Chapter 7

Graph Coverage

Date last generated: September 29, 2014.

DRAFT: Prepublication draft, Fall 2014, George Mason University
Redistribution is forbidden without the express permission of the authors

If you build a better mouse trap you will catch better mice.
– George Gobel

This chapter introduces some of the most widely known test coverage criteria. This chapter uses graphs to define criteria and design tests. This starts our progession into the RIPR model by ensuring that tests "reach" certain locations in a graph model of the artifact being tested. The chapter starts with basic theory as a way to make the practical and applied portions of the chapter easier to follow. We first emphasize a generic view of a graph without regard to the graph's source. After this model is established, the rest of the chapter turns to practical applications by demonstrating how graphs can be obtained from various software artifacts and how the generic versions of the criteria are adapted to those graphs.

7.1 Overview

Directed graphs form the foundation for many coverage criteria. They come from many sources and types of software artifacts, including control flow graphs from source, design structures, finite state machines, statecharts, and use cases, among others. We use the term *artifact* in the most general way, to be anything associated with the software, including the requirements, design documents, implementation, tests, user manuals, and many others. Graph criteria usually require the tester to "cover" the graph in some way, usually by traversing specific portions of the graph. This overview presents graphs in general terms,

[0]© Ammann & Offutt, 2014, *Introduction to Software Testing*

and overlaps standard texts on discrete math, algorithms, and graph theory. Unlike those theoretical treatments, we focus only on the ideas needed for testing and introduce some new terminology that enable test design.

Given an artifact under test, the idea is to extract a graph from that artifact. For example, the most common graph abstraction for source code maps executable statements and branches to a control flow graph. It is important to recognize that the graph is not the same as the artifact, and usually omits certain details. It is also possible for the same artifact to have several useful, but different, graph abstractions. The same abstraction that produces the graph from the artifact also maps test cases for the artifact to paths in the graph. Accordingly, a graph-based coverage criterion evaluates a test set for an artifact in terms of how the paths corresponding to the test cases "cover" the artifact's graph abstraction.

We give our basic notion of a graph below and will add additional structures later in the chapter when needed. A graph G formally is:

- a set N of *nodes*
- a set N_0 of *initial nodes*, where $N_0 \subseteq N$
- a set N_f of *final nodes*, where $N_f \subseteq N$
- a set E of *edges*, where E is a subset of $N \times N$

For a graph to be useful for generating tests, it is necessary for N, N_0, and N_f to contain at least one node each. Sometimes, it helps to consider only part of a graph. A *subgraph* of a graph is also a graph, and is defined by a subset of N, along with the corresponding subsets of N_0, N_f, and E. Specifically, if N_{sub} is a subset of N, then for the subgraph defined by N_{sub}, the set of initial nodes is $N_{sub} \cap N_0$, the set of final nodes is $N_{sub} \cap N_f$, and the set of edges is $(N_{sub} \times N_{sub}) \cap E$.

Note that more than one initial node can be present; that is, N_0 is a set. Having multiple initial nodes is necessary for some software artifacts, for example, if a class has multiple entry points, but sometimes we will restrict the graph to having one initial node. Edges are considered to be *from* one node and *to* another and written as (n_i, n_j). The edge's initial node n_i is sometimes called the *predecessor* and n_j is called the *successor*.

We always identify final nodes, and there must be at least one final node. The reason is that every test must start in some initial node, and end in some final node. The concept of a final node depends on the kind of software artifact the graph represents. Some test criteria require tests to end in a particular final node. Other test criteria are satisfied with any node for a final node, in which case the set N_f is the same as the set N.

The term "node" has various synonyms. Graph theory texts sometimes call a node a *vertex*, and testing texts typically identify a node with the structure it represents, often a statement, a state, a method, or a basic block. Similarly, graph theory texts sometimes call an edge an *arc*, and testing texts typically identify an edge with the structure it represents, often a branch or a transition. This section discusses graph criteria in a generic way; thus we use general graph terms.

Graphs are often drawn with bubbles and arrows. Figure 7.1 shows three example graphs. The nodes with incoming edges but no predecessor nodes are the initial nodes. The nodes with heavy borders are final nodes. Figure 7.1(a) has a single initial node. Figure 7.1(b) has three initial nodes. Figure 7.1(c) has no initial nodes, and so is not useful for generating test cases.

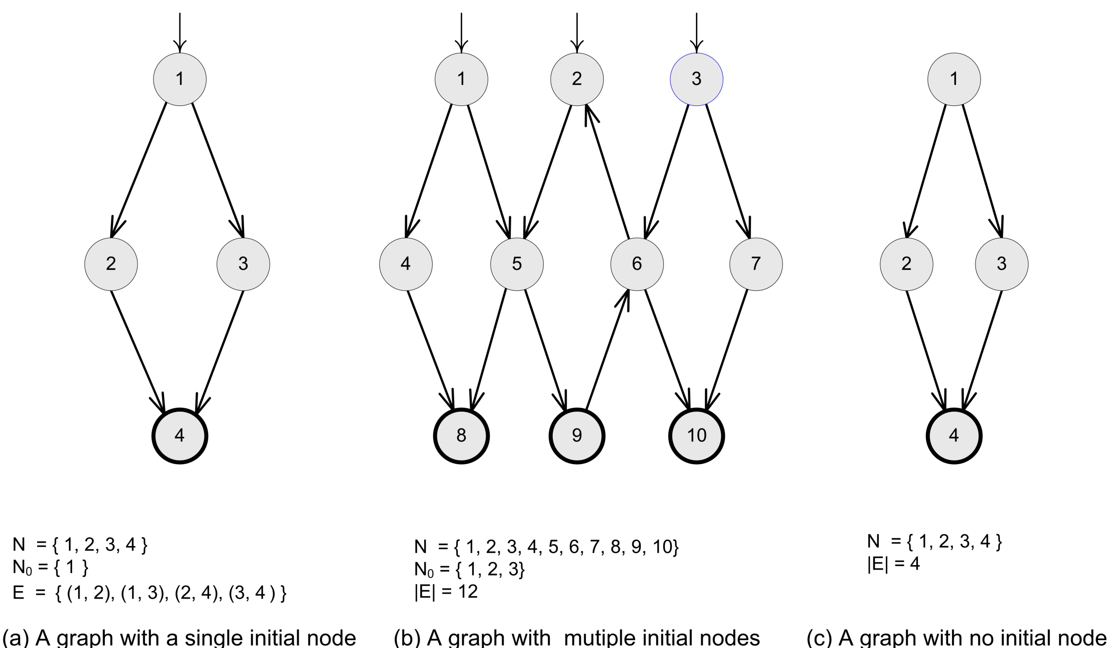


Figure 7.1: Graph (a) has a single initial node, graph (b) multiple initial nodes, and graph (c) (rejected) with no initial nodes.

A *path* is a sequence $[n_1,\ n_2,\ ...,\ n_M]$ of nodes, where each pair of adjacent nodes, $(n_i,\ n_{i+1})$, $1 \leq i < M$, is in the set E of edges. The length of a path is defined as the number of edges it contains. We sometimes consider paths and subpaths of length zero. A *subpath* of a path p is a subsequence of p (possibly p itself). Following the notation for edges, we say a path is *from* the first node in the path and *to* the last node in the path. It is also useful to be able to say that a path is *from* (or *to*) an edge e, which simply means that e is the first (or last) edge in the path. A *cycle* is a path that begins and ends at the same node. For example, the path $[2,\ 5,\ 9,\ 6,\ 2]$ in Figure 7.1(b) is a cycle.

Figure 7.2 shows a graph along with several example paths, and several examples that are not paths. For instance, the sequence $[1,\ 8]$ is not a path because the two nodes are not connected by an edge.

Many test criteria require inputs that start at one node and end at another. This is only possible if those nodes are connected by a path. When we apply these criteria on specific graphs, we sometimes find that we have asked for a path that for some reason cannot be executed. For example, a path may demand that a loop be executed zero times in a situation where the program always executes the loop at least once. This kind of problem is based on

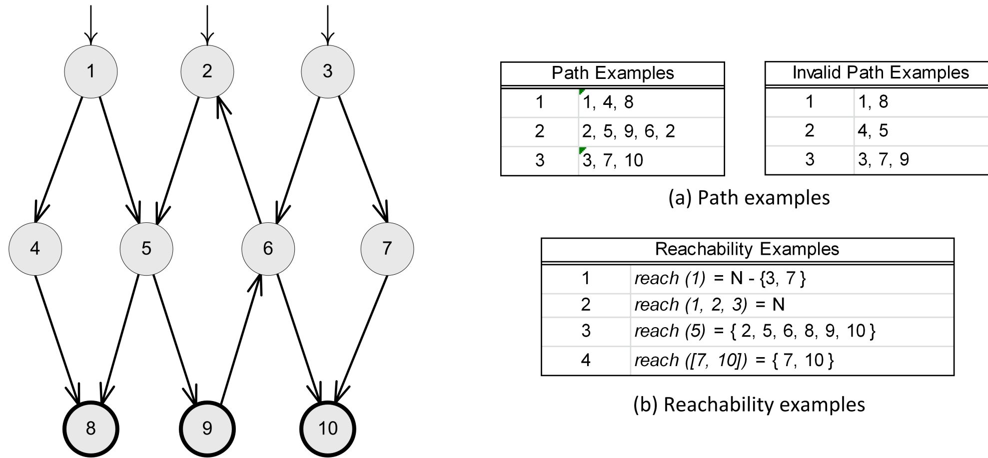


Figure 7.2: Example of paths.

the **semantics** of the software artifact that the graph represents. For now, we emphasize that we are looking only at the **syntax** of the graph.

We say that a node n (or an edge e) is *syntactically reachable* from node n_i if there exists a path from node n_i to n (or edge e). A node n (or edge e) is also *semantically reachable* if it is possible to execute at least one of the paths with some input. We can define the function $reach_G(x)$ as the portion of a graph that is syntactically reachable from the parameter x. The parameter for $reach_G()$ can be a node, an edge, or a set of nodes or edges. Then $reach_G(n_i)$ is the subgraph of G that is syntactically reachable from node n_i, $reach_G(N_0)$ is the subgraph of G that is syntactically reachable from any initial node, $reach_G(e)$ is the subgraph of G syntactically reachable from edge e, and so on. In our use, $reach_G()$ includes the starting nodes. For example, both $reach_G(n_i)$ and $reach_G([n_i,\ n_j])$ always include n_i, and $reach_G([n_i,\ n_j])$ includes edge $([n_i,\ n_j])$. Some graphs have nodes or edges that cannot be syntactically reached from any of the initial nodes in N_0. Since they are unreachable, they make it impossible to fully satisfy a coverage criterion, so we restrict attention to $reach_G(N_0)$.[1]

Consider the examples in Figure 7.2. From 1, it is possible to reach all nodes except 3 and 7. From the entire set of initial nodes $\{1,\ 2,\ 3\}$, it is possible to reach all nodes. If we start at 5, it is possible to reach all nodes except 1, 3, 4, and 7. If we start at edge $(7, 10)$, it is possible to reach only 7, 10 and edge $(7, 10)$. In addition, some graphs (such as finite state machines) have explicit edges from a node to itself, that is, (n_i, n_i).

Basic graph algorithms, usually given in standard data structures texts, can be used to compute syntactic reachability.

A test path represents the execution of a set of test cases. The reason test paths must start in N_0 is that test cases always begin from an initial node. It is important to note that a single test path may correspond to a very large number of test cases on the software. It is

[1] By way of example, typical control flow graphs have very few, if any, syntactically unreachable nodes, but call graphs, especially for object-oriented programs, often do.

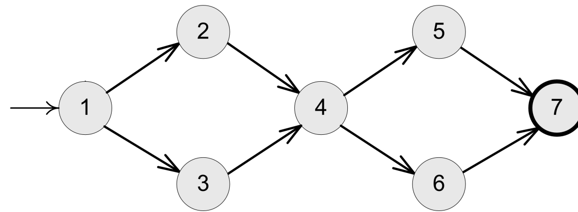


Figure 7.3: A Single-Entry Single-Exit graph.

also possible that a test path may correspond to zero test cases if the test path is infeasible. We return to the crucial but theoretical issue of infeasibility later, in Section 7.2.1.

Definition 7.1 Test path: *A path p, possibly of length zero, that starts at some node in N_0 and ends at some node in N_f.*

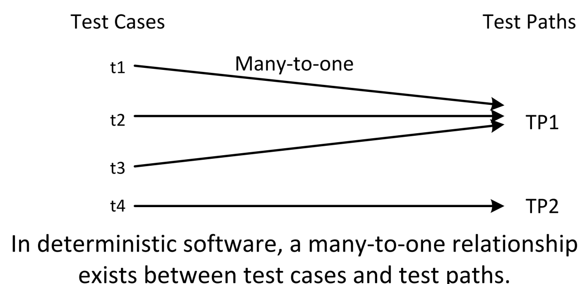
For some graphs, all test paths start at one node and end at a single node. We call these *single entry/single exit* or *SESE* graphs. For SESE graphs, the set N_0 has exactly one node, called n_0, and the set N_f also has exactly one node, called n_f, which may be the same as n_0. We require that n_f be syntactically reachable from every node in N, and that no node in N (except n_f) be syntactically reachable from n_f (unless n_0 and n_f are the same node). In other words, no edges start at n_f, except when n_0 and n_f happen to be the same node.

Figure 7.3 is an example of a SESE graph. This particular structure is sometimes called a "double-diamond" graph, and corresponds to the control flow graph for a sequence of two `if-then-else` statements. The initial node, 1, is designated with an incoming arrow (remember we only have one initial node), and the final node, 7, is designated with a thick circle. Exactly four test paths exist in the double-diamond graph: [1, 2, 4, 5, 7], [1, 2, 4, 6, 7], [1, 3, 4, 5, 7], and [1, 3, 4, 6, 7].

We need some terminology to express the notion of nodes, edges, and subpaths that appear in test paths, and choose familiar terminology from traveling. A test path p is said to *visit* node n if n is in p. Test path p is said to *visit* edge e if e is in p. The term visit applies well to single nodes and edges, but sometimes we want to consider subpaths. For subpaths, we use the term **tour**. Test path p is said to *tour* subpath q if q is a subpath of p. The first path of Figure 7.3, [1, 2, 4, 5, 7], visits nodes 1 and 2, visits edges (1, 2) and (4, 5), and tours the subpath [2, 4, 5] (among others, these lists are not complete). Since the subpath relationship is reflexive, the tour relationship is also reflexive. That is, any given path p always tours itself.

We define a mapping $path_G$ for tests, so for a test case t, $path_G(t)$ is the test path in graph G that is executed by t. If it is obvious which graph we are discussing, we omit the subscript G. We also define the set of paths toured by a set of tests. For a test set T, $path(T)$ is the set of test paths that are executed by the tests in T: $path_G(T) = \{path_G(t) \mid t \in T\}$.

Except for nondeterministic structures, which we do not consider until **Chapter ???,** each test case will tour exactly one test path in graph G. Figure 7.4 illustrates the difference with respect to test case/test path mapping for deterministic vs. nondeterministic software.



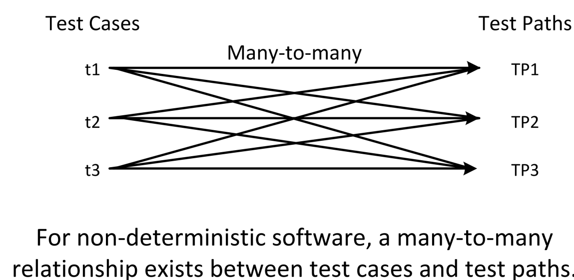


Figure 7.4: Test case mappings to test paths.

Figure 7.5 illustrates a set of test cases and corresponding test paths on a SESE graph with the final node $n_f = 3$. Some edges are annotated with predicates that describe the conditions under which that edge is traversed. (This notion is formalized later in this chapter.) So, in the example, if a is less than b, the only path is from 1 to 2 and then on to 4 and 3. This book describes all of the graph coverage criteria in terms of relationships of test paths to the graph in question, but it is important to realize that testing is carried out with test cases, and that the test path is simply a model of the test case in the abstraction captured by the graph. To reduce cost, we usually want the fewest test paths that will satisfy our test requirements. A *minimal* set of test paths has the property that if we take any test path out, it will no longer satisfy our criterion.

Exercises, Section 7.1.

1. Give the sets N, N_0, N_f, and E for the graph in Figure 7.2.
2. Give a path that is not a test path in Figure 7.2.
3. List all test paths in Figure 7.2.
4. In Figure 7.5, find test case inputs such that the corresponding test path visits edge (2, 4).

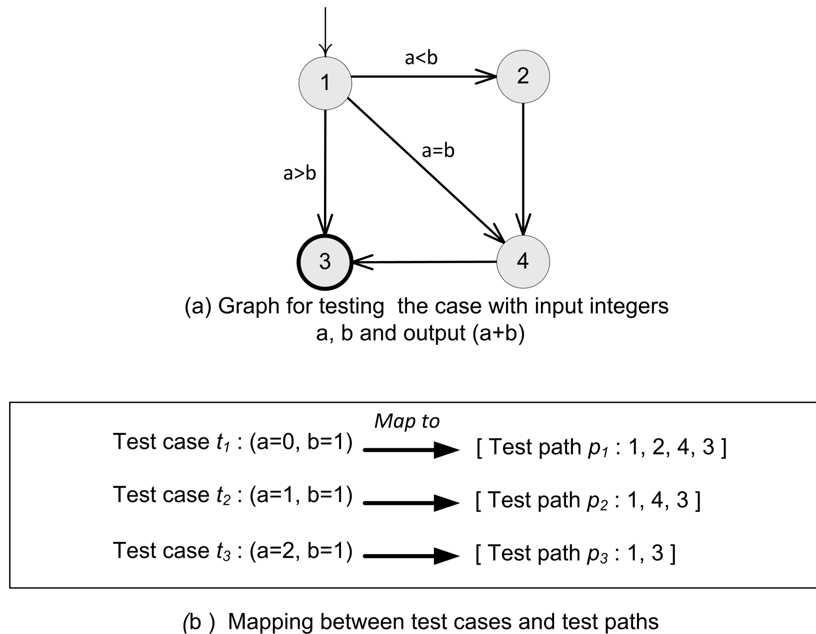


Figure 7.5: A set of test cases and corresponding test paths.

7.2 Graph Coverage Criteria

The structure in Section 7.1 is adequate to define coverage on graphs. As is usual in the testing literature, we divide these criteria into two types. The first are usually referred to as *control flow coverage* criteria, or more generally, *structural graph coverage criteria*. The other criteria are based on the flow of data through the software artifact represented by the graph, and are called *data flow coverage* criteria. Following the discussion in Chapter 1, we identify the appropriate test requirements and then define each criterion in terms of the test requirements. In general, for any graph-based coverage criterion, the idea is to identify the test requirements in terms of various structures in the graph.

For graphs, coverage criteria define test requirements, TR, in terms of properties of test paths in a graph G. A typical test requirement is *met* by *visiting* a particular node or edge or by *touring* a particular path. The definitions we have given so far for a *visit* are adequate, but the notion of a *tour* requires more development. We return to the issue of touring later in this chapter, and then refine it further in the context of data flow criteria. The following definition is a refinement of the definition of coverage given in Chapter 5:

Definition 7.2 Graph Coverage: *Given a set TR of test requirements for a graph criterion C, a test set T satisfies C on graph G if and only if for every test requirement tr in TR, there is at least one test path p in $path(T)$ such that p meets tr.*

This is a very general statement that must be refined for different kinds of graphs.

7.2.1 Structural Coverage Criteria

We define graph coverage criteria by specifying a set of test requirements, TR. We will start by defining criteria to visit every node and then every edge in a graph. The first criterion is probably familiar and is based on the old notion of executing every statement in a program. This concept has variously been called "statement coverage," "block coverage," "state coverage," and "node coverage." We use the general graph term Node Coverage. This concept is probably familiar and simple, so we use it to introduce some additional notation. The notation initially seems to complicate the criterion, but ultimately has the effect of making subsequent criteria cleaner and mathematically precise, avoiding confusion with more complicated situations.

The requirements produced by a graph criterion are technically predicates that can have either the value true (the requirement has been met) or false (the requirement has **not** been met). For the double-diamond graph in Figure 7.3, the test requirements for Node Coverage are: $TR = \{visit\ 1,\ visit\ 2,\ visit\ 3,\ visit\ 4,\ visit\ 5,\ visit\ 6,\ visit\ 7\}$. That is, we must satisfy a predicate for each node, where the predicate asks whether the node has been visited or not. With this in mind, the formal definition of Node Coverage is as follows[2]:

Definition 7.3 Node Coverage (Formal Definition): *For each node* $n \in reach_G(N_0)$*,* TR *contains the predicate "visit n."*

This notation, although mathematically precise, is too cumbersome for practical use. Thus we choose to introduce a simpler version of the definition that abstracts the issue of predicates in the test requirements.

CRITERION **7.1 Node Coverage (NC):** *TR contains each reachable node in G.*

With this definition, it is left as understood that the term "contains" actually means "contains the predicate $visit_n$." This simplification allows us to shorten the writing of the test requirements for Figure 7.3 to only contain the nodes: $TR = \{1, 2, 3, 4, 5, 6, 7\}$. Test path $p_1 = [1, 2, 4, 5, 7]$ meets the first, second, fourth, fifth, and seventh test requirements, and test path $p_2 = [1, 3, 4, 6, 7]$ meets the first, third, fourth, sixth, and seventh. Therefore, if a test set T contains $\{t_1, t_2\}$, where $path(t_1) = p_1$ and $path(t_2) = p_2$, then T satisfies Node Coverage on G.

The usual definition of Node Coverage omits the intermediate step of explicitly identifying the test requirements, and is often stated as given below. Notice the economy of the form used above with respect to the standard definition.

Definition 7.4 Node Coverage (NC) (Standard Definition): *Test set T satisfies node coverage on graph G if and only if for every syntactically reachable node n in N, there is some path p in $path(T)$ such that p visits n.*

[2] Our mathematician readers might notice that this definition is constructive in that it defines what is in the set TR, but does not actually bound the set. It is certainly our intention that TR contains no other elements.

The exercises at the end of the section have the reader reformulate the definitions of some of the remaining coverage criteria in both the formal way and the standard way. We choose the intermediate definition because it is more compact, avoids the extra verbiage in a standard coverage definition, and focuses just on the part of the definition of coverage that changes from criterion to criterion.

Node Coverage is implemented in many commercial testing tools, most often in the form of statement coverage. So is the next common criterion of Edge Coverage, usually implemented as branch coverage:

CRITERION **7.2 Edge Coverage (EC):** *TR contains each reachable path of length up to 1, inclusive, in G.*

The reader might wonder why the test requirements for Edge Coverage also explicitly include the test requirements for Node Coverage–that is, why the phrase "up to" is included in the definition. All the graph coverage criteria are developed like this. The motivation is subsumption for graphs that do not contain more complex structures. For example, consider a graph with a node that has no edges. Without the "up to" clause in the definition, Edge Coverage would not cover that node. Intuitively, we would like edge testing to be at least as demanding as node testing. This style of definition is the best way to achieve this property. To make our TR sets readable, we list only the maximal length paths.

Figure 7.6 illustrates the difference between Node and Edge Coverage. In program statement terms, this is a graph of the common "if-else" structure without the "else."

