad-paths $\not\longrightarrow$ all-e-defs	
I2	(a)	all-e-defs, all-throw	all-catch	all-e-defs $\not\longrightarrow$ all-catch	I10(a)
	(b)			all-catch $\not\longrightarrow$ all-throw	
I3	(a)	all-e-acts	all-catch	all-e-acts $\not\longrightarrow$ all-catch	
	(b)			all-catch $\not\longrightarrow$ all-e-acts	
I4	(a)	all-du-paths, all-uses, all-p-uses/some-c-uses, all-p-uses, all-edges, all-nodes	all-e-ad-paths, all-e-deacts, all-e-acts	all-du-paths $\not\longrightarrow$ all-e-acts	
	(b)			all-e-ad-paths $\not\longrightarrow$ all-nodes	
I5	(a)	all-uses	all-e-du-paths	all-uses $\not\longrightarrow$ all-e-du-paths	
	(b)			all-e-du-paths $\not\longrightarrow$ all-uses	I7(b)
I6	(a)	all-p-uses/some-c-uses	all-e-du-paths, all-e-uses	all-p-uses/some-c-uses $\not\longrightarrow$ all-e-uses	
	(b)			all-e-du-paths $\not\longrightarrow$ all-p-uses/some-c-uses	I7(b)
I7	(a)	all-p-uses, all-edges, all-nodes	all-e-du-paths, all-e-uses, all-e-defs	all-p-uses $\not\longrightarrow$ all-e-defs	
	(b)			all-e-du-paths $\not\longrightarrow$ all-nodes	
I8	(a)	all-c-uses/some-p-uses, all-c-uses	all-e-du-paths, all-e-uses	all-c-uses/some-p-uses $\not\longrightarrow$ all-e-uses	
	(b)			all-e-du-paths $\not\longrightarrow$ all-c-uses	
I9	(a)	all-c-uses/some-p-uses, all-c-uses	all-e-ad-paths, all-e-deacts all-e-acts	all-c-uses/some-p-uses $\not\longrightarrow$ all-e-acts	I4(a)
	(b)			all-e-ad-paths $\not\longrightarrow$ all-c-uses	
I10	(a)	all-defs	all-e-du-paths, all-e-uses, all-catch	all-defs $\not\longrightarrow$ all-catch	
	(b)			all-e-du-paths $\not\longrightarrow$ all-defs	
I11	(a)	all-defs	all-e-ad-paths, all-e-deacts, all-e-acts	all-defs $\not\longrightarrow$ all-e-acts	I4(a)
	(b)			all-e-ad-paths $\not\longrightarrow$ all-defs	I1(b)
I12	(a)	all-c-uses	all-e-defs	all-c-uses $\not\longrightarrow$ all-e-defs	
	(b)			all-e-defs $\not\longrightarrow$ all-c-uses	I8(b)

Proof: Outlined in Reference [2], page 1323. □

We now state and prove the theorem that establishes the correctness of the hierarchy shown in Figure 3.

Theorem 1: The family of data-flow testing and exception testing criteria is partially ordered by strict subsumption as shown in Figure 3. Furthermore, criterion C_1 strictly subsumes criterion C_2 if and only if it is explicitly shown to be so in Figure 3, or follows from the transitivity of the relationship.

Proof: We first prove the subsumption relations among the exception testing criteria. We then prove the subsumption relations between the data-flow testing criteria and the exception testing criteria. Finally, we demonstrate that pairs of criteria that are not shown to be related in Figure 3 are incomparable.

(S1) *all-paths strictly subsumes all-e-du-paths.*

(a) all-paths subsumes all-e-du-paths. Suppose that all-paths does not subsume all-e-du-paths. Then there exist a program P and a test suite T such that (P, T) satisfies all-paths but does not satisfy all-e-du-paths. Because (P, T) does not satisfy all-e-du-paths, there exists an e-du-path in P that is covered by no test in T. This contradicts the assumption that tests in T cover all paths in P.

(b) all-e-du-paths does not subsume all-paths. Implied by I7(b). □

(S2) *all-e-du-paths strictly subsumes all-e-uses.*

(a) all-e-du-paths subsumes all-e-uses. Suppose that all-e-du-paths does not subsume all-e-uses. Then there exist a program P and a test suite T such that (P, T) satisfies all-e-du-paths but does not satisfy all-e-uses. Let a be an e-du association that is not covered by tests in T. Let a be of the form (i, j, v). By definition of a, $v \in$ e-def(i), $v \in$ e-use(j), and there exists a def-clear path $p = (i, m_1, m_2, \ldots, m_k, j)$ with respect to v from i to j. Then by Lemma 3 and the definition of e-du-path, there exists a path $q = (i, n_1, n_2, \ldots, n_l, j)$ in the flow graph of P such that q is an e-du-path with respect to v. Because no test in T covers a, no test in T traverses the path q, which contradicts the assumption that (P, T) satisfies all-e-du-paths. Similarly, we can arrive at a contradiction for the case when the uncovered association a is of the form (i, j, $v \rightarrow w$).

(b) all-e-uses does not subsume all-e-du-paths. Implied by I5(a). □

(S3) *all-e-uses strictly subsumes all-e-defs.*

(a) all-e-uses subsumes all-e-defs. Suppose that all-e-uses does not subsume all-e-defs. Then there exist a program P and a test suite T such that (P, T) satisfies all-e-uses but does not satisfy all-e-defs. Let $v \in$ e-def(i) such that the definition is covered by no test in T. Then for all $j \in$ e-du(v, i), no test in T covers e-du association (i, j, v), which contradicts the assumption that (P, T) satisfies all-e-uses.

(b) all-e-defs does not subsume all-e-uses. Implied by I2(a). □

(S4) *all-e-uses strictly subsumes all-catch.*

(a) all-e-uses subsumes all-catch. Suppose that all-e-uses does not subsume all-catch. Then there exist a program P and a test suite T such that (P, T) satisfies all-e-uses but does not satisfy all-catch. Let j be a catch node that is covered by no test in T. Because we assume that each `catch` handler in P is reachable, there is a `throw` statement in P such that the exception raised at that statement is handled by the `catch` handler corresponding to j; let i be the throw node for that `throw` statement. Then $evar_{active} \in$ e-def(i), $evar_{active} \in$ e-use(j), and there exists a def-clear path with respect to $evar_{active}$ from i to j. Therefore, (i, j, $evar_{active}$) is an e-du association. Because no test in T covers catch node j, no test in T covers the e-du association (i, j, $evar_{active}$). This contradicts the assumption that (P, T) satisfies all-e-uses.

(b) all-catch does not subsume all-e-uses. Implied by I2(b). □

(S5) *all-e-defs strictly subsumes all-throw.*

(a) all-e-defs subsumes all-throw. Suppose that all-e-defs does not subsume all-throw. Then there exist a program P and a test suite T such that (P, T) satisfies all-e-defs but does not satisfy all-throw. Let i be a throw node that is covered by no test case in T. Because we assume that no exception is propagated out of P, there is a `catch` handler in P such that the exception raised at the `throw` statement corresponding to i is caught by that handler; let j be the catch node for that `catch` handler. Then $evar_{active} \in$ e-def(i), $evar_{active} \in$ e-use(j), and there exists a def-clear path with respect to $evar_{active}$ from i to j. Therefore, (i, j, $evar_{active}$) is an e-du association. Because no test in T covers throw node i, for all j, no test in T covers e-du association (i, j, $evar_{active}$). This contradicts the assumption that (P, T) satisfies all-e-defs.

