MODULE-03
STRUCTURAL TESTING

Path Testing

The distinguishing characteristic of structural testing methods is that they are all based on the source code of the program being tested, and not on the definition. Because of this absolute basis, structural testing methods are very amenable to rigorous definitions, mathematical analysis, and precise measurement. In this chapter, we will examine the two most common forms of path testing. The technology behind these has been available since the mid-1970s, and the originators of these methods now have companies that market very successful tools that implement the techniques. Both techniques start with the program graph;

3.1 Program Graphs

Definition

Given a program written in an imperative programming language, its ***program graph*** is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represent flow of control. If i and j are nodes in the program graph, there is an edge from node i to node j iff the statement (fragment) corresponding to node j can be executed immediately after the statement (fragment) corresponding to node i.

3.1.1 Style Choices for Program Graphs

Deriving a program graph from a given program is an easy process. It is illustrated here with four of the basic structured programming constructs (Figure 3.1), and also with our pseudocode implementation of the triangle program from Chapter 2. Line numbers refer to statements and statement fragments. An element of judgment can be used here: sometimes it is convenient to keepa fragment as a separate node; other times it seems better to include this with another portion of a Statement. For example, in Figure 3.2, line 14 could be split into two lines:

14 Then If (a = b) AND (b = c)
14a Then
14b If (a = b) AND (b = c)

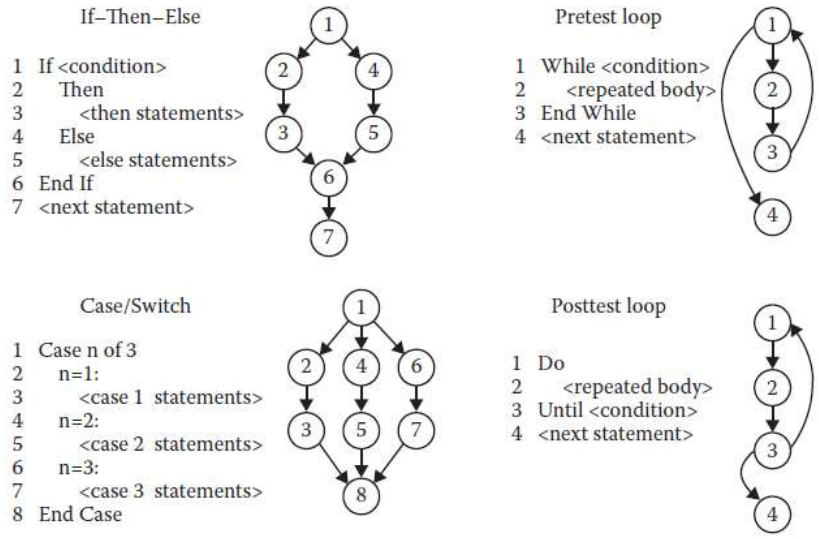

Figure 3.1 Program graphs of four structured programming constructs.

This latitude collapses onto a unique DD-path graph, so the differences introduced by differing judgments are moot. (A mathematician would make the point that, for a given program, several distinct program graphs might be used, all of which reduce to a unique DD-path graph.) We also need to decide whether to associate nodes with nonexecutable statements such as variable and type declarations; here we do not. A program graph of the second version of the triangle problem (see Chapter 2) is given in Figure 3.2.

Nodes 4 through 8 are a sequence, nodes 9 through 12 are an if–then–else construct, and nodes 13 through 22 are nested if–then–else constructs. Nodes 4 and 23 are the program source and sink nodes, corresponding to the single entry, single-exit criteria. No loops exist, so this is a directed acyclic graph. The importance of the program graph is that program executions correspond to paths from the source to the sink nodes. Because test cases force the execution of some such program path, we now have a very explicit description of the relationship between a test case and the part of the program it exercises. We also have an elegant, theoretically respectable way to deal with the potentially large number of execution paths in a program.

There are detractors of path-based testing. Figure 3.3 is a graph of a simple (but unstructured!) program; it is typical of the kind of example detractors use to show the (practical) impossibility of completely testing even simple programs. (This example first appeared in Schach [1993].) In this program, five paths lead from node B to node F in the interior of the loop. If the loop may have up to 18 repetitions, some 4.77 trillion distinct program execution paths exist. (Actually, it is 4,768,371,582,030 paths.) The detractor's argument is a good example of the logical fallacy of extension—take a situation, extend it to an extreme, show that the extreme supports your point, and then apply it back to the original question. The detractors miss the point of code-based testing later in this chapter, we will see how this enormous number can be reduced, with good reasons, to a more manageable size.

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10    Then IsATriangle = True
11    Else IsATriangle = False
12 EndIf
13 If IsATriangle
14   Then  If (a = b) AND (b = c)
15           Then Output ("Equilateral")
16           Else  If (a≠b) AND (a≠c) AND (b≠c)
17                    Then Output ("Scalene")
18                    Else Output ("Isosceles")
19                EndIf
20           EndIf
21   Else  Output("Nota a Triangle")
22 EndIf
23 End triangle2
```

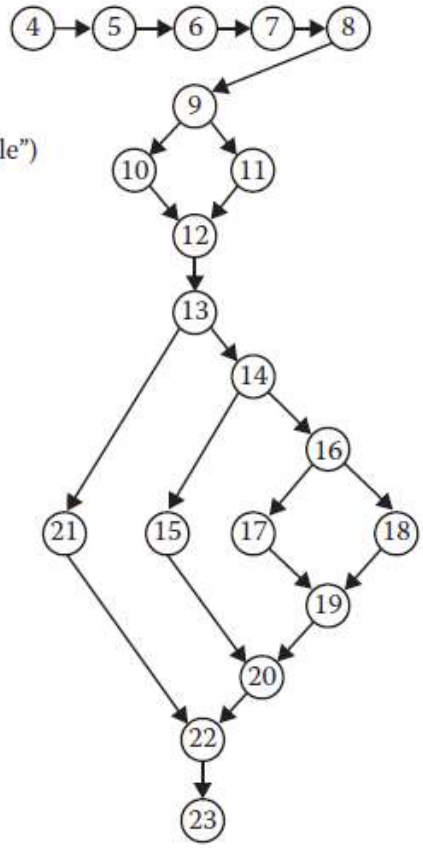

Figure 3.2 Program graph of triangle program.

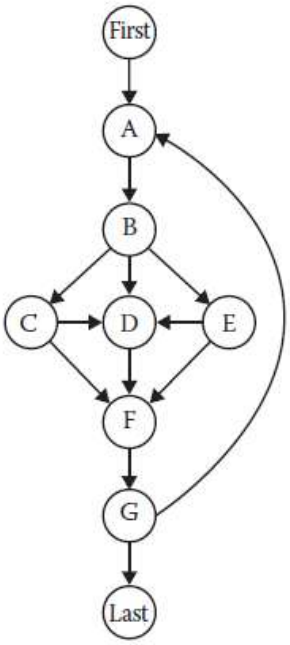

Figure 3.3 Trillions of paths

3.2 DD-Paths

The best-known form of code-based testing is based on a construct known as a decision-todecision path (DD-path) (Miller, 1977). The name refers to a sequence of statements that, in Miller's words, begins with the "outway" of a decision statement and ends with the "inway" of the next decision statement. No internal branches occur in such a sequence, so the corresponding code is like a row of dominoes lined up so that when the first falls, all the rest in the sequence fall. Miller's original definition works well for second-generation languages like FORTRAN II, because decision-making statements (such as arithmetic IFs and DO loops) use statement labels to refer to target statements. With modern languages (e.g., Pascal, AdaR, C, Visual Basic, Java), the notion of statement fragments resolves the difficulty of applying Miller's original definition. Otherwise, we end up with program graphs in which some statements are members of more than one DD-path. In the ISTQB literature, and also in Great Britain, the DD-path concept is known as a "linear code sequence and jump" and is abbreviated by the acronym LCSAJ. Same idea, longer name. We will define DD-paths in terms of paths of nodes in a program graph. In graph theory, these paths are called chains, where a chain is a path in which the initial and terminal nodes are distinct, and every interior node has indegree = 1 and outdegree = 1. (See Chapter 4 for a formal definition.) Notice that the initial node is 2-connected to every other node in the chain, and no instances of 1- or 3-connected nodes occur, as shown in Figure 3.4. The length (number of edges) of the chain in Figure 3.4 is 6.

Definition

A *DD-path* is a sequence of nodes in a program graph such that
Case 1: It consists of a single node with indeg = 0.
Case 2: It consists of a single node with outdeg = 0.
Case 3: It consists of a single node with indeg ≥ 2 or outdeg ≥ 2.
Case 4: It consists of a single node with indeg = 1 and outdeg = 1.
Case 5: It is a maximal chain of length ≥ 1.
Cases 1 and 2 establish the unique source and sink nodes of the program graph of a structured Program as initial and final DD-paths. Case 3 deals with complex nodes; it assures that no node is contained in more than one DD-path. Case 4 is needed for "short branches"; it also preserves the one-fragment, one DD-path principle. Case 5 is the "normal case," in which a DD-path is a single entry, single-exit sequence of nodes (a chain). The "maximal" part of the case 5 definition is used to determine the final node of a normal (nontrivial) chain.

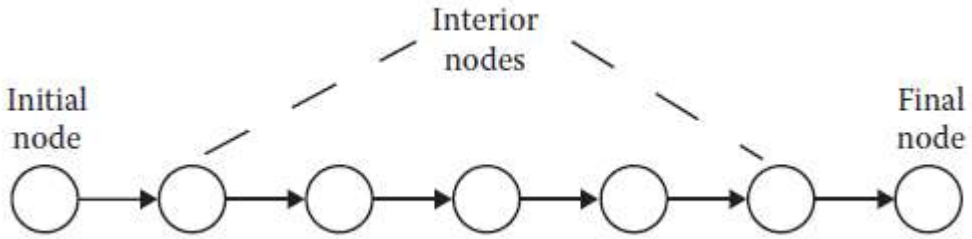

Figure 3.4 Chain of nodes in a directed graph.

