Proceedings of the International Conference on Software Maintenance, Oxford, England, August-September 1999 ©*IEEE*

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
{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 testing 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, integration testing.

1 Introduction

Structural testing techniques [15] 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., [7, 13]). *Data-flow-based* structural testing criteria use data-flow relationships in a program to guide the selection of test cases (e.g., [2, 5, 10, 14, 16]). 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., [4, 6, 11]). 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 [2, 15, 16]. Criterion A *subsumes* criterion B if and only if any test suite that satisfies A also satisfies B [2, 16].

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 indicates that such constructs occur frequently [18]. Despite the frequency of their occurrences, the behavior of exception-handling constructs is often the least understood and poorly tested part of a program [17]. 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.

Testing 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 [21]. Other informally-stated criteria require all exceptions to be raised in a program under test. These criteria lack a systematic and structured approach to testing the behavior of

[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.

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 [18], 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 present a subsumption hierarchy of the criteria, and illustrate the relationship of the criteria to those found in traditional data-flow testing criteria [2, 16]. We describe techniques for generating the testing 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.

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 [3]. We also discuss our control-flow representations that account for exception-handling constructs.

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` statement.

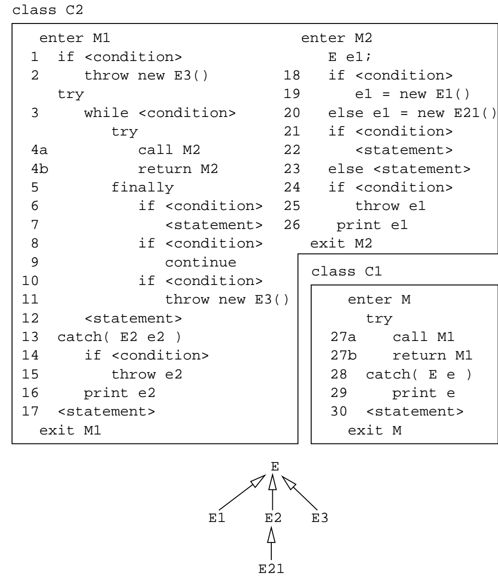

Figure 1. Pseudo-code for sample program that uses Java-like exception-handling constructs (above); hierarchy of exceptions used in the sample program (below).