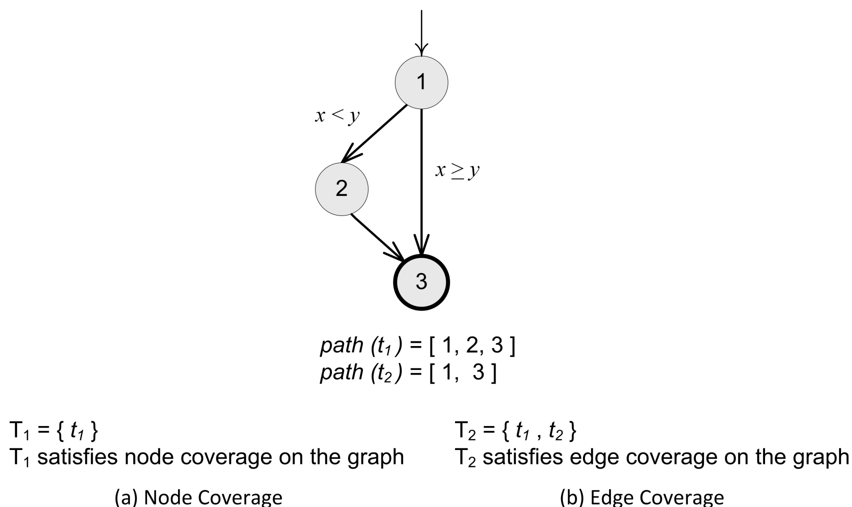


Figure 7.6: A graph showing Node Coverage and Edge Coverage.

Other coverage criteria use only the graph definitions introduced so far. For example, one requirement is that each path of length (up to) two be toured by some test path. With this context, Node Coverage could be redefined to contain each path of length zero. Clearly,

this idea can be extended to paths of any length, although possibly with diminishing returns. We formally define one of these criteria; others are left as exercises for the interested reader.

CRITERION **7.3 Edge-Pair Coverage (EPC):** *TR contains each reachable path of length up to 2, inclusive, in G.*

One useful testing criterion is to start the software in some state (that is, a node in the finite state machine) and then follow transitions (that is, edges) so that the last state is the same as the start state. This type of testing is used to verify that the system is not changed by certain inputs. Shortly we will formalize this notion as round trip coverage.

Before defining round trip coverage, we need a few more definitions. A path from n_i to n_j is *simple* if no node appears more than once in the path, with the exception that the first and last nodes may be identical. That is, simple paths have no internal loops, although the entire path itself may wind up being a loop. One useful aspect of simple paths is that any path can be created by composing simple paths.

Even fairly small programs may have a very large number of simple paths. Most of these simple paths are not worth addressing explicitly since they are subpaths of other simple paths. For a coverage criterion for simple paths we would like to avoid enumerating the entire set of simple paths. To this end we list only maximal length simple paths. To clarify this notion, we introduce a formal definition for a maximal length simple path, which we call a *prime path*, and we adopt the name "prime" for the criterion:

Definition 7.5 Prime Path: *A path from n_i to n_j is a prime path if it is a simple path and it does not appear as a proper subpath of any other simple path.*

CRITERION **7.4 Prime Path Coverage (PPC):** *TR contains each prime path in G.*

While this definition of prime path coverage has the practical advantage of keeping the number of test requirements down, it suffers from the problem that a given infeasible prime path may well incorporate many feasible simple paths. The solution is direct: replace the infeasible prime path with relevant feasible subpaths. For simplicity, we leave this replacement out of the definitions, but assume it when discussing prime path coverage later.

Prime path coverage has two special cases that we include below for historical reasons. From a practical perspective, it is usually better simply to adopt prime path coverage. Both special cases involve treatment of loops with "round trips."

A *round trip* path is a prime path of nonzero length that starts and ends at the same node. One type of round trip test coverage requires at least one round trip path to be taken for each node, and another requires all possible round trip paths.

CRITERION **7.5 Simple Round Trip Coverage (SRTC):** *TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.*

CRITERION **7.6 Complete Round Trip Coverage (CRTC):** *TR contains all round-trip paths for each reachable node in G.*

Next we turn to path coverage, which is traditional in the testing literature.

CRITERION **7.7 Complete Path Coverage (CPC):** *TR contains all paths in G.*

Sadly, Complete Path Coverage is useless if a graph has a cycle, since this results in an infinite number of paths, and hence an infinite number of test requirements. A variant of this criterion is, however, useful. Suppose that instead of requiring all paths, we consider a specified set of paths. For example, these paths might be given by a customer in the form of usage scenarios.

CRITERION **7.8 Specified Path Coverage (SPC):** *TR contains a set S of test paths, where S is supplied as a parameter.*

Complete Path Coverage is not feasible for graphs with cycles, hence the reason for developing the other alternatives listed above. Figure 7.7 contrasts Prime Path Coverage with Complete Path Coverage. Figure 7.7(a) shows the "diamond" graph, which contains no loops. Both Complete Path Coverage and Prime Path Coverage can be satisfied on this graph with the two paths shown. Figure 7.7(b), however, includes a loop from 2 to 4 to 5 to 2, thus the graph has an infinite number of possible test paths, and Complete Path Coverage is not possible. The requirements for Prime Path Coverage, however, can be toured with two test paths, for example, [1, 2, 3] and [1, 2, 4, 5, 2, 4, 5, 2, 3].

7.2.2 Touring, Sidetrips, and Detours

An important but subtle point to note is that while simple paths do not have internal loops, we do **not** require the test paths that tour a simple path to have this property. That is, we distinguish between the path that **specifies** a test requirement and the portion of the test path that **meets** the requirement. The advantage of separating these two notions has to do with the issue of infeasible test requirements. Before describing this advantage, let us refine the notion of a tour.

Testing researchers have come up with many schemes to get around the problem of loops introducing an infinite number of paths. These range from the practical to the clever to the impractical to the hopeless. We introduce a subtle but elegant distinction that clarifies the problem and allows previous ideas to be folded together cleanly.

We previously defined "visits" and "tours," and recall that using a path p to tour a subpath [2, 3, 4] means that the subpath is a subpath of p. This is a rather strict definition because each node and edge in the subpath must be visited **exactly** in the order that they

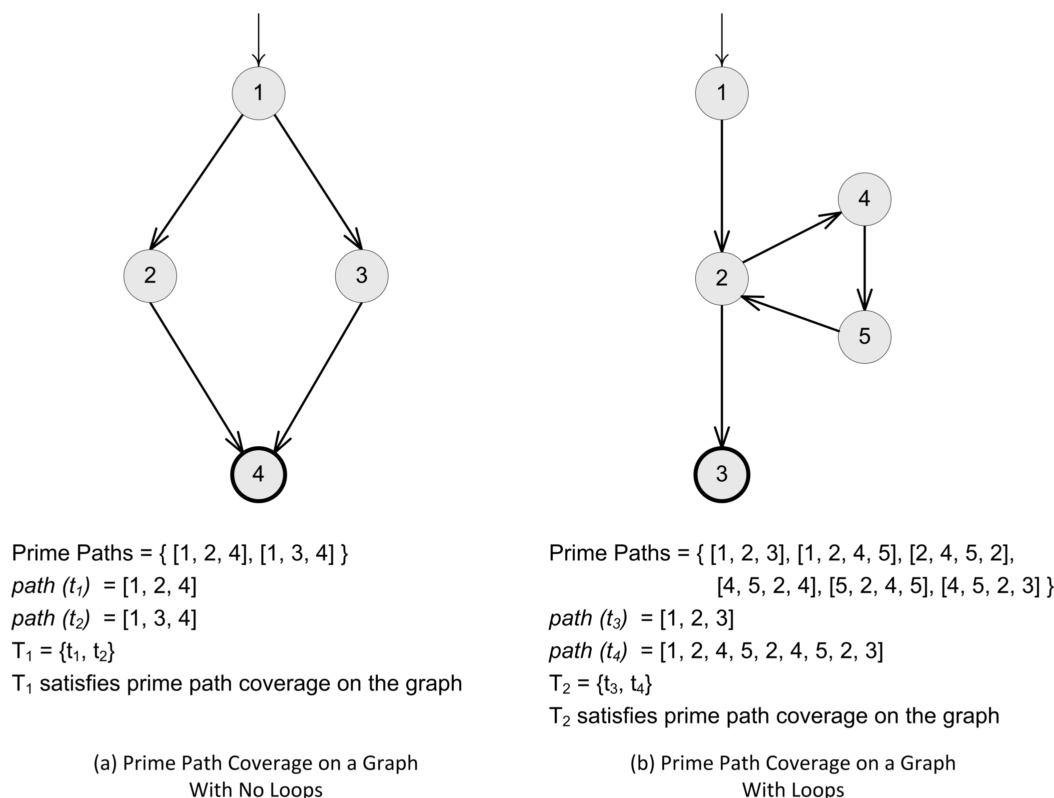


Figure 7.7: Two graphs showing prime path coverage.

appear in the subpath. We would like to relax this a bit to allow loops to be included in the tour. Consider the graph in Figure 7.8, which features a small loop from `b` to `c` and back.

If we are required to tour subpath $q = [2, 3, 5]$, the strict definition of tour prohibits us from meeting the requirement with any path that contains `4`, such as $p = [1, 2, 3, 4, 3, 5, 6]$, because we do not visit `2`, `3`, and `5` in exactly the same order. We relax the tour definition in two ways. The first allows the tour to include "sidetrips," where we can leave the path temporarily from a node and then return to the same node. The second allows the tour to include more general "detours" where we can leave the path from a node and then return to the **next** node on the path (skipping an edge). In the following definitions, q is a

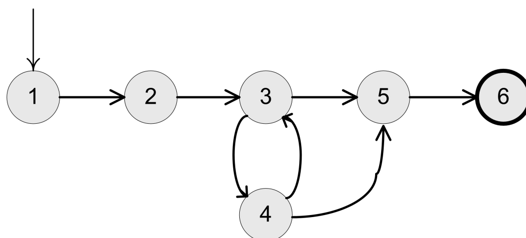


Figure 7.8: Graph with a loop.

simple subpath that is required.

Definition 7.6 Tour: *Test path p is said to tour subpath q if and only if q is a subpath of p.*

Definition 7.7 Tour With Sidetrips: *Test path p is said to tour subpath q with sidetrips if and only if every* **edge** *in q is also in p in the same order.*

Definition 7.8 Tour With Detours: *Test path p is said to tour subpath q with detours if and only if every* **node** *in q is also in p in the same order.*

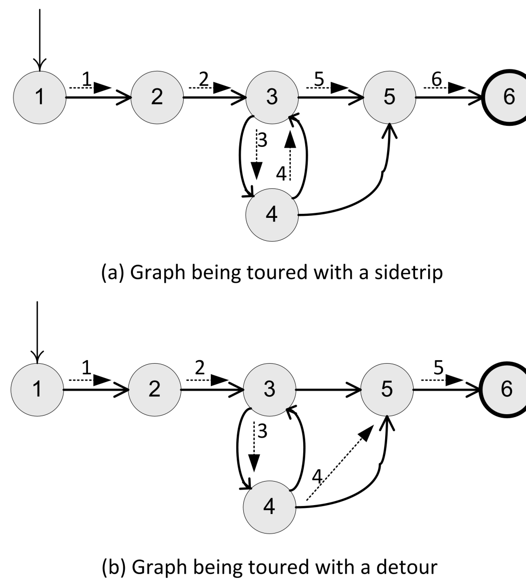


Figure 7.9: Tours, sidetrips, and detours in graph coverage.

The graphs in Figure 7.9 illustrate sidetrips and detours on the graph from Figure 7.8. In Figure 7.9(a), the dashed lines show the sequence of edges that are executed in a tour with a sidetrip. The numbers on the dashed lines indicate the order in which the edges are executed. In Figure 7.9(b), the dashed lines show the sequence of edges that are executed in a tour with a detour.

While these differences are rather small, they have far-reaching consequences. The difference between sidetrips and detours can be seen in Figure 7.9. The subpath [3, 4, 3] is a **sidetrip** to [2, 3, 5] because it leaves the subpath at node 3 and then returns to the subpath at node 3. Thus, every edge in the subpath [2, 3, 5] is executed in the same order. The subpath [3, 4, 5] is a **detour** to [2, 3, 5] because it leaves the subpath at node 3 and then returns to a node in the subpath at a later point, bypassing the edge (3, 5). That is, every node [2, 3, 5] is executed in the same order but every edge is not. Detours have the potential to drastically change the behavior of the intended test. That is, a test that takes

the edge (4, 5) may exhibit different behavior and test different aspects of the program than a test that takes the edge (3, 5).

To use the notion of sidetrips and detours, one can "decorate" each appropriate graph coverage criterion with a choice of touring. For example, Prime Path Coverage could be defined strictly in terms of tours, less strictly to allow sidetrips, or even less strictly to allow detours.

The position taken in this book is that sidetrips are a practical way to deal with infeasible test requirements, as described below. Hence we include them explicitly in our criteria. Detours seem less practical, and so we do not include them further.

Dealing with Infeasible Test Requirements

If sidetrips are not allowed, a large number of infeasible requirements can exist. Consider again the graph in Figure 7.9. In many programs it will be impossible to take the path from a to d without going through node c at least once because, for example, the loop body is written such that it cannot be skipped. If this happens, we need to allow sidetrips. That is, it may not be possible to tour the path $[a, b, d]$ without a sidetrip.

The argument above suggests dropping the strict notion of touring and simply allowing test requirements to be met with sidetrips. However, this is not always a good idea! Specifically, if a test requirement can be met without a sidetrip, then doing so may be superior to meeting the requirement with a sidetrip. Consider the loop example again. If the loop can be executed zero times, then the path $[a, b, d]$ should be toured without a sidetrip.

The argument above suggests a hybrid treatment with desirable practical and theoretical properties. The idea is to meet test requirements first with strict tours, and then allow sidetrips for unmet test requirements. Clearly, the argument could easily be extended to detours, but, as mentioned above, we elect not to do so.

Definition 7.9 Best Effort Touring: *Let TR_{tour} be the subset of test requirements that can be toured and $TR_{sidetrip}$ be the subset of test requirements that can be toured with sidetrips. Note that $TR_{tour} \subseteq TR_{sidetrip}$. A set T of test paths achieves best effort touring if for every path p in TR_{tour}, some path in T tours p directly and for every path p in $TR_{sidetrip}$, some path in T tours p either directly or with a sidetrip.*

Best Effort Touring has the practical benefit that as many test requirements are met as possible, yet each test requirement is met in the strictest possible way. As we will see in Section 7.2.4 on subsumption, Best Effort Touring has desirable theoretical properties with respect to subsumption.

Finding Prime Test Paths

It turns out to be relatively simple to find all prime paths in a graph, and test paths to tour the prime paths can be constructed automatically. The book website contains a graph coverage web application tool that will compute prime paths (and other criteria) on general graphs. We illustrate the process with the example graph in Figure 7.10. It has seven nodes

and nine edges, including a loop and an edge from node 5 to itself (sometimes called a "self-loop.")

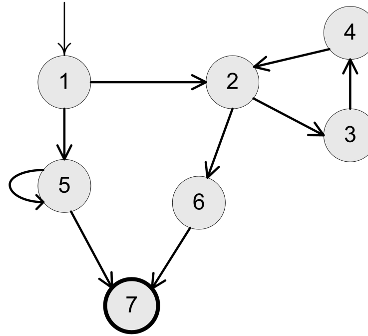


Figure 7.10: An example for prime test paths.

Prime paths can be found by starting with paths of length 0, then extending to length 1, and so on. Such an algorithm collects all simple paths, whether prime or not. The prime paths can then be filtered from this set. The set of paths of length 0 is simply the set of nodes, and the set of paths of length 1 is simply the set of edges. For simplicity, we list the node numbers in this example.

Simple paths of length 0 (7):
1) [1]
2) [2]
3) [3]
4) [4]
5) [5]
6) [6]
7) [7] !

The exclamation point on the path [7] tells us that this path cannot be extended. Specifically, the final node 7 has no outgoing edges, and so paths that end with 7 are not extended further. The simple paths of length 1 are computed by adding the successor nodes for each edge that starts with the last node in each simple path of length 0.

Simple paths of length 1 (9):
8) [1, 2]
9) [1, 5]
10) [2, 3]
11) [2, 6]
12) [3, 4]
13) [4, 2]
14) [5, 5] *
15) [5, 7] !
16) [6, 7] !

The asterisk on the path [5, 5] tells us that path can go no further because the first node

is the same as the last (it is already a cycle). For paths of length 2, we identify each path of length 1 that is not a cycle or ends in a node that has no outgoing edges. We then extend the path with every node that can be reached from the last node in the path unless that node is already in the path and not the first node. The first path of length 1, [1, 2], is extended to [1, 2, 3] and [1, 2, 6]. The second, [1, 5], is extended to [1, 5, 7] but not [1, 5, 5], because node 5 is already in the path (that is, [1, 5, 5] is not simple and thus is not prime).

Simple paths of length 2 (8):
17) [1, 2, 3]
18) [1, 2, 6]
19) [1, 5, 7] !
20) [2, 3, 4]
21) [2, 6, 7] !
22) [3, 4, 2]
23) [4, 2 ,3]
24) [4, 2 ,6]

Paths of length 3 are computed in a similar way.

Simple paths of length 3 (7):
25) [1, 2, 3, 4] !
26) [1, 2, 6, 7] !
27) [2, 3, 4, 2] *
28) [3, 4, 2, 3] *
29) [3, 4, 2, 6]
30) [4, 2 ,3, 4] *
31) [4, 2 ,6, 7] !

Finally, only one path of length 4 exists. Three paths of length 3 cannot be extended because they are cycles; two others end with node 7. Of the remaining two, the path that ends in node 4 cannot be extended because [1, 2, 3, 4, 2] is **not** simple and thus is not prime.

Simple path of length 4 (1):
32) [3, 4, 2, 6, 7]!

The prime paths can be computed by eliminating any path that is a (proper) subpath of some other simple path. Note that every simple path without an exclamation mark or asterisk is eliminated as it can be extended and is thus a proper subpath of some other simple path. The graph in Figure 7.10 has eight prime paths:

14) [5, 5] *
19) [1, 5, 7] !
25) [1, 2, 3, 4] !
26) [1, 2, 6, 7] !

27) [2, 3, 4, 2] *
28) [3, 4, 2, 3] *
30) [4, 2 ,3, 4] *
32) [3, 4, 2, 6, 7]!

This process is guaranteed to terminate because the length of the longest possible prime path is the number of nodes. Although graphs often have many simple paths (32 in this example, of which 8 are prime), they can usually be toured with far fewer test paths. Many possible algorithms can find test paths to tour the prime paths, two of which are implemented in the graph coverage web application on the book website. We can do this by hand with the graph in Figure 7.10. For example, it can be seen that the four test paths [1, 2, 6, 7], [1, 2, 3, 4, 2, 3, 4, 2, 6, 7], [1, 5, 7], and [1, 5, 5, 7] are enough. This approach, however, is error-prone. The easiest thing to do is to tour the loop [2, 3, 4] only once, which omits the prime paths [3, 4, 2, 3] and [4, 2, 3, 4].

With more complicated graphs, a mechanical approach is needed. By hand, we recommend starting with the longest prime paths and extending them to the beginning and end nodes in the graph. For our example, this results in the test path [1, 2, 3, 4, 2, 6, 7]. The test path [1, 2, 3, 4, 2, 6, 7] tours 3 prime paths: 25, 27, and 32.

The next test path is constructed by extending one of the longest remaining prime paths; we will continue to work backward and choose 30. The resulting test path is [1, 2, 3, 4, 2, 3, 4, 2, 6, 7], which tours 2 prime paths, 28 and 30 (it also tours paths 25 and 27).

The next test path is constructed by using the prime path 26 [1, 2, 6, 7]. This test path tours only maximal prime path 26.

Continuing in this fashion yields two more test paths, [1, 5, 7] for prime path 19, and [1, 5, 5, 7] for prime path 14. The complete set of test paths is then:

1) [1, 2, 3, 4, 2, 6, 7]
2) [1, 2, 3, 4, 2 ,3, 4, 2, 6, 7]
3) [1, 2, 6, 7]
4) [1, 5, 7]
5) [1, 5, 5, 7]

This can be used as is, or optimized if the tester desires a smaller test set. It is clear that test path 2 tours the prime paths toured by test path 1, so 1 can be eliminated, leaving the four test paths identified informally earlier in this section. Simple algorithms such as implemented in the graph covreage web application on the book website can automate this process.

Exercises, Section 7.2.2.

1. Redefine *Edge Coverage* in the standard way (see the discussion for *Node Coverage*).

2. Redefine *Complete Path Coverage* in the standard way (see the discussion for *Node Coverage*).

3. Subsumption has a significant weakness. Suppose criterion C_{strong} subsumes criterion C_{weak} and that test set T_{strong} satisfies C_{strong} and test set T_{weak} satisfies C_{weak}. It is not necessarily the case that T_{weak} is a subset of T_{strong}. It is also not necessarily the case that T_{strong} reveals a fault if T_{weak} reveals a fault. Explain these facts.

4. Answer questions a-d for the graph defined by the following sets:
   - $N = \{1,\ 2,\ 3,\ 4\}$
   - $N_0 = \{1\}$
   - $N_f = \{4\}$
   - $E = \{(1,2),\ (2,3),\ (3,2),\ (2,4)\}$

   (a) Draw the graph.
   (b) List test paths that achieve Node Coverage, but not Edge Coverage.
   (c) List test paths that achieve Edge Coverage, but not Edge Pair Coverage.
   (d) List test paths that achieve Edge Pair Coverage.

5. Answer questions a-g for the graph defined by the following sets:
   - $N = \{1,2,3,4,5,6,7\}$
   - $N_0 = \{1\}$
   - $N_f = \{7\}$
   - $E = \{(1,2),(1,7),(2,3),(2,4),(3,2),(4,5),(4,6),(5,6),(6,1)\}$

   Also consider the following (candidate) test paths:

   - $t0 = [1,\ 2,\ 4,\ 5,\ 6,\ 1,\ 7]$
   - $t1 = [1,\ 2,\ 3,\ 2,\ 4,\ 6,\ 1,\ 7]$

   (a) Draw the graph.
   (b) List the test requirements for Edge-Pair Coverage. (Hint: You should get 12 requirements of length 2).
   (c) Does the given set of test paths satisfy Edge-Pair Coverage? If not, identify what is missing.
   (d) Consider the simple path [**3, 2, 4, 5, 6**] and test path [**1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7**]. Does the test path tour the simple path directly? With a sidetrip? If so, identify the sidetrip.
   (e) List the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage on the graph.
   (f) List test paths that achieve Node Coverage but not Edge Coverage on the graph.
   (g) List test paths that achieve Edge Coverage but not Prime Path Coverage on the graph.

6. Answer questions a-c for the graph in Figure 7.2.

   (a) Enumerate the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage on the graph.
   (b) List test paths that achieve Node Coverage but not Edge Coverage on the graph.
   (c) List test paths that achieve Edge Coverage but not Prime Path Coverage on the graph.

7. Answer questions a-d for the graph defined by the following sets:
   - $N = \{1,\ 2,\ 3\}$
   - $N_0 = \{1\}$

- $N_f = \{3\}$
- $E = \{(1,2),\ (1,3),\ (2,1),\ (2,3),\ (3,1)\}$

Also consider the following (candidate) paths:

- $p_0 = [1,\ 2,\ 3,\ 1]$
- $p_1 = [1,\ 3,\ 1,\ 2,\ 3]$
- $p_2 = [1,\ 2,\ 3,\ 1,\ 2,\ 1,\ 3]$
- $p_3 = [2,\ 3,\ 1,\ 3]$
- $p_4 = [1,\ 2,\ 3,\ 2,\ 3]$

(a) Which of the listed paths are test paths? Explain the problem with any path that is not a test path.
(b) List the eight test requirements for Edge-Pair Coverage (only the length two subpaths).
(c) Does the set of **test** paths (part a) above satisfy Edge-Pair Coverage? If not, identify what is missing.
(d) Consider the prime path [3, 1, 3] and path p_2. Does p_2 tour the prime path directly? With a sidetrip?

8. Design and implement a program that will compute all prime paths in a graph, then derive test paths to tour the prime paths. Although the user interface can be arbitrarily complicated, the simplest version will be to accept a graph as input by reading a list of nodes, initial nodes, final nodes, and edges.