Definition

Given a program written in an imperative language, its *DD-path graph* is the directed graph in which nodes are DD-paths of its program graph, and edges represent control flow between successor DD-paths.

This is a complex definition, so we will apply it to the program graph in Figure 3.2. Node 4 is a case 1 DD-path; we will call it "first." Similarly, node 23 is a case 2 DD-path, and we will call it "last." Nodes 5 through 8 are case 5 DD-paths. We know that node 8 is the last node in this DD-path because it is the last node that preserves the 2-connectedness property of the chain. If we go beyond node 8 to include node 9, we violate the indegree = outdegree = 1 criterion of a chain. If we stop at node 7, we violate the "maximal" criterion. Nodes 10, 11, 15, 17, 18, and 21 are case 4 DD-paths. Nodes 9, 12, 13, 14, 16, 19, 20, and 22 are case 3 DD-paths. Finally, node 23 is a case 2 DD-path. All this is summarized in Figure 3.5.

In effect, the DD-path graph is a form of condensation graph (see Chapter 4); in this condensation, 2-connected components are collapsed into individual nodes that correspond to case 5 DD-paths. The single-node DD-paths (corresponding to cases 1–4) are required to preserve the convention that a statement (or statement fragment) is in exactly one DD-path. Without this convention, we end up with rather clumsy DD-path graphs, in which some statement fragments are in several DD-paths.

This process should not intimidate testers—high-quality commercial tools are available, which generate the DD-path graph of a given program. The vendors make sure that their products work for a wide variety of programming languages. In practice, it is reasonable to manually create DD-path graphs for programs up to about 100 source lines. Beyond that, most testers look for a tool.

Figure 8.2 Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2

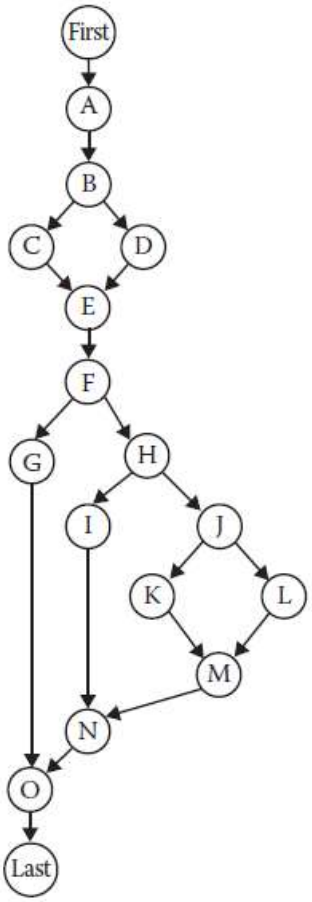

Figure 3.5 DD-path graph for triangle program.

Part of the confusion with this example is that the triangle problem is logic intensive and Computationally sparse. This combination yields many short DD-paths. If the THEN and ELSE clauses contained blocks of computational statements, we would have longer chains, as we will see in the commission problem.

3.3 Test Coverage Metrics

The *raison d' etre* of DD-paths is that they enable very precise descriptions of test coverage. Recall (from Chapters 5 through 7) that one of the fundamental limitations of specification-based testing is that it is impossible to know either the extent of redundancy or the possibility of gaps corresponding to the way a set of functional test cases exercises a program. Back in Chapter 1, we had a Venn diagram showing relationships among specified, programmed, and tested behaviors. Test coverage metrics are a device to measure the extent to which a set of test cases covers (or exercises) a program.

3.3.1 Program Graph–Based Coverage Metrics

Given a program graph, we can define the following set of test coverage metrics. We will use them to relate to other published sets of coverage metrics.

Definition

Given a set of test cases for a program, they constitute *node coverage* if, when executed on the program, every node in the program graph is traversed. Denote this level of coverage as *G*node, where the *G* stands for program graph.
Since nodes correspond to statement fragments, this guarantees that every statement fragment is executed by some test case. If we are careful about defining statement fragment nodes, this also guarantees that statement fragments that are outcomes of a decision-making statement are executed.

Definition

Given a set of test cases for a program, they constitute *edge coverage* if, when executed on the program, every edge in the program graph is traversed. Denote this level of coverage as *G*edge.
The difference between *G*node and *G*edge is that, in the latter, we are assured that all outcomes of a decision-making statement are executed. In our triangle problem (see Figure 8.2), nodes 9, 10, 11, and 12 are a complete if–then–else statement. If we required nodes to correspond to full statements, we could execute just one of the decision alternatives and satisfy the statement coverage criterion. Because we allow statement fragments, it is natural to divide such a statement into separate nodes (the condition test, the true outcome, and the false outcome). Doing so results in predicate outcome coverage. Whether or not our convention is followed, these coverage metrics require that we find a set of test cases such that, when executed, every node of the program graph is traversed at least once.

Definition

Given a set of test cases for a program, they constitute *chain coverage* if, when executed on the program, every chain of length greater than or equal to 2 in the program graph is traversed. Denote this level of coverage as *G*chain.

The *G*chain coverage is the same as node coverage in the DD-path graph that corresponds to the given program graph. Since DD-paths are important in E.F. Miller's original formulation of test covers (defined in Section 3.3.2), we now have a clear connection between purely program graph constructs and Miller's test covers.

Definition

Given a set of test cases for a program, they constitute *path coverage* if, when executed on the program, every path from the source node to the sink node in the program graph is traversed. Denote this level of coverage as *G*path.

This coverage is open to severe limitations when there are loops in a program (as in Figure 3.3). E.F. Miller partially anticipated this when he postulated the *C*2 metric for loop coverage. Referring back to Chapter 4, observe that every loop in a program graph represents a set of strongly (3-connected) nodes. To deal with the size implications of loops, we simply exercise every loop, and then form the condensation graph of the original program graph, which must be a directed acyclic graph.

3.3.2 E.F. Miller's Coverage Metrics

Several widely accepted test coverage metrics are used; most of those in Table 3.1 are due to the early work of Miller (1977). Having an organized view of the extent to which a program is tested makes it possible to sensibly manage the testing process. Most quality organizations now expect the *C*1 metric (DD-path coverage) as the minimum acceptable level of test coverage.

These coverage metrics form a lattice in which some are equivalent and some are implied by others. The importance of the lattice is that

there are always fault types that can be revealed at one level and can escape detection by inferior levels of testing. Miller (1991) observes that when DD-path coverage is attained by a set of test cases, roughly 85% of all faults are revealed. The test coverage metrics in Table 3.1 tell us what to test but not how to test it. In this section, we take a closer look at techniques that exercise source code. We must keep an important distinction in mind: Miller's test coverage metrics are based on program graphs in which nodes are full statements, whereas our formulation allows statement fragments (which can be entire statements) to be nodes.

3.3.2.1 Statement Testing and Predicate Testing

Because our formulation allows statement fragments to be individual nodes, the statement and predicate levels (C_0 and C_1) to collapse into one consideration. In our triangle example (see Figure

3.1), nodes 8, 9, and 10 are a complete Pascal IF-THEN-ELSE statement. If we required nodes to correspond to full statements, we could execute just one of the decision alternatives and satisfy the statement coverage criterion. Because we allow statement fragments, it is "natural" to divide such a statement into three nodes. Doing so results in predicate outcome coverage.

Whether or not our convention is followed, these coverage metrics require that we find a set of test cases such that, when executed, every node of the program graph is traversed at least once.

Metric	*Description of Coverage*
C_0	Every statement
C_1	Every DD-path
C_{1p}	Every predicate to each outcome
C_2	C_1 coverage + loop coverage
C_d	C_1 coverage + every dependent pair of DD-paths
C_{MCC}	Multiple condition coverage
C_{ik}	Every program path that contains up to k repetitions of a loop (usually $k = 2$)
C_{stat}	"Statistically significant" fraction of paths
C_∞	All possible execution paths

Table 3.1 Miller's/structure Test Coverage Metrics

3.3.2.2 DD-Path Testing

When every DD-path is traversed (the *C*1 metric), we know that each predicate outcome has been executed; this amounts to traversing every edge in the DD-path graph (or program graph). Therefore, the *C*1 metric is exactly our *G*chain metric.
For if–then and if–then–else statements, this means that both the true and the false branches are covered (*C*1p coverage). For CASE statements, each clause is covered. Beyond this, it is useful to ask how we might test a DD-path. Longer DD-paths generally represent complex computations, which we can rightly consider as individual functions. For such DD-paths, it may be appropriate to apply a number of functional tests, especially those for boundary and special values.

3.3.2.3 Dependent Pairs of DD-Paths

Identification of dependencies must be made at the code level. This cannot be done just by considering program graphs. The *C*d metric foreshadows the topic of Chapter 9—data flow testing. The most common dependency among pairs of DD-paths is the define/reference relationship, in which a variable is defined (receives a value) in one DD-path and is referenced in another DD-path. The importance of these dependencies is that they are closely related to the problem of infeasible paths. We have good examples of dependent pairs of DD-paths: in Figure 3.5, C and H are such a pair,

as are DD-paths D and H. The variable IsATriangle is set to TRUE at node C, and FALSE at node D. Node H is the branch taken when IsATriangle is TRUE win the condition at node F. Any path containing nodes D and H is infeasible. Simple DD-path coverage might not exercise these dependencies; thus, a deeper class of faults would not be revealed.