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

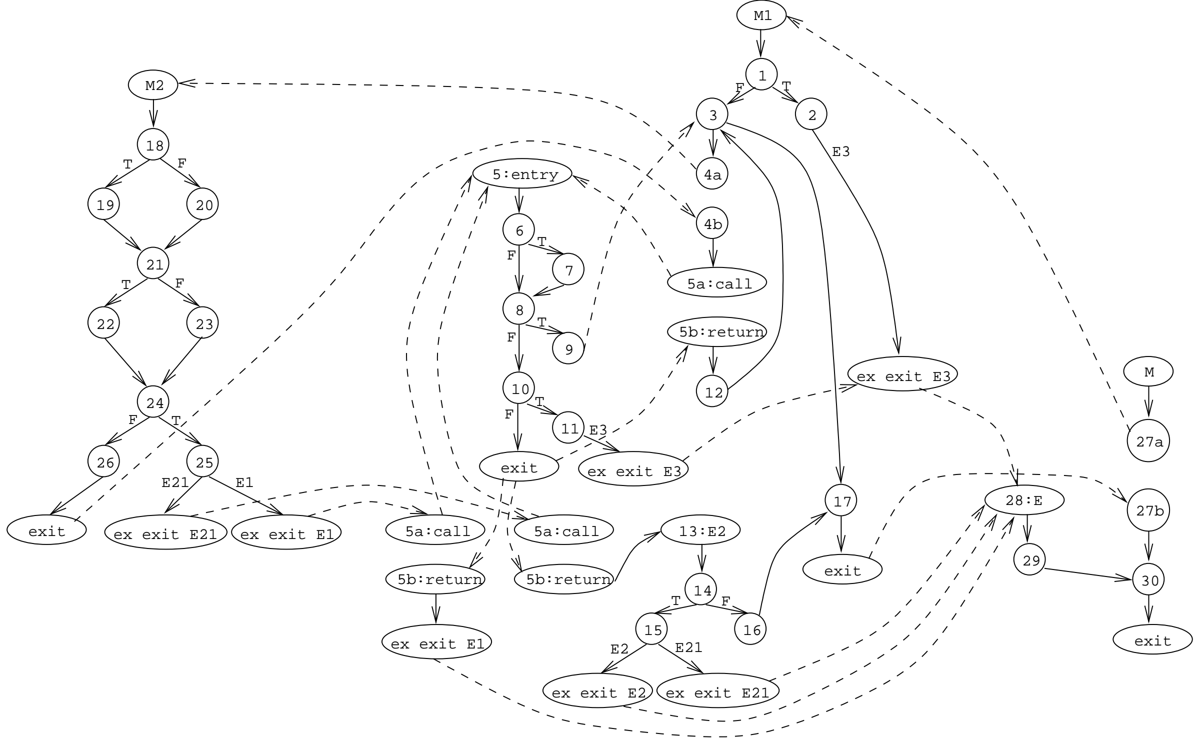

Figure 2. Interprocedural control-flow graph for the sample program.

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 [18] 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 Interprocedural Control-Flow Graph

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). The CFG-construction algorithm [18] 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 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 by 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 handler. To model propagation of exceptions out of a method, the technique creates an exceptional-exit node for each type of exception that can be propagated out of the method. For example, method `M2` propagates exception types `E1` and `E21`; therefore, the CFG for `M2` contains two exceptional-exit nodes (labeled "ex exit E1" and "ex exit 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 under one of two contexts: a normal context or an exceptional context. A `finally` block executes under *normal context* when (1) control reaches the end of a `try` block or a `catch` block, or (2) when control leaves a `try` statement because of a transfer-of-control statement, such as `break`, `continue`, or `return`. A `finally` block executes under *exceptional context* when control leaves a `try` statement because of the propagation of an exception. The context of execution of a `finally` block determines where control flows from that `finally` block. The CFG-construction algorithm treats each `finally` block as a separate method: it creates a separate CFG for each `finally` block, and inserts call nodes to those `finally` blocks appropriately. Figure 2 illustrates the CFG created for the `finally` block that appears in line 5 of the sample program. The figure also contains, call and return nodes for calls to the `finally` block.

To create an interprocedural representation, the algorithm constructs an interprocedural control-flow graph (ICFG) by connecting CFGs 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. The technique also creates exceptional-return edges that connect an exceptional-exit node to a catch node, a finally-call node, or an exceptional-exit node. For example, in Figure 2, exceptional-exit node E1 in the CFG for method `M2` is connected to finally-call node 5a by an exceptional-return edge.

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 [2, 16], 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 a null value, 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 graph G (G can be a CFG or another representation, such as 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 $evar_{active}$ and 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

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`

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 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.[2] 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.

A *definition-clear path* (def-clear path) with respect to exception variable v is a path $(i, n_1, \cdots, n_m, j)$ in G, $m \geq 0$, such that there is no definition of v in $n_1, \cdots, n_m$.[3]

To generate testing 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. 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 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.

[2] 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 [2, 16]. For simplicity, we use *e-use(i)* to represent both of those types of uses.

[3] If G is an ICFG, we consider only realizable paths in G [9]. Reference [19] defines realizable paths in the ICFG for a program that contains exception-handling constructs.

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	($evar_2$→`e`, 2)	29
($evar_{active}$, 2)	28	($evar_{11}$→`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		

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

- For throw node i and an exception variable v ($v \in$ e-use(i)), a *throw-variable definition set*, *tvar-def*(v, i), contains the set of nodes j such that $v \in$ e-def(j) and there exists a def-clear path with respect to v from j to i. tvar-def(v, i) stores nodes that contain definitions of v that reach the use of v at throw node i. 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 $(i, n_1, \cdots, n_m, j)$, $m \geq 0$, in G. For example, e-map(`e1`, 25) = $\{<$`e2`, 13$>$, $<$`e`, 28$>\}$.