7.2.3 Data Flow Criteria

Meta Discussion

We debated whether to include data flow in the second edition. On the negative side, prime path coverage subsumes all the data flow criteria, and since PPC is simpler to understand and compute, some argue that the data flow criteria are now obsolete. Additionally, we are not aware of any companies who uses data flow criteria in practice. On the positive side, many educators believe that if a student learns testing, the student should know something about data flow coverage. It is also possible that a tester may want to use all-uses coverage without the additional expense of prime path coverage. Also, it is used in later sections of the book in situations where prime path coverage is not used. Additionally, it may be important for data flow programming languages. Finally, it is often the first software analysis technique, and is considered basic for more advanced analysis techniques such as symbolic execution and slicing.

After considering all factors, we decided to include data flow with this explicit note to instructors: It is possible to omit data flow entirely from a course on testing. The concepts in this subsection are used in sections 7.3.2 and 7.4.2. If you choose to cover any of those sections later, you will need section 7.2.3.

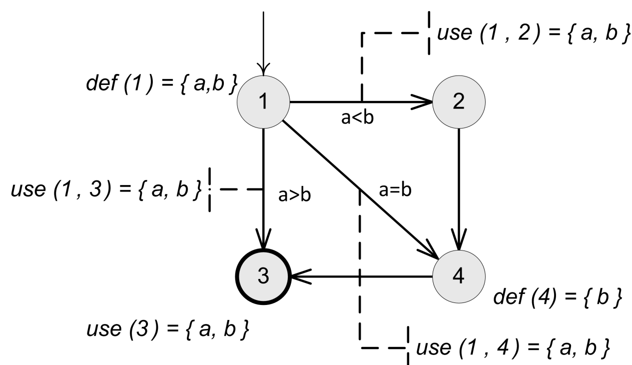


Figure 7.11: A graph showing variables, def sets and use sets.

The next few testing criteria are based on the assumption that to test a program adequately, we should focus on the flows of data values. Specifically, we should try to ensure that the values created at one point in the program are created and used correctly. This is done by focusing on definitions and uses of values. A *definition (def)* is a location where a value for a variable is stored into memory (assignment, input, etc.). A *use* is a location where a variable's value is accessed. Data flow testing criteria use the fact that values are carried from defs to uses. We call these *du-pairs* (they are also known as *definition-use*, *def-use*, and *du associations* in the testing literature). The idea of data flow criteria is to exercise du-pairs in various ways.

First we must integrate data flow into the existing graph model. Let V be a set of variables that are associated with the program artifact being modeled in the graph. Each node n and edge e is considered to define a subset of V; this set is called $def(n)$ or $def(e)$. (Although graphs from programs cannot have defs on edges, other software artifacts such as finite state machines allow defs as side-effects on edges.) Each node n and edge e is also considered to use a subset of V; this set is called $use(n)$ or $use(e)$.

Figure 7.11 gives an example of a graph annotated with defs and uses. All variables involved in a decision are assumed to be used on the associated edges, so a and b are in the use set of all three edges $(1,\ 2)$, $(1,\ 3)$, and $(1,\ 4)$.

An important concept when discussing data flow criteria is that a def of a variable may or may not reach a particular use. The most obvious reason that a def of a variable v at location l_i (a location could be a node or an edge) will not reach a use at location l_j is because no path goes from l_i to l_j. A more subtle reason is that the variable's value may be changed by another def before it reaches the use. Thus, a path from l_i to l_j is *def-clear* with respect to variable v if for every node n_k and every edge e_k on the path, $k \neq i$ and $k \neq j$, v is not in $def(n_k)$ or in $def(e_k)$. That is, no location between l_i and l_j changes the value. If a def-clear path goes from l_i to l_j with respect to v, we say that the def of v at l_i *reaches* the use at l_j.

For simplicity, we will refer to the start and end of a du-path as nodes, even if the definition or the use occurs on an edge. We discuss relaxing this convention later. Formally,

a *du-path* with respect to a variable v is a simple path that is def-clear with respect to v from a node n_i for which v is in $def(n_i)$ to a node n_j for which v is in $use(n_j)$. We want the paths to be simple to ensure a reasonably small number of paths. Note that a du-path is always associated with a specific variable v, a du-path always has to be simple, and there may be intervening uses on the path.

Figure 7.12 gives an example of a graph annotated with defs and uses. Rather than displaying the actual sets, we show the full program statements that are associated with the nodes and edges. This is common and often more informative to a human, but the actual sets are simpler for automated tools to process. Note that the parameters (*subject* and *pattern*) are considered to be *explicitly defined* by the first node in the graph. That is, the def set of node 1 is $def(1) = \{subject, pattern\}$. Also note that decisions in the program (for example, $if\ subject[iSub] == pattern[0]$) result in uses of the associated variables for both edges in the decision. That is, $use(4, 10) \equiv use(4, 5) \equiv \{subject, iSub, pattern\}$. The parameter *subject* is used at node 2 (with a reference to its *length* attribute) and at edges (4, 5), (4, 10), (7, 8), and (7, 9), thus du-paths exist from node 1 to node 2 and from node 1 to each of those four edges.

Figure 7.13 shows the same graph, but this time with the def and use sets explicitly marked on the graph.[3] Note that node 9 both defines and uses the variable $iPat$. This is because of the statement $iPat++$, which is equivalent to $iPat = iPat + 1$. In this case, the use occurs before the def, so for example, a def-clear path goes from node 5 to node 9 with respect to $iPat$.

The test criteria for data flow will be defined as sets of du-paths. This makes the criteria quite simple, but first we need to categorize the du-paths into several groups.

The first grouping of du-paths is according to definitions. Specifically, consider all of the du-paths with respect to a given variable defined in a given node. Let the *def-path* set $du(n_i, v)$ be the set of du-paths with respect to variable v that start at node n_i. Once we have clarified the notion of touring for dataflow coverage, we will define the All-defs criterion by simply asking that at least one du-path from each def-path set be toured. Because of the large number of nodes in a typical graph, and the potentially large number of variables defined at each node, the number of def-path sets can be quite large. Even so, the coverage criterion based on the def-path groupings tends to be quite weak.

Perhaps surprisingly, it is *not* helpful to group du-paths by uses, and so we will not provide a definition of "use-path" sets that parallels the definition of def-path sets given above.

The second, and more important, grouping of du-paths is according to pairs of definitions and uses. We call this the *def-pair* set. After all, the heart of data flow testing is allowing definitions to flow to uses. Specifically, consider all of the du-paths with respect to a given variable that are defined in one node and used in another (possibly identical) node. Formally, let the *def-pair* set $du(n_i, n_j, v)$ be the set of du-paths with respect to variable v that start at node n_i and end at node n_j. Informally, a def-pair set collects together all the (simple)

[3] The reader might wonder why `NOTFOUND` fails to appear in the set $use(2)$. The reason, as explained in Section 7.3.2 is that the use is *local*.

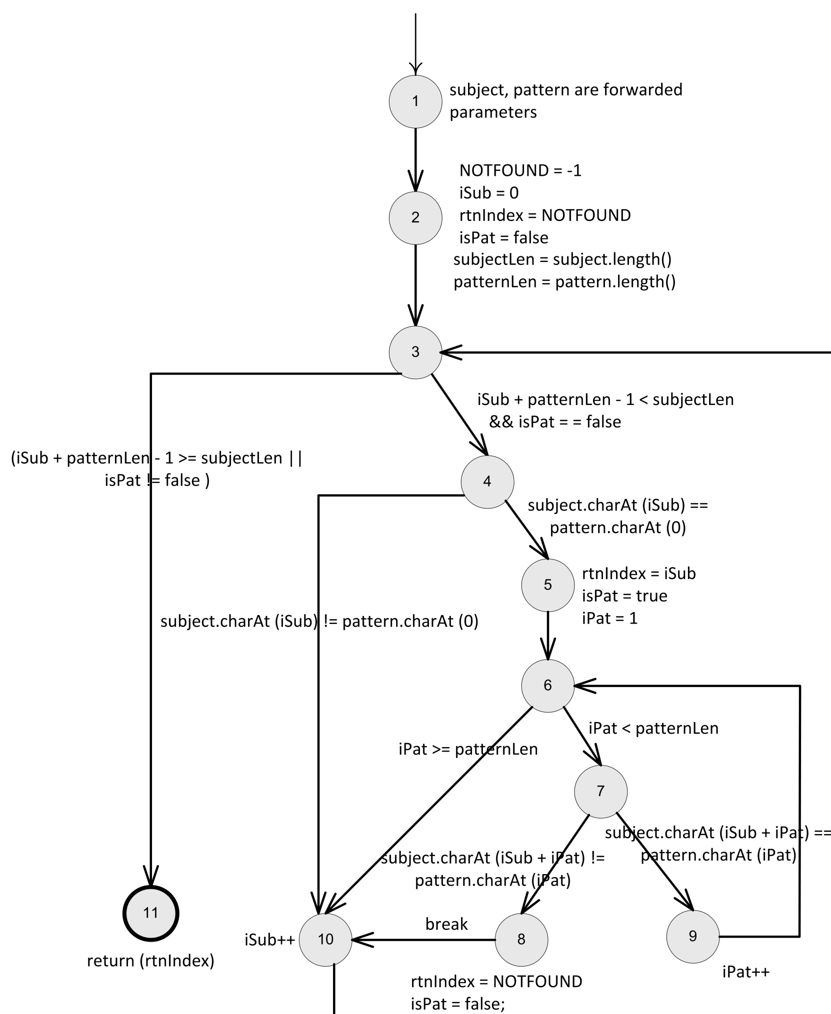


Figure 7.12: A graph showing an example of du-paths.

ways to get from a given definition to a given use. Once we have clarified the notion of touring for dataflow coverage, we will define the All-Uses criterion by simply asking that at least one du-path from each def-pair set be toured. Since each definition can typically reach multiple uses, there are usually many more def-path sets than def-pair sets.

In fact, the def-path set for a def at node n_i is the union of all the def-pair sets for that def. More formally: $du(n_i,\ v)\ =\ \cup_{n_j}\ du(n_i,\ n_j,\ v)$.

To illustrate the notions of def-path sets and def-pair sets, consider du-paths with respect to the variable *iSub*, which has one of its definitions in node 10 in Figure 7.13. There are du-paths with respect to *iSub* from node 10 to nodes 5 and 10, and to edges (3,4), (3,11), (4,5), (4, 10), (7, 8), and (7,9).

The def-path set for the use of *iSub* at node 10 is:

$du(10, iSub) = \{[10, 3, 4], [10, 3, 4, 5], [10, 3, 4, 5, 6, 7, 8], [10, 3, 4, 5, 6, 7, 9], [10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10], [10, 3, 11]\}$

This def-path set can be broken up into the following def-pair sets:
$du(10, 4, iSub) = \{[10, 3, 4]\}$
$du(10, 5, iSub) = \{[10, 3, 4, 5]\}$
$du(10, 8, iSub) = \{[10, 3, 4, 5, 6, 7, 8]\}$
$du(10, 9, iSub) = \{[10, 3, 4, 5, 6, 7, 9]\}$
$du(10, 10, iSub) = \{[10, 3, 4, 5, 6, 10],\ [10, 3, 4, 5, 6, 7, 8, 10],\ [10, 3, 4, 10]\}$
$du(10, 11, iSub) = \{[10, 3, 11]\}$

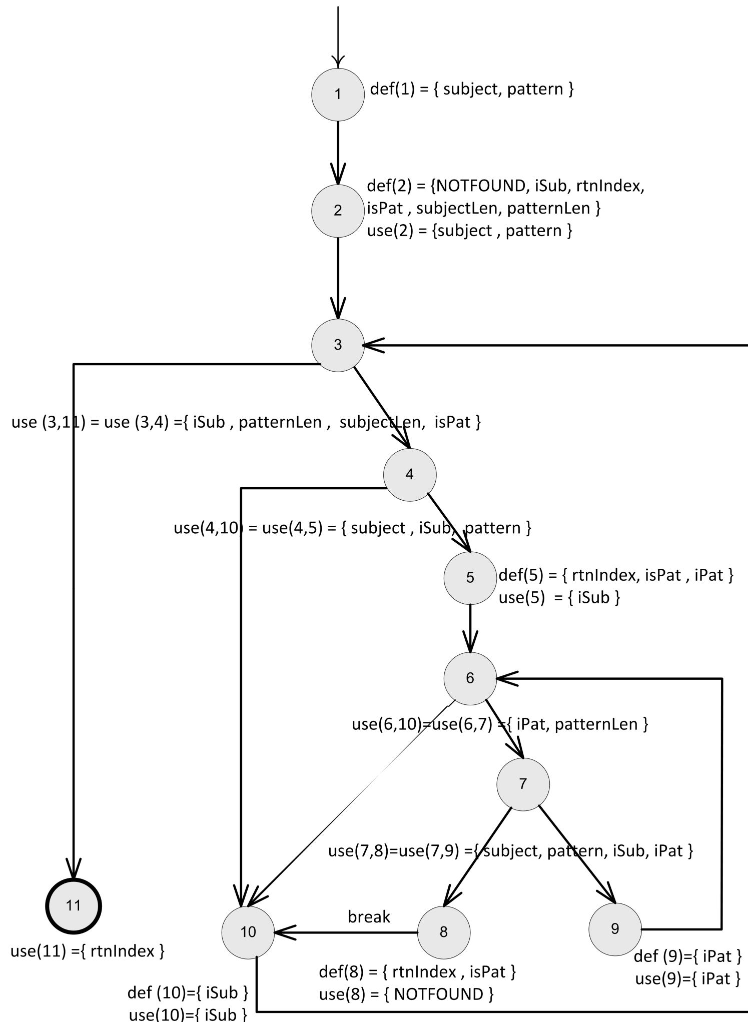


Figure 7.13: Graph showing explicit def and use sets.

Next, we extend the definition of *tour* to apply to du-paths. A test path p is said to *du tour* subpath d with respect to v if p tours d and the portion of p to which d corresponds is def-clear with respect to v. It is possible to allow or disallow def-clear sidetrips with respect to v when touring a du-path. Because def-clear sidetrips make it possible to tour more du-paths, we define the dataflow coverage criteria given below to allow sidetrips where

necessary.

Now we can define the primary data flow coverage criteria. The three most common are best understood informally. The first requires that each def reaches **at least one use**, the second requires that each def reaches **all possible uses**, and the third requires that each def reaches all possible uses through **all possible du-paths**. As mentioned in the development of def-path sets and def-pair sets, the formal definitions of the criteria are simply appropriate selections from the appropriate set. For each test criterion below, we assume Best Effort Touring (see Section 7.2.2), where sidetrips are required to be def-clear with respect to the variable in question.

CRITERION **7.9 All-Defs Coverage (ADC):** *For each def-path set* $S = du(n, v)$, TR *contains at least one path* d *in* S.

Remember that the def-path set $du(n, v)$ represents all def-clear simple paths from n to all uses of v. So All-Defs requires us to tour at least one path to at least one use.

CRITERION **7.10 All-Uses Coverage (AUC):** *For each def-pair set* $S = du(n_i, n_j, v)$, TR *contains at least one path* d *in* S.

Remember that the def-pair set $du(n_i, n_j, v)$ represents all the def-clear simple paths from a def of v at n_i to a use of v at n_j. So All-Uses requires us to tour at least one path for every def-use pair[4].

CRITERION **7.11 All-du-Paths Coverage (ADUPC):** *For each def-pair set* $S = du(n_i, n_j, v)$, TR *contains every path* d *in* S.

The definition could also simply be written as "include every du-path." We chose the given formulation because it highlights that the key difference between All-Uses and All-du-Paths is a change in quantifier. Specifically, the "at least one du-path" directive in All-Uses is changed to "every path" in All-du-Paths. Thought of in terms of def-use pairs, All-Uses requires *some* def-clear simple path to each use, whereas All-du-Paths requires *all* def-clear simple paths to each use.

To simplify the development above, we assumed that definitions and uses occurred on nodes. Naturally, definitions and uses can occur on edges as well. It turns out that the development above also works for uses on edges, so data flow on program flow graphs can be easily defined (uses on program flow graph edges are sometimes called "p-uses"). However,

[4]Despite the names of the criteria, All-Defs and All-Uses treat definitions and uses differently. Specifically, replacing the term "def" with "use" in All-Defs does *not* result in All-Uses. While All-Defs focuses on definitions, All-Uses focuses on def-use *pairs*. While the naming convention might be misleading, and that a name such as "All-Pairs" might be clearer than All-Uses, we use the standard usage from the dataflow literature.

the development above does not work if the graph has definitions on edges. The problem is that a du-path from an edge to an edge is no longer necessarily simple, since instead of simply having a common first and last *node*, such a du-path now might have a common first and last *edge*. It is possible to modify the definitions to explicitly mention definitions and uses on edges as well as nodes, but the definitions tend to get messier. The bibliographic notes contain pointers for this type of development.

Figure 7.14 illustrates the differences among the three data flow coverage criteria with the double-diamond graph. The graph has one def, so only one path is needed to satisfy all-defs. The def has two uses, so two paths are needed to satisfy all-uses. Since two paths go from the def to each use, four paths are needed to satisfy all-du-paths. Note that the definitions of the data flow criteria leave open the choice of touring. The literature uses various choices–in some cases demanding direct touring, and, in other cases, allowing def-clear sidetrips. Our recommendation is Best Effort Touring, a choice that, in contrast to the treatments in the literature, yields the desired subsumption relationships even in the case of infeasible test requirements. From a practical perspective, Best Effort Touring also makes sense–each test requirement is satisfied as rigorously as possible.

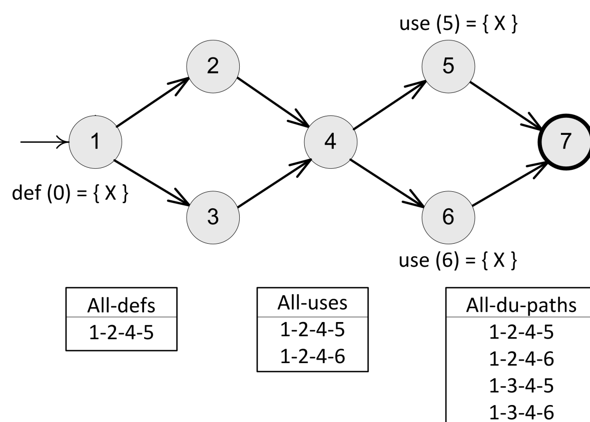


Figure 7.14: Example of the differences among the three data flow coverage criteria.

Exercises, Section 7.2.3.

1. Below are four graphs, each of which is defined by the sets of nodes, initial nodes, final nodes, edges, and defs and uses. Each graph also contains a collection of test paths. Answer the following questions about each graph.

Graph I.
$N = \{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 2\}$
$N_0 = \{1\}$
$N_f = \{7\}$
$E = \{(1,2),\ (2,3),\ (2,8),\ (3,4),\ (3,5),\ (4,3),$
$(5,6),\ (5,7),\ (6,7),\ (7,2)\}$
$def(1) = def(4) = use(5) = use(8) = \{x\}$
Test Paths:
$t1 = [1,\ 2,\ 8]$
$t2 = [1,\ 2,\ 3,\ 5,\ 7,\ 2,\ 8]$
$t3 = [1,\ 2,\ 3,\ 5,\ 6,\ 7,\ 2,\ 8]$
$t4 = [1,\ 2,\ 3,\ 4,\ 3,\ 5,\ 7,\ 2,\ 8]$
$t5 = [1,\ 2,\ 3,\ 4,\ 3,\ 4,\ 3,\ 5,\ 6,\ 7,\ 2,\ 8]$
$t6 = [1,\ 2,\ 3,\ 4,\ 3,\ 5,\ 7,\ 2,\ 3,\ 5,\ 6,\ 7,\ 2,\ 8]$

Graph II.
$N = \{1,\ 2,\ 3,\ 4,\ 5,\ 6\}$
$N_0 = \{1\}$
$N_f = \{6\}$
$E = \{(1,2),\ (2,3),\ (2,6),\ (3,4),\ (3,5),\ (4,5),\ (5,2)\}$
$def(1) = def(2) = use(5) = use(6) = \{x\}$
// Assume the use of x in 4 precedes the def
Test Paths:
$t1 = [1,\ 2,\ 6]$
$t2 = [1,\ 2,\ 3,\ 4,\ 5,\ 2,\ 3,\ 5,\ 2,\ 6]$
$t3 = [1,\ 2,\ 3,\ 5,\ 2,\ 3,\ 4,\ 5,\ 2,\ 6]$
$t4 = [1,\ 2,\ 3,\ 5,\ 2,\ 6]$

Graph III.
$N = \{1,\ 2,\ 3,\ 4,\ 5,\ 6\}$
$N_0 = \{1\}$
$E = \{(1,2),\ (2,3),\ (3,4),\ (3,5),\ (4,5),\ (5,2),\ (2,6)\}$
$def(1) = def(4) = use(3) = use(5) = use(6) = \{x\}$

Test Paths:
$t_1 = [1,\ 2,\ 3,\ 5,\ 2,\ 6]$
$t_2 = [1,\ 2,\ 3,\ 4,\ 5,\ 2,\ 6]$

Graph IV.
$N = \{1,\ 2,\ 3,\ 4,\ 5,\ 6\}$
$N_0 = \{1\}$
$N_f = \{6\}$
$E = \{(1,2),\ (2,3),\ (2,6),\ (3,4),\ (3,5),\ (4,5),\ (5,2)\}$
$def(1) = def(5) = use(5) = use(6) = \{x\}$
// Assume the use of x in 5 precedes the def
Test Paths:
$t1 = [1,\ 2,\ 6]$
$t2 = [1,\ 2,\ 3,\ 4,\ 5,\ 2,\ 3,\ 5,\ 2,\ 6]$
$t3 = [1,\ 2,\ 3,\ 5,\ 2,\ 3,\ 4,\ 5,\ 2,\ 6]$

(a) Draw the graph.
(b) List all of the du-paths with respect to x. (Note: Include all du-paths, even those that are subpaths of some other du-path).
(c) For each test path, determine which du-paths that test path tours. For this part of the exercise, you should consider both direct touring and sidetrips. Hint: A table is a convenient format for describing this relationship.
(d) List a minimal test set that satisfies *all defs* coverage with respect to x. (Direct tours only.) If possible, use the given test paths. If not, provide additional test paths to satisfy the criterion.
(e) List a minimal test set that satisfies *all uses* coverage with respect to x. (Direct tours only.) If possible, use the given test paths. If not, provide additional test paths to satisfy the criterion.
(f) List a minimal test set that satisfies *all du-paths* coverage with respect to x. (Direct tours only.) If possible, use the given test paths. If not, provide additional test paths to satisfy the criterion.

7.2.4 Subsumption Relationships among Graph Coverage Criteria

Recall from Chapter 1 that coverage criteria are often related to one another by *subsumption*. The first relation to note is that Edge Coverage subsumes Node Coverage. In most cases, this is because if we traverse every edge in a graph, we will visit every node. However, if a graph has a node with no incoming or outgoing edges, traversing every edge will not reach that node. Thus, Edge Coverage is defined to include every path of length **up to 1**, that is, of length 0 (all nodes) and length 1 (all edges). The subsumption does not hold in the reverse

direction. Recall that Figure 7.6 gave an example test set that satisfied Node Coverage but not Edge Coverage. Hence, Node Coverage does not subsume Edge Coverage.

It may seem surprising that Prime Path Coverage does not subsume Edge-Pair Coverage. In most situations, it does. The exception is when a node n has a self-loop, the subpath from its predecessor m creates the edge-pair $[m, n, n]$, which of course, is not a prime path. We have several other subsumption relations among the criteria. Where applicable, the structural coverage relations assume Best-Effort Touring. Because Best-Effort touring is assumed, the subsumption results hold even if some test requirements are infeasible.