(b) all-throw does not subsume all-e-defs. Implied by I1(b). □

(S6) *all-paths strictly subsumes all-e-ad-paths.*

(a) all-paths subsumes all-e-ad-paths. Suppose that all-paths does not subsume all-e-ad-paths. Then there exist a program P and a test suite T such that (P, T) satisfies all-paths but does not satisfy all-e-ad-paths. Because (P, T) does not satisfy all-e-ad-paths, there exists an e-ad-path in P that is covered by no test in T. This contradicts the assumption that tests in T cover all paths in P.

```
  enter M
1   read i
2   sum = 0
3   while ( i<10 )
      try
        try
4a        call B
4b        return B
5       finally
6         if ( i==9 )
7           print "i = 9"
8     catch E e1
9       return
10    i = i + 1
11  print sum
  exit M
```

```
  enter B
12  read j
13  if ( j<0 )
14    throw new E1()
15  if ( j==0 )
16    throw new E2()
17  sum = sum + j
  exit B
```

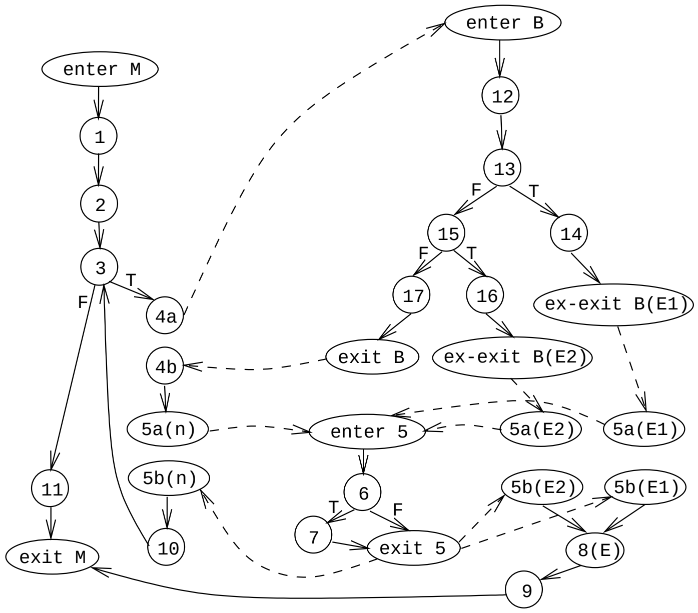

Table 1

du associations
(1, 3, i)*
(1, 6, i)*
(1, 10, i)
(2, 11, sum)
(2, 17, sum)
(10, 3, i)*
(10, 6, i)*
(10, 10, i)
(12, 13, j)*
(12, 15, j)*
(12, 17, j)
(17, 11, sum)
(17, 17, sum)

Table 2

e-du associations	e-du-paths
(14, 14, $evar_{14}$)	(14)
(14, 8, $evar_{active}$)	(14, ex exit B(E1), 5a(E1), enter 5, 6, exit 5, 5b(E1), 8(E))
	(14, ex exit B(E1), 5a(E1), enter 5, 6, 7, exit 5, 5b(E1), 8(E))
(16, 16, $evar_{16}$)	(16)
(16, 8, $evar_{active}$)	(16, ex exit B(E2), 5a(E2), enter 5, 6, exit 5, 5b(E2), 8(E))
	(16, ex exit B(E2), 5a(E2), enter 5, 6, 7, exit 5, 5b(E2), 8(E))

Table 3

e-ad associations	e-ad-paths
(14, 8, $eobj_{14}$)	(14, ex exit B(E1), 5a(E1), enter 5, 6, exit 5, 5b(E1), 8(E))
	(14, ex exit B(E1), 5a(E1), enter 5, 6, 7, exit 5, 5b(E1), 8(E))
(16, 8, $eobj_{16}$)	(16, ex exit B(E2), 5a(E2), enter 5, 6, exit 5, 5b(E2), 8(E))
	(16, ex exit B(E2), 5a(E2), enter 5, 6, 7, exit 5, 5b(E2), 8(E))

Figure 4: Program `Sum1` (top left). ICFG of `Sum1` (top right). Different types of associations in `Sum1` (bottom): Table 1 lists du associations; Table 2 lists e-du associations, and e-du-paths for each e-du association; Table 3 lists e-ad associations, and e-ad-paths for each e-ad association; each row in Tables 1 and 2 lists associations for one definition; each row in Table 3 lists associations for one activation; each du or e-du association that corresponds to a p-use is marked with an asterisk.

(b) all-e-ad-paths does not subsume all-paths. Implied by I4(b). □

(S7) *all-e-ad-paths strictly subsumes all-e-deacts.*

(a) all-e-ad-paths subsumes all-e-deacts. Suppose that all-e-ad-paths does not subsume all-e-deacts. Then there exist a program P and a test suite T such that (P, T) satisfies all-e-ad-paths but does not satisfy all-e-deacts. Let $(i,\ j,\ eobj_k)$ be an e-ad association that is covered by no test in T. By the definition of an e-ad association, $eobj_k \in$ e-act(i), $eobj_k \in$ e-deact(j), and there exists a deact-clear path $p = (i, m_1, m_2, \ldots, m_k, j)$ with respect to $eobj_k$ from i to j. Then by Lemma 3 and the definition of e-ad-path, there exists a path $q = (i, n_1, n_2, \ldots, n_l, j)$ in the flow graph of P such that q is an e-ad-path with respect to $eobj_k$. Because no test in T covers the association $(i,\ j,\ eobj_k)$, no test in T traverses the path q, which contradicts the assumption that (P, T) satisfies all-e-ad-paths.

(b) all-e-deacts does not subsume all-e-ad-paths. Let P be the program `Sum1` shown in Figure 4. `Sum1`

```
  enter M
1   sum = 0
    try
2a    call B
2b    return B
3   catch E e2
4     print e2
5     return
    try
6a    call B
6b    return B
7   catch E e3
8     return
9   print sum
  exit M
```

```
  enter M
10  read j
11  if ( j<-2 )
12    e = new E1()
13  else if ( j>2 )
14    e = new E1()
    else
15    e = new E2()
16  e1 = e
17  if ( j mod 2 != 0 )
18    throw e1
19  sum = sum + j
  exit j
```

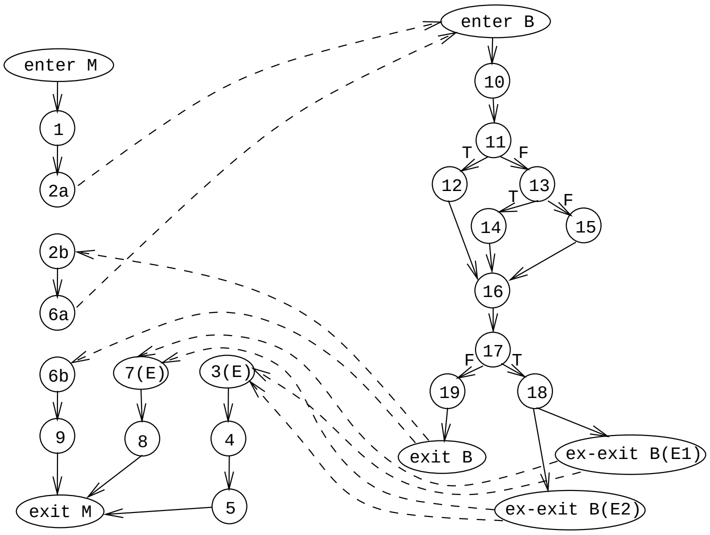

Table 1

e-ad associations
(18, 3, $eobj_{12}$)
(18, 7, $eobj_{12}$)
(18, 3, $eobj_{14}$)
(18, 7, $eobj_{14}$)
(18, 3, $eobj_{15}$)
(18, 7, $eobj_{15}$)

Table 2

du associations	du-paths
(1, 19, `sum`)	(1, 2a, enter B, 10, 11, 12, 16, 17, 19)
	(1, 2a, enter B, 10, 11, 13, 14, 16, 17, 19)
	(1, 2a, enter B, 10, 11, 13, 15, 16, 17, 19)
(10, 11, `j`)*	(10, 11)
(10, 13, `j`)*	(10, 11, 13)
(10, 17, `j`)*	(10, 11, 12, 16, 17)
	(10, 11, 13, 14, 16, 17)
	(10, 11, 13, 15, 16, 17)
(10, 19, `j`)	(10, 11, 12, 16, 17, 19)
	(10, 11, 13, 14, 16, 17, 19)
	(10, 11, 13, 15, 16, 17, 19)
(19, 9, `sum`)	(19, exit B, 6b, 9)
(19, 19, `sum`)	(19, exit B, 2b, 6a, enter B, 10, 11, 12, 16, 17, 19)
	(19, exit B, 2b, 6a, enter B, 10, 11, 13, 14, 16, 17, 19)
	(19, exit B, 2b, 6a, enter B, 10, 11, 13, 15, 16, 17, 19)