3.3.2.4 Multiple Condition Coverage

Look closely at the compound conditions in DD-Paths A and E. Rather than simply traversing such predicates to their TRUE and FALSE outcomes; we should investigate the different ways that each outcome can occur.

One possibility is to make a truth table; a compound condition of three simple conditions would have eight rows, yielding eight test cases. Another possibility is to reprogram compound predicates into nested simple IF-THEN-ELSE logic, which will result in more DD-Paths to cover. We see an interesting trade-off: statement complexity versus path complexity. Multiple condition coverage assures that this complexity isn't swept under the DD-Path coverage rug.

3.3.2.5 Loop Coverage

The condensation graphs provide us with an elegant resolution to the problems of testing loops. Loop testing has been studied extensively, and with good reason — loops are a highly fault prone portion of source code.

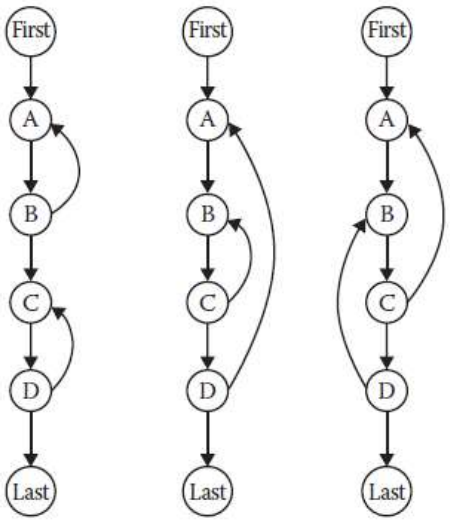

Figure 3.6 Concatenated, nested, and knotted loops.

To start, an amusing taxonomy of loops occurs (Beizer, 1984): concatenated, nested, and horrible, shown in Figure 3.6.

Concatenated loops are simply a sequence of disjoint loops, while nested loops are such that one is contained inside another. Knotted (Beizer calls them "horrible") loops cannot occur when the structured programming precepts are followed, but they can occur in languages like Java with try/catch. When it is possible to branch into (or out from) the middle of a loop, and these branches are internal to other loops, the result is Beazer's knotted loop.

We can also take a modified boundary value approach, where the loop index is given its minimum, nominal, and maximum values We can push this further to full boundary value testing and even robustness testing. If the body of a simple loop is a DD-path that performs a complex calculation, this should also be tested, as discussed previously.

Once a loop has been tested, the tester condenses it into a single node. If loops are nested, this process is repeated starting with the innermost loop and workin outward. This results in the same multiplicity of test cases we found with boundary value analysis, which makes sense, because each loop index variable acts like an input variable. As a preview, consider the infinite loop that could occur if one loop tampers with the value of the other loop's index.

3.3.5 Test Coverage Analyzers

Coverage analyzers are a class of test tools that offer automated support for this approach to testing
Management. With a coverage analyzer, the tester runs a set of test cases on a program that has been "instrumented" by the coverage analyzer. The analyzer then uses information produced by the instrumentation code to generate a coverage report.
In the common case of DD-path coverage, for example, the instrumentation identifies and labels all DD-paths in an original program. When the instrumented program is executed with test cases, the analyzer tabulates the DD-paths traversed by each test case. In this way, the tester can experiment with different sets of test cases to determine the coverage of each set.

3.4 Basis Path Testing

The mathematical notion of a "basis" has attractive possibilities for structural testing. Certain sets can have a basis; and when they do, the basis has very important properties with respect to the entire set. Mathematicians usually define a basis in terms of a structure called a "vector space," which is a set of elements (called vectors) as well as operations that correspond to multiplication and addition defined for the vectors. If a half dozen other criteria apply, the structure is said to be a vector space, and all vector spaces have a basis (in fact they may have several bases). The basis of a vector space is a set of vectors that are independent of each other and "span" the entire vector space in the sense that any other vector in the space can be expressed in terms of the basis vectors. Thus, a set of basis vectors somehow represents "the essence" of the full vector space: everything else in the space can be expressed in terms of the basis, and if one basis element is deleted, this spanning property is lost. The potential application of this theory for testing is that, if we can view a program as a vector space, then the basis for such a space would be a very interesting set of elements to test. If the basis is okay, we could hope that everything that can be expressed in terms of the basis is also okay. In this section, we examine the early work of Thomas McCabe, who recognized this possibility in the mid-1970s.

3.4.1 McCabe's Basis Path Method

Figure 3.7 is taken from McCabe (1982). It is a directed graph that we might take to be the program graph (or the DD-path graph) of some program.

For the convenience of readers who have encountered this example elsewhere (McCabe, 1987; Perry, 1987), the original notation for nodes and edges is repeated here. (Notice that this is not a graph derived from a structured program: nodes B and C are a loop with two exits, and the edge from B to E is a branch into the if–then statement in nodes D, E, and F.) The program does have a single entry (A) and a single exit (G).

McCabe based his view of testing on a major result from graph theory, which states that the cyclomatic number (see Chapter 4) of a strongly connected graph is the number of linearly independent circuits in the graph. (A circuit is similar to a chain: no internal loops or decisions occur, but the initial node is the terminal node.

A circuit is a set of 3-connected nodes.) We can always create a strongly connected graph by adding an edge from the (every) sink node to the (every) source node. (Notice that, if the single-entry, single-exit precept is violated, we greatly increase the cyclomatic number because we need to add edges from each sink node to each source node.) The right side of Figure 3.7 shows the result of doing this; it also contains edge labels that are used in the discussion that follows.

Some confusion exists in the literature about the correct formula for cyclomatic complexity. Some sources give the formula as $V(G) = e - n + p$, while others use the formula $V(G) = e - n + 2p$; everyone agrees that e is the number of edges, n is the number of nodes, and p is the number of connected regions. The confusion apparently comes from the transformation of an arbitrary directed graph (such as the one in Figure 3.7, left side) to a strongly connected, directed graph obtained by adding one edge from the sink to the source node (as in Figure 3.7, right side). Adding an edge clearly affects value computed by the formula, but it should not affect the number of circuits. Counting or not counting the added edge accounts for the change to the coefficient of p, the number of connected regions. Since p is usually 1, adding the extra edge means we move from $2p$ to p. Here is a way to resolve the apparent inconsistency. The number of linearly independent paths from the source node to the sink node of the graph on the left side of Figure 3.7 is

$$\mathbf{V(G) = e - n + 2p}$$

$$\mathbf{= 10 - 7 + 2(1) = 5}$$

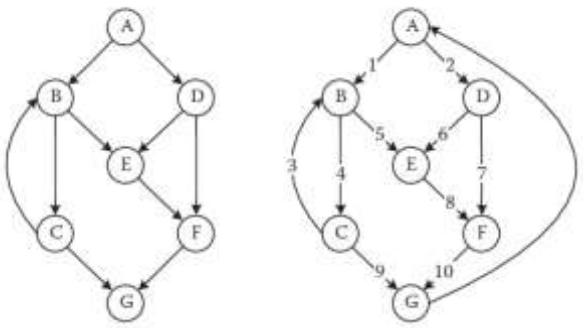

Figure 3.7 McCabe's control graph and derived strongly connected graph.

The number of linearly independent circuits of the graph on the right side of the graph in Figure 3.7 is

$$\boldsymbol{V(G) = e - n + p}$$

=11-7+1=5

The cyclomatic complexity of the strongly connected graph in Figure 8.10 is 5; thus, there are five linearly independent circuits. If we now delete the added edge from node G to node A, these five circuits become five linearly independent paths from node A to node G. In small graphs, we can visually identify independent paths. Here, we identify paths as sequences of nodes:

p1: A, B, C, G
p2: A, B, C, B, C, G
p3: A, B, E, F, G
p4: A, D, E, F, G
p5: A, D, F, G

Table 3.2 shows the edges traversed by each path, and also the number of times an edge is traversed. We can force this to begin to look like a vector space by defining notions of addition and scalar multiplication: path addition is simply one path followed by another path, and multiplication corresponds to repetitions of a path. With this formulation, McCabe arrives at a vector space of program paths. His illustration of the basis part of this framework is that the path A, B, C, B, E, F, G is the basis sum p2 + p3 – p1, and the path A, B, C, B, C, B, C, G is the linear combination 2p2 – p1. It is easier to see this addition with an incidence matrix (see Chapter 4) in which rows correspond to paths, and columns correspond to edges, as in Table 8.5. The entries in this table are obtained by following a path and noting which edges are traversed. Path p1, for example, traverses edges 1, 4, and 9, while path p2 traverses the following edge sequence: 1, 4, 3, 4, 9. Because edge 4 is traversed twice by path p2, that is the entry for the edge 4 column. We can check the independence of paths p1 – p5 by examining the first five rows of this incidence matrix. The bold entries show edges that appear in exactly one path, so paths p2 – p5 must

Path/Edges Traversed	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	1	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	**1**	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	**1**	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	**1**	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Table 3.2 Path/Edge Traversal

be independent. Path p1 is independent of all of these, because any attempt to express p1 in terms of the others introduces unwanted edges. None can be deleted, and these five paths span the set of all paths from node A to node G. At this point, you might check the linear combinations of the two example paths. (The addition and multiplication are performed on the column entries.)