The subsumption results for data flow criteria are based on three assumptions: (1) every use is preceded by a def, (2) every def reaches at least one use, and (3) for every node with multiple outgoing edges, at least one variable is used on each out edge, and the same variables are used on each out edge. If we satisfy All-Uses Coverage, then we will have implicitly ensured that every def was used. Thus All-Defs is also satisfied and All-Uses subsumes All-Defs. Likewise, if we satisfy All-du-Paths Coverage, then we will have implicitly ensured that every def reached every possible use. Thus All-Uses is also satisfied and All-du-Paths subsumes All-Uses. Additionally, each edge is based on the satisfaction of some predicate, so each edge has at least one use. Therefore All-Uses will guarantee that each edge is executed at least once, so All-Uses subsumes Edge Coverage.

Finally, each du-path is also a simple path, so Prime Path Coverage subsumes All-du-Paths Coverage.[5] This is a significant observation, since computing prime paths is considerably simpler than analyzing data flow relationships. Figure 7.15 shows the subsumption relationships among the structural and data flow coverage criteria.

7.3 Graph Coverage for Source Code

Most of the graph coverage criteria were developed for source code, and these definitions match the definitions in Section 7.2 very closely. As in Section 7.2, we first consider structural coverage criteria and then data flow criteria.

7.3.1 Structural Graph Coverage for Source Code

The most widely used graph coverage criteria are defined on source code. Although precise details vary from one programming language to another, the basic pattern is the same for most common languages. To apply one of the graph criteria, the first step is to define the graph, and for source code, the most common graph is a *control flow graph (CFG)*. Control flow graphs associate an edge with each possible branch in the program, and a node

[5] This is a bit of an overstatement, and, as usual, the culprit is infeasibility. Specifically, consider a du-path with respect to variable x that can only be toured with a sidetrip. Further, suppose that there are two possible sidetrips, one of which is def-clear with respect to x, and one of which is not. The relevant test path from the All du-Paths test set necessarily tours the former sidetrip, where as the corresponding test path from the Prime Path test set is free to tour the latter side trip. Our opinion is that in most situations it is reasonable for the test engineer to ignore this special case and simply proceed with Prime Path Coverage.

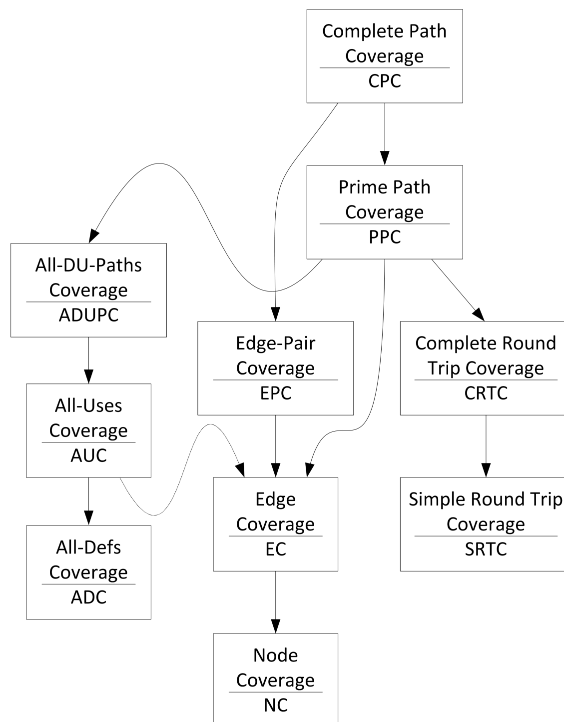


Figure 7.15: Subsumption relations among graph coverage criteria.

with sequences of statements. Formally, a *basic block* is a maximum sequence of program statements such that if any one statement of the block is executed, all statements in the block are executed. A basic block has only one entry point and one exit point. Our first example language structure is an `if` statement with an `else` clause, shown as Java code followed by the corresponding CFG in Figure 7.16. The `if-else` structure results in two basic blocks.

Note that the two statements in the `then` part of the `if` statement both appear in the same node. Node 1, which represents the conditional test `x < y` has two out-edges, and is called a *decision* node. Node 4, which has more than one in-edge, is called a *junction* node.

Next we turn to the degenerate case of an `if` statement without an `else` clause, shown in Figure 7.17. This is the same graph previously seen in Figure 7.6, but this time based on actual program statements. The control flow graph for this structure has only three nodes. The reader should note that a test with $x < y$ traverses all of the nodes in this control flow graph, but not all of the edges.

The graph changes if the loop body contains a `return` statement. Figure 7.18 shows this example. Nodes 2 and 3 are final nodes, and there is no edge from node 2 to node 3.

Representing loops is a little tricky because we have to include nodes that are not directly derived from program statements. The simplest kind of loop is a `while` loop with an initializing statement, as shown in Figure 7.19. (Assume that `y` has a value at this point in the program.)

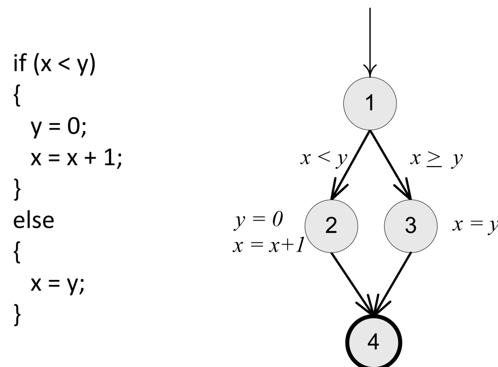


Figure 7.16: CFG fragment for the **if-else** structure.

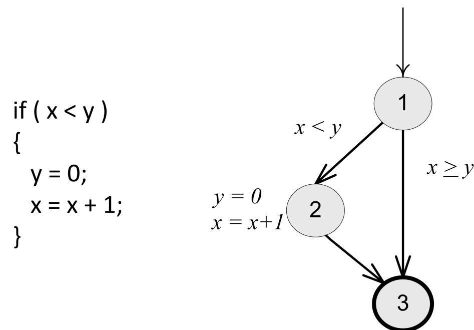


Figure 7.17: CFG fragment for the **if** structure without an **else**.

The graph for the `while` structure has a decision node, which is needed for the conditional test, and a single node for the body of the `while` loop. Node 2 is sometimes called a "dummy node," because it does not represent any statements, but gives the iteration edge (3, 2) somewhere to go. Node 2 can also be thought of as representing a decision. A common mistake for beginners is to try to have the edge go to 1; this is not correct because that would mean the initialization step is done each iteration of the loop. Note that the method call `f(x,y)` is not expanded in this particular graph; we return to this issue later.

Now, consider a `for` loop. The example in Figure 7.20 behaves equivalently to the prior `while` loop. The graph becomes a little more complicated, essentially because the `for` structure is at a very high level of abstraction.

Although the initialization, test, and increment of the loop control variable `x` are all on the same line in the program, they need to be associated with different nodes in the graph. The control flow graph for the `for` loop is slightly different from that of the `while` loop. Specifically, we show the increment of `x` in a different node than the method call `y = f(x,y)`. Technically speaking, this violates the definition of a basic block and the two nodes should be combined, but it is often easier to develop templates for the various possible program structures and then plug the control flow graph for the relevant code into the correct spot in the template. Commercial tools typically do this to make the graph

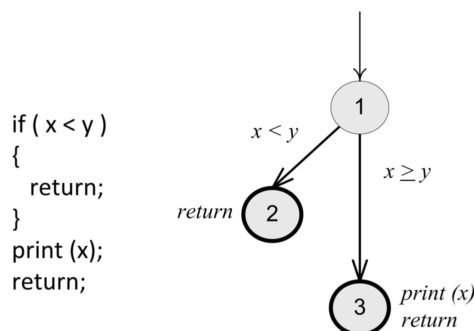


Figure 7.18: CFG fragment for the **if** structure with a **return**.

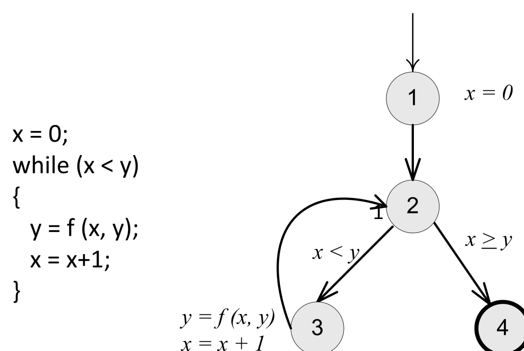


Figure 7.19: CFG fragment for the **while** loop structure.

generation simpler. In fact, commercial tools often do not follow the strict definition of the basic block and sometimes add seemingly random nodes. This can have trivial effects on the bookkeeping (for example, we might cover 67 of 73 instead of 68 of 75), but is not really important for testing.

The `do-while` loop is similar, but simpler. The loop body is always executed at least once, so the statements are associated with node 2 in Figure 7.21.

Figure 7.22 shows how we handle `break` and `continue` statements in `while` loops. If the `break` statement at node 4 is reached, control immediately transfers out of the loop to node 8. If the `continue` statement at node 6 is reached, control transfers to the next iteration of the loop at node 2, without going through the statement after the `if` test (node 7).

The next language structure is the `case` statement, or `switch` in Java. The `case` structure can be graphed either as a single node with multi-way branching or as a series of `if-then-else` structures. We choose to illustrate the `case` structure with multi-way branching, as in Figure 7.23.

If the programmer omits a `break` statement, this must be reflected in the graph. For example, if the `break` in the 'N' case is omitted, the graph would contain an edge from node 2 to node 3, reflecting the "fall-through" semantics in Java, and not from node 2 to 5.

Our final language structure is exception handling, which in Java uses the `try-catch` statement. Figure 7.24 shows an input statement with three exceptions, one called by the run

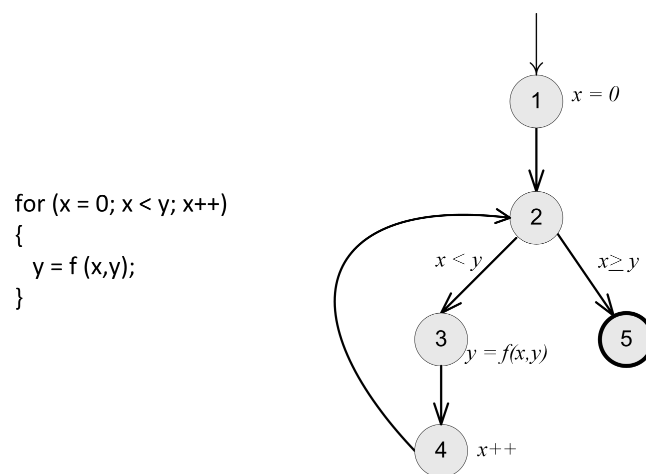


Figure 7.20: CFG fragment for the **for** loop structure.

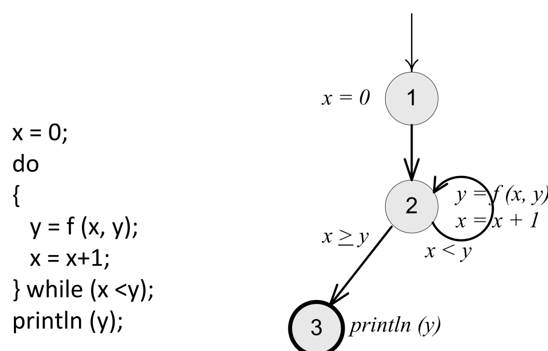


Figure 7.21: CFG fragment for the **do-while** structure.

time system (`IOException`) and the other two called by the program (`Exception`). The edge from node 1 to 2 reflects the `IOException` that can be raised if the `readLine()` statement fails. The subpaths [3, 4, 6] and [5, 7, 6] represent the programmer-raised exceptions. If the string is too long or too short, then the `throw` statement is run, and control transfers to the `catch` block.

The coverage criteria from the previous section can now be applied to graphs from source code. The application is direct with only the names being changed. Node Coverage is often called *Statement Coverage* or *Basic Block Coverage* and Edge Coverage is often called *Branch Coverage*.

7.3.2 Data Flow Graph Coverage for Source Code

This section applies the data flow criteria to the code examples given in the prior section. Before we can do this, we need to define what constitutes a *def* and what constitutes a *use*. A *def* is a location in the program where a value for a variable is stored into memory

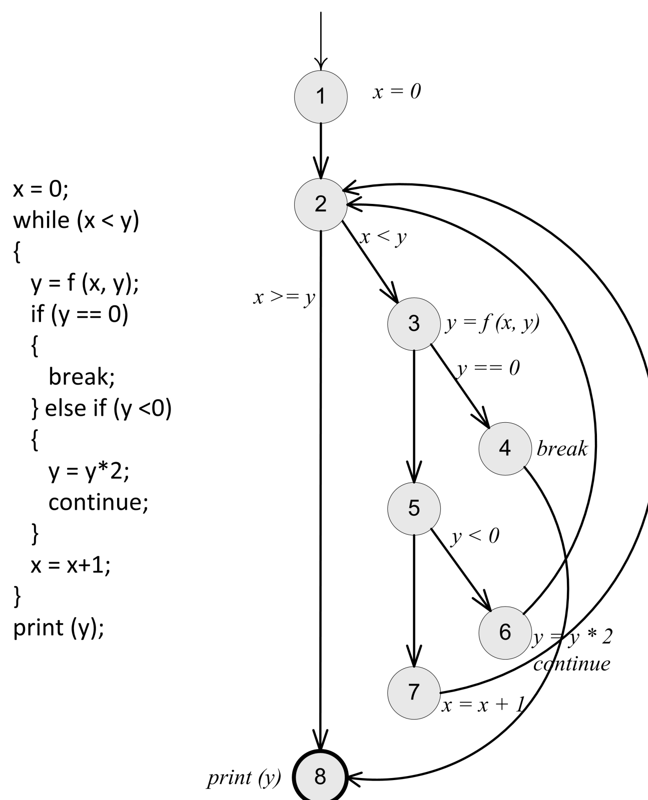


Figure 7.22: CFG fragment for the **while** loop with a **break** structure.

(assignment, input, etc.). A *use* is a location where a variable's value is accessed.

A def may occur for variable `x` in the following situations:

1. `x` appears on the left side of an assignment statement
2. `x` is an actual parameter in a call site and its value is changed within the method
3. `x` is a formal parameter of a method (an implicit def when the method begins execution)
4. `x` is an input to the program

Some features of programming languages greatly complicate this seemingly simple definition. For example, is a def of an array variable a def of the entire array, or of just the element being referenced? What about objects; should the def consider the entire object, or only a particular instance variable inside the object? If two variables reference the same location, that is, the variables are aliases, how is the analysis done? What is the relationship between coverage of the original source code, coverage of the optimized source code, and coverage of the machine code? We omit these complicating issues in our presentation and refer advanced readers to the bibliographic notes.

If a variable has multiple definitions in a single basic block, the last definition is the only one that is relevant to data flow analysis.

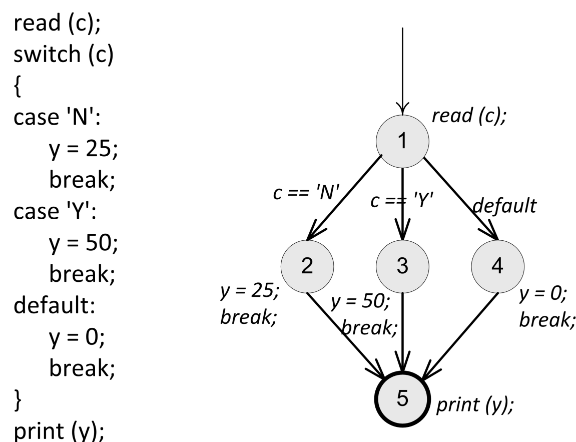


Figure 7.23: CFG fragment for the **case** structure.

A use may occur for variable `x` in the following situations:

1. `x` appears on the right side of an assignment statement
2. `x` appears in a conditional test (note that such a test is always associated with at least two edges)
3. `x` is an actual parameter to a method
4. `x` is an output of the program
5. `x` is an output of a method in a `return` statement or returned through a parameter

Not all uses are relevant for data flow analysis. Consider the following statements that reference local variables (ignoring concurrency):

```
y = z;
x = y + 2;
```

The use of `y` in the second statement is called a *local use*; it is impossible for a def in another basic block to reach it. The reason is that the definition of `y` in `y = z` *always* overrides any definition of `y` from any other basic block. That is, no def-clear path goes from any other def to that use. In contrast, the use of `z` is called *global*, because the definition of `z` used in this basic block must originate in some other basic block. Data flow analysis only considers global uses.

The `PatternIndex` example in Figure 7.25 is used to illustrate dataflow analysis for a simple string pattern matching method called `patternIndex()`. The CFG for `patternIndex()` was previously shown in Figure 7.12, with the actual Java statements annotated on the nodes and edges.

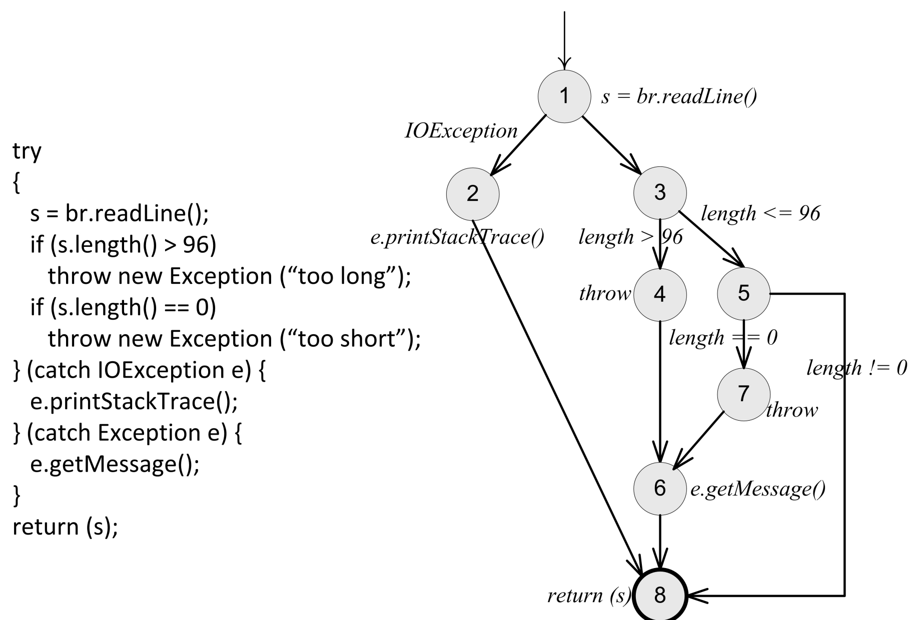


Figure 7.24: CFG fragment for the **try-catch** structure.

The CFG for `patternIndex()` with def and use sets explicitly marked was shown in Figure 7.13. While numerous tools can create CFGs for programs, it helps students to create CFGs by hand. When doing so, a good habit is to draw the CFG first with the statements, then redraw it with the def and use sets.

Table 7.1 lists the defs and uses at each node in the CFG for `patternIndex()`. This simply repeats the information in Figure 7.13, but in a convenient form. Table 7.2 contains the same information for edges. We suggest that beginning students check their understanding of these definitions by verifying that the contents of these two tables are correct.

Finally, we list the du-paths for each variable in `patternIndex()` followed by all the du-paths for each du-pair in Table 7.3. The first column gives the variable name, and the second gives the def node number and variable (that is, the left side of the formula that lists all the du-paths with respect to the variable, as defined in Section 7.2.3). The third column lists all the du-paths that start with that def. If a du-pair has more than one path to the same use, they are listed on multiple rows with subpaths that end with the same node number. The fourth column, "prefix?", is a notational convenience that is explained below. This information is extremely tedious to derive by hand, and testers tend to make many errors. This analysis is best done automatically.

Several def/use pairs have more than one du-path in `patternIndex()`. For example, the variable `iSub` is defined in node 2 and used in node 10. Three du-paths go from node 2 to 10, `[2,3,4,10](iSub)`, `[2,3,4,5,6,10](iSub)`, and `[2,3,4,5,6,7,8,10](iSub)`.

One optimization uses the fact that a du-path must be toured by any test that tours an extension of that du-path. These du-paths are marked with the annotation "Yes" in the `prefix?` column of the table. For example, `[2,3,4](iSub)` is necessarily toured by

```
/**
 * Find index of pattern in subject string
 *
 * @param subject String to search
 * @param pattern String to find
 * @return index (zero-based) of first occurrence of pattern in subject;
 *              -1 if not found
 * @throws NullPointerException if subject or pattern is null
 */
public static int patternIndex (String subject, String pattern)
{
  final int NOTFOUND = -1;
  int  iSub = 0, rtnIndex = NOTFOUND;
  boolean isPat  = false;
  int subjectLen = subject.length();
  int patternLen = pattern.length();

  while (isPat == false && iSub + patternLen - 1 < subjectLen)
  {
    if (subject.charAt (iSub) == pattern.charAt (0))
    {
      rtnIndex = iSub; // Starting at zero
      isPat = true;
      for (int iPat = 1; iPat < patternLen; iPat ++)
      {
        if (subject.charAt (iSub + iPat) != pattern.charAt (iPat))
        {
          rtnIndex = NOTFOUND;
          isPat = false;
          break;  // out of for loop
        }
      }
    }
    iSub ++;
  }
  return (rtnIndex);
}
```

Figure 7.25: Method patternIndex() for data flow example.

Table 7.1: Defs and uses at each node in the CFG for patternIndex().

node	def	use
1	{subject, pattern}	
2	{NOTFOUND, isPat, iSub, rtnIndex, subjectLen, patternLen}	{subject, pattern}
3		
4		
5	{rtnIndex, isPat, iPat}	{iSub}
6		
7		
8	{rtnIndex, isPat}	{NOTFOUND}
9	{iPat}	{iPat}
10	{iSub}	{iSub}
11		{rtnIndex}

Table 7.2: Defs and uses at each edge in the CFG for patternIndex().

edge	use
(1, 2)	
(2, 3)	
(3, 4)	{iSub, patternLen, subjectLen, isPat}
(3, 11)	{iSub, patternLen, subjectLen, isPat}
(4, 5)	{subject, iSub, pattern}
(4, 10)	{subject, iSub, pattern}
(5, 6)	
(6, 7)	{iPat, patternLen}
(6, 10)	{iPat, patternLen}
(7, 8)	{subject, iSub, iPat, pattern}
(7, 9)	{subject, iSub, iPat, pattern}
(8, 10)	
(9, 6)	
(10, 3)	

any test that tours the du-path `[2,3,4,5,6,7,8](iSub)`, because `[2,3,4]` is a prefix of `[2,3,4,5,6,7,8]`. Thus, the path is not considered in the subsequent table that relates du-paths to test paths that tour them. One has to be a bit careful with this optimization, since the extended du-path may be infeasible even if the prefix is not.

Table 7.4 shows that a relatively small set of 11 test cases satisfies all du-paths coverage on this example. (One du-path is infeasible.) The reader may wish to evaluate this test set with the non-dataflow graph coverage criteria. These tests are available on the book website in `DataDrivenPatternIndexTest.java`.