Table 3

e-du associations	e-du-paths
(3, 4, `e2`)	(3, 4)
(12, 16, `e`)	(12, 16)
(14, 16, `e`)	(14, 16)
(15, 16, `e`)	(15, 16)
(16, 18, `e1`)	(16, 17, 18)
(16, 4, `e1`→`e2`)	(16, 17, 18, ex exit B(E1), 3(E), 4)
	(16, 17, 18, ex exit B(E2), 3(E), 4)
(18, 3, $evar_{active}$)	(18, ex exit B(E1), 3(E))
	(18, ex exit B(E2), 3(E))
(18, 7, $evar_{active}$)	(18, ex exit B(E1), 7(E))
	(18, ex exit B(E2), 7(E))

Figure 5: Program `Sum2` (top left). ICFG of `Sum2` (top right). Different types of associations in `Sum2` (bottom): Table 1 lists e-ad associations; Table 2 lists du associations, and du-paths for each du association; Table 3 lists e-du associations, and e-du-paths for each e-du association; each row in Table 1 lists associations for one activation; each row in Tables 2 and 3 lists associations for one definition; each du or e-du association that corresponds to a p-use is marked with an asterisk.

---

computes $\sum_{i=1}^{n} x_i$, where each $x_i > 0$; the program exits if it reads an $x_i \leq 0$. Let T be the following test suite:

Test	Input	Path traversed
t_1	`i`=1 `j`=−1	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 14, ex-exit B(E1), 5a(E1), enter 5, 6, exit 5, 5b(E1), 8(E), 9, exit M)
t_2	`i`=1 `j`=0	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 16, ex-exit B(E2), 5a(E2), enter 5, 6, exit 5, 5b(E2), 8(E), 9, exit M)

Then $(P,\, T)$ satisfies all-e-deacts because tests in T cover all e-ad associations listed in Column 1 of Table 3 in Figure 4. $(P,\, T)$ does not satisfy all-e-ad-paths because no test in T covers the e-ad-path (14, ex-exit B(E1), 5a(E1), enter 5, 6, 7, exit 5, 5b(E1), 8(E)). □

(S8) *all-e-deacts strictly subsumes all-e-acts.*

(a) all-e-deacts subsumes all-e-acts. Suppose that all-e-deacts does not subsume all-e-acts. Then there exist a program P and a test suite T such that $(P,\, T)$ satisfies all-e-deacts but does not satisfy all-e-acts. Let node i represent a `throw` statement that activates exception object $eobj_k$ such that the activation is covered by no test in T. Then for all $j \in$ e-ad($eobj_k$, i), no test in T covers e-ad association (i, j, $eobj_k$), which contradicts the assumption that (P, T) satisfies all-e-deacts.

(b) all-e-acts does not subsume all-e-deacts. Implied by I3(a). □

(S9) *all-e-deacts strictly subsumes all-catch.*

(a) all-e-deacts subsumes all-catch. Suppose that all-e-deacts does not subsume all-catch. Then there exist a program P and a test suite T such that (P, T) satisfies all-e-deacts but does not satisfy all-catch. Let j be a catch node that is covered by no test in T. Because all `catch` handlers in P are reachable through an exception raised in P, there exists a `throw` statement in P (with corresponding node i in the flow graph of P) such that the exception raised at that `throw` statement is caught by the handler that corresponds to j. Let $eobj_k \in$ e-act(i). Then $eobj_k \in$ e-deact(j) and there exists a deact-clear path with respect to $eobj_k$ from i to j. Therefore, (i, j, $eobj_k$) is an e-ad association. Because no test in T covers catch node j, no test is T covers the e-ad association (i, j, $eobj_k$). This contradicts the assumption that (P, T) satisfies all-e-deacts.

(b) all-catch does not subsume all-e-deacts. Implied by I2(b). □

(S10) *all-e-acts strictly subsumes all-throw.*

(a) all-e-acts subsumes all-throw. Suppose that all-e-acts does not subsume all-throw. Then there exist a program P and a test suite T such that (P, T) satisfies all-e-acts but does not satisfy all-throw. Let i be a throw node that is covered by no test in T. Then for any $eobj_k$ that is activated at the `throw` statement that corresponds to i, no test in T covers the activation. This contradicts the assumption that (P, T) satisfies all-e-acts.

(b) all-throw does not subsume all-e-acts. Let P be the program `Sum2` shown in Figure 5. `Sum2` computes $\sum_{i=1}^{2} x_i$, where each x_i is an even integer; the program exits if it reads an odd integer. Let T be the following test suite:

Test	Input	Path traversed
t_1	j=−3	(enter M, 1, 2a, enter B, 10, 11, 12, 16, 17, 18, ex-exit B(E1), 3(E), 4, 5, exit M)

Then (P, T) satisfies all-throw because tests in T cover the only `throw` statement in `Sum1` — that corresponds to throw node 18. (P, T) does not satisfy all-e-acts because no test in T covers the activation of $eobj_{14}$ at throw node 18. □

(S11) *all-c-uses strictly subsumes all-catch.*

(a) all-c-uses subsumes all-catch. Suppose that all-c-uses does not subsume all-catch. Then there exist a program P and test suite T such that (P, T) satisfies all-c-uses but does not satisfy all-catch. Let j be a catch node that is covered by no test in T. Then $evar_{active} \in$ e-use(j), and the use of $evar_{active}$ at j is a c-use. Because all catch handlers in P are reachable through some exception that is explicitly raised in P, there exists a `throw` statement (with corresponding node i is the flow graph P) such that the exception raised at that statement is caught by the handler that corresponds to j. Then $evar_{active} \in$ e-def(i), and there exists a def-clear path with respect to $evar_{active}$ from i to j. Therefore, (i, j, $evar_{active}$) is an e-du association that must be covered by a test in T in order for (P, T) to satisfy all-c-uses. Because no test in T covers catch node j, no test in T covers the e-du association (i, j, $evar_{active}$). This contradicts the assumption that (P, T) satisfies all-c-uses.

(b) all-catch does not subsume all-c-uses. Implied by I8(b). □

(S12) *all-c-uses strictly subsumes all-throw.*

(a) all-c-uses subsumes all-throw. Suppose that all-c-uses does not subsume all-throw. Then there exist a program P and test suite T such that (P, T) satisfies all-c-uses but does not satisfy all-throw. Let i be a throw node that is not covered by the tests in T. Then $evar_{active} \in$ e-def(j). Because we assume that no exception is propagated out of P, there exists a `catch` handler in P that handles the exception raised at the `throw` statement corresponding to i; let j be the catch node for that handler. Then j contains a c-use of $evar_{active}$, and there exists a def-clear path with respect to $evar_{active}$ from i to j. Therefore, (i, j, $evar_{active}$) is an e-du association that must be covered by a test in T in order for (P, T) to satisfy all-c-uses. Because no test in T covers throw node i, no test in T covers the e-du association (i, j, $evar_{active}$). This contradicts the assumption that (P, T) satisfies all-c-uses.