Original	p1: A–B–C–E–F–H–J–K–M–N–O–Last	Scalene
Flip p1 at B	p2: A–B–D–E–F–H–J–K–M–N–O–Last	Infeasible
Flip p1 at F	p3: A–B–C–E–F–G–O–Last	Infeasible
Flip p1 at H	p4: A–B–C–E–F–H–I–N–O–Last	Equilateral
Flip p1 at J	p5: A–B–C–E–F–H–J–L–M–N–O–Last	Isosceles

Table 3.3 Basis Paths in Figure 3.2

McCabe next develops an algorithmic procedure (called the baseline method) to determine a set of basis paths. The method begins with the selection of a baseline path, which should correspond to some "normal case" program execution. This can be somewhat arbitrary; McCabe advises choosing a path with as many decision nodes as possible. Next, the baseline path is retraced, and in turn each decision is "flipped"; that is, when a node of outdegree ≥ 2 is reached, a different edge must be taken. Here we follow McCabe's example, in which he first postulates the path through nodes A, B, C, B, E, F, G as the baseline. (This was expressed in terms of paths p1 – p5 earlier.) The first decision node (outdegree ≥ 2) in this path is node A; thus, for the next basis path, we traverse edge 2 instead of edge 1. We get the path A, D, E, F, G, where we retrace nodes E, F, G in path 1 to be as minimally different as possible. For the next path, we can follow the second path, and take the other decision outcome of node D, which gives us the path A, D, F, G. Now, only decision nodes B and C have not been flipped; doing so yields the last two basis paths, A, B, E, F, G and A, B, C, G. Notice that this set of basis paths is distinct from the one in Table 3.3: this is not problematic because a unique basis is not required.

3.4.2 Observations on McCabe's Basis Path Method

If you had trouble following some of the discussion on basis paths and sums and products of these, you may have felt a haunting skepticism—something along the lines of, "Here's another academic oversimplification of a real-world problem." Rightly so, because two major soft spots occur in the McCabe view: one is that testing the set of basis paths is sufficient (it is not), and the other has to do with the yoga-like contortions we went through to make program paths look like a vector space. McCabe's example that the path A, B, C, B, C, B, C, G is the linear combination 2p2 – p1 is very unsatisfactory. What does the 2p2 part mean? Execute path p2 twice? (Yes, according to the math.) Even worse, what does the –p1 part mean? Execute path p1 backward? Undo the most recent execution of p1? Do not do p1 next time? Mathematical sophistries like this are a real turnoff to practitioners looking for solutions to their very real problems. To get a better understanding of these problems, we will go back to the triangle program example.
Start with the DD-path graph of the triangle program in Figure 3.2. We begin with a baseline path that corresponds to a scalene triangle, for example, with sides 3, 4, 5. This test case will traverse the path p1 (see Table 3.2). Now, if we flip the decision at node B, we get path p2. Continuing the procedure, we flip the decision at node F, which yields the path p3. Now, we continue to flip decision nodes in the baseline path p1; the next node with outdegree = 2 is node H. When we flip node H, we get the path p4. Next, we flip node J to get p5. We know we are done because there are only five basis paths; they are shown in Table 3.2.

Time for a reality check: if you follow paths p2 and p3, you find that they are both infeasible. Path p2 is infeasible because passing through node D means the sides are not a triangle; so the outcome of the decision at node F must be node G. Similarly, in p3, passing through node C means the sides do form a triangle; so node G cannot be traversed. Paths p4 and p5 are both feasible and correspond to equilateral and isosceles triangles, respectively. Notice that we do not have a basis path for the NotATriangle case.
Recall that dependencies in the input data domain caused difficulties for boundary value testing and that we resolved these by going to decision table-based specification-based testing, where we addressed data dependencies in the decision table. Here, we are dealing with code-level dependencies, which are absolutely incompatible with the latent assumption that basis paths are independent. McCabe's procedure successfully identifies basis paths that are topologically independent; however, when these contradict semantic dependencies, topologically possible paths are seen to be logically infeasible. One solution to this problem is to always require that flipping a decision results in a semantically feasible path. Another is to reason about logical dependencies. If we think about this problem, we can identify two rules:
If node C is traversed, then we must traverse node H.
If node D is traversed, then we must traverse node G.

Taken together, these rules, in conjunction with McCabe's baseline method, will yield the following feasible basis path set. Notice that logical dependencies reduce the size of a basis set when basis paths must be feasible.

p1: A–B–C–E–F–H–J–K–M–N–O–Last	Scalene
p6: A–B–D–E–F–G–O–Last	Not a triangle
p4: A–B–C–E–F–H–I–N–O–Last	Equilateral
p5: A–B–C–E–F–H–J–L–M–N–O–Last	Isosceles

The triangle problem is atypical in that no loops occur. The program has only eight topologically possible paths; and of these, only the four basis paths listed above are feasible. Thus, for this special case, we arrive at the same test cases as we did with special value testing and output range testing. For a more positive observation, basis path coverage guarantees DD-path coverage: the process of flipping decisions guarantees that every decision outcome is traversed, which is the same as DD-path coverage. We see this by example from the incidence matrix description of basis paths and in our triangle program feasible basis paths. We could push this a step further and observe that the set of DD-paths acts like a basis because any program path can be expressed as a linear combination of DD-paths.

3.4.3 Essential Complexity

Part of McCabe's work on cyclomatic complexity does more to improve programming than testing.In this section, we take a quick look at this elegant blend of graph theory, structured programming, and the implications these have for testing. This whole package centers on the notion of essential complexity (McCabe, 1982), which is only the cyclomatic complexity of yet another form of condensation graph. Recall that condensation graphs are a way of simplifying an existing graph; thus far, our simplifications have been based on removing either strong components or DD-paths. Here, we condense around the structured programming constructs, which are repeated as Figure 3.8

The basic idea is to look for the graph of one of the structured programming constructs, collapse it into a single node, and repeat until no more structured programming constructs can be found. This process is followed in Figure 3.9, which starts with the DD-path graph of the pseudocode triangle program. The if–then–else construct involving nodes B, C, D, and E is condensed into node a, and then the three if–then constructs are condensed onto nodes b, c, and d. The remaining if–then–else (which corresponds to the IF IsATriangle statement) is condensed into node e, resulting in a condensed graph with cyclomatic complexity $V(G) = 1$. In general, when a program is well structured (i.e., is composed solely of the structured programming constructs), it can always be reduced to a graph with one path. The graph in Figure 3.7 cannot be reduced in this way (try it!). The loop with nodes B and C cannot be condensed because of the edge from B to E. Similarly, nodes D, E, and F look like an if–then construct, but the edge from B to E violates the structure. McCabe (1976) went on to find elemental "unstructured" that violate the precepts of structured programming. These are shown in Figure 3.10 Each of these violations contains three distinct paths, as opposed to the two paths present in the corresponding structured programmingconstructs.

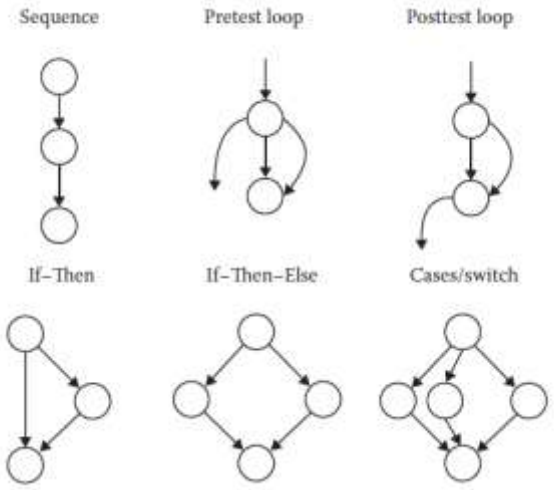

Figure 3.8 Structured programming constructs.

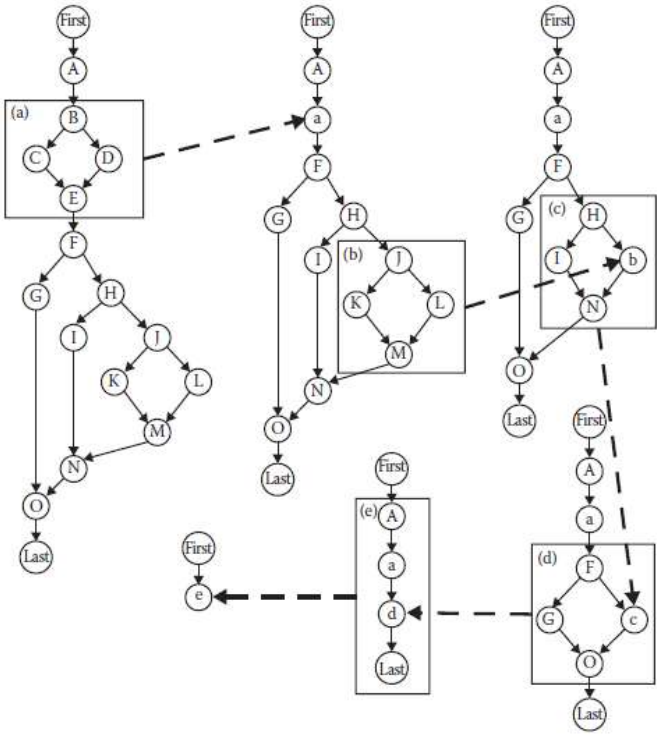

Figure 3.9 Condensing with respect to structured programming constructs.

so one conclusion is that such violations increase cyclomatic complexity. The *piece de resistance* of McCabe's analysis is that these violations cannot occur by themselves: if one occurs in a program, there must be at least one more, so a program cannot be only slightly unstructured. Because these increase cyclomatic complexity, the minimum number of test cases is thereby increased. In the next chapter, we will see that the violations have interesting implications for data flow testing.The bottom line for testers is this: programs with high cyclomatic complexity require moretesting. Of the organizations that use the cyclomatic complexity metric, most set some guideline for maximum acceptable complexity; $V(G) = 10$ is a common choice.

What happens if a unit has a higher complexity? Two possibilities: either simplify the unit or plan to do more testing. If the unit is well structured, its essential complexity is 1; so it can be simplified easily.
If the unit has an essential complexity greater than 1, often the best choice is to eliminate the violations.