Table 7.3: du-path sets for each variable in patternIndex().

variable	du-path set	du-paths	prefix?
NOTFOUND	du (2, NOTFOUND)	[2,3,4,5,6,7,8]	
rtnIndex	du (2, rtnIndex)	[2,3,11]	
	du (5, rtnIndex)	[5,6,10,3,11]	
	du (8, rtnIndex)	[8,10,3,11]	
iSub	du (2, iSub)	[2,3,4]	Yes
		[2,3,4,5]	Yes
		[2,3,4,5,6,7,8]	Yes
		[2,3,4,5,6,7,9]	
		[2,3,4,5,6,10]	
		[2,3,4,5,6,7,8,10]	
		[2,3,4,10]	
		[2,3,11]	
	du (10, iSub)	[10,3,4]	Yes
		[10,3,4,5]	Yes
		[10,3,4,5,6,7,8]	Yes
		[10,3,4,5,6,7,9]	
		[10,3,4,5,6,10]	
		[10,3,4,5,6,7,8,10]	
		[10,3,4,10]	
		[10,3,11]	
iPat	du (5, iPat)	[5,6,7]	Yes
		[5,6,10]	
		[5,6,7,8]	
		[5,6,7,9]	
	du (9, iPat)	[9,6,7]	Yes
		[9,6,10]	
		[9,6,7,8]	
		[9,6,7,9]	
isPat	du (2, isPat)	[2,3,4]	
		[2,3,11]	
	du (5, isPat)	[5,6,10,3,4]	
		[5,6,10,3,11]	
	du (8, isPat)	[8,10,3,4]	
		[8,10,3,11]	
subject	du (1, subject)	[1,2]	Yes
		[1,2,3,4,5]	Yes
		[1,2,3,4,10]	
		[1,2,3,4,5,6,7,8]	
		[1,2,3,4,5,6,7,9]	
pattern	du (1, pattern)	[1,2]	Yes
		[1,2,3,4,5]	Yes
		[1,2,3,4,10]	
		[1,2,3,4,5,6,7,8]	
		[1,2,3,4,5,6,7,9]	
subjectLen	du (2, subjectLen)	[2,3,4]	
		[2,3,11]	
patternLen	du (2, patternLen)	[2,3,4]	Yes
		[2,3,11]	
		[2,3,4,5,6,7]	
		[2,3,4,5,6,10]	

Table 7.4: Test paths to satisfy all du-paths coverage on patternIndex().

test case (subject,pattern,output)	test path(t)
(a, bc, -1)	[1,2,3,11]
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]
(ab, b, 1)	[1,2,3,4,10,3,4,5,6,10,3,11]
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]
(abc, abc, 0)	[1,2,3,4,5,6,7,9,6,7,9,6,10,3,11]
(abc, abd, -1)	[1,2,3,4,5,6,7,9,6,7,8,10,3,11]
(abc, ac -1)	[1,2,3,4,5,6,7,8,10,3,4,10,3,11]
(abc, ba, -1)	[1,2,3,4,10,3,4,5,6,7,8,10,3,11]
(abc, bc, 1)	[1,2,3,4,10,3,4,5,6,7,9,6,10,3,11]

Exercises, Section 7.3.

1. Use the following program fragment for questions a-e below.

```
w = x;          // node 1
if (m > 0)
{
   w++;         // node 2
}
else
{
   w=2*w;       // node 3
}
// node 4 (no executable statement)
if (y <= 10)
{
   x = 5*y;     // node 5
}
else
{
   x = 3*y+5;   // node 6
}
z = w + x;      // node 7
```

   (a) Draw a control flow graph for this program fragment. Use the node numbers given above.
   (b) Which nodes have defs for variable w?
   (c) Which nodes have uses for variable w?
   (d) Are there any du-paths with respect to variable w from node 1 to node 7? If not, explain why not. If any exist, show one.
   (e) Enumerate all of the du-paths for variables w and x.

Table 7.5: Test paths and du-paths covered on patternIndex().

test case (subject,pattern, output)	test path(t)	du-path toured
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4,5,6,7,8](NOTFOUND)
(a, bc, -1)	[1,2,3,11]	[2,3,11](rtnIndex)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[5,6,10,3,11](rtnIndex)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[8,10,3,11](rtnIndex)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[2,3,4,5,6,7,9] (iSub)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[2,3,4,5,6,10](iSub)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4,5,6,7,8,10](iSub)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[2,3,4,10](iSub)
(a, bc, -1)	[1,2,3,11]	[2,3,11] (iSub)
(abc, bc, 1)	[1,2,3,4,10,3,4,5,6,7,9,6,10,3,11]	[10,3,4,5,6,7,9](iSub)
(ab, b, 1)	[1,2,3,4,10,3,4,5,6,10,3,11]	[10,3,4,5,6,10](iSub)
(abc, ba, -1)	[1,2,3,4,10,3,4,5,6,7,8,10,3,11]	[10,3,4,5,6,7,8,10](iSub)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[10,3,4,10](iSub)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[10,3,11](iSub)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[5,6,10](iPat)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[5,6,7,8](iPat)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[5,6,7,9](iPat)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[9,6,10](iPat)
(abc, abd, -1)	[1,2,3,4,5,6,7,9,6,7,8,10,3,11]	[9,6,7,8](iPat)
(abc, abc, 0)	[1,2,3,4,5,6,7,9,6,7,9,6,10,3,11]	[9,6,7,9](iPat)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4](isPat)
(a, bc, -1)	[1,2,3,11]	[2,3,11](isPat)
No test case	Infeasible	[5,6,10,3,4](isPat)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[5,6,10,3,11](isPat)
(abc, ac -1)	[1,2,3,4,5,6,7,8,10,3,4,10,3,11]	[8,10,3,4](isPat)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[8,10,3,11](isPat)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[1,2,3,4,10](subject)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[1,2,3,4,5,6,7,8](subject)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[1,2,3,4,5,6,7,9](subject)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[1,2,3,4,10](pattern)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[1,2,3,4,5,6,7,8](pattern)
(ab, ab, 0)	[1,2,3,4,5,6,7,9,6,10,3,11]	[1,2,3,4,5,6,7,9](pattern)
(ab, c, -1)	[1,2,3,4,10,3,4,10,3,11]	[2,3,4](subjectLen)
(a, bc, -1)	[1,2,3,11]	[2,3,11](subjectLen)
(a, bc, -1)	[1,2,3,11]	[2,3,11](patternLen)
(ab, ac, -1)	[1,2,3,4,5,6,7,8,10,3,11]	[2,3,4,5,6,7](patternLen)
(ab, a, 0)	[1,2,3,4,5,6,10,3,11]	[2,3,4,5,6,10](patternLen)

2. Select a commercial coverage tool of your choice. Note that some have free trial evaluations. Choose a tool, download it, and run it on some software. You can use one of the examples from this text, software from your work environment, or software available over the Web. Write up a short summary report of your experience with the tool. Be sure to include any problems installing or using the tool. The main grading criterion is that you actually collect some coverage data for a reasonable set of tests on some program.

3. Consider the pattern matching example in Figure 7.25. Instrument the code so as to be able to produce the execution paths reported in the text for this example. That is, on a given test execution, your instrumentation program should compute and print the corresponding test path. Run the instrumented program on the test cases listed at the end of Section 7.3.

4. Consider the pattern matching example in Figure 7.25. In particular, consider the final table of tests in Section 7.3. Consider the variable *iSub*. Number the (unique) test cases, starting at 1, from the top of the *iSub* part of the table. For example, $(ab, c, -1)$, which appears twice in the *iSub* portion of the table, should be labeled test $t4$.

   (a) Give a minimal test set that satisfies *all defs* coverage. Use the test cases given.
   (b) Give a minimal test set that satisfies *all uses* coverage.
   (c) Give a minimal test set that satisfies *all du-paths* coverage.

5. Again consider the pattern matching example in Figure 7.25. Instrument the code so as to produce the execution paths reported in the text for this example. That is, on a given test execution, your tool should compute and print the corresponding test path. Run the following three test cases and answer questions a-g below:

   - *subject* = "brown owl" *pattern* = "wl" *expected output* = 7
   - *subject* = "brown fox" *pattern* = "dog" *expected output* = -1
   - *subject* = "fox" *pattern* = "brown" *expected output* = -1

   (a) Find the actual path followed by each test case.
   (b) For each path, give the du-paths that the path tours in the table at the end of Section 7.3. To reduce the scope of this exercise, consider only the following du-paths: *du (10, iSub)*, *du (2, isPat)*, *du (5, isPat)*, and *du (8, isPat)*.
   (c) Explain why the du-path [5, 6, 10, 3, 4] cannot be toured by any test path.
   (d) Select tests from the table at the end of Section 7.3 to complete coverage of the (feasible) du-paths that are uncovered in question a.
   (e) From the tests above, find a minimal set of tests that achieves All-Defs Coverage with respect to the variable *isPat*.
   (f) From the tests above, find a minimal set of tests that achieves All-Uses Coverage with respect to the variable *isPat*.
   (g) Is there any difference between All-Uses Coverage and all du-paths coverage with respect to the variable *isPat* in the *pat* method?

6. Use the method **fmtRewrap()** for questions a-e below. A compilable version is available on the book website in the file `FmtRewrap.java`. A line numbered version suitable for this exercise is available on the book website in the file `FmtRewrap.num`.

(a) Draw the control flow graph for the **fmtRewrap()** method.
(b) For **fmtRewrap()**, find a test case such that the corresponding test path visits the edge that connects the beginning of the **while** statement to the **S = new String(SArr) + CR;** statement **without** going through the body of the while loop.
(c) Enumerate the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage for the graph for **fmtRewrap()**.
(d) List test paths that achieve Node Coverage but not Edge Coverage on the graph.
(e) List test paths that achieve Edge Coverage but not Prime Path Coverage on the graph.

7. Use the method **printPrimes()** for questions a-f below. A compilable version is available on the book website in the file `PrintPrimes.java`. A line numbered version suitable for this exercise is available on the book website in the file `PrintPrimes.num`.

(a) Draw the control flow graph for the **printPrimes()** method.
(b) Consider test cases $t1 = (n = 3)$ and $t2 = (n = 5)$. Although these tour the same prime paths in *printPrimes()*, they do not necessarily find the same faults. Design a simple fault that $t2$ would be more likely to discover than $t1$ would.
(c) For *printPrimes()*, find a test case such that the corresponding test path visits the edge that connects the beginning of the **while** statement to the **for** statement **without** going through the body of the while loop.
(d) Enumerate the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage for the graph for *printPrimes()*.
(e) List test paths that achieve Node Coverage but not Edge Coverage on the graph.
(f) List test paths that achieve Edge Coverage but not Prime Path Coverage on the graph.

8. Consider the *equals()* method from the *java.util.AbstractList* class:

```
public boolean equals (Object o)
{
  if (o == this) // A
     return true;
  if (!(o instanceof List)) // B
     return false;

  ListIterator e1 = listIterator();
  ListIterator e2 = ((List) o).listIterator();
  while (e1.hasNext() && e2.hasNext()) // C
  {
     E o1 = e1.next();
     Object o2 = e2.next();
     if (!(o1 == null ? o2 == null : o1.equals (o2))) // D
        return false;
  }
  return !(e1.hasNext() || e2.hasNext()); // E
}
```

(a) Draw a control flow graph for this method. There are several possible values for the number of nodes in the graph. Choose something reasonable.
(b) Label edges and nodes in the graph with the corresonding code fragments. You may abbreviate predicates as follows when labelling your graph:

```
A: o == this
B: !(o instanceof List)
C: e1.hasNext() && e2.hasNext()
C: e1.hasNext() && e2.hasNext()
D: !(o1 == null ? o2 == null : o1.equals(o2))
E: !(e1.hasNext() $\parallel$ e2.hasNext())
```

(c) Node coverage requires (at least) four tests on this graph. Explain why.

(d) Provide four tests (as calls to *equals()*) that yield node coverage on this graph. Make your tests short. You need to include output assertions. Assume that each test is independent and starts with the following state:

List<String> list1 = new ArrayList<String>();
List<String> list2 = new ArrayList<String>();
Use the constants null, "ant", "bat", etc. as needed

7.4 Graph Coverage for Design Elements

Use of data abstraction and object-oriented software has led to an increased emphasis on modularity and reuse. This means that testing of software based on various parts of the design (*design elements*) is becoming more important than in the past. These activities are usually associated with integration testing. One benefit of modularity is that the software components can be tested independently, which is usually done by programmers during unit and module testing.

7.4.1 Structural Graph Coverage for Design Elements

Graph coverage for design elements usually starts by creating graphs that are based on couplings between software components. *Coupling* measures the dependency relations between two units by reflecting their interconnections; faults in one unit may affect the coupled unit. Coupling provides summary information about the design and the structure of the software. Most test criteria for design elements require that connections among program components be visited.

The most common graph used for structural design coverage is the *call graph*. In a call graph, the nodes represent methods (or units) and the edges represent method calls. Figure 7.26 represents a small program that contains six methods. Method `A` calls `B`, `C`, and `D`, `C` calls `E` and `F`, and `D` also calls `F`.

The coverage criteria from Section 7.2.1 can be applied to call graphs. Node Coverage requires that each method be called at least once and is also called *Method Coverage*. Edge Coverage requires that each call be executed at least once and is also called *Call Coverage*.

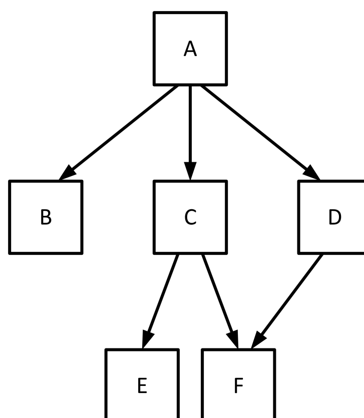


Figure 7.26: A simple call graph.

For the example in Figure 7.26, Node Coverage requires that each method be called at least once, whereas Edge Coverage requires that `F` be called at least twice, once from `C` and once from `D`.

Application to Modules

Recall from Chapter 2 that a module is a collection of related units, for example a class is Java's version of a module. As opposed to complete programs, the units in a class may not all call each other. Thus, instead of being able to obtain one connected call graph, we may generate several disconnected call graphs. In a simple degenerative case (such as for a simple stack), there may be no calls between units. In these cases, module testing with this technique is not appropriate. Techniques based on sequences of calls are needed.

Inheritance and Polymorphism

The object-oriented language features of inheritance and polymorphism introduce new abilities for designers and programmers, but also new problems for testers. The research community is still developing ways to test these language features, so this text introduces the current state of knowledge. The interested reader is encouraged to keep up with the literature for continuing results and techniques for testing OO software. The bibliographic notes give some current references, and further ideas are discussed in **Chapter ???**. The most obvious graph to create for testing these features (collectively called "the OO language features") is the inheritance hierarchy. Figure 7.27 represents a small inheritance hierarchy with four classes. Classes `C` and `D` inherit from `B`, and `B` in turn inherits from `A`.

The coverage criteria from Section 7.2.1 can be applied to inheritance hierarchies in ways that are superficially simple, but have some subtle problems. In OO programming, classes are not directly tested because they are not executable. In fact, the edges in the inheritance hierarchy do not represent execution flow at all, but rather inheritance dependencies. To apply any type of coverage, we first need a model for what coverage means. The first step is

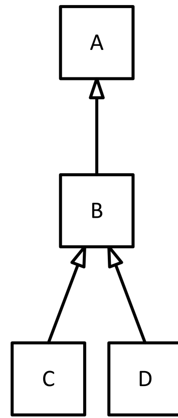


Figure 7.27: A simple inheritance hierarchy.

to require that objects be instantiated for some or all of the classes. Figure 7.28 shows the inheritance hierarchy from Figure 7.27 with one object instantiated for each class.

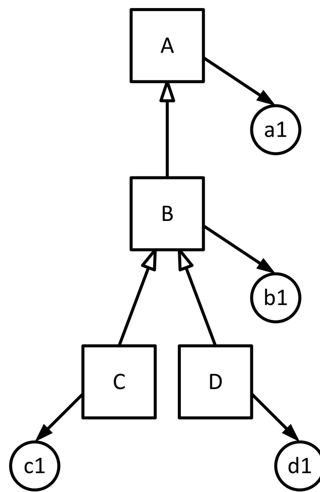


Figure 7.28: An inheritance hierarchy with objects instantiated.

The most obvious interpretation of Node Coverage for this graph is to require that at least one object be created for each class. However, this seems weak because it says nothing about execution. The logical extension is to require that for each object of each class, the call graph must be covered according to the Call Coverage criterion above. We call this the *OO Call Coverage* criterion, and it can be considered an "aggregation criterion" because it requires Call Coverage to be applied on at least one object for each class.

An extension of this is the *All Object Call* criterion, which requires that Call Coverage is satisfied for every object that is instantiated for every class.

7.4.2 Data Flow Graph Coverage for Design Elements

Control connections among design elements are simple and straightforward and tests based on them are probably not very effective at finding faults. On the other hand, data flow connections are often very complex and difficult to analyze. For a tester, that should immediately suggest that they are a rich source for software faults. The primary issue is where the defs and uses occur. When testing program units, the defs and uses are in the same unit. During integration testing, defs and uses are in different units. This section starts with some standard compiler/program analysis terms.

A *caller* is a unit that invokes another unit, the *callee*. The statement that makes the call is the *call site*. An *actual parameter* is in the caller; its value is assigned to a *formal parameter* in the callee. The *call interface* between two units is the mapping of actual to formal parameters.

The underlying premise of the data flow testing criteria for design elements is that to achieve confidence in the interfaces between integrated program units, it must be ensured that variables defined in caller units be appropriately used in callee units. This technique can be limited to the unit interfaces, allowing us to restrict our attention to the **last** definitions of variables just **before** calls to and returns from the called units, and the **first** uses of variables just **after** calls to and returns from the called unit.

Figure 7.29 illustrates the relationships that the data flow criteria will test. The criteria require execution from definitions of actual parameters through calls to uses of formal parameters.

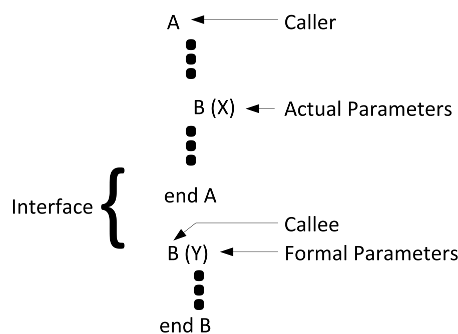


Figure 7.29: An example of parameter coupling.

Three types of data flow couplings have been identified. The most obvious is *parameter coupling*, where parameters are passed in calls. *Shared data coupling* occurs when two units access the same data object as a global or other non-local variable, and *external device coupling* occurs when two units access the same external medium such as a file. In the following, all examples and discussion will be in terms of parameters and it will be understood that the concepts apply equally to shared data and external device coupling. We use the general term *coupling variable* for variables that are defined in one unit and used in another.

This form of data flow is concerned only with last-defs before calls and returns and first-uses after calls and returns. That is, it is concerned only with defs and uses immediately surrounding the calls between methods. The last-defs before a call are locations with defs that reach uses at call sites and the last-defs before a return are locations with defs that reach a return statement. The following definitions assume a variable that is defined in either the caller or the callee, and used in the other.

Definition 7.10 Last-def: *The set of nodes that define a variable* `x` *for which there is a def-clear path from the node through the call site to a use in the other unit.*

The variable can be passed as a parameter, a return value, or a shared variable reference. If the function has no return statement, an implicit return statement is assumed to exist at the last statement in the method.

The definition for first-use is complementary to that of last-def. It depends on paths that are not only def-clear, but also use-clear. A path from n_i to n_j is *use-clear* with respect to variable v if for every node n_k on the path, $k \neq i$ and $k \neq j$, v is not in $use(n_k)$. Assume that the variable y is used in one of the units after having been defined in the other. Further assume that y has received a value that has been passed from the other unit, either through parameter passing, a return statement, shared data, or other value passing.

Definition 7.11 First-use: *The set of nodes that have uses of a variable* y *and for which there exists a path that is def-clear and use-clear from the entry point (if the use is in the callee) or the call site (if the use is in the caller) to the nodes.*

Figure 7.30 shows a caller `F()` and a callee `G()`. The call site has two du-pairs; `x` in `F()` is passed to `a` in `G()` and `b` in `G()` is returned and assigned to `y` in `F()`. Note that the assignment to `y` in `F()` is explicitly **not** the use, but considered to be part of the transfer. Its use is further down, in the `print(y)` statement.

This definition allows for one anomaly when a return value is not explicitly assigned to a variable, as in the statement `print (f(x))`. In this case, an implicit assignment is assumed and the first-use is in the `print(y)` statement.

Figure 7.31 illustrates last-defs and first-uses between two units with two partial CFGs. The unit on the left, the caller, calls the callee `B`, with one actual parameter, `X`, which is assigned to formal parameter `y`. `X` is defined at nodes 1, 2 and 3, but the def at node 1 cannot reach the call site at node 4, thus the last-defs for `X` is the set $\{2, 3\}$. The formal parameter `y` is used at nodes 11, 12, and 13, but no use-clear path goes from the entry point at node 10 to 13, so the first-uses for `y` is the set $\{11, 12\}$.

Recall that a du-path is a path from a def to a use in the same graph. This notion is refined to a *coupling du-path* with respect to a coupling variable x. A coupling du-path is a path from a last-def to a first-use.

The coverage criteria from Section 7.2.3 can now be applied to coupling graphs. All-Defs Coverage requires that a path be executed from every last-def to at least one first-use. In this context, all-defs is called *All-Coupling-Def* coverage. All-Uses Coverage requires that a path be executed from every last-def to every first-use. In this context, all-uses is also called

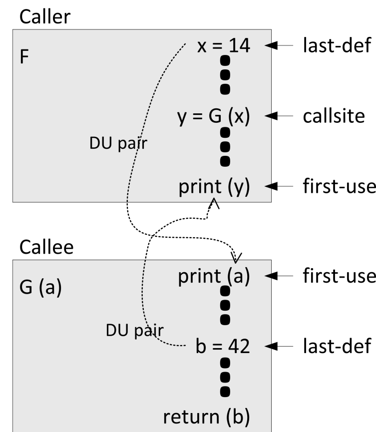


Figure 7.30: Coupling du-pairs.

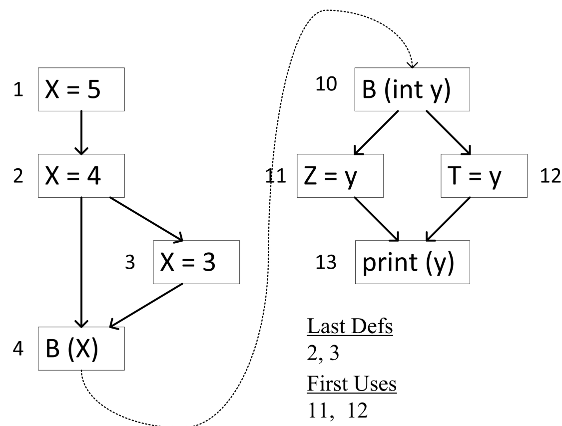


Figure 7.31: Last-defs and first-uses.

All-Coupling-Use coverage.

Finally, All-du-Paths coverage requires that we tour every simple path from every last-def to every first-use. As before, the All-du-Paths criterion can be satisfied by tours that include sidetrips. In this context, All-du-Paths is also called *All-Coupling-du-Paths* coverage.

Concrete Example

Now we will use a concrete example to illustrate coupling data flow. Class `Quadratic` in Figure 7.32 computes the quadratic root of an equation, given three integer coefficients. The call to `Root()` on line 34 in `main` passes in three parameters. Each of the variables `X`, `Y`, and `Z` have three last-defs in the caller at lines 16, 17, 18, lines 23, 24, and 25, and lines 30, 31, and 32. They are mapped to formal parameters `A`, `B`, and `C` in `Root()`. All three variables have a first-use at line 47. The class variables `Root1` and `Root2` are defined in the callee and

used in the caller. Their last-defs are at lines 53 and 54 and the first-use is at line 37.

The value of local variable `Result` is returned to the caller, with two last-defs at lines 50 and 55 and a first-use in the caller at line 35.