(b) all-throw does not subsume all-c-uses. Implied by I8(b). □

(S13) *all-du-paths strictly subsumes all-e-du-paths.*

(a) all-du-paths subsumes all-e-du-paths. Suppose that all-du-paths does not subsume all-e-du-paths. Then there exist a program P and test suite T such that (P, T) satisfies all-du-paths but does not satisfy all-e-du-paths. Let $p = (i, n_1, n_2, \ldots, n_m, j)$ be an e-du-path in P that is not covered by the tests in T. p can be either an e-du-path(v) or an e-du-path($v \rightarrow w$) depending on whether p is defined with respect to exception variable v or the mapping $v \rightarrow w$. Let p be an e-du-path(v). By the definition of an e-du-path(v), $v \in$ e-def(i), $v \in$ e-use(j), and $(i, n_1, n_2, \ldots, n_m, j)$ is a def-clear simple path with respect to v from i to j. But then p is also a du-path [18], which contradicts the assumption that (P, T) satisfies all-du-paths. Let p be an e-du-path($v \rightarrow w$). By the definition of an e-du-path($v \rightarrow w$), $i \in$ tvar-def(v, k), $j \in$ cvar-use(w, l), $<w, l> \in$ e-map(v, k), and there exists a def-clear simple path with respect to v from k to j. Then $v \in$ e-def(i), $w \in$ e-use(j), and there exist def-clear paths with respect to v from i to k and with respect to w from l to j. However, in our exception-handling model, v and w are aliases.[8] Therefore, by the above definitions, it follows that p is also a du-path, and p would be included in the all-du-paths test requirements generated using a data-flow testing approach that considers the effects of aliases [13, 17]. This contradicts the assumption that (P, T) satisfies all-du-paths.

(b) all-e-du-paths does not subsume all-du-paths. Implied by I7(b). □

(S14) *all-uses strictly subsumes all-e-uses.*

(a) all-uses subsumes all-e-uses. Suppose that all-uses does not subsume all-e-uses. Then there exist a program P and test suite T such that (P, T) satisfies all-uses but does not satisfy all-e-uses. Let a be an e-du-association in P that is not covered by the tests in T. Then a can be either of the form (i, j, v) or of the form $(i, j, v \rightarrow w)$. Let a be of the form (i, j, v). By the definition of a, $v \in$ e-def(i), $v \in$ e-use(j), and there exists a def-clear path with respect to v from i to j. Then (i, j, v) is also a du association [18], which contradicts the assumption that (P, T) satisfies all-uses. Let a be of the form $(i, j, v \rightarrow w)$. By the definition of a, $i \in$ tvar-def(v, k), $j \in$ cvar-use(w, l), $<w, l> \in$ e-map(v, k), and there exists a def-clear path with respect to v from k to j. However, in our exception-handling model, v and w are aliases. By the definitions of tvar-def and a, the path from i to j is def-clear with respect to v. In our exception-handling model, w is aliased to v at the catch node l, the scope of w is limited to the corresponding `catch` handler, and by the definition of cvar-use, the path from l to j is def-clear with respect to w. Therefore, (i, j, v) is a du-association that is included in the all-uses test requirements generated using a data-flow testing approach that considers the effects of aliases [13, 17]. But that association is not covered by tests in T,

[8] An *alias* occurs at a program point if two names refer to the same memory location at that point.

which contradiacts the assumption that (P, T) satisfies all-uses.

(b) all-e-uses does not subsume all-uses. Implied by I7(b). □

(S15) *all-defs strictly subsumes all-e-defs.*

(a) all-defs subsumes all-e-defs. The all-defs and all-e-defs criteria can generate a set of alternative test requirements for a program P; let $\mathcal{R}_{defs}$ and $\mathcal{R}_{edefs}$ be the sets of alternative test requirements for P that are generated by the all-defs and all-e-defs criteria, respectively. In general, given any pair of test requirements $R_1 \in \mathcal{R}_{defs}$ and $R_2 \in \mathcal{R}_{edefs}$, a test suite that covers the associations in R_1 may not cover all associations in R_2. However, for any $R_1 \in \mathcal{R}_{defs}$, there exists an element $R_2 \in \mathcal{R}_{edefs}$ such that the associations in R_2 form a subset of the associations in R_1.[9] This follows from the fact that the all-e-defs criterion is a specialization of the all-defs criterion — all-e-defs considers definitions of only exception variables, whereas all-defs considers definitions of all variables — and requires the coverage of a subset of the associations required by all-defs. Therefore, given a test suite T, such that (P,T) satisfies all-defs by covering the associations in R_1, (P,T) also satisfies all-e-defs because T covers all associations in R_2.

(b) all-e-defs does not subsume all-defs. Implied by I10(b). □

(S16) *all-nodes strictly subsumes all-throw.*

all-nodes strictly subsumes all-catch.

(a) Each exception testing criterion listed above is a specialization of the corresponding data-flow testing criterion, and requires a subset of the associations that are required by the corresponding data-flow testing criterion. Therefore, each of the data-flow testing criterion subsumes the corresponding exception testing criterion.

(b) all-catch does not subsume all-nodes. Implied by I7(b).

(c) all-throw does not subsume all-nodes. Implied by I7(b). □

(I1) *Each of all-e-du-paths, all-e-uses, and all-e-defs is incomparable to each of all-e-ad-paths, all-e-deacts, and all-e-acts.*

(a) all-e-du-paths does not subsume all-e-acts. Implied by I4(a).

(b) all-e-ad-paths does not subsume all-e-defs. Let P be the program `Sum3` shown in Figure 6. `Sum3` computes $\sum_{i=1}^{2} x_i$, where each $x_i \geq -1$; the program exits if it reads an $x_i < -1$. Let T be the following test suite:

Test	Input	Path traversed
t_1	i=1 j=−2	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 13, ex-exit B(E), 5(E), 6, 7, exit M)

Then (P, T) satisfies all-e-ad-paths because tests in T cover all e-ad-paths listed in Table 3 in Figure 6. (P, T) does not satisfy all-e-defs because no test in T covers the definition of `sum` at node 16. □

(I2) *Each of all-e-defs and all-throw is incomparable to all-catch.*

(a) all-e-defs does not subsume all-catch. Implied by I10(a).

(b) all-catch does not subsume all-throw. Let P be `Sum1`, and let T be the following test suite:

Test	Input	Path traversed
t_1	i=1 j=−1	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 14, ex-exit B(E1), 5a(E1), enter 5, 6, exit 5, 5b(E1), 8(E), 9, exit M)

Then (P, T) satisfies all-catch because tests in T cover catch node 8. (P, T) does not satisfy all-throw because no test in T covers throw node 16. □

[9] For a program that contains no exception variables, $\mathcal{R}_{edefs}$ is empty, and the all-e-defs criterion is trivially satisfied by any test suite.

```
  enter M
1   read i
2   sum = 0
3   while ( i<10 )
      try
4a      call B
4b      return B
5     catch E e1
6       print e1
7       return
8     i = i + 1
9   print sum
  exit M

  enter C
17  read j
18  if ( j<=0 )
19    e = new E()
    else
20    e = null
21 if ( e!=null )
22    print "j <= 0"
  exit C
```

```
    enter B
10a  call C
10b  return C
11   if ( j<0 )
12     if ( j<-1 )
13       throw new E()
       else
14     if ( e!=null )
15       return
16   sum = sum + j
  exit B
```

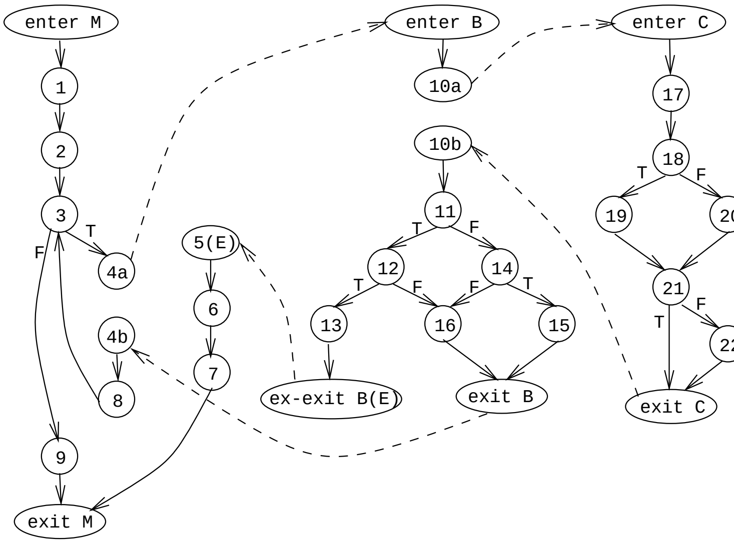

Table 1

du associations
(1, 3, `i`)*
(1, 8, `i`)
(2, 9, `sum`)
(2, 16, `sum`)
(8, 3, `i`)*
(8, 8, `i`)
(16, 9, `sum`)
(16, 16, `sum`)
(17, 11, `j`)*
(17, 12, `j`)*
(17, 16, `j`)
(17, 18, `j`)*