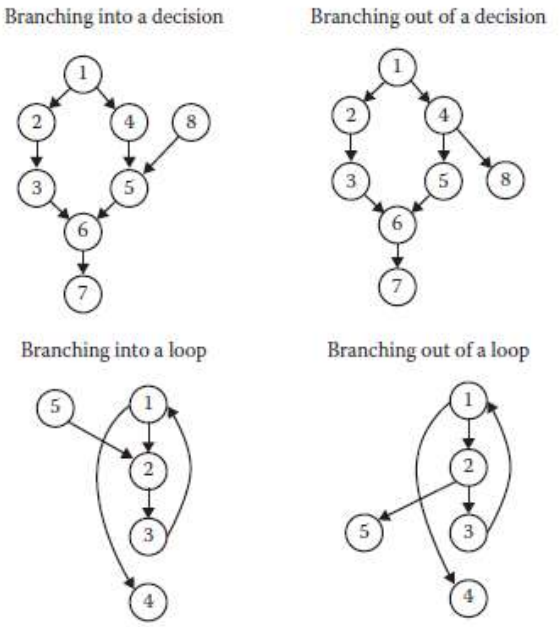

Figure 3.10 Violations of structured programming constructs.

3.5 Guidelines and Observations

In our study of specification-based testing, we observed that gaps and redundancies can both exist and, at the same time, cannot be recognized. The problem was that specification-based testing removes us too far from the code. The path testing approaches to code-based testing represent the case where the pendulum has swung too far the other way: moving from code to directed graph representations and program path formulations obscures important information that is present in the code, in particular the distinction between feasible and infeasible paths. Also, no form of code-based testing can reveal missing functionality that is specified in the requirements. In the next chapter, we look at data flow-based testing. These techniques move closer to the code, so the pendulum will swing back from the path analysis extreme.
McCabe (1982) was partly right when he observed, "It is important to understand that these are purely criteria that measure the quality of testing, and not a procedure to identify test cases." He was referring to the DD-path coverage metric and his basis path heuristic based on cyclomatic complexity metric. Basis path testing therefore gives us a lower boundary on how much testing is necessary.

Path-based testing also provides us with a set of metrics that act as crosschecks on specification-based testing. We can use these metrics to resolve the gaps and redundancies question. When we find that the same program path is traversed by several functional test cases, we suspect that this redundancy is not revealing new faults. When we fail to attain DD-path coverage, we know that there are gaps in the functional test cases. As an example, suppose we have a program that contains extensive error handling, and we test it with boundary value test cases (min, min+, nom, max–, and max). Because these are all permissible values, DD-paths corresponding to the error-handling code will not be traversed. If we add test cases derived from robustness testing or traditional equivalence class testing, the DD-path coverage will improve.

Beyond this rather obvious use of coverage metrics, an opportunity exists for real testing craftsmanship. Any of the coverage metrics in Section 8.3 can operate in two ways: either as a blanket-mandated standard (e.g., all units shall be tested to attain full DD-path coverage) or as a mechanism to selectively test portions of code more rigorously than others. We might choose multiple-condition coverage for modules with complex logic, while those with extensive iteration might be tested in terms of the loop coverage techniques. This is probably the best view of structural testing: use the properties of the source code to identify appropriate coverage metrics, and then use these as a crosscheck on functional test cases. When the desired coverage is not attained, follow interesting paths to identify additional (special value) test cases.

Data Flow Testing

Data flow testing is an unfortunate term, because most software developers immediately think about some connection with dataflow diagrams. Data flow testing refers to forms of structural testing that focus on the points at which variables receive values and the points at which these values are used (or referenced). We will see that data flow testing serves as a "reality check" on path testing; indeed, many of the data flow testing proponents (and researchers) see this approach as a form of path testing. We will look at two mainline forms of data flow testing: one provides a set of basic definitions and a unifying structure of test coverage metrics, while the second is based on a concept called a "program slice". Both of these formalize intuitive behaviors (and analyses) of testers, and although they both start with a program graph, both move back in the direction of functional testing.

Most programs deliver functionality in terms of data. Variables that represent data somehow receive values, and these values are used to compute values for other variables. Since the early 1960s, programmers have analyzed source code in terms of the points (statements) at which variables receive values and points at which these values are used. Many times, their analyses were based on concordances that list statement numbers in which variable names occur. Concordances were popular features of second generation language compilers (they are still popular with COBOL programmers). Early "data flow" analyses often centered on a set of faults that are now known as define/reference anomalies:

- a variable that is defined but never used (referenced)
- a variable that is used but never defined
- a variable that is defined twice before it is used

3.6 Define/Use Testing

Much of the formalization of define/use testing was done in the early 1980s [Rapps 85]; the definitions in this section are compatible with those in [Clarke 89], an article which summarizes most of define/use testing theory. This body of research is very compatible with the formulation we

developed in chapters 4 and 9. It presumes a program graph in which nodes are statement fragments (a fragment may be an entire statement), and programs that follow the structured programming precepts.

The following definitions refer to a program P that has a program graph G(P), and a set of program variables V. The program graph G(P) is constructed as in Chapter 4, with statement fragments as nodes, and edges that represent node sequences. G(P) has a single entry node, and a single exit node.

Definition

Node n G(P) is a ***defining node*** of the variable v V, written as DEF(v,n), iff the value of the variable v is defined at the statement fragment corresponding to node n. Input statements, assignment statements, loop control statements, and procedure calls are all examples of statements that are defining nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables are changed.

Definition

Node n G(P) is a ***usage node*** of the variable v V, written as USE(v, n), iff the value of the variablev is used at the statement fragment corresponding to node n. Output statements, assignment statements, conditional statements, loop control statements, and procedure calls are all examples of statements that are usage nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables remain unchanged.

Definition

A usage node USE(v, n) is a ***predicate use*** (denoted as P-use) iff the statement n is a predicate statement; otherwise USE(v, n) is a ***computation use*** , (denoted C-use). The nodes corresponding to predicate uses always have an outdegree ≥ 2, and nodes corresponding to computation uses always have outdegree ≤ 1.

Definition

A ***definition-use (sub)path*** with respect to a variable v (denoted du-path) is a (sub)path in PATHS(P) such that, for some v V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are the initial and final nodes of the (sub)path.

Definition

A ***definition-clear (sub)path*** with respect to a variable v (denoted dc-path) is a definition-use (sub)path in PATHS(P) with initial and final nodes DEF (v, m) and USE (v, n) such that no other node in the (sub)path is a defining node of v. Testers should notice how these definitions capture the essence of computing with stored data values. Du-paths and dc-paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. Du-paths that are not definition-clear are potential trouble spots.

3.6.1 Example

We will use the Commission Problem and its program graph to illustrate these definitions. The numbered source code is given next, followed by a program graph constructed according to the procedures we discussed in Chapter 4. This program computes the commission on the sales of four salespersons, hence het outer For-loop that repeats four times. During each repetition, a salesperson's name is read from the input device, and the input from that person is read to compute
the total numbers of locks, stocks, and barrels sold by the person. The While-loop is a classical sentinel controlled loop in which a value of -1 for locks signifies the end of that person's data. The totals are accumulated as the data lines are read in the While-loop. After printing this preliminary information, the sales value is computed, using the constant item prices defined at the beginning of the program. The sales value is then used to compute the commission in the conditional portion of the program.

Figure 3.12 shows the decision-to-decision path (DD-path) graph of the program graph in Figure 3.11. More compression exists in this DD-path graph because of the increased computation in the commission problem. Table 3.4 details the statement fragments associated with DD-paths. Some DD-paths (per the definition in Chapter 8) are combined to simplify the graph. We will need this figure later to help visualize the differences among DD-paths, du-paths, and program slices.

Table 3.5 lists the define and usage nodes for the variables in the commission problem. We use this information in conjunction with the program graph in Figure 3.11 to identify various definition/ use and definition-clear paths. It is a judgment call whether no executable statements such as constant and variable declaration statements should be considered as defining nodes. Such nodes are not very interesting when we follow what happens along their du-paths; but if something is wrong, it can be helpful to include them. Take your pick. We will refer to the various paths as sequences of node numbers.

Tables 3.6 and 3.7 present some of the du-paths in the commission problem; they are named by their beginning and ending nodes (from Figure 3.11). The third column in Table 3.6 indicates the du-paths are definition clear. Some of the du-paths are trivial—for example, those for lockPrice, stockPrice, and barrelPrice. Others are more complex: the while loop (node sequence <14, 15, 16, 17, 18, 19, 20>) inputs and accumulated values for total Locks, totalStocks, and total- Barrels. Table 3.6 only shows the details for the totalStocks variable. The initial value definition for totalStocks occurs at node 11, and it is first used at node 17. Thus, the path (11, 17), which consists of the node sequence <11, 12, 13, 14, 15, 16, 17>, is definition clear. The path (11, 22), which consists of the node sequence <11, 12, 13, (14, 15, 16, 17, 18, 19, 20)*, 21, 22>, is not definition because values of totalStocks are defined at node 11 and (possibly several times at) node 17. (The asterisk after the while loop is the Kleene Star notation used both in formal logic and regular expressions to denote zero or more repetitions.)