The coupling du-pairs are listed as pairs of triples. Each triple gives a unit name, variable name, and a line number. The first triple in a pair says where the variable is defined, and the second where it is used. The complete set of coupling du-pairs for class `Quadratic` is:

```
(main(), X, 16) — (Root(), A, 47)
(main(), Y, 17) — (Root(), B, 47)
(main(), Z, 18) — (Root(), C, 47)
(main(), X, 23) — (Root(), A, 47)
(main(), Y, 24) — (Root(), B, 47)
(main(), Z, 25) — (Root(), C, 47)
(main(), X, 30) — (Root(), A, 47)
(main(), Y, 31) — (Root(), B, 47)
(main(), Z, 32) — (Root(), C, 47)
(Root(), Root1, 53) — (main(), Root1, 37)
(Root(), Root2, 54) — (main(), Root2, 37)
(Root(), Result, 50) — (main(), ok, 35)
(Root(), Result, 55) — (main(), ok, 35)
```

A couple of things are important to remember about coupling data flow. First, only variables that are used or defined in the callee are considered. That is, last-defs that have no corresponding first-uses are not useful for testing. Second, we must remember implicit initialization of class and global variables. In some languages (such as Java and C), class and instance variables are given default values. These definitions can be modeled as occurring at the beginning of appropriate units. For example, class-level initializations may be considered to occur in the `main()` method or in constructors. Although other methods that access class variables may use the default values on the first call, it is also possible for such methods to use values written by other methods, and hence the normal coupling data flow analysis methods should be employed. Also, this analysis is specifically not considering "transitive du-pairs." That is, if unit A calls B, and B calls C, last-defs in A do **not** reach first-uses in C. This type of analysis is prohibitively expensive with current technologies and of questionable value. Finally, data flow testing has traditionally taken an abstract view of array references. Identifying and keeping track of individual array references is an undecidable problem in general and very expensive even in finite cases. So, most tools consider a reference to one element of an array to be a reference to the entire array.

Inheritance and Polymorphism (*Advanced topic*)

The previous discussion covers the most commonly used form of data flow testing as applied beyond the method level. However, the flow of data along couplings between callers and callees is only one type of a very complicated set of data definition and use pairs. Consider Figure 7.33, which illustrates the types of du-pairs discussed so far. On the left is a method,

```
 1  // Program to compute the quadratic root for two numbers
 2  import java.lang.Math;
 3
 4  class Quadratic
 5  {
 6  private static double Root1, Root2;
 7
 8  public static void main (String[] argv)
 9  {
10     int X, Y, Z;
11     boolean ok;
12     if (argv.length == 3)
13     {
14        try
15        {
16           X = Integer.parseInt (argv[0]);
17           Y = Integer.parseInt (argv[1]);
18           Z = Integer.parseInt (argv[2]);
19        }
20        catch (NumberFormatException e)
21        {
22           System.out.println ("Inputs not integers, using 8, 10, -33.");
23           X = 8;
24           Y = 10;
25           Z = -33;
26        }
27     }
28     else
29     {
30        X = 8;
31        Y = 10;
32        Z = -33;
33     }
34     ok = Root (X, Y, Z);
35     if (ok)
36        System.out.println
37           ("Quadratic: Root 1 = " + Root1 + ", Root 2 = " + Root2);
38     else
39        System.out.println ("No solution.");
40  }
41
42  // Finds the quadratic root, A must be non-zero
43  private static boolean Root (int A, int B, int C)
44  {
45     double D;
46     boolean Result;
47     D = (double)(B*B) - (double)(4.0*A*C);
48     if (D < 0.0)
49     {
50        Result = false;
51        return (Result);
52     }
53     Root1 = (double) ((-B + Math.sqrt(D)) / (2.0*A));
54     Root2 = (double) ((-B - Math.sqrt(D)) / (2.0*A));
55     Result = true;
56     return (Result);
57  } // End method Root
58
59  } // End class Quadratic
```

Figure 7.32: Quadratic root program.

`A()`, which contains a def and a use. (For this discussion we will omit the variable for simplicity.) The right illustrates two types of inter-procedural du-pairs.

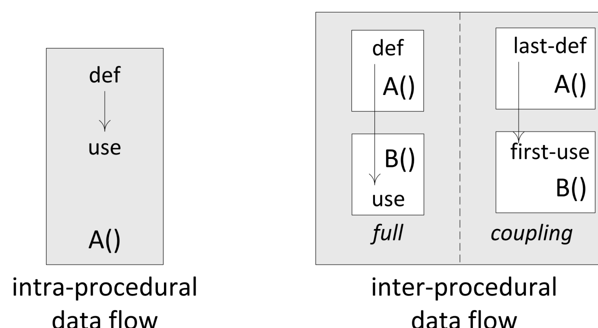


Figure 7.33: Def-use pairs under intra-procedural and inter-procedural data flow.

Full inter-procedural data flow identifies **all** du-pairs between a caller (`A()`) and a callee (`B()`). *Coupling inter-procedural* data flow is as described in Section 7.4.2; identifying du-pairs between last-defs and first-uses.

Figure 7.34 illustrates du-pairs in object-oriented software. DU pairs are usually based on the *class* or *state* variables defined for the class. The left picture in Figure 7.34 shows the "direct" case for OO du-pairs. A *coupling method*, `F()`, calls two methods, `A()` and `B()`. `A()` defines a variable and `B()` uses it. For the variable reference to be the same, both `A()` and `B()` must be called through the same *instance context*, or object reference. That is, if the calls are `o.A()` and `o.B()`, they are called through the instance context of `o`. If the calls are not made through the same instance context, the definition and use will be to different instances of the variable.

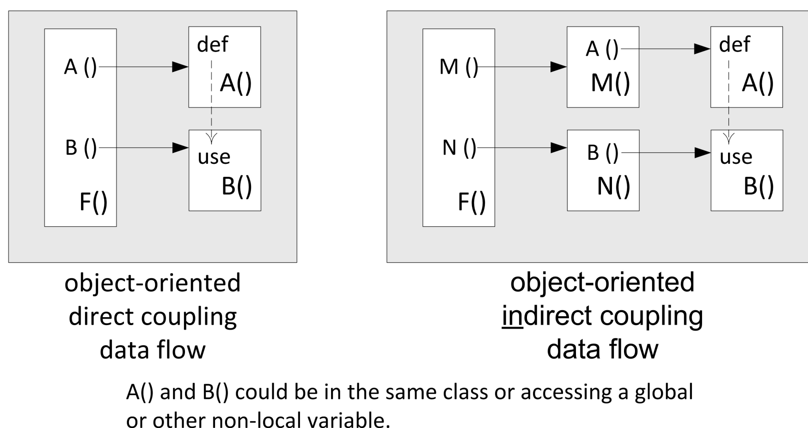


Figure 7.34: Def-use pairs in object-oriented software.

The right side of Figure 7.34 illustrates "indirect" du-pairs. In this scenario, the coupling method `F()` calls methods `M()` and `N()`, which in turn call two other methods, `A()` and `B()`.

The def and use are in `A()` and `B()`, so the reference is indirect. The analysis for indirect du-pairs is considerably more complicated than for direct du-pairs. It should be obvious that there can be more than one call between the coupling method and the methods with the def and use.

In OO data flow testing, the methods `A()` and `B()` could be in the same class, or they could be in different classes and accessing the same global variables.

Finally, Figure 7.35 illustrates du-pairs in distributed software. `P1` and `P2` are two processes, threads, or other distributed software components, and they call `A()` and `B()`, which define and use the same variable. The distribution and communication could use any of a number of methods, including HTTP (Web-based), remote method invocation (RMI), or CORBA. `A()` and `B()` could be in the same class or could access a persistent variable such as a web session variable or permanent data store. While this sort of "very loosely coupled" software can be expected to have far fewer du-pairs, identifying them, finding def-clear paths between them, and test cases to cover them is quite complicated.

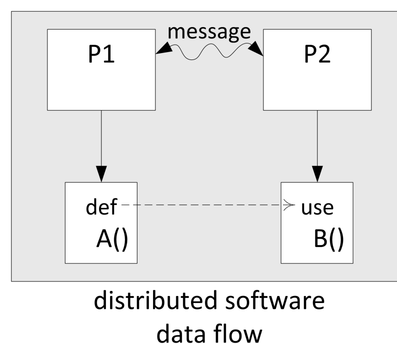


Figure 7.35: Def-use pairs in web applications and other distributed software.

Exercises, Section 7.4.

1. Use the class *Watch* in Figures 7.38 and 7.39 in Section 7.5 to answer questions a-d below.
   (a) Draw control flow graphs for the methods in Watch.
   (b) List all the call sites.
   (c) List all coupling du-pairs for each call site.
   (d) Create test data to satisfy *All-Coupling-Use Coverage* for Watch.
2. Use the class **Stutter()** for questions a-d below. A compilable version is available on the book website in the file `Stutter.java`. A line numbered version suitable for this exercise is available on the book website in the file `Stutter.num`.

(a) Draw a control flow graph for Stutter.
(b) List all the call sites.
(c) List all coupling du-pairs for each call site.
(d) Create test data to satisfy *All-Coupling Use Coverage* for Stutter.

3. Use the following program fragment for questions a-e below.

```
public static void f1 (int x, int y)
{
   if (x < y) { f2 (y); } else { f3 (y); };
}
public static void f2 (int a)
{
   if (a % 2 == 0) { f3 (2*a); };
}
public static void f3 (int b)
{
   if (b > 0) { f4(); } else { f5(); };
}
public static void f4() {... f6()....}
public static void f5() {... f6()....}
public static void f6() {...}
```

Use the following test inputs:

- $t1 = f1\ (0,\ 0)$
- $t2 = f1\ (1,\ 1)$
- $t3 = f1\ (0,\ 1)$
- $t4 = f1\ (3,\ 2)$
- $t5 = f1\ (3,\ 4)$

(a) Draw the call graph for this program fragment.
(b) Give the path in the graph followed by each test.
(c) Find a minimal test set that achieves Node Coverage.
(d) Find a minimal test set that achieves Edge Coverage.
(e) Give the (maximal) prime paths in the graph. Give the maximal prime path that is not covered by any of the test paths above.

4. Use the following methods **trash()** and **takeOut()** to answer questions a-c.

```
 1 public void trash (int x)     15 public int takeOut (int a, int b)
 2 {                             16 {
 3    int m, n;                  17    int d, e;
 4                               18
 5    m = 0;                     19    d = 42*a;
 6    if (x > 0)                 20    if (a > 0)
 7       m = 4;                  21       e = 2*b+d;
 8    if (x > 5)                 22    else
 9       n = 3*m;                23       e = b+d;
10    else                       24    return (e);
11       n = 4*m;                25 }
12    int o = takeOut (m, n);
13    System.out.println ("o is: " + o);
14 }
```

(a) Give all call sites using the line numbers given.
(b) Give all pairs of *last-def*s and *first-use*s.
(c) Provide test inputs that satisfy *all-coupling-uses* (note that **trash()** only has one input).

7.5 Graph Coverage for Specifications

Testers can also use software specifications as sources for graphs. The literature presents many techniques for generating graphs and criteria for covering those graphs, but most of them are in fact very similar. We begin by looking at graphs based on *sequencing constraints* among methods in classes, then graphs that represent state behavior of software.

7.5.1 Testing Sequencing Constraints

We pointed out in Section 7.4.1 that call graphs for classes often wind up being disconnected and in many cases, such as with small Abstract Data Types (ADTs), methods in a class share no calls at all. However, the order of calls is almost always constrained by rules. For example, many ADTs must be initialized before being used, we cannot pop an element from a stack until something has been pushed onto it, and we cannot remove an element from a queue until an element has been put on it. These rules impose constraints on the order in which methods may be called. Generally, a *sequencing constraint* is a rule that imposes restriction on the order in which certain methods may be called.

Sequencing constraints are sometimes explicitly expressed, sometimes implicitly expressed, and sometimes not expressed at all. Sometimes they are encoded as a precondition or other specification, but not directly as a sequencing condition. For example, consider the following precondition for the common `DeQueue()` on a queue ADT:

```
public int DeQueue ()
{
// Pre: At least one element must be on the queue.
.
:
public EnQueue (int e)
{
// Post: e is on the end of the queue.
```

Although it is not said explicitly, a programmer can infer that the only way an element can "be on the queue" is if `EnQueue()` has previously been called. Thus, an implicit sequencing constraint occurs between `EnQueue()` and `DeQueue()`.

Of course, formal specifications can help make the relationships more precise. A wise tester will certainly use formal specifications when available, but a responsible tester must look for formal relationships even when they are not explicitly stated. Also, note that sequencing constraints do not capture all the behavior, but only abstract certain key aspects. The sequence constraint that `EnQueue()` must be called before `DeQueue()` does not capture

the fact that if we only `EnQueue()` one item, and then try to `DeQueue()` two items, the queue will be empty. The precondition may capture this fact, but usually not in a formal way that automated tools can use. This kind of relationship is beyond the ability of a simple sequencing constraint but can be dealt with by some of the state behavior techniques in the next section.

This relationship is used in two places during testing. We illustrate them with a small example of a class that encapsulates operations on a file. Our class `FileADT` will have three methods:

- `open (String fName)` // Opens the file with the name `fName`
- `close (String fName)` // Closes the file and makes it unavailable for use
- `write (String textLine)` // Writes a line of text to the file

This class has several sequencing constraints. The following statements use "must" and "should" in very specific ways. When "must" is used, it implies that violation of the constraint is a fault. When "should" is used, it implies that violation of the constraint is a potential fault, but the software will not necessarily fail.

1. An `open(F)` **must** be executed before every `write(t)`
2. An `open(F)` **must** be executed before every `close()`
3. A `write(t)` **must** not be executed after a `close()` unless an `open(F)` appears in between
4. A `write(t)` **should** be executed before every `close()`
5. A `close()` **must** not be executed after a `close()` unless an `open(F)` appears in between
6. An `open(F)` **must** not be executed after an `open(F)` unless a `close()` appears in between

Constraints are used in testing in two ways to evaluate software that uses the class (a "client"), based on the CFG of Section 7.3.1. Consider the two (partial) CFGs in Figure 7.36, representing two units that use `FileADT`. We can use this graph to test the use of the `FileADT` class by checking for sequence violations. This can be done both statically and dynamically.

Static checks (not considered to be traditional testing) proceed by checking each constraint. First consider the `write(t)` statements at nodes 2 and 5 in graph (a). We can check to see whether paths exist from the `open(F)` at node 1 to nodes 2 and 5 (constraint 1). We can also check whether a path exists from the `open(F)` at node 1 to the `close()` at node 6 (constraint 2). For constraints 3 and 4, we can check to see if a path goes from

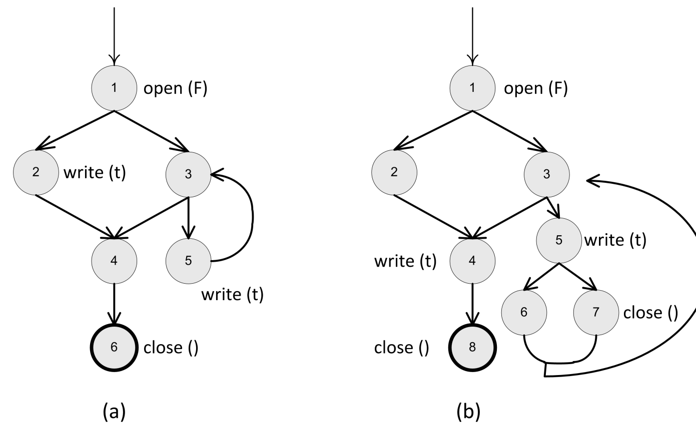


Figure 7.36: Control flow graph using the File ADT.

the `close()` at node 6 to any of the `write(t)` statements, and see if a path exists from the `open(F)` to the `close()` that does not go through at least one `write(t)`. This will uncover one possible problem, the path [1, 3, 4, 6] goes from an `open(F)` to a `close()` with no intervening `write(t)` calls.

For constraint 5, we can check if a path exists from a `close()` to a `close()` that does not go through an `open(F)`. For constraint 6, we can check if a path exists from an `open(F)` to an `open(F)` that does not go through a `close()`.

This process will find a more serious problem with graph (b) in 7.36. A path exists from the `close()` at node 7 to the `write(t)` at node 5 and to the `write(t)` at node 4. While this may seem simple enough not to require formalism for such small graphs, this process is quite difficult with large graphs containing dozens or hundreds of nodes.

Dynamic testing follows a slightly different approach. Consider the problem in graph (a) where no `write()` appears on the possible path [1, 3, 4, 6]. It is quite possible that the logic of the program dictates that the edge (3, 4) can *never* be taken unless the loop [3, 5, 3] is taken at least once. Because deciding whether the path [1, 3, 4, 6] can be taken or not is formally undecidable, this situation can be checked only by executing the program. Thus we generate test requirements to try to *violate* the sequencing constraints. For the `FileADT` class, we generate the following sets of test requirements:

1. Cover every path from the start node to every node that contains a `write(t)` such that the path does not go through a node containing an `open(F)`.

2. Cover every path from the start node to every node that contains a `close()` such that the path does not go through a node containing an `open(F)`.

3. Cover every path from every node that contains a `close()` to every node that contains a `write(t)` such that the path does not contain an `open(F)`.

4. Cover every path from every node that contains an `open(F)` to every node that contains a `close()` such that the path does not go through a node containing a `write(t)`.

5. Cover every path from every node that contains an `open(F)` to every node that contains an `open(F)`.

Of course, all of these test requirements will be infeasible in well written programs. However, any tests created as a result of these requirements will almost certainly reveal a fault if one exists.

7.5.2 Testing State Behavior of Software

The other major method for using graphs based on specifications is to model state behavior of the software by developing some form of finite state machine (FSM). Over the last 25 years, many suggestions have been made for creating FSMs and how to test software based on the FSM. The topic of how to create, draw, and interpret an FSM has filled entire textbooks, and authors have gone into great depth and effort to define what exactly goes into a state, what can go onto edges, and what causes transitions. Rather than using any particular language, we choose to define a very generic model for FSMs that can be adapted to virtually any notation. These FSMs are essentially graphs, and the graph testing criteria already defined can be used to test software that is based on the FSM.

One advantage of basing tests on FSMs is that huge numbers of practical software applications are based on an FSM model, or can be modeled as FSMs. Virtually all embedded software fits in this category, including software in remote controls, household appliances, watches, cars, cell phones, airplane flight guidance, traffic signals, railroad control systems, network routers, and factory automation. Indeed, most software can be modeled with FSMs, the primary limitation being the number of states needed to model the software. Word processors, for example, contain so many commands and states that modeling them as FSMs may be impractical.

Creating FSMs often has great value. If the test engineer creates an FSM to describe existing software, he or she will almost certainly detect design flaws. Some would even argue the converse; if the designers created FSMs, the testers should not bother creating them because problems will be rare. This would probably be true if programmers were perfect.

FSMs can be annotated with different types of actions, including actions on transitions, entry actions on nodes, and exit actions on nodes. Many languages are used to describe FSMs, including UML statecharts, finite automata state tables (SCR), and Petri nets. This book presents examples with basic features that are common to many languages. It is closest to UML statecharts, but not exactly the same.

A *Finite State Machine* is a graph whose nodes represent states in the execution behavior of the software and edges represent transitions among the states. A *state* represents a recognizable situation that remains in existence over some period of time. A state is defined by specific values for a set of variables; as long as those variables have those values the software is considered to be in that state. (Note that these variables are defined at the design modeling level and may not necessarily correspond to variables in an implementation.) A *transition* is thought of as occurring in zero time and usually represents a change to the values of one or more variables. When the variables change, the software is considered

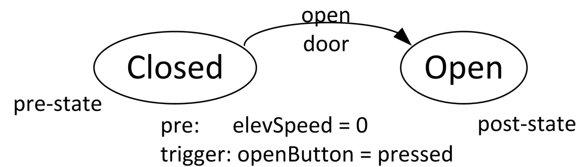


Figure 7.37: Elevator door open transition.

to move from the transition's *pre-state* (predecessor) to its *post-state* (successor). (If a transition's pre-state and post-state are the same, then values of state variables will not change.) FSMs often define *preconditions* or *guards* on transitions, which define values that specific variables must have for the transition to be enabled, and *triggering events*, which are changes in variable values that cause the transition to be taken. A triggering event "triggers" the change in state. For example, the modeling language SCR calls these WHEN conditions and triggering events. The values the triggering events have before the transition are called *before-values*, and the values after the transition are called *after-values*. When graphs are drawn, transitions are often annotated with the guards and the values that change.

Figure 7.37 illustrates this model with a simple transition that opens an elevator door. If the elevator button is pressed (the triggering event), the door opens only if the elevator is not moving (the precondition, $elevSpeed = 0$).

When preparing FSMs for testing, it is important to note that FSMs do not necessarily have final nodes. They often represent the behavior of a device that runs for a long time, ideally forever, like with the watch in the following subsection. But a test graph that abstracts an FSM needs initial and final nodes so we can derive test paths. Sometimes it is pretty much arbitrary which nodes are designated as initial and final.

Given this type of graph, many of the previous criteria can be defined directly. Node Coverage requires that each state in the FSM be visited at least once and is called *State Coverage*. Edge Coverage is applied by requiring that each transition in the FSM be visited at least once, which is called *Transition Coverage*. The Edge-Pair Coverage criterion was originally defined for FSMs and is also called *transition-pair* and *two-trip*.

The data flow coverage criteria are a bit more troublesome for FSMs. In most formulations of FSMs, nodes are not allowed to have defs or uses of variables. That is, all of the action is on the transitions. Unlike with code-based graphs, different edges from the same node in an FSM need not have the same set of defs and uses. In addition, the semantics of the triggers imply that the effects of a change to the variables involved are felt immediately by taking a transition to the next state. That is, defs of triggering variables immediately reach uses.

Thus, the All-Defs and All-Uses criteria can only be applied meaningfully to variables involved in guards. This also brings out a more practical problem, which is that the FSMs do not always model assignment to all variables. That is, the uses are clearly marked in the FSM, but defs are not always easy to find. Because of these reasons, few attempts have been made to apply data flow criteria to FSMs.

Deriving Finite State Machine Graphs

One difficulty of applying graph techniques to FSMs is deriving the FSM model in the first place. As we said earlier, FSM models of the software may or may not already exist. If not, the tester can dramatically increase his or her understanding of the software by deriving the FSMs. However, it is not necessarily obvious how to derive an FSM, so we offer some suggestions. This is not a complete tutorial on constructing FSMs; indeed, complete texts exist on the subject and we recommend that the interested reader study them.

This section offers simple and straightforward suggestions to help readers who are unfamiliar with FSMs get started and avoid some of the more obvious mistakes. We offer the suggestions in terms of a running example, the class `Watch` in Figures 7.38 and 7.39. Class `Watch` implements part of a digital watch, using inner class `Time`.

Classes `Watch` and `Time` each have one interesting method, `doTransition()` and `changeTime()`. When left to their own devices, students will usually pick one of four strategies for generating FSMs from code. Unfortunately, the first two are not effective or recommended. Each of these is discussed in turn.

1. Combining control flow graphs
2. Using the software structure
3. Modeling state variables
4. Using the implicit or explicit specifications

1. Combining control flow graphs: For programmers who have little or no knowledge of FSMs, this is often the most natural approach to deriving FSMs. Our experience has been that the majority of students will use this approach if not guided away from it. A control flow graph-based "FSM" for class `Watch` is given in Figure 7.40.

The graph in Figure 7.40 is **not** an FSM at all and this is not the way to form graphs from software. This method has several problems, the first being that the nodes are not states. The methods must return to the appropriate call sites, which means that the graphs contain built-in non-determinism. For example, Figure 7.40 has three edges from node 12 in `changeTime()`, to nodes 6, 8, and 10 in `doTransition()`. Which edge is taken depends on whether `changeTime()` was entered from node 6, 8, or 10 in `doTransition()`. A second problem is the implementation must be finished before the graph can be built; remember from Chapter 1 that one of our goals is to prepare tests as early as possible. Most importantly, however, this kind of graph does not scale to large software products. The graph is complicated enough with small `Watch`, and gets much worse with larger programs.