Table 2

e-du associations
(5, 6, `e1`)
(13, 13, $evar_{13}$)
(13, 5, $evar_{active}$)
(19, 14, `e`)*
(19, 21, `e`)*
(20, 14, `e`)*
(20, 21, `e`)*

Table 3

e-ad associations	e-ad-paths
(13, 5, $eobj_{13}$)	(13, ex exit B(E), 5(E))

Figure 6: Program `Sum3` (top left). ICFG of `Sum3` (top right). Different types of associations in `Sum3` (bottom): Table 1 lists du associations; Table 2 lists e-du associations; Table 3 lists e-ad associations, and e-ad-paths for each e-ad association; each row in Tables 1 and 2 lists associations for one definition; each row in Table 3 lists associations for one activation; each du or e-du association that corresponds to a p-use is marked with an asterisk.

(I3) *all-e-acts is incomparable to all-catch.*

(a) all-e-acts does not subsume all-catch. Let P be `Sum2`, and let T be the following test suite:

Test	Input	Path traversed
t_1	j=−3	(enter M, 1, 2a, enter B, 10, 11, 12, 16, 17, 18, ex-exit B(E1), 3(E), 4, 5, exit M)
t_2	j=3	(enter M, 1, 2a, enter B, 10, 11, 13, 14, 16, 17, 18, ex-exit B(E1), 3(E), 4, 5, exit M)
t_3	j=1	(enter M, 1, 2a, enter B, 10, 11, 13, 15, 16, 17, 18, ex-exit B(E2), 3(E), 4, 5, exit M)

Then (P, T) satisfies all-e-acts because tests in T cover at least one e-ad association from each row of Table 1 in Figure 5. (P, T) does not satisfy all-catch because no test in T covers catch node 7.

(b) all-catch does not subsume all-e-acts. Let P be `Sum2`, and let T be the following test suite:

Test	Input	Path traversed
t_1	j=−3	(enter M, 1, 2a, enter B, 10, 11, 12, 16, 17, 18, ex-exit B(E1), 3(E), 4, 5, exit M)
t_1	j=4, 3	(enter M, 1, 2a, enter B, 10, 11, 13, 14, 16, 17, 19, exit B, 2b, 6a, enter B, 10, 11, 13, 14, 16, 17, 18, ex-exit B(E2), 7(E), 8, exit M)

Then (P, T) satisfies all-catch because tests in T cover all catch nodes in `Sum2`, but (P, T) does not satisfy all-e-acts because no test in T covers the activation of $eobj_{15}$ at throw node 18. □

(I4) *Each of all-du-paths, all-uses, all-p-uses/some-c-uses, all-p-uses, all-edges, and all-nodes is incomparable to each of all-e-ad-paths, all-e-deacts, and all-e-acts.*

(a) all-du-paths does not subsume all-e-acts. Let P be `Sum2`, and let T be the following test suite:

Test	Input	Path traversed
t_1	j=−4, 4	(enter M, 1, 2a, enter B, 10, 11, 12, 16, 17, 19, exit B, 2b, 6a, enter B, 10, 11, 13, 14, 16, 17, 19, exit B, 6b, 9, exit M)
t_2	j=4, 2	(enter M, 1, 2a, enter B, 10, 11, 13, 14, 16, 17, 19, exit B, 2b, 6a, enter B, 10, 11, 13, 15, 16, 17, 19, exit B, 6b, 9, exit M)
t_3	j=2, −4	(enter M, 1, 2a, enter B, 10, 11, 13, 15, 16, 17, 19, exit B, 2b, 6a, enter B, 10, 11, 12, 16, 17, 19, exit B, 6b, 9, exit M)
t_4	j=3	(enter M, 1, 2a, enter B, 10, 11, 13, 14, 16, 17, 18, ex-exit B(E1), 3(E), 4, 5, exit M)
t_5	j=1	(enter M, 1, 2a, enter B, 10, 11, 13, 15, 16, 17, 18, ex-exit B(E2), 3(E), 4, 5, exit M)
t_6	j=4, 5	(enter M, 1, 2a, enter B, 10, 11, 13, 14, 16, 17, 19, exit B, 2b, 6a, enter B, 10, 11, 13, 14, 16, 17, 18, ex-exit B(E1), 7(E), 8, exit M)
t_7	j=2, 1	(enter M, 1, 2a, enter B, 10, 11, 13, 15, 16, 17, 19, exit B, 2b, 6a, enter B, 10, 11, 13, 15, 16, 17, 18, ex-exit B(E2), 7(E), 8, exit M)

Then (P, T) satisfies all-du-paths because tests in T cover each du-path and e-du-path listed in Tables 2 and 3 in Figure 5. (P, T) does not satisfy all-e-acts because no test in T covers the activation of $eobj_{12}$ at throw node 18.

(b) all-e-ad-paths does not subsume all-nodes. Let P be `Sum3`, and let T be the following test suite:

Test	Input	Path traversed
t_1	i=1 j=−2	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 13, ex-exit B(E), 5(E), 6, 7, exit M)

Then (P, T) satisfies all-e-ad-paths because tests in T cover all e-ad-paths listed in Table 3 of Figure 6. (P, T) does not satisfy all-nodes because no test in T covers node 16. □

(I5) *all-uses is incomparable to all-e-du-paths.*

(a) all-uses does not subsume all-e-du-paths. Let P be `Sum1`, and let T be the following test suite:

Test	Input	Path traversed
t_1	i=10	(enter M, 1, 2, 3, 11, exit M)
t_2	i=1 j=−1	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 14, ex-exit B(E1), 5a(E1), enter 5, 6, exit 5, 5b(E1), 8(E), 9, exit M)
t_3	i=1 j=0	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 16, ex-exit B(E2), 5a(E2), enter 5, 6, exit 5, 5b(E2), 8(E), 9, exit M)
t_4	i=8 j=1, 2	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 17, exit B, 4b, 5a(n), enter 5, 6, exit 5, 5b(n), 10, 3, 4a, enter B, 12, 13, 15, 17, exit B, 4b, 5a(n), enter 5, 6, exit 5, 5b(n), 10, 3, 11, exit M)

Then (P, T) satisfies all-uses because tests in T cover each du and e-du association listed in Tables 1 and 2 in Figure 4. (P, T) does not satisfy all-e-du-paths because no test in T covers the e-du-path (14, ex-exit

B(E1), 5a(E1), enter 5, 6, 7, exit 5, 5b(E1), 8(E)).

(b) all-e-du-paths does not subsume all-uses. Implied by I7(b). □

(I6) all-p-uses/some-c-uses is incomparable to each of all-e-du-paths and all-e-uses.

(a) all-p-uses/some-c-uses does not subsume all-e-uses. Let P be `Sum2`, and let T be the following test suite:

Test	Input	Path traversed
t_1	j=4, −6	(enter M, 1, 2a, enter B, 10, 11, 13, 14, 16, 17, 19, exit B, 2b, 6a, enter B, 10, 11, 12, 16, 17, 19, exit B, 6b, 9, exit M)
t_2	j=1	(enter M, 1, 2a, enter B, 10, 11, 13, 15, 16, 17, 18, ex-exit B(E2), 3(E), 4, 5, exit M)

Then (P, T) satisfies all-p-uses/some-c-uses because tests in T cover each du and e-du association that corresponds to a p-use (listed in Tables 2 and 3 in Figure 5); in addition, for each definition that has no p-uses, tests in T cover at least one association that corresponds to a c-use. (P, T) does not satisfy all-e-uses because no test in T covers the e-du association (18, 7, $evar_{active}$).