An *exception definition-use set*, *e-du*, contains, for each

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		$eobj_2, eobj_{11}, eobj_{19}, eobj_{20}$

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`$\rightarrow$`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 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.

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

A *deactivation-clear path* (deact-clear path) with respect to exception object $eobj_k$ is a sequence of nodes $(i, n_1, \cdots, n_m, j)$, $m \geq 0$, such that there is no deactivation of $eobj_k$ in $n_1, \cdots, n_m$. 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 *simple path* is path in which all nodes, except possibly the first and last, are distinct. A path $(i, n_1, \cdots, n_m, j)$, $m \geq 0$, is an *e-du-path with respect to exception variable* v if $v \in$ e-def(i), $v \in$ e-use(j), and $(i, n_1, \cdots, n_m, j)$ is a def-clear simple path with respect to v. A path $(i, n_1, \cdots, n_m, j)$, $m \geq 2$, is an *e-du-path with respect to mapping* $v \rightarrow w$ 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 $(i, n_1, \cdots, n_m, j)$, $m \geq 0$, 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 $(i, n_1, \cdots, n_m, j)$ 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.

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

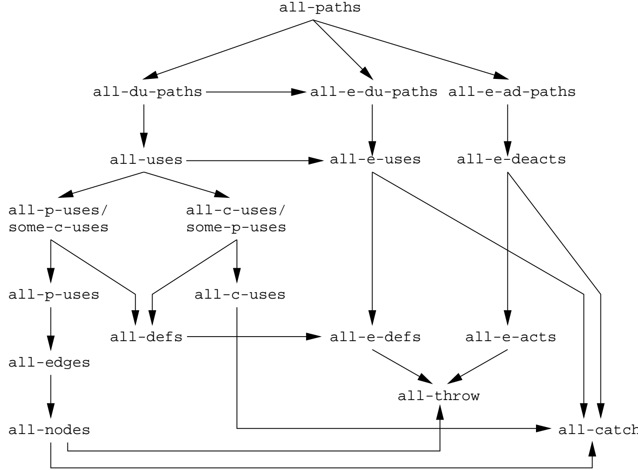

Figure 3. The subsumption hierarchies of the data-flow testing criteria (left), and the exception testing criteria (right).

based on the assumption that a complete program is being tested. In the next section, we relax the restriction and illustrate how the criteria can be used 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.

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 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 [2, 16]. The *all-e-defs* criterion requires the coverage of every definition of each exception variable to some some reachable use. The *all-e-uses* criterion requires the coverage of every 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.

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))

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.

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 criteria. Reference [19] proves the subsumption relationships depicted in the figure.

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; more detail can be found in Reference [19]. We then discuss the way in which the criteria can be used during unit testing and integration testing.

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 a definition 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

Figure 4. Control-flow representation of class `C2` that is used for unit testing of the class.

alias[4] information about the program using a technique at the desired level of precision (e.g., [12, 20]), 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 using the interprocedural representation of the program. We then compute the e-act set for each activation point. Finally, we compute e-deact sets and e-ad associations by computing 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;[5] Figure 4 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.

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

Table 6. Associations required by select criterion for unit testing of class `C2`.

Criterion	Associations required: before addition of dummy deacts	Associations required: 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, $eobj_{20}$)

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 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 4 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 6 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 ac-

[5] In an object-oriented program, a class is usually considered to be the basic unit of testing [4].

tivations 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.

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 only test the interactions of the new module that is added to the partially-built system at each integration step. Testing 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 testing 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 7. 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 7 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`.

Table 7. 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	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, $eobj_{20}$)	

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 testing 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 testing 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] 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.

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

[4] M. J. Harrold and G. Rothermel. Performing dataflow testing on classes. In *Proc. of the 2nd ACM SIGSOFT Symp. on Foundations of Softw. Eng.*, pages 154–163, Dec. 1994.

[5] 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.

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

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

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

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

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

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

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

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

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

[18] 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, Nov. 1998.

[19] S. Sinha and M. J. Harrold. Criteria for testing exception-handling constructs in Java programs. Technical Report OSU-CISRC-6/99-TR16, The Ohio State Univ., June 1999.

[20] 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.

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