2. Using the software structure: A more experienced programmer may consider the overall flow of operations in the software. This might lead to something like the graph in Figure 7.41, where methods are mapped to states.

```
public class Watch
{
   // Constant values for the button (inputs)
   private static final int NEXT = 0;
   private static final int UP   = 1;
   private static final int DOWN = 2;

   // Constant values for the state
   private static final int TIME      = 5;
   private static final int STOPWATCH = 6;
   private static final int ALARM     = 7;

   // Primary state variable
   private int mode = TIME;

   // Three separate times, one for each state
   private Time watch, stopwatch, alarm;

   // Inner class keeps track of hours and minutes
   public class Time
   {
      private int hour   = 0;
      private int minute = 0;

      // Increases or decreases the time.
      // Rolls around when necessary.
      public void changeTime (int button)
      {
         if (button == UP)
         {
            minute += 1;
            if (minute >= 60)
            {
               minute = 0;
               hour += 1;
               if (hour >= 12)
                  hour = 0;
            }
         }
         else if (button == DOWN)
         {
            minute -= 1;
            if (minute < 0)
            {
               minute = 59;
               hour -= 1;
               if (hour <= 0)
                  hour = 12;
            }
         }
      }  // end changeTime()

      public String toString ()
      {
         return (hour + ":" + minute);
      }  // end toString()
   }  // end class Time
```

Figure 7.38: Watch – Part A.

```
   public Watch () // Constructor
   {
      watch = new Time();
      stopwatch = new Time();
      alarm = new Time();
   }  // end Watch constructor

   public String toString ()  // Converts values
   {
      return ("watch is: " + watch + "\n"
              + "stopwatch is: " + stopwatch + "\n"
              + "alarm is: " + alarm);
   }  // end toString()

   public void doTransition (int button) // Handles inputs
   {
      switch (mode)
      {
         case TIME:
            if (button == NEXT)
               mode = STOPWATCH;
            else
               watch.changeTime (button);
            break;
         case STOPWATCH:
            if (button == NEXT)
               mode = ALARM;
            else
               stopwatch.changeTime (button);
            break;
         case ALARM:
            if (button == NEXT)
               mode = TIME;
            else
               alarm.changeTime (button);
            break;
         default:
            break;
      }
   }  // end doTransition()
}  // end Watch
```

Figure 7.39: Watch – Part B.

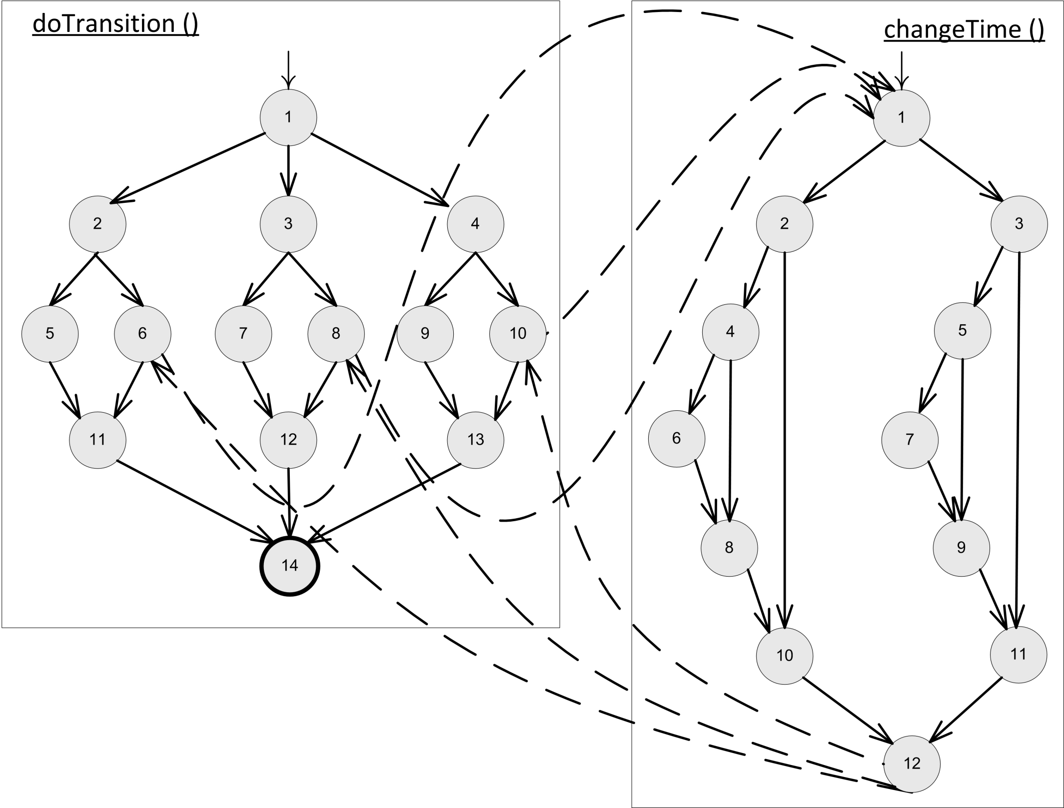


Figure 7.40: An FSM representing Watch, based on control flow graphs of the methods.

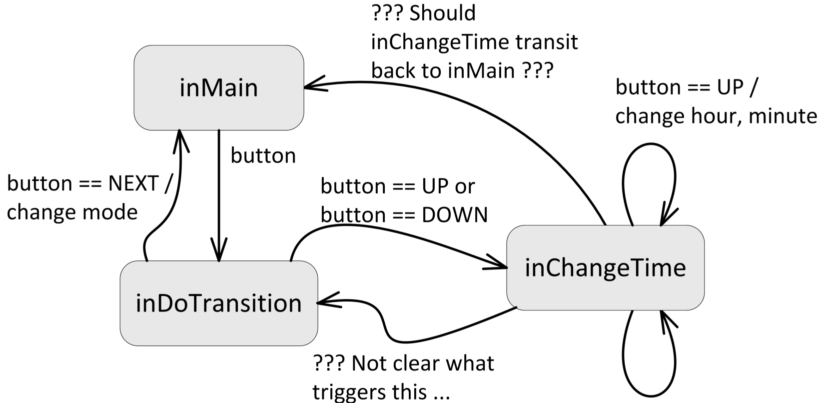


Figure 7.41: An FSM representing Watch, based on the structure of the software.

Although an improvement over the control flow graph, methods are not really states. This kind of derivation is also very subjective, meaning different testers will draw different graphs, introducing inconsistency in the testing. It also requires in-depth knowledge of the software, is not possible until the detailed design is ready, and is hard to scale to large programs.

3. Modeling state variables: A more mechanical method for deriving FSMs is to consider the values of the state variables in the program. These are usually defined early in design. The first step is to identify the state variables, then choose which are actually relevant to the FSM (for example, global and class variables).

The class level variables in `Watch` can be divided into constants (NEXT, UP, DOWN, TIME, STOPWATCH, and ALARM) and non-constants (`mode`, `watch`, `stopwatch`, and `alarm`). The constants are not relevant to defining the state of a watch and should be omitted from the model. The three variables of time `Time` (`watch`, `stopwatch`, and `alarm`) are objects. The can be modeled hierarchically, but we choose to replace them with the state variables in class `Time` (`hour` and `minute`). Thus we model the states with `mode`, `watch::hour`, `watch::minute`, `stopwatch::hour`, `stopwatch::minute`, `alarm::hour`, and `alarm::minute`.

Theoretically, each combination of values for the state variables defines a different state. In practice however, this can result in a very large number of states; even infinite for some programs. For example, `mode` can have only three values, but the `hour` and `minute` variables are of type *int*, so can be considered to have an infinite number of values. Alternatively, since they represent units of time, each `minute` could be assumed to have 60 possible values, and each `hour` could be assumed to have 24 possible values. Even this simplification results in $1440 * 1440 * 1440 * 3 = 8,957,952,000$ possible states!

This is clearly too many, so we further simplify the model. First, instead of representing 1440 values for each `Time` object, we combine values into groups that are similar semantically. In this example, we assume that the rollovers at noon and midnight are special cases, as are the rollovers from one hour to the next. This leads to choosing the value 12:00, and the ranges 12:01 ... 12:59 and 01:00 ... 11:59. That is:

```
mode: TIME, STOPWATCH, ALARM
watch : 12:00, 12:01...12:59, 01:00...11:59
stopwatch : 12:00, 12:01...12:59, 01:00...11:59
alarm : 12:00, 12:01...12:59, 01:00...11:59
```

This results in $3*3*3*3 = 81$ states. Our next observation is that the `mode` is not really independent of the three `Time` objects. For example, if `mode` == `TIME`, only the `watch` is relevant. So we only really care about $3 + 3 + 3 = 9$ states.

The resulting FSM is shown in Figure 7.42. Actually, Figure 7.42 does not show two kinds of transitions. The state (`mode` = `TIME`; `watch` = `12:00`) has three outgoing transitions on `next`, one each to the states where `mode` = `STOPWATCH`. In the complete FSM, each state would have three outgoing transitions on `next`. We omit those transitions because they make the figure hard to read.

Second, the states with a range of values for watch should have "self-loops," that is, transitions back to themselves on UP and DOWN. Some FSM styles say to omit these self-loops, whereas others say to include them. If a variable is changed but does not put the FSM into a new state, then the self loop can be assumed. But sometimes it is valuable to include them. When our goal is to transform an FSM into a generic graph, and then derive tests, it might be useful to include transitions from a state to itself. Both of these situations are illustrated with dashed lines in the figure.

Having three outgoing transitions on `next` introduces a form of non-determinism into the graph, but it is important to note that this non-determinism is not reflected in the implementation. During execution, which transition is taken depends on the current state of the other `Time` object. The 81-state model would not have this non-determinism, and whether to have a smaller, non-deterministic, model or a larger, deterministic, model is an important test design decision. This situation could also be handled by a hierarchy of FSMs, where each watch is in a separate FSM and they are organized together.

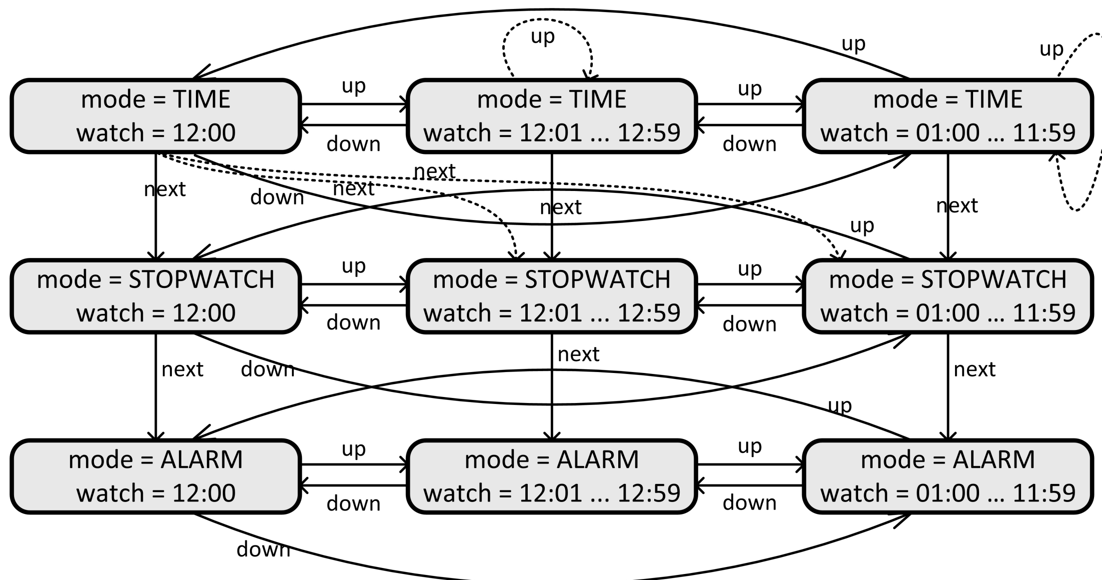


Figure 7.42: An FSM representing Watch, based on modeling state variables. This figure omits numerous transitions. The full diagram would have transitions on `next` from each node to three other nodes, two of which are shown as dotted line examples.

The mechanical process of this strategy is appealing because we can expect different testers to derive the same or similar FSMs. This strategy also does not have the disadvantages of the first two methods. It is not yet possible at this time to completely automate this process because of the difficulty of determining transitions from the source and because the decision of which variables to model requires judgment. The software is not necessary for this diagram to be derived, but the design is needed. The FSMs that are derived by modeling state variables may not completely reflect the software.

4. Using the implicit or explicit specifications: The last method for deriving FSMs relies on explicit requirements or formal specifications describing the software's behavior. A natural language specification for the `Watch` is:

> **Specification for class Watch**
> Class `Watch` will store and update time for three watches: the current time, the stopwatch, and an alarm. It will implement behavior for three external buttons. A `next` button will change the `Watch` from the current time, to the stopwatch, to the alarm, and back to the current time. An `up` button will increase the time by one minute for the current watch. A `down` button will decrease the time by one minute for the current watch. The watches will function in 12-hour format, that is, the hours will be from 1 to 12.

These requirements will lead to an FSM that looks very much like the FSM in Figure 7.42 that models state variables. FSMs based on specifications are usually cleaner and easier to understand. If the software is designed well, this type of FSM should contain the same information that UML statecharts contain.

Exercises, Section 7.5.

1. Use the class **BoundeQueue2()** for questions a-f below. A compilable version is available on the book website in the file `BoundedQueue2.java`. The queue is managed in the usual circular fashion.

   Suppose we build an FSM where states are defined by the representation variables of **BoundedQueue2**. That is, a state is a 4-tuple defined by the values for $[elements,\ size,\ front,\ back]$. For example, the initial state has the value $[[null,\ null],\ 0,\ 0,\ 0]$, and the state that results from pushing an object obj onto the queue in its initial state is $[[obj,\ null],\ 1,\ 0,\ 1]$.

   (a) We do not care which specific objects are in the queue. Consequently, there are really just four useful values for the variable *elements*. Enumerate them.
   (b) How many states are there?
   (c) How many of these states are reachable?
   (d) Show the reachable states in a drawing.
   (e) Add edges for the *enqueue()* and *dequeue()* methods. (For this assignment, ignore the exceptional returns, although you should observe that when exceptional returns are taken, none of the instance variables are modified.)
   (f) Define a small test set that achieves Edge Coverage. Implement and execute this test set. You might find it helps to write a method that shows the internal variables at each call.

2. For the following questions a-c, consider the FSM that models a (simplified) programmable thermostat. Suppose the variables that define the state and the methods that transition between states are:

```
partOfDay : {Wake, Sleep}
temp      : {Low, High}

// Initially "Wake" at "Low" temperature

// Effects: Advance to next part of day
public void advance();

// Effects: Make current temp higher, if possible
public void up();

// Effects: Make current temp lower, if possible
public void down();
```

(a) How may states are there?
(b) Draw and label the states (with variable values) and transitions (with method names). Notice that all of the methods are total, that is, their behaviors are defined for all possible inputs.
(c) A test case is simply a sequence of method calls. Provide a test set that satisfies Edge Coverage on your graph.

7.6 Graph Coverage for Use Cases

UML use cases are widely used to clarify and express software requirements. They are meant to describe sequences of actions that software performs as a result of inputs from the users, that is, they help express the *workflow* of a computer application. Because use cases are developed early in software development, they can help the tester start testing activities early.

Many books and papers can help the reader develop use cases. As with FSMs, it is not the purpose of this book to explain how to develop use cases, but how to use them to create useful tests. The technique for using graph coverage criteria to develop tests from use cases is expressed through an example.

Figure 7.43 shows three common use cases for an automated teller machine (ATM). In use cases, *actors* are humans or other software systems that use the software being modeled. They are drawn as simple stick figures. In Figure 7.43, the actor is an ATM customer who has three potential use cases; `Withdraw Funds`, `Get Balance`, and `Transfer Funds`.

While Figure 7.43 is a graph, it is not a very useful graph for testing. About the best we could do as a tester is to use Node Coverage, which amounts to "try each use case once." However, use cases are usually elaborated, or "documented" with a more detailed textual description. The description describes the details of operation and includes *alternatives*,

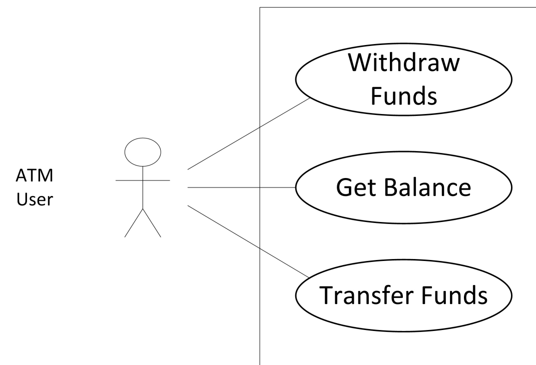


Figure 7.43: ATM actor and use cases.

which model choices or conditions during execution. The `Withdraw Funds` use case from Figure 7.43 can be described as follows:

Use Case Name: Withdraw Funds
Summary: Customer uses a valid card to withdraw funds from a valid bank account.
Actor: ATM Customer
Precondition: ATM is displaying the idle welcome message
Description:

1. Customer inserts an ATM Card into the ATM Card Reader.
2. If the system can recognize the card, it reads the card number.
3. System prompts the customer for a PIN.
4. Customer enters PIN.
5. System checks the expiration date and whether the card has been stolen or lost.
6. If card is valid, the system checks whether the PIN entered matches the card PIN.
7. If the PINs match, the system finds out what accounts the card can access.
8. System displays customer accounts and prompts the customer to choose a type of transaction. Three types of transactions are Withdraw Funds, Get Balance, and Transfer Funds.
   The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.
9. Customer selects Withdraw Funds, selects account number, and enters the amount.
10. System checks that the account is valid, makes sure that the customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
11. If all four checks are successful, the system dispenses the cash.
12. System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
13. System ejects card.

14. System displays the idle welcome message.

Alternatives:

- If the system cannot recognize the card, it is ejected and a welcome message is displayed.
- If the current date is past the card's expiration date, the card is confiscated and a welcome message is displayed.
- If the card has been reported lost or stolen, it is confiscated and a welcome message is displayed.
- If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
- If the customer enters an incorrect PIN three times, the card is confiscated and a welcome message is displayed.
- If the account number entered by the user is invalid, the system displays an error message, ejects the card, and a welcome message is displayed.
- If the request for withdrawal exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card, and a welcome message is displayed.
- If the request for withdrawal exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card, and a welcome message is displayed.
- If the customer enters Cancel, the system cancels the transaction, ejects the card, and a welcome message is displayed.
- If the request for withdrawal exceeds the amount of funds in the account, the system displays an apology message, cancels the transaction, ejects the card, and a welcome message is displayed.

Postcondition: Funds have been withdrawn from the customer's account.

At this point, some testing students will be wondering why this discussion is included in a chapter on graph coverage. That is, there is little obvious relationship with graphs thus far. We want to reiterate the first phrase in Beizer's admonition: "testers find a graph, then cover it." In fact, there is a nice graph structure in the use case textual description, which may be up to the tester to express. This graph can be modeled as the Transaction Flow Graphs in Beizer's Chapter 4, or can be drawn as a UML Activity Diagram.

An activity diagram shows the flow among activities. Activities can be used to model a variety of things, including state changes, returning values, and computations. We advocate using them to model use cases as graphs by considering activities as *user level steps*. Activity diagrams have two kinds of nodes, action states and sequential branches[6].

We construct activity graphs as follows. The numeric items in the use case **Description** express steps that the actors undertake. These correspond to inputs to or outputs from the

[6]As in previous chapters, we explicitly leave out concurrency, so concurrent forks and joins are not considered.

software and appear as **nodes** in the activity diagram as action states. The **Alternatives** in the use case represent decisions that the software or actors make and are represented as **nodes** in the activity diagram as sequential branches.

The activity diagram for the withdraw funds scenario is shown in Figure 7.44. Several things are **expected** but not **required** of activity diagrams constructed from use cases. First, they usually do not have many loops, and most loops they do contain are tightly bounded or determinate. For example, the graph in Figure 7.44 contains a three-iteration loop when the PIN is entered incorrectly. This means that Complete Path Coverage is often feasible and sometimes reasonable. Second, it is very rare to see a complicated predicate that contains multiple clauses. This is because the use case is usually expressed in terms that the users can understand. This means that the logic coverage criteria in Chapter 8 are usually not useful. Third, there are no obvious data definition-use pairs. This means that data flow coverage criteria are not applicable.

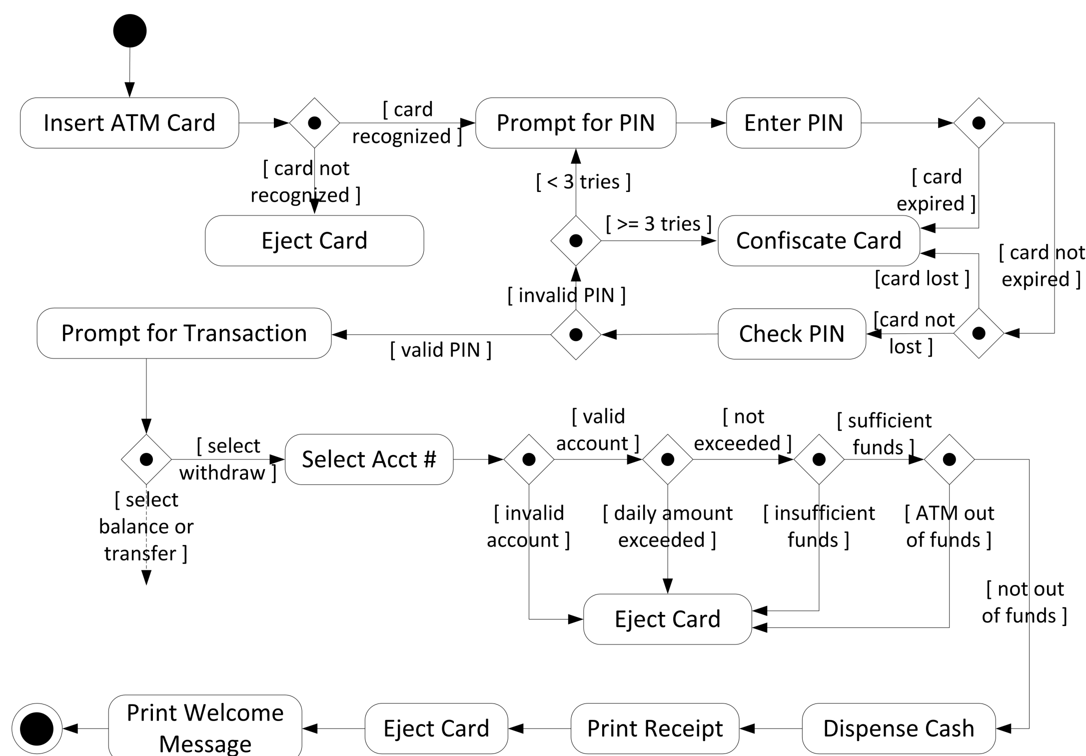


Figure 7.44: Activity graph for ATM withdraw funds.

The two criteria that are most obviously applicable to use case graphs are Node Coverage and Edge Coverage. Test case values are derived from interpreting the nodes and predicates as inputs to the software. One other criterion for use case graphs is based on the notion of "scenarios."

7.6.1 Use Case Scenarios

A use case scenario is an *instance* of, or a complete path through, a use case. A scenario should make some sense semantically to the users and is often derived when the use cases are constructed. If the use case graph is finite (as is usually the case), then it is possible to list all possible scenarios. However, domain knowledge can be used to reduce the number of scenarios that are useful or interesting from either a modeling or test case perspective. Note that *Specified Path Coverage*, defined at the beginning of this chapter, is exactly what we want here. The set S for Specified Path Coverage is simply the set of all scenarios.