(b) all-e-du-paths does not subsume all-p-uses/some-c-uses. Implied by I7(b). □

(I7) Each of all-p-uses, all-edges, and all-nodes is incomparable to each of all-e-du-paths, all-e-uses, and all-e-defs.

(a) all-p-uses does not subsume all-e-defs. Let P be `Sum3`, and let T be the following test suite:

Test	Input	Path traversed
t_1	i=8 j=−1, 0	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 16, exit B, 4b, 8, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 14, 15, exit B, 4b, 8, 3, 9, exit M)
t_2	i=9 j=1	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 20, 21, exit C, 10b, 11, 14, 16, exit B, 4b, 8, 3, 9, exit M)

Then (P, T) satisfies all-p-uses because tests in T cover each du and e-du association that corresponds to a p-use (listed in Tables 1 and 2 in Figure 6). (P, T) does not satisfy all-e-defs because no test in T covers the definition of `e1` at node 5.

(b) all-e-du-paths does not subsume all-nodes. Let P be `Sum1`, and let T be the following test suite:

Test	Input	Path traversed
t_1	i=1 j=−1	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 14, ex-exit B(E1), 5a(E1), enter 5, 6, exit 5, 5b(E1), 8(E), 9, exit M)
t_2	i=9 j=1, −1	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 17, exit B, 4b, 5a(n), enter 5, 6, exit 5, 5b(n), 10, 3, 4a, enter B, 12, 13, 14, ex-exit B(E1), 5a(E1), enter 5, 6, 7, exit 5, 5b(E1), 8(E), 9, exit M)
t_3	i=1 j=0	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 16, ex-exit B(E2), 5a(E2), enter 5, 6, exit 5, 5b(E2), 8(E), 9, exit M)
t_4	i=9 j=0	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 16, ex-exit B(E2), 5a(E2), enter 5, 6, 7, exit 5, 5b(E2), 8(E), 9, exit M)

Then (P, T) satisfies all-e-du-paths because tests in T cover each e-du-path listed in Table 2 in Figure 4. (P, T) does not satisfy all-nodes because no test in T covers node 11. □

(I8) Each of all-c-uses/some-p-uses and all-c-uses is incomparable to each of all-e-du-paths and all-e-uses.

(a) all-c-uses/some-p-uses does not subsume all-e-uses. Let P be `Sum3`, and let T be the following test suite:

Test	Input	Path traversed
t_1	`i=10`	(enter M, 1, 2, 3, 9, exit M)
t_2	`i=1 j=−2`	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 13, ex-exit B(E), 5(E), 6, 7, exit M)
t_3	`i=8 j=−1, −1`	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 16, exit B, 4b, 8, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 16, exit B, 4b, 8, 3, 9, exit M)
t_4	`i=9 j=1`	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 20, 21, exit C, 10b, 11, 14, 16, exit B, 4b, 8, 3, 9, exit M)

Then (P, T) satisfies all-c-uses/some-p-uses because tests in T cover each du and e-du association that corresponds to a p-use (listed in Tables 1 and 2 in Figure 6); in addition, for each definition that has no c-uses, tests in T cover at least one association that corresponds to a p-use. (P, T) does not satisfy all-e-uses because no test in T covers the e-du association (19, 14, `e`).

(b) all-e-du-paths does not subsume all-c-uses. Let P be `Sum1`, and let T be the following test suite:

Test	Input	Path traversed
t_1	`i=1 j=−1`	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 14, ex-exit B(E1), 5a(E1), enter 5, 6, exit 5, 5b(E1), 8(E), 9, exit M)
t_2	`i=9 j=1, −1`	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 17, exit B, 4b, 5a(n), enter 5, 6, exit 5, 5b(n), 10, 3, 4a, enter B, 12, 13, 14, ex-exit B(E1), 5a(E1), enter 5, 6, 7, exit 5, 5b(E1), 8(E), 9, exit M)
t_3	`i=1 j=0`	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 16, ex-exit B(E2), 5a(E2), enter 5, 6, exit 5, 5b(E2), 8(E), 9, exit M)
t_4	`i=9 j=0`	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 16, ex-exit B(E2), 5a(E2), enter 5, 6, 7, exit 5, 5b(E2), 8(E), 9, exit M)

Then (P, T) satisfies all-e-du-paths because tests in T cover each e-du-path listed in Table 2 in Figure 4. (P, T) does not satisfy all-c-uses because no test in T covers du association (17, 11, `sum`) that corresponds to a c-use. □

(I9) Each of all-c-uses/some-p-uses and all-c-uses is incomparable to each of all-e-ad-paths, all-e-deacts, and all-e-acts.

(a) all-c-uses/some-p-uses does not subsume all-e-acts. Implied by I4(a).

(b) all-e-ad-paths does not subsume all-c-uses. Let P be `Sum3`, and let T be the following test suite:

Test	Input	Path traversed
t_1	`i=1 j=−2`	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 13, ex-exit B(E), 5(E), 6, 7, exit M)

Then (P, T) satisfies all-e-ad-paths because tests in T cover all e-ad-paths listed in Table 3 in Figure 6. (P, T) does not satisfy all-c-uses because no test in T covers du association (16, 9, `sum`) that corresponds to a c-use. □

(I10) *all-defs is incomparable to each of all-e-du-paths, all-e-uses, and all-catch.*

(a) all-defs does not subsume all-catch. Let P be `Sum2`, and let T be the following test suite:

Test	Input	Path traversed
t_1	`j=4, −6`	(enter M, 1, 2a, enter B, 10, 11, 13, 14, 16, 17, 19, exit B, 2b, 6a, enter B, 10, 11, 12, 16, 17, 19, exit B, 6b, 9, exit M)
t_2	`j=1`	(enter M, 1, 2a, enter B, 10, 11, 13, 15, 16, 17, 18, ex-exit B(E2), 3(E), 4, 5, exit M)

Then (P, T) satisfies all-defs because tests in T cover at least one du and e-du association from each row in Tables 2 and 3 in Figure 5. (P, T) does not satisfy all-catch because no test in T covers catch node 7.

(b) all-e-du-paths does not subsume all-defs. Let P be `Sum1`, and let T be the following test suite:

Test	Input	Path traversed
t_1	i=1 j=−1	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 14, ex-exit B(E1), 5a(E1), enter 5, 6, exit 5, 5b(E1), 8(E), 9, exit M)
t_2	i=9 j=1, −1	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 17, exit B, 4b, 5a(n), enter 5, 6, exit 5, 5b(n), 10, 3, 4a, enter B, 12, 13, 14, ex-exit B(E1), 5a(E1), enter 5, 6, 7, exit 5, 5b(E1), 8(E), 9, exit M)
t_3	i=1 j=0	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 16, ex-exit B(E2), 5a(E2), enter 5, 6, exit 5, 5b(E2), 8(E), 9, exit M)
t_4	i=9 j=0	(enter M, 1, 2, 3, 4a, enter B, 12, 13, 15, 16, ex-exit B(E2), 5a(E2), enter 5, 6, 7, exit 5, 5b(E2), 8(E), 9, exit M)

Then (P, T) satisfies all-e-du-paths because tests in T cover each e-du-path listed in Table 2 in Figure 4. (P, T) does not satisfy all-c-uses because no test in T covers the du association $(17, 11, \texttt{sum})$ that corresponds to a c-use. □

(I11) *all-defs is incomparable to each of all-e-ad-paths, all-e-deacts, and all-e-acts.*

(a) all-defs does not subsume all-e-acts. Implied by I4(a).