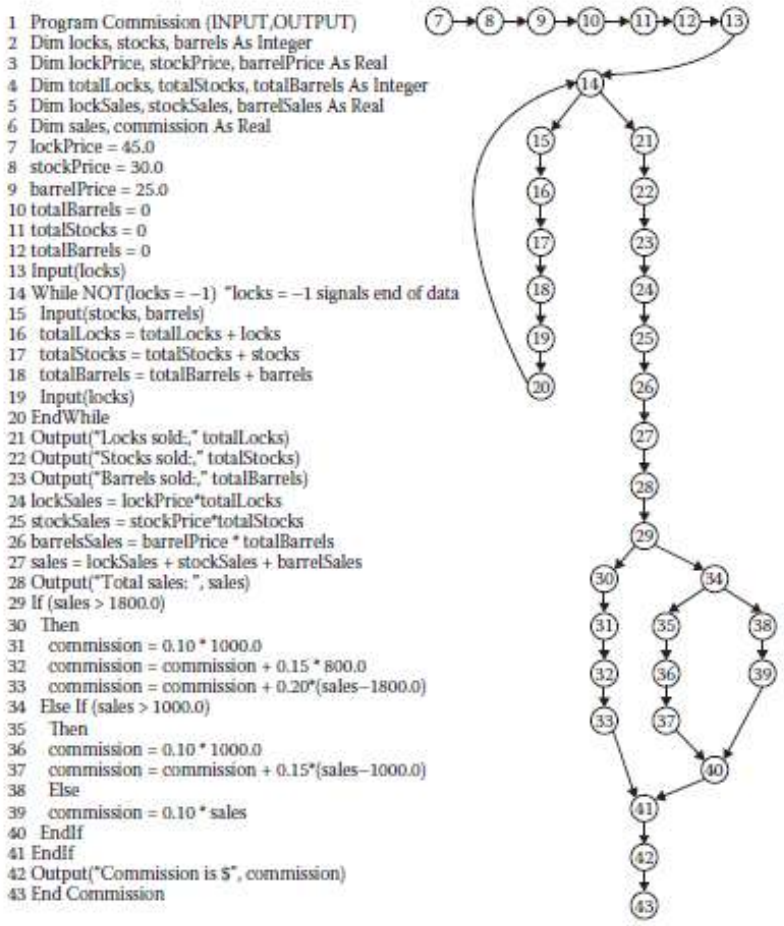

Figure 3.11 Commission problem and its program graph.

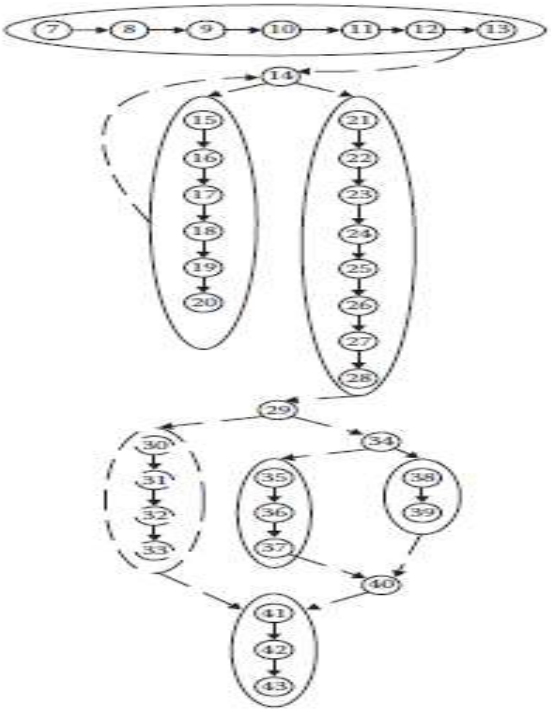

Figure 3.12 DD-path graph of commission problem pseudocode (in Figure 3.11).

3.6.2 Du-paths for Stocks

First, let us look at a simple path: the du-path for the variable stocks. We have DEF(stocks, 15) and USE(stocks, 17), so the path <15, 17> is a du-path with respect to stocks. No other defining nodes are used for stocks; therefore, this path is also definition clear.

3.6.3 Du-paths for Locks

Two defining and two usage nodes make the locks variable more interesting: we have DEF(locks, 13), DEF(locks, 19), USE(locks, 14), and USE(locks, 16). These yield four du-paths; they are shown in Figure 3.13.
p1 = <13, 14>
p2 = <13, 14, 15, 16>
p3 = <19, 20, 14>
p4 = <19, 20, 14, 15, 16>

Note: du-paths p1 and p2 refer to the priming value of locks, which is read at node 13. The locks variable has a predicate use in the while statement (node 14), and if the condition is true (as in path p2), a computation use at statement 16. The other two du-paths start near the end of the while loop and occur when the loop repeats. These paths provide the loop coverage discussed in Chapter 8—bypass the loop, begin the loop, repeat the loop, and exit the loop. All these du-paths are definition clear.

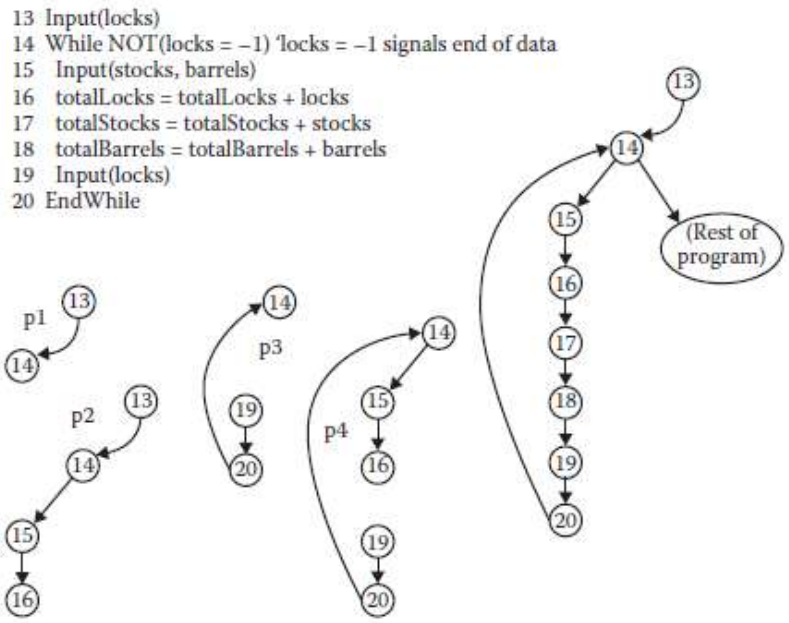

Figure 3.13 Du-paths for locks.

DD-path	Nodes
A	7, 8, 9, 10, 11, 12, 13
B	14
C	15, 16, 17, 18, 19, 20
D	21, 22, 23, 24, 25, 26, 27, 28
E	29
F	30, 31, 32, 33
G	34
H	35, 36, 37
I	38
J	39
K	40
L	41, 42

Table 3.4 DD-paths in Figure 3.11

Variable	Defined at Node	Used at Node
lockPrice	7	24
stockPrice	8	25
barrelPrice	9	26
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26
Locks	13, 19	14, 16
Stocks	15	17
Barrels	15	18
lockSales	24	27
stockSales	25	27
barrelSales	26	27
Sales	27	28, 29, 33, 34, 37, 38
Commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41

Table 3.5 Define/Use Nodes for Variables in Commission Problem

Variable	*Path (Beginning, End) Nodes*	*Definition Clear?*
lockPrice	7, 24	Yes
stockPrice	8, 25	Yes
barrelPrice	9, 26	Yes
totalStocks	11, 17	Yes
totalStocks	11, 22	No
totalStocks	11, 25	No
totalStocks	17, 17	Yes
totalStocks	17, 22	No
totalStocks	17, 25	No
Locks	13, 14	Yes
Locks	13, 16	Yes
Locks	19, 14	Yes
Locks	19, 16	Yes
Sales	27, 28	Yes
Sales	27, 29	Yes
Sales	27, 33	Yes
Sales	27, 34	Yes
Sales	27, 37	Yes
Sales	27, 38	Yes

Table 3.6 Selected Define/Use Paths

Variable	*Path (Beginning, End) Nodes*	*Feasible?*	*Definition Clear?*
Commission	31, 32	Yes	Yes
Commission	31, 33	Yes	No
Commission	31, 37	No	N/A
Commission	31, 41	Yes	No
Commission	32, 32	Yes	Yes
Commission	32, 33	Yes	Yes
Commission	32, 37	No	N/A
Commission	32, 41	Yes	No
Commission	33, 32	No	N/A
Commission	33, 33	Yes	Yes
Commission	33, 37	No	N/A
Commission	33, 41	Yes	Yes
Commission	36, 32	No	N/A
Commission	36, 33	No	N/A
Commission	36, 37	Yes	Yes
Commission	36, 41	Yes	No
Commission	37, 32	No	N/A
Commission	37, 33	No	N/A
Commission	37, 37	Yes	Yes
Commission	37, 41	Yes	Yes
Commission	38, 32	No	N/A
Commission	38, 33	No	N/A
Commission	38, 37	No	N/A
Commission	38, 41	Yes	Yes

Table 3.7 Define/Use Paths for Commission

3.6.4 Du-paths for totalLocks

The du-paths for totalLocks will lead us to typical test cases for computations. With two defining nodes (DEF(totalLocks, 10) and DEF(totalLocks, 16)) and three usage nodes (USE(totalLocks, 16), USE(totalLocks, 21), USE(totalLocks, 24)), we might expect six du-paths. Let us take a closer look. Path p5 = <10, 11, 12, 13, 14, 15, 16> is a du-path in which the initial value of totalLocks (0) has a computation use. This path is definition clear. The next path is problematic:

p6 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21>

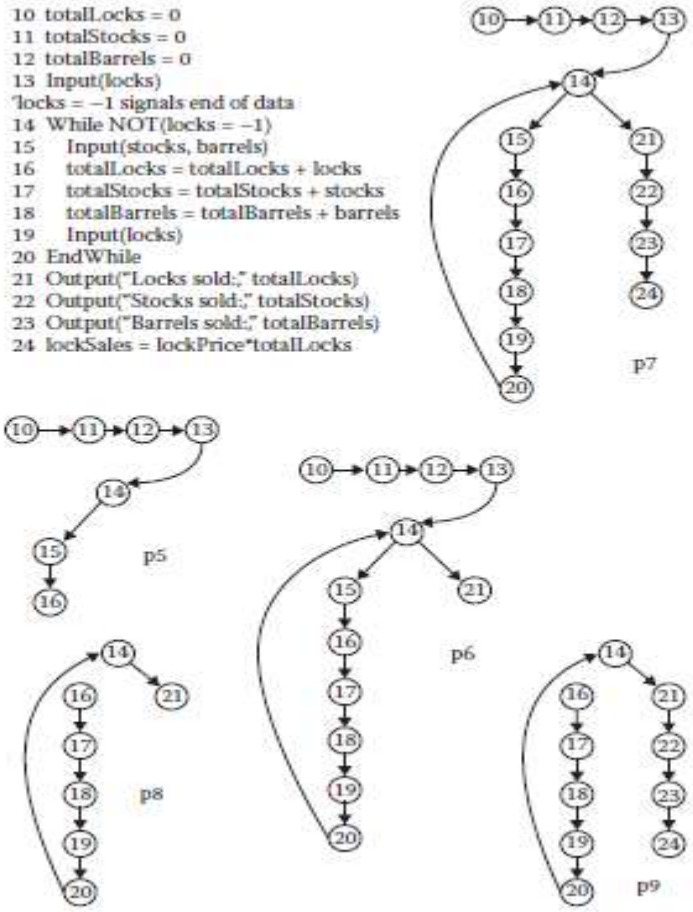

Figure 3.14 Du-paths for totalLocks.