If the tester or requirements writer chooses all possible paths as scenarios, then Specified Path Coverage is equivalent to Complete Path Coverage. The scenarios are chosen by people and they depend on domain knowledge. Thus it is **not** guaranteed that Specified Path Coverage subsumes Edge Coverage or Node Coverage. That is, it is possible to choose a set of scenarios that do not include every edge. This would probably be a mistake, however. So in practical terms, Specified Path Coverage can be expected to cover all edges.

Exercises, Section 7.6.

1. Construct two separate use cases and use case scenarios for interactions with a bank Automated Teller Machine. Do not try to capture all the functionality of the ATM into one graph; think about two different people using the ATM and what each one might do.

   Design test cases for your scenarios.

7.7 Bibliographic Notes

During the research for the first edition of this book, one thing that became abundantly clear is that this field has had a significant amount of parallel discovery of the same techniques by people working independently. Some individuals have discovered various aspects of the same technique, which were subsequently polished into very pretty test criteria. Others have invented the same techniques, but based them on different types of graphs or used different names. Thus, ascribing credit for software testing criteria is a perilous task. We do our best, but claim only that the bibliographic notes in this book are starting points for further study in the literature.

The research into covering graphs seems to have started with generating tests from finite state machines (FSMs), which has a long and rich history. Some of the earliest papers were in the 1970s [15, 42, 44, 57, 68]. The primary focus of most of these papers was on using FSMs to generate tests for telecommunication systems that were defined with standard finite automata, although much of the work pertained to general graphs. The control flow graph seems to have been invented (or should it be termed "discovered"?) by Legard in 1975

[53]. In papers published in 1975, Huang [44] suggested covering each edge in the FSM, and Howden [42] suggested covering complete trips through the FSM, but without looping. In 1976, McCabe [57] suggested the same idea on control flow graphs as the primary application of his cyclomatic complexity metric. In 1976, Pimont and Rault [68] suggested covering pairs of edges, or "switches," a technique that they referred to as "switch-testing," and which has also been called "switch cover." In 1978, Chow [15] suggested generating a spanning tree from an FSM and then basing test sequences on paths through this tree. He also generalized the idea of a switch to "n-switch," which are sequences of n edges. Fujiwara et al. [32] referred to Chow's approach with the term "W-method," and developed the "partial" W-method (the "Wp-method"). They also attributed the idea of switches to Chow's paper instead of Pimont and Rault's. The idea of covering pairs of edges was rediscovered in the 1990s. The British Computer Society Standard for Software Component Testing called it *two-trip* [11] and Offutt et al. [62], called it *transition-pair*.

Other test generation methods based on FSMs include tour [58], the distinguished sequence method [35], and unique input-output method [72]. Their objectives are to detect output errors based on state transitions driven by inputs. FSM-based test generation has been used to test a variety of applications including lexical analyzers, real-time process control software, protocols, data processing, and telephony. One early realization when developing the first edition of this book is that the criteria for covering finite state machines are not substantially different from criteria for other graphs.

This book has introduced the explicit inclusion of Node Coverage requirements in Edge Coverage requirements (the "up to" clause). This inclusion is not necessary for typical control flow graphs, where, indeed, subsumption of Node Coverage by Edge Coverage is often presented as a basic theorem, but is often required for graphs derived from other artifacts.

Several later papers focused on automatic test data generation to cover structural elements in the program [8, 9, 16, 21, 23, 43, 48, 50, 61, 69]. Much of this work was based on the analysis techniques of symbolic evaluation [13, 18, 20, 21, 22, 42], dynamic symbolic evaluation [61, 49, 50], and slicing [73, 74]. Some of these ideas are discussed in **Chapter ???.** NEED TO ADD CONCOLIC AND SEACH-BASED

The problem of handling loops has plagued graph-based criteria from the beginning. It seems obvious that we want to cover paths, but loops create infinite numbers of paths. In Howden's 1975 paper [42], he specifically addressed loops by covering complete paths "without looping," and Chow's 1978 suggestion to use spanning trees was an explicit attempt to avoid having to execute loops [15]. Binder's book [7] used the technique from Chow's paper, but changed the name to *round trip*, which is the name used in this book.

Another early suggestion was based on testing loop free programs [14], which is certainly interesting from a theoretical view, but not particularly practical.

White and Wiszniewski [76] suggested limiting the number of loops that need to be executed based on specific patterns. Weyuker, Weiss and Hamlet tried to choose specific loops to test based on data definitions and uses [75].

The notion of *subpath sets* was developed by Offutt et al. [46, 60] to support inter-class

path testing and is essentially equivalent to tours with detours as presented here. Prime paths were introduced in an unpublished manuscript by Ammann and Offutt in 2004, and first appeared in the research literature in an experimental comparison paper by Li, Praphamontripong, and Offutt [54]. The ideas of touring, sidetrips and detours were introduced in the first edition of this book.

The earliest reference we have found on data flow testing was a technical report in 1974 by Osterweil and Fosdick [67]. This technical report was followed by a 1976 paper in ACM Computing Surveys [25], along with an almost simultaneous publication by Herman in the Australian Computer Journal [39]. The seminal data flow analysis procedure (without reference to testing) was due to Allen and Cocke [5].

Other fundamental and theoretical references are by Laski and Korel in 1983 [52], who suggested executing paths from definitions to uses, Rapps and Weyuker in 1985 [70], who defined criteria and introduced terms such as All-Defs and All-Uses, and Frankl and Weyuker in 1988 [31]. These papers refined and clarified the idea of data flow testing, and are the basis of the presentation in this text. Stated in the language in this text, [31] requires direct tours for the All-du-Paths Coverage, but allows sidetripsfor All-Defs Coverage and All-Uses Coverage. This text allows sidetrips (or not) for all of the data-flow criteria. The pattern matching example used in this text has been employed in the literature for decades; as far as we know, Frankl and Weyuker [31] were the first to use the example to illustrate data flow coverage.

Forman also suggested a way to detect data flow anomalies without running the program [24].

Some detailed problems with data flow testing have been recurring. These include the application of data flow when paths between definitions and uses cannot be executed [30], and handling pointers and arrays [61, 75].

The method of defining data flow criteria in terms of sets of du-paths is original to this book, as is the explicit suggestion for Best Effort touring.

Many papers present empirical studies of various aspects of data flow testing. One of the earliest was by Clarke, Podgurski, Richardson and Zeil, who compared some of the different criteria [17]. Comparisons with mutation testing (introduced in Chapter 9) started with Mathur in 1991 [55], which was followed by Mathur and Wong [56], Wong and Mathur [77], Offutt, Pan, Tewary and Zhang [63], and Frankl, Weiss and Hu [28]. Comparisons of data flow with other test criteria have been published by Frankl and Weiss [27], Hutchins, Foster, Goradia and Ostrand [45], and Frankl and Deng [26].

Several tools have also been built by researchers to support data flow testing. Most worked by taking a program and tests as inputs, and deciding whether one or more data flow criteria have been satisfied (a *recognizer*). Frankl, Weiss and Weyuker built ASSET in the mid-80s [29], Girgis and Woodward built a tool to implement both data flow and mutation testing in the mid-80s [34], and Laski built STAD in the late-80s [51]. Researchers at Bellcore developed the ATAC data flow tool for C programs in the early '90s [40, 41], and the first tool that included a test data generator for data flow criteria was built by Offutt, Jin and Pan in the late '90s [61].

Coupling was first discussed as a design metric by Constantine and Yourdon [19] and its use for testing was introduced implicitly by Harrold, Soffa and Rothermel [36, 37] and explicitly by Jin and Offutt [46], who introduced the use of *first-uses* and *last-defs*.

Kim, Hong, Cho, Bae and Cha used a graph-based approach to generate tests from UML state diagrams [47].

The USA's Federal Aviation Authority (FAA) has recognized the increased importance of modularity and integration testing by imposing requirements on structural coverage analysis of software that "the analysis should confirm the data coupling and control coupling between the code components" [71], pg. 33, section 6.4.4.2.

Data flow testing has also been applied to integration testing by Harrold and Soffa [37], Harrold and Rothermel [36], and Jin and Offutt [46]. This work focused on class-level integration issues, but did not address inheritance or polymorphism. Data flow testing has been applied to inheritance and polymorphism in object-oriented software by Alexander and Offutt [3, 2, 4], and Buy, Orso and Pezze [12, 66]. Gallagher and Offutt modeled classes as interacting state machines, and tested concurrency and communication issues among them [33].

Generating tests to satisfy sequencing constraints is due to Olender and Osterweil [64, 65].

SCR was first discussed by Henninger [38] and its use in model checking and testing was introduced by Atlee [6].

Constructing tests from UML diagrams is a more recent development, though relatively straightforward. It was first suggested by Abdurazik and Offutt [59, 1], and soon followed by Briand and Labiche [10]. This has since led to an entire field called model-based testing, with dozens of papers every year and workshops such as the annual workshop on model-based testing.

Bibliography

[1] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In *Proceedings of the Third International Conference on the Unified Modeling Language (UML '00)*, pages 383–395, York, England, October 2000.

[2] Roger T. Alexander and Jeff Offutt. Analysis techniques for testing polymorphic relationships. In *Proceedings of the Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99)*, pages 104–114, Santa Barbara CA, August 1999. IEEE Computer Society Press.

[3] Roger T. Alexander and Jeff Offutt. Criteria for testing polymorphic relationships. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 15–23, San Jose CA, October 2000. IEEE Computer Society Press.

[4] Roger T. Alexander and Jeff Offutt. Coupling-based testing of O-O programs. *Journal of Universal Computer Science*, 10(4):391–427, April 2004. http://www.jucs.org/jucs_10_4/coupling_based_testing_of.

[5] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–146, March 1976.

[6] J. M. Atlee. Native model-checking of SCR requirements. In *Fourth International SCR Workshop*, November 1994.

[7] Robert Binder. *Testing Object-oriented Systems*. Addison-Wesley Publishing Company Inc., New York, New York, 2000.

[8] Juris Borzovs, Audris Kalniņš, and Inga Medvedis. Automatic construction of test sets: Practical approach. In *Lecture Notes in Computer Science, Vol 502*, pages 360–432. Springer Verlag, 1991.

[9] R. S. Boyer, B. Elpas, and K. N. Levitt. Select-a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, June 1975. SIGPLAN Notices, vol. 10, no. 6.

[10] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. In *Proceedings of the Fourth International Conference on the Unified Modeling Language (UML '01)*, pages 194–208, Toronto, Canada, October 2001.

[11] Special Interest Group in Software Testing British Computer Society. *Standard for Software Component Testing, Working Draft 3.3*. British Computer Society, 1997. http://www.rmcs.cranfield.ac.uk/~cised/sreid/BCS_SIG/.

[12] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. In *Proceedings of the 2000 International Symposium on Software Testing, and Analysis (ISSTA '00)*, pages 39–48, Portland OR, August 2000. IEEE Computer Society Press.

[13] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering*, 5(4), July 1979.

[14] J. C. Cherniavsky. On finding test data sets for loop free programs. *Information Processing Letters*, 8(2):106–107, February 1979.

[15] T. Chow. Testing software designs modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[16] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[17] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15:1318–1332, November 1989.

[18] Lori A. Clarke and Debra J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software, Elsevier*, 5(1):15–35, January 1985.

[19] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice-Hall, Englewood Cliffs NJ, 1979.

[20] J. A. Darringer and J. C. King. Applications of symbolic execution to program testing. *IEEE Computer*, 11(4), April 1978.

[21] Richard A. DeMillo and Jeff Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[22] R. E. Fairley. An experimental program testing facility. *IEEE Transactions on Software Engineering*, SE-1:350–3571, December 1975.

[23] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering Methodology*, 5(1):63–86, January 1996.

[24] I. R. Forman. An algebra for data flow anomaly detection. In *Proceedings of the Seventh International Conference on Software Engineering*, pages 278–286. IEEE Computer Society Press, March 1984.

[25] L. D. Fosdick and L. J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[26] Phyllis G. Frankl and Yuetang Deng. Comparison of delivered reliability of branch, data flow and operational testing: A case study. In *Proceedings of the 2000 International Symposium on Software Testing, and Analysis (ISSTA '00)*, pages 124–134, Portland OR, August 2000. IEEE Computer Society Press.

[27] Phyllis G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[28] Phyllis G. Frankl, Steven N. Weiss, and Cang Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software, Elsevier*, 38(3):235–253, 1997.

[29] Phyllis G. Frankl, Stewart N. Weiss, and Elaine J. Weyuker. ASSET: A system to select and evaluate tests. In *Proceedings of the Conference on Software Tools*, New York NY, April 1985. IEEE Computer Society Press.

[30] Phyllis G. Frankl and Elaine J. Weyuker. Data flow testing in the presence of unexecutable paths. In *Proceedings of the Workshop on Software Testing*, pages 4–13, Banff, Alberta, July 1986. IEEE Computer Society Press.

[31] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[32] S. Fujiwara, G. Bochman, F. Khendek, M. Amalou, and A. Ghedasmi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[33] Leonard Gallagher, Jeff Offutt, and Tony Cincotta. Integration testing of object-oriented components using finite state machines. *Software Testing, Verification, and Reliability, Wiley*, 17(1):215–266, January 2007.

[34] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 313–319, London UK, August 1985. IEEE Computer Society Press.

[35] G. Gonenc. A method for the design of fault-detection experiments. *IEEE Transactions on Computers*, C-19:155–558, June 1970.

[36] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Symposium on Foundations of Software Engineering*, pages 154–163, New Orleans, LA, December 1994. ACM SIGSOFT.

[37] Mary Jean Harrold and Mary Lou Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[38] K. Henninger. Specifiying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[39] P. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, November 1976.

[40] J. R. Horgan and S. London. Data flow coverage and the C languages. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 87–97, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[41] J. R. Horgan and S. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, pages 2–10, New Orleans, LA, May 1992. IEEE Computer Society Press.

[42] W. E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Software Engineering*, SE-24, May 1975.

[43] W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[44] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, September 1975.

[45] Marlie Hutchins, H. Foster, Thomas Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[46] Zhenyi Jin and Jeff Offutt. Coupling-based criteria for integration testing. *Software Testing, Verification, and Reliability, Wiley*, 8(3):133–154, September 1998.

[47] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings-Software*, 146(4):187–192, August 1999.

[48] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[49] Bogdan Korel. A dynamic approach of test data generation. In *Conference on Software Maintenance-1990*, pages 311–317, San Diego, CA, 1990.

[50] Bogdan Korel. Dynamic method for software test data generation. *Software Testing, Verification, and Reliability, Wiley*, 2(4):203–213, 1992.

[51] Janusz Laski. Data flow testing in STAD. *Journal of Systems and Software, Elsevier*, 12:3–14, 1990.

[52] Janusz Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[53] H. Legard and M. Marcotty. A generalogy of control structures. *Communications of the ACM*, 18:629–639, Nov 1975.

[54] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Fifth IEEE Workshop on Mutation Analysis (Mutation 2009)*, Denver CO, April 2009.

[55] Aditya P. Mathur. On the relative strengths of data flow and mutation based test adequacy criteria. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, Portland OR, 1991. Lawrence and Craig.

[56] Aditya P. Mathur and W. Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification, and Reliability, Wiley*, 4(1):9–31, March 1994.

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

[58] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Proceedings Fault Tolerant Computing Systems*, pages 238–243. IEEE Computer Society Press, 1981.

[59] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. Springer-Verlag Lecture Notes in Computer Science Volume 1723.

[60] Jeff Offutt, Aynur Abdurazik, and Roger T. Alexander. An analysis tool for coupling-based integration testing. In *The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, pages 172–178, Tokyo, Japan, September 2000. IEEE Computer Society Press.

[61] Jeff Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction approach to test data generation. *Software-Practice and Experience*, 29(2):167–193, January 1999.

[62] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification, and Reliability, Wiley*, 13(1):25–53, March 2003.

[63] Jeff Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software-Practice and Experience*, 26(2):165–176, February 1996.

[64] K. M. Olender and L. J. Osterweil. Specification and static evaluation of sequencing constraints in software. In *Proceedings of the Workshop on Software Testing*, pages 2–9, Banff, Alberta, July 1986. IEEE Computer Society Press.

[65] K. M. Olender and L. J. Osterweil. Cesar: A static sequencing constraint analyzer. In *Proceedings of the Third Workshop on Software Testing, Verification and Analysis*, pages 66–74, Key West Florida, December 1989. ACM SIGSOFT.

[66] Alex Orso and Mauro Pezze. Integration testing of procedural object oriented programs with polymorphism. In *Proceedings of the Sixteenth International Conference on Testing Computer Software*, pages 103–114, Washington DC, June 1999. ACM SIGSOFT.

[67] L. J. Osterweil and L. D. Fosdick. Data flow analysis as an aid in documentation, assertion generation, validation, and error detection. Technical report cu-cs-055-74, Department of Computer Science, University of Colorado, Boulder CO, September 1974.

[68] S. Pimont and J. C. Rault. A software reliability assessment based on a structural behavioral analysis of programs. In *Proceedings of the Second International Conference on Software Engineering*, pages 486–491, San Francisco, CA, October 1976.

[69] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, December 1976.

[70] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[71] RTCA-DO-178B. Software considerations in airborne systems and equipment certification, December 1992.

[72] K. Sabnani and A. Dahbura. A protocol testing procedure. *Computer Networks and ISDN Systems*, 14(4):285–297, 1988.

[73] F. Tip. A survey of program slicing techniques. Technical report CS-R-9438, Computer Science/Department of Software Technology, Centrum voor Wiskunde en Informatica, 1994.

[74] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[75] Elaine J. Weyuker, Steward N. Weiss, and Richard G. Hamlet. Data flow-based adequacy analysis for languages with pointers. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 74–86, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[76] Lee White and Bogdan Wiszniewski. Path testing of computer programs with loops using a tool for simple loop patterns. *Software-Practice and Experience*, 21(10):1075–1102, October 1991.

[77] W. Eric Wong and Aditya P. Mathur. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, March 1995.

Index

activity diagram , 67–68
alias, 32
All Object Call, 44
All-Coupling-Def, 46
All-Coupling-Use, 47
arc, 2
ASSET, 71
ATAC, 71
automatic test data generation, 70

basic block, 2, 28–31, 33
    definition, 28
best effort touring, *see* tour, best effort

Call Coverage, 42
call graph, 4, 43, 44, 52, 53
    definition, 42
    example, 43
call site, 45
    actual parameter, 45
    callee, 45, 46
    caller, 45, 46
    former parameter, 45
CFG, *see* control flow graph
class variable, 50
Complete Path Coverage, 68
component, 42, 51
control flow, 7
control flow graph, 1, 4, 5, 29, 38, 41, 51, 55, 58, 69, 70
    definition, 27
CORBA, 51
coupling, 42, 45–48, 50–52, 72
    coupling method, 50
    coupling variable, 45
    external device coupling, 45
    full inter-procedural, 50
    inter-procedural, 50
    parameter coupling, 45
    shared data coupling, 45
criteria
    ADC, 24, 28, 46, 57, 71
    ADUPC, 24–25, 27–28, 47, 71
    AUC, 24–25, 27–28, 46–47, 57, 71
    CPC, 11, 28
    CRTC, 11, 28
    EC, 9, 26–28, 31, 42–43, 57, 68–70
    EPC, 10, 27, 28, 57
    NC, 8–9, 26–28, 31, 42–44, 57, 65, 68–70
    PPC, 10–11, 14, 27–28
    SPC, 11, 69
    SRTC, 11, 28
cycle
    definition, 3
cyclomatic complexity, 70

data flow, 7, 27, 57, 68, 71, 72
    all-defs, 24
    all-du-paths, 24
    all-uses, 24
    def, 20–21
        last-def, 46, 50
    def-clear, 20
    def-pair set, 21
    def-path set, 21
    design, 45–53
    du-pair, 20, 34–36, 46–48
        interprocedural, 48–51
    du-path, 20–25, 27, 34–36, 46
    including, 19–20
    local use, 33
    p-uses, 24
    reach, 27, 33, 46, 48, 57
    reaches, 20
    source, 31–36

theory, 20–25
use, 20–21
first-use, 50
use-clear, 46
definition, 46
data flow graph coverage, 7
data processing, 70
dataflow
location
definition, 20
reach, 20, 22, 24
decision, 28
def, *see* data flow, def
def-use, *see* data flow, du-pair
definition, *see* data flow, def
definition-use, *see* data flow, du-pair
design element, 42
detour, 11–14
definition, 13
explanation, 13–14
distinguished sequence method, 70
du-associations, *see* data flow, du-pair
du-pair, *see* data flow, du-pair
du-path, *see* data flow, du-path, 71

edge, 2–4, 13
example
call graph, 43
graph, 4
Java
BoundedQueue2, 64
patternIndex, 35
Quadratic, 49
takeOut(), 52
trash(), 52
Watch, 58

FAA, 72
finite automata,, 56
finite state machines, 4, 10, 20, 56
definition, 56
deriving, 58–64
first-use, *see* data flow, use, first-use
first-uses, 72
FSM, *see* finite state machine

graph, 2–27
case structure, 33
do while structure, 31
double-diamond, 5, 8, 25
for structure, 31
if structure, 29, 30
if-else structure, 29
self loop, 15
SESE, 5
try-catch structure, 34
visit, 8
while break structure, 32
while structure, 30
graph coverage
definition, 7
guards, 57

HTTP, 51

infeasible, 11, 14, 27, 36, 56
inheritance, 43–44, 72
inheritance hierarchy, 43
instance context, 50
integration testing, 42, 45, 72

junction, 28

last-def, *see* data flow, def, last-def
last-defs, 72
lexical analyzers, 70

Method Coverage, 42
minimal, 26, 40, 52
definition, 6
model checking, 72
model-based testing, 72
module, 43

n-switch, 70
node, 2–4, 13
final, 2
initial, 2
node coverage
definition, 8
non-determinism, 58, 63

OO Call Coverage, 44

path, 3–6
  du-path, *see* data flow, du-path
  prime, 27
    definition, 10
    deriving, 14–17
    examples, 12
  simple, 10–11, 15, 27
  test-path, 4, 8
    definition, 5
Petri nets, 56
polymorphism, 43–44, 72
post-state, 57
pre-state, 57
preconditions, 57
predecessor, 2
prime path, *see* path, prime
protocols, 70

reach, 4–5, 8–11, 26, 30, 64
  definition, 4
  semantic, 4
  syntactic, 4
reachability, 1
real-time, 70
recognizer, 71
RIPR, 1
RMI, 51
round trip, 10, 70

SCR, 56, 57, 72
Sequencing Constraints, 53–55
sequencing constraints, 53–56, 72
SESE graph, *see* graph, SESE
sidetrip, 11–14, 27
  definition, 13
  explanation, 13–14
simple path, *see* path, simple
spanning tree, 70
specification, 53–65
Specified Path Coverage, 69
state, 56
State Coverage, 57
state variable, 50
structural graph coverage, 7
subgraph, 2
subpath, 3, 13
subpath sets, 70
subsumption, 9, 14, 18, 25–27
successor, 2
switch cover, 70

telecommunication systems, 69
telephony, 70
test engineer, 27, 56
test path, *see* path, test-path
tour, 5, 7, 11–14, 47
  best effort, 24, 25, 71
    definition, 14
  definition, 13
  explanation, 13–14
transaction flow graph, 67
transition, 56
Transition Coverage, 57
transition-pair, 57, 70
triggering events, 57
two-trip, 57, 70

UML, 72
UML Activity Diagram, 67
UML statecharts, 56
undecidable, 48, 55
unique input-output method, 70
use, *see* data flow, use
use case, 65–69

vertex, 2
visit, 5, 7

W-method, 70