(b) all-e-ad-paths does not subsume all-defs. Implied by I1(b). □

(I12) *all-c-uses is incomparable to all-e-defs.*

(a) all-c-uses does not subsume all-e-defs. Let P be `Sum3`, and let T be the following test suite:

Test	Input	Path traversed
t_1	i=10	(enter M, 1, 2, 3, 9, exit M)
t_2	i=1 j=−2	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 13, ex-exit B(E), 5(E), 6, 7, exit M)
t_3	i=8 j=−1, −1	(enter M, 1, 2, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 16, exit B, 4b, 8, 3, 4a, enter B, 10a, enter C, 17, 18, 19, 21, 22, exit C, 10b, 11, 12, 16, exit B, 4b, 8, 3, 9, exit M)

Then (P, T) satisfies all-c-uses because tests in T cover each du and e-du association that corresponds to a c-use (listed in Tables 1 and 2 in Figure 6). (P, T) does not satisfy all-e-defs because no test in T covers the definition of `e` at node 20.

(b) all-e-defs does not subsume all-c-uses. Implied by I8(b). □

4 Application of the Criteria

In this section, we briefly discuss the approach we use to compute test requirements that satisfy the exceptions testing criteria.

4.1 Computation of Test Requirements

To generate the test requirements, we add a definition of $evar_{active}$ at each throw node, and a use of $evar_{active}$ followed by an undefinition of $evar_{active}$ at each catch node. To compute e-du associations, we use a reaching definitions, data-flow analysis algorithm. This algorithm first gathers alias[8] information about the program using a technique at the desired level of precision (e.g., [12, 21]), then it uses the alias information to compute, reaching definitions, and finally it uses the reaching definitions to compute the du associations. We then select those du associations that are associated with exception objects.

For the e-ad associations, we first determine activation and deactivation points. We then compute the e-act set for each activation point. Finally, we compute e-deact sets and e-ad associations by computing

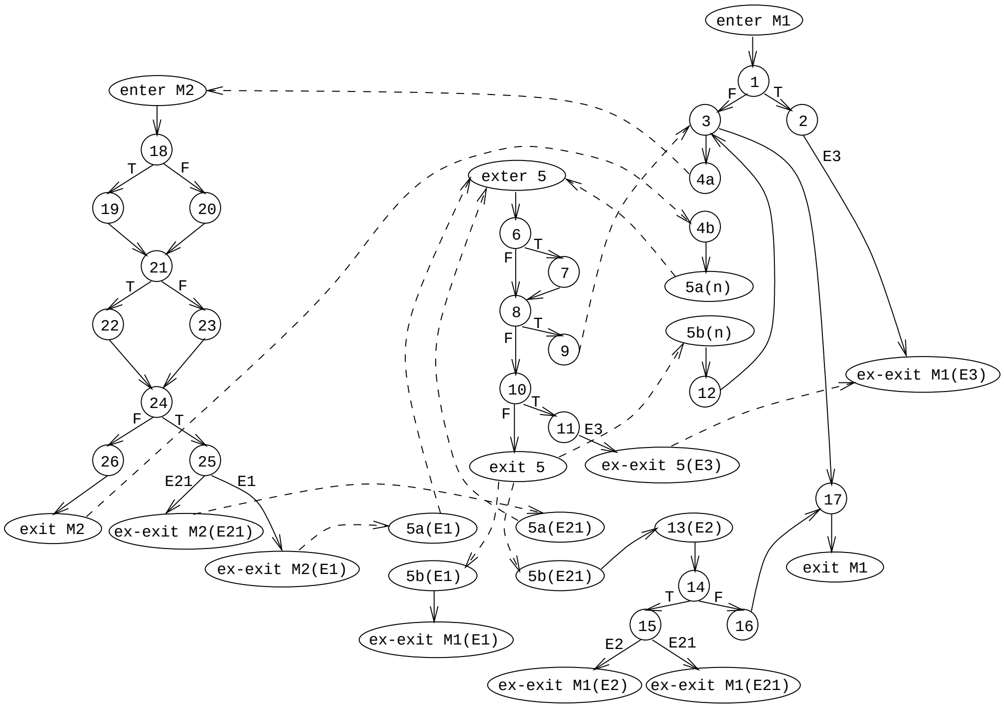

Figure 7: Control-flow representation of class `C2` that is used for unit testing of the class.

reaching activations for each deactivation. An exception activation at node i reaches a deactivation at node j if there exists a def-clear path with respect to $evar_{active}$ from i to j.

4.2 Using the Criteria for Unit Testing

Unit testing focuses on testing individual modules of a program. The goal of unit testing is to establish confidence in the correctness of a module independent of the interaction of that module with other modules. During unit testing, drivers and stubs are used to simulate the interactions of the module being tested with other modules.

The exception testing criteria described in the previous section fails to provide the required test selection when applied to the testing of an incomplete program, such as a unit. For example, if an exception that is activated in the unit being tested is deactivated only outside the unit, the all-e-acts criterion does not require an association that covers the activation of that exception. Consider class `C2` as the module that is being unit tested;[10] Figure 7 shows the control-flow representation for the class that is used for applying the exception testing criteria. As the figure illustrates, the activation of $eobj_{20}$ in statement 15 has no reachable deactivation in the same unit; that deactivation occurs in a different unit (`C2`) that is being tested independently.

A similar problem occurs for covering deactivations: if all reaching activations for a deactivation lie outside the unit being tested, that deactivation is not covered by the all-e-deacts criterion. For example, the `catch` handler in line 28 of method `M1` deactivates several exception activations in the program, but all those activations lie outside the unit being tested. Therefore, the all-e-deacts criterion does not require a

[10] In an object-oriented program, a class is usually considered to be the basic unit of testing [5].

Table 10: Associations required by select criterion for unit testing of class `C2`.

	Associations required	
Criterion	before addition of dummy deacts	after addition of dummy deacts
all-throw	2, 11, 15, 25	2, 11, 15, 25
all-e-acts	(25, 13, $eobj_{20}$)	(25, 13, $eobj_{20}$) (2, ex-exit E3, $eobj_2$) (11, ex-exit E3, $eobj_{11}$) (15, ex-exit E21, $eobj_{20}$)
all-e-deacts	(25, 9, $eobj_{19}$) (25, 9, $eobj_{20}$) (25, 11, $eobj_{19}$) (25, 11, $eobj_{20}$) (25, 13, $eobj_{20}$)	(25, 9, $eobj_{19}$) (25, 9, $eobj_{20}$) (25, 11, $eobj_{19}$) (25, 11, $eobj_{20}$) (25, 13, $eobj_{20}$) (2, ex-exit E3, $eobj_2$) (11, ex-exit E3, $eobj_{11}$) (15, ex-exit E21, $eobj_{20}$) (25, ex-exit E1, $eobj_{19}$)

test case to cover that handler. Such omissions not only lower our confidence in the quality of unit testing, but also limit the applicability of the exception testing criteria.

When applied to incomplete programs, some of the subsumption relationships shown in Figure 3 do not hold. For example, the all-e-acts criterion fails to be equivalent to the all-throw criterion; instead, the all-e-acts criterion is subsumed by the all-throw criterion. This is illustrated in Figure 7 where the all-throw criterion requires the coverage of statement 15, whereas the all-e-acts criterion does not require that statement to be covered.

To overcome the deficiencies of the exception testing criteria for testing incomplete programs, we add dummy uses of exception variables and dummy deactivations of exception objects at relevant exceptional-exit nodes. For example, in the CFG for `M1`, we add the following dummy deactivations: e-deact(ex-exit E21) = $\{eobj_{20}\}$, e-deact(ex-exit E3) = $\{eobj_2, eobj_{11}\}$, and e-deact(ex-exit E1) = $\{eobj_{19}\}$.

Table 10 illustrates the differences in the required associations for the all-e-acts criterion before and after the addition of the dummy deactivations. Before the addition of the nodes, all-e-acts did not subsume all-throw, whereas after the addition it does.