Path p6 ignores the possible repetition of the while loop. We could highlight this by noting that the subpath <16, 17, 18, 19, 20, 14, 15> might be traversed several times. Ignoring this for now, we still have a du-path that fails to be definition clear. If a problem occurs with the value of totalLocks at node 21 (the Output statement), we should look at the intervening DEF(totalLocks, 16) node. The next path contains p6; we can show this by using a path name in place of its corresponding node sequence:

p7 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21, 22, 23, 24>
p7 = <p6, 22, 23, 24>

Du-path p7 is not definition clear because it includes node 16. Subpaths that begin with node 16 (an assignment statement) are interesting. The first, <16, 16>, seems degenerate. If we "expanded" it into machine code, we would be able to separate the define and usage portions. We will disallow these as du-paths.

Technically, the usage on the right-hand side of the assignment refers to a value defined at node 10 (see path p5). The remaining two du-paths are both subpaths of p7:

p8 = <16, 17, 18, 19, 20, 14, 21>

p9 = <16, 17, 18, 19, 20, 14, 21, 22, 23, 24>

Both are definition clear, and both have the loop iteration problem we discussed before. The du-paths for totalLocks are shown in Figure 3.14.

3.6.5 Du-paths for Sales

There is one defining node for sales; therefore, all the du-paths with respect to sales must be definition clear. They are interesting because they illustrate predicate and computation uses. The first three du-paths are easy:

p10 = <27, 28>

p11 = <27, 28, 29>

p12 = <27, 28, 29, 30, 31, 32, 33>

Notice that p12 is a definition-clear path with three usage nodes; it also contains paths p10 and p11. If we were testing with p12, we know we would also have covered the other two paths. We will revisit this toward the end of the chapter. The IF, ELSE IF logic in statements 29 through 40 highlights an ambiguity in the original research. Two choices for du-paths begin with path p11: one choice is the path <27, 28, 29, 30, 31, 32, 33>, and the other is the path <27, 28, 29, 34>.

The remaining du-paths for sales are

p13 = <27, 28, 29, 34>

p14 = <27, 28, 29, 34, 35, 36, 37>

p15 = <27, 28, 29, 34, 38>

3.6.6 Du-paths for Commission

If you have followed this discussion carefully, you are probably dreading the analysis of du-paths with respect to commission. You are right—it is time for a change of pace. In statements 29 through 41, the calculation of commission is controlled by ranges of the variable sales. Statements 31 to 33 build up the value of commission by using the memory location to hold intermediate values. This is a common programming practice, and it is desirable because it shows how the final value is computed. (We could replace these lines with the statement "commission: = 220 + 0.20 * (sales –1800)," where 220 is the value of 0.10 * 1000 + 0.15 * 800, but this would be hard for a maintainer to understand.) The "built-up" version uses intermediate values, and these will appear as define and usage nodes in the du-path analysis. We decided to disallow du-paths from assignment statements like 31 and 32, so we will just consider du-paths that begin with the three "real" defining nodes: DEF(commission, 33), DEF(commission, 37), and DEF(commission, 39). Only one usage node is used: USE (commission, 41).

3.6.7 Define/Use Test Coverage Metrics

The whole point of analyzing a program with definition/use paths is to define a set of test coverage metrics known as the Rapps–Weyuker data flow metrics (Rapps and Weyuker, 1985). The first three of these are equivalent to three of E.F. Miller's metrics in Chapter 8: All-Paths, All-Edges, and All-Nodes. The others presume that define and usage nodes have been identified for all program variables, and that du-paths have been identified with respect to each variable. In the following definitions, *T* is a set of

paths in the program graph *G*(*P*) of a program *P*, with the set *V* of variables. It is not enough to take the cross product of the set of DEF nodes with the set of USE nodes for a variable to define du-paths.

This mechanical approach can result in infeasible paths. In the next definitions, we assume that the define/use paths are all feasible.

Definition

The set *T* satisfies the *All-Defs criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition-clear paths from every defining node of *v* to a use of *v*.

Definition

The set *T* satisfies the *All-Uses criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition-clear paths from every defining node of *v* to every use of *v*, and to the successor node of each USE(*v*, *n*).

Definition

The set *T* satisfies the *All-P-Uses/Some C-Uses criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition-clear paths from every defining node of *v* to every predicate use of *v*; and if a definition of *v* has no P-uses, a definition-clear path leads to at least one computation use.

Definition

The set *T* satisfies the *All-C-Uses/Some P-Uses criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition clear paths from every defining node of *v* to every computation use of *v*; and if a definition of *v* has no C-uses, a definition-clear path leads to at least one predicate use.

Definition

The set *T* satisfies the *All-DU-paths criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition-clear paths from every defining node of *v* to every use of *v* and to the successor node of each USE(*v*, *n*), and that these paths are either single loop traversals or they are cycle free. These test coverage metrics have several set-theory-based relationships, which are referred to as “subsumption” in Rapps and Weyuker (1985). These relationships are shown in Figure 9.5. We now have a more refined view of structural testing possibilities between the extremes of the (typically unattainable) All-Paths metric and the generally accepted minimum, All-Edges. What good is all this? Define/use testing provides a rigorous, systematic way to examine points at which faults may occur.

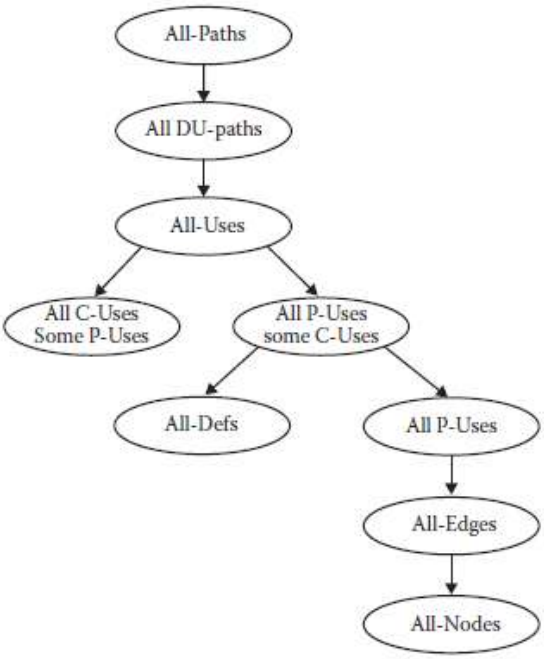

Figure 3.15 Rapps–Weyuker hierarchy of data flow coverage metrics.

3.6.8 Define/Use Testing for Object-Oriented Code

All of the define/use definitions thus far make no mention of where the variable is defined and where it is used. In a procedural code, this is usually assumed to be within a unit, but it can involve procedure calls to improperly coupled units. We might make this distinction by referring to these definitions as "context free"; that is, the places where variables are defined and used are independent. The object-oriented paradigm changes this—we must now consider the define and use locations with respect to class aggregation, inheritance, dynamic binding, and polymorphism. The bottom lineis that data flow testing for object-oriented code moves from the unit level to the integration level.

3.7 Slice-Based Testing

Program slices have surfaced and submerged in software engineering literature since the early 1980s. They were proposed in Mark Weiser's dissertation in 1979 (Weiser, 1979), made more generally available in Weiser (1985), used as an approach to software maintenance in Gallagher and Lyle (1991), and more recently used to quantify functional cohesion in Bieman (1994). During the early 1990s, there was a flurry of published activity on slices, including a paper (Ball and Eick, 1994) describing a program to visualize program slices. This latter paper describes a tool used in industry. (Note that it took about 20 years to move a seminal idea into industrial practice.) Part of the utility and versatility of program slices is due to the natural, intuitively clear intent of the concept. Informally, a program slice is a set of program statements that contributes to, or affects the value of, a variable at some point in a program. This notion of slice corresponds to other disciplines as well. We might study history in terms of slices—US history, European history, Russian history, Far East history, Roman history, and so on. The way such historical slices interact turns out to be very analogous to the way program slices interact. We will start by growing our working definition of a program slice. We continue with the notation we used for define/use paths: a program P that has a program graph $G(P)$ and a set of program variables V. The first try refines the definition in Gallagher and Lyle (1991) to allow nodes in $P(G)$ to refer to statement fragments.