Similar to exit points, we add dummy definitions and activations at relevant catch nodes. For example, catch node 28 in method `M` requires dummy activations. The addition of dummy activations to that node causes an exception testing criterion, such as all-e-acts, to require more rigorous coverage of method `M`. Because `M` is tested independent of the contexts in which it may be used, to gain confidence in the correctness of the `catch` handler in `M`, the handler must be tested for all possible exception types that can be deactivated at that handler: all subtypes of the declared type of the handler. During the actual testing process, the stub that is called at the call site in line 27 is suitably implemented to activate exceptions of each type.

4.3 Using the Criteria for Integration Testing

Integration testing combines the individually tested modules of a program to incrementally build a working program [11]. During integration testing, the emphasis shifts from testing the algorithmic correctness of an individual module to testing the interactions of modules.

Table 11: Differences in testing requirements for the all-e-deacts criterion for exhaustive and selective testing of the integration of classes C1 and C2.

Criterion	Associations required for exhaustive testing	Associations required for selective testing
all-e-deacts	(2, 28, $eobj_2$)	(2, 28, $eobj_2$)
	(11, 28, $eobj_{11}$)	(11, 28, $eobj_{11}$)
	(15, 28, $eobj_{20}$)	(15, 28, $eobj_{20}$)
	(25, 28, $eobj_{19}$)	(25, 28, $eobj_{19}$)
	(25, 9, $eobj_{19}$)	
	(25, 9, $eobj_{20}$)	
	(25, 11, $eobj_{19}$)	
	(25, 11, $eobj_{20}$)	
	(25, 13, $eobj_{20}$)	

Because integration testing incrementally builds a system, a complete program is not available until the final integration step. Integration testing exposes the same deficiencies in the exception testing criteria as unit testing did because both these testing techniques share the common property that they are incomplete programs. These deficiencies are overcome through a similar solution that creates dummy uses/deactivations at exit points of the partially-built system, and dummy definitions/activations at entry points of the system.

Ideally, integration testing should test only the interactions of the new module that is added to the partially-built system at each integration step. Test requirements that are not generated based on these interactions repeat much of the testing that was done at previous integration steps or during unit testing of the individual modules, and therefore, waste time and resources.

Consider the situation in which classes `C1` and `C2` of the sample program are being integrated together after being unit-tested individually. The exception testing criteria can be used to identify test requirements for testing the behavior of exception-handling constructs at this integration step. However, the criteria should be adapted so that they do not generate redundant test requirements, but only the necessary ones. For example, a naive application of the all-e-deacts criterion at the integration step uses the testing requirement described in Table 5, and generates the associations shown in the left column in Table 11. An adapted version of the all-e-deacts criterion, however, generates only those ad-associations in which the activation occurs in class `C2`, and the deactivation occurs in class `C1`. The right column of Table 11 lists the associations generated by the adapted criteria. The data illustrates that the adapted criterion avoids selecting five redundant associations: these associations were exercised during the unit testing of `C2` and provide no additional benefit when they are reexercised during the integration of `C1` and `C2`.

A general adaptation of the all-e-deacts criterion to the integration of modules A and B occurs as follows: if a method in module A invokes a method in module B, generate those e-ad associations in which the activation occurs in B and the deactivation occurs in A. Each exception testing criterion can be adapted similarly to avoid generating redundant test requirements during integration testing.

5 Related Work

Chatterjee and Ryder [1] identify definition-use relationships between `throw` and `catch` statements caused by exception objects. They also identify other definition-use relationships (with respect to ordinary program variables) that arise along control-flow paths induced by the flow of exceptions. The du associations arising from these relationships should be covered by a data-flow-based testing strategy. Chatterjee and Ryder provide an algorithm for computing such du associations for a language model that provides a subset of the Java exception-handling mechanism; their language model excludes `finally` blocks, and includes only one version of the `throw` statement. The intent of their work is to compute such du associations and others that arise because of aliasing between method parameters, polymorphism, and dynamic dispatch; their intent is not to explore ways to test the behavior of exception-handling constructs. Therefore, they do not investigate in any detail the relationships introduced by exception-handling constructs, and they do not define adequacy criteria that cause these relationships to be exercised.

6 Conclusions

In this paper, we have presented a set of adequacy criteria for use in testing exception-handling constructs. Although we described the testing criteria for a Java-like exception-handling model, with some modifications, the criteria can be applied to exception-handling constructs in other languages, such as Ada and C++.

We described the relationships among our testing criteria, and described the relationships among our criteria and the well-known data-flow criteria. These relationships, depicted as subsumption hierarchies, show that our criteria subsume the criteria used in most commercial tools. Therefore, we expect that testing with our criteria would provide a greater degree of confidence in the correctness of the behavior of exception-handling constructs. These relationships also show that data-flow criteria do not subsume some of our criteria. Thus, our criteria provide additional coverage of exceptions over that provided by data-flow coverage, and focus the testing effort on the behavior of exceptions.

We also described the way in which test requirements can be computed for a program using our intraprocedural and interprocedural representations, and presented a methodology for applying the criteria to unit and integration testing of programs that contain exception-handling constructs. Our criteria are thus applicable at different levels of testing.

In this paper, we evaluated the exception testing criteria only in terms of subsumption relationships. Other issues, such as cost of applying the criteria and fault-detection capabilities of the criteria, are also important; our future work will address these issues.

We have implemented an analysis system, written in Java, to provide program analyses of Java subjects. This analysis system takes a set of compiled Java byte-code files, constituting a program, as input, and builds a control-flow graph for every method in the subject. The system uses data-flow information and type inferencing to connect them into one interprocedural control-flow graph. Our future work includes experiments for comparing path covers for the various exception testing criterion, comparing exception testing criteria with data-flow testing criteria, and evaluating the effectiveness of exception testing criteria in detecting faults in Java programs. In future work, we will also investigate ways to extend the exception testing criteria to include implicitly-raised exceptions.

7 Acknowledgments

This work was supported in part by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University.

References

[1] R Chatterjee and B. G. Ryder. Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-382, Rutgers University, Mar. 1999.

[2] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[3] 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.

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

[5] M. J. Harrold and G. Rothermel. Performing dataflow testing on classes. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, December 1994.

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

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

[8] W. E. Howden. Methodology for the generation of program test data. *IEEE Trans. on Computers*, C-24(5):554–559, May 1975.

[9] C. Huang, J. An approach to program testing. *ACM Computing Surveys*, 7(3):114–128, Sep. 1975.

[10] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. on Softw. Eng.*, SE-9(3):347–354, May 1983.

[11] H. K. N. Leung and L. J. White. A study of integration testing and software regression at the integration level. In *Proc. of the Conf. on Softw. Maint.*, pages 290–300, November 1990.

[12] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proc. of 7th Intl. Symp. on the Foundations of Softw. Eng.*, Sep. 1999. to appear.

[13] D. I. S. Marx and P. G. Frankl. The path-wise approach to data flow testing with pointer variables. In *Proc. of the ACM Int'l. Symp. on Softw. Testing and Analysis*, pages 135–146, January 1996.

[14] J. McCabe, T. A complexity measure. *IEEE Transaction on Software Engineering*, SE-2(4):308–320, Dec. 1976.

[15] S. Ntafos. On required elements testing. *IEEE Trans. on Softw. Eng.*, SE-10(6):795–803, Nov. 1984.

[16] S. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. on Softw. Eng.*, 14(6):868–874, June 1988.

[17] T. J. Ostrand and E. J. Weyuker. Data flow-based test adequacy analysis for languages with pointers. In *Proc. of the Third Symp. on Softw. Testing, Analysis, and Verification*, pages 74–86, October 1991.

[18] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Softw. Eng.*, (4):367–375, Apr. 1985.

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

[20] S. Sinha and M. J. Harrold. Analysis of programs that contain exception-handling constructs. In *Proc. of Int'l Conf. on Softw. Maint.*, pages 348–357, Bethesda, MD, Nov. 1998.

[21] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. of POPL'96 ACM Symp. on Prin. of Prog. Lang.*, pages 32–41, 1996.

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