Definition

Given a program P and a set V of variables in P, a *slice on the variable set V at statement n*, written $S(V, n)$, is the set of all statement fragments in P that contribute to the values of variables in V at node n.

One simplifying notion—in our discussion, the set V of variables consists of a single variable, v. Extending this to sets of more than one variable is both obvious and cumbersome. For sets V with more than one variable, we just take the union of all the slices on the individual variables of V. There are two basic questions about program slices, whether they are backward or forward slices, and whether they are static or dynamic. Backward slices refer to statement fragments that contribute to the value of v at statement n. Forward slices refer to all the program statements that are affected by the value of v and statement n. This is one place where the define/use notions are helpful. In a backward slice $S(v, n)$, statement n is nicely understood as a Use node of the variable v, that is, Use(v, n). Forward slices are not as easily described, but they certainly depend on predicate uses and computation uses of the variable v.

The static/dynamic dichotomy is more complex. We borrow two terms from database technology to help explain the difference. In database parlance, we can refer to the intension and extensions of a database. The intension (it is unique) is the fundamental database structure, presumably expressed in a data modeling language.

Populating a database creates an extension, and changes to a populated database all result in new extensions. With this in mind, a static backward slice $S(v, n)$ consists of all the statements in a program that determine the value of variable v at statement n, independent of values used in the statements.

Dynamic slices refer to execution–time execution of portions of a static slice with specific values of all variables in $S(v, n)$. This is illustrated in Figures 9.6 and 9.7.

Listing elements of a slice $S(V, n)$ will be cumbersome because, technically, the elements are program statement fragments. It is much simpler to list the statement fragment numbers in $P(G)$, so we make the following trivial change.

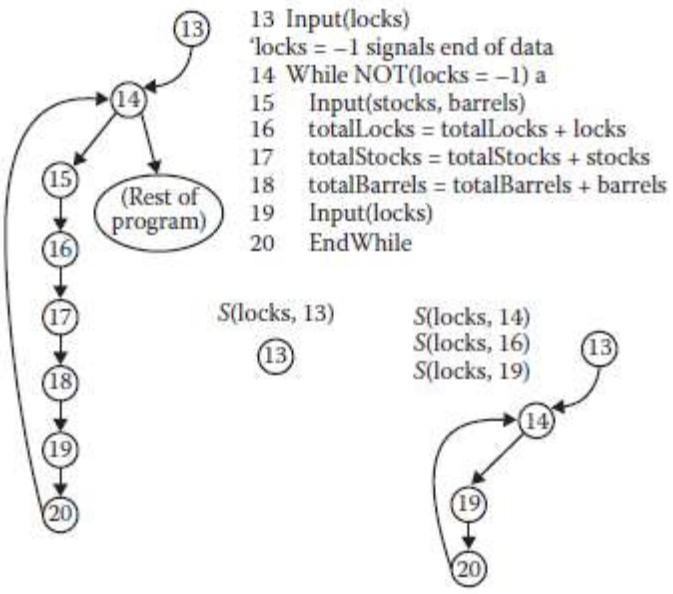

Figure 3.16 Selected slices on locks.

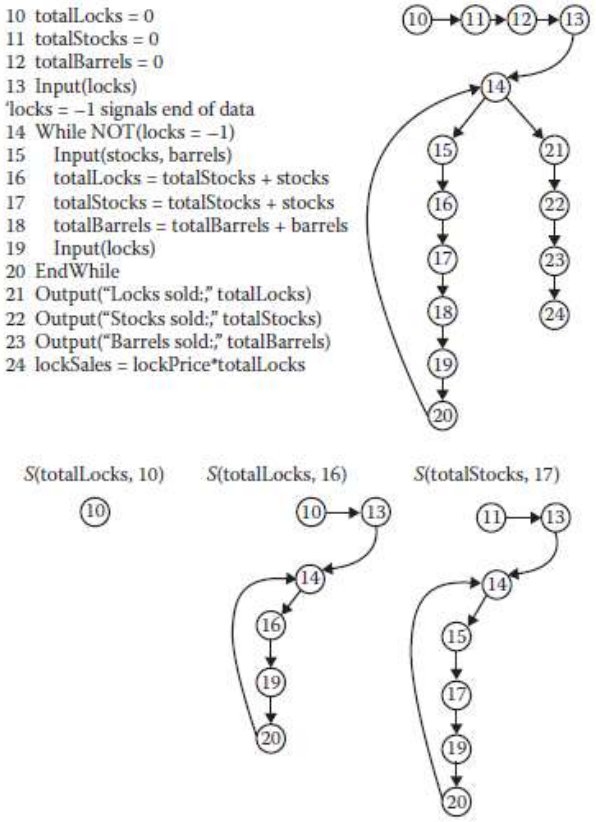

Figure 3.17 Selected slices in a loop.

lattice (a directed, acyclic graph) of static slices, in which nodes are slices and edges correspond to the subset relationship. The "contribute" part is more complex. In a sense, data declaration statements have an effect on the value of a variable. For now, we only include all executable statements. The notion of contribution is partially clarified by the predicate (P-use) and computation (C-use) usage distinction of Rapps and Weyuker (1985), but we need to refine these forms of variable usage. Specifically, the USE relationship pertains to five forms of usage:

P-use used in a predicate (decision)
C-use used in computation
O-use used for output
L-use used for location (pointers, subscripts)
I-use iteration (internal counters, loop indices)

Most of the literature on program slices just uses P-uses and C-uses. While we are at it, we Identify two forms of definition nodes:

I-def defined by input

A-def defined by assignment

Recall our simplification that the slice *S*(*V*, *n*) is a slice on one variable; that is, the set *V* consists of a single variable, *v*. If statement fragment *n* is a defining node for *v*, then *n* is included in the slice. If statement fragment *n* is a usage node for *v*, then *n* is not included in the slice. If a statement is both a defining and a usage node, then it is included in the slice. In a static slice, P-uses and C-uses of other variables (not the *v* in the slice set *V*) are included to the extent that their execution affects the value of the variable *v*. As a guideline, if the value of *v* is the same whether a statement fragment is included or excluded, exclude the statement fragment.

L-use and I-use variables are typically invisible outside their units, but this hardly precludes the problems such variables often create. Another judgment call: here (with some peril), we choose to exclude these from the intent of "contribute." Thus, O-use, L-use, and I-use nodes are excluded from slices.

3.7.1 Example

The commission problem is used in this book because it contains interesting data flow properties, and these are not present in the triangle problem (nor in NextDate). In the following, except where specifically noted, we are speaking of static backward slices and we only include nodes corresponding to executable statement fragments. The examples refer to the source code for the commission problem in Figure 3.11. There are 42 "interesting" static backward slices in our example. They are named in Table 3.8. We will take a selective look at some interesting slices.

The first six slices are the simplest—they are the nodes where variables are initialized.

S_1: S(lockPrice, 7)	S_{15}: S(barrels, 18)	S_{29}: S(barrelSales, 26)
S_2: S(stockPrice, 8)	S_{16}: S(totalBarrels, 18)	S_{30}: S(sales, 27)
S_3: S(barrelPrice, 9)	S_{17}: S(locks, 19)	S_{31}: S(sales, 28)
S_4: S(totalLocks, 10)	S_{18}: S(totalLocks, 21)	S_{32}: S(sales, 29)
S_5: S(totalStocks, 11)	S_{19}: S(totalStocks, 22)	S_{33}: S(sales, 33)
S_6: S(totalBarrels, 12)	S_{20}: S(totalBarrels, 23)	S_{34}: S(sales, 34)
S_7: S(locks, 13)	S_{21}: S(lockPrice, 24)	S_{35}: S(sales, 37)
S_8: S(locks, 14)	S_{22}: S(totalLocks, 24)	S_{36}: S(sales, 39)
S_9: S(stocks, 15)	S_{23}: S(lockSales, 24)	S_{37}: S(commission, 31)
S_{10}: S(barrels, 15)	S_{24}: S(stockPrice, 25)	S_{38}: S(commission, 32)
S_{11}: S(locks, 16)	S_{25}: S(totalStocks, 25)	S_{39}: S(commission, 33)
S_{12}: S(totalLocks, 16)	S_{26}: S(stockSales, 25)	S_{40}: S(commission, 36)
S_{13}: S(stocks, 17)	S_{27}: S(barrelPrice, 26)	S_{41}: S(commission, 37)
S_{14}: S(totalStocks, 17)	S_{28}: S(totalBarrels, 26)	S_{42}: S(commission, 39)

Table 3.8 Slices in Commission Problem

S1: S(lockPrice, 7) = {7}
S2: S(stockPrice, 8) = {8}
S3: S(barrelPrice, 9) = {9}
S4: S(totalLocks, 10) = {10}
S5: S(totalStocks, 11) = {11}
S6: S(totalBarrels, 12) = {12}

Slices 7 through 17 focus on the sentinel controlled while loop in which the totals for locks, stocks, and barrels are accumulated. The locks variable has two uses in this loop: a P-use at fragment 14 and C-use at statement 16. It also has two defining nodes, at statements 13 and 19. The stocks and barrels variables have a defining node at 15, and computation uses at nodes 17 and 18, respectively. Notice the presence of all relevant statement fragments in slice 8. The slices on locks are shown in Figure 3.16.

S7: S(locks, 13) = {13}
S8: S(locks, 14) = {13, 14, 19, 20}
S9: S(stocks, 15) = {13, 14, 15, 19, 20}
S10: S(barrels, 15) = {13, 14, 15, 19, 20}
S11: S(locks, 16) = {13, 14, 19, 20}
S12: S(totalLocks, 16) = {10, 13, 14, 16, 19, 20}
S13: S(stocks, 17) = {13, 14, 15, 19, 20}
S14: S(totalStocks, 17) = {11, 13, 14, 15, 17, 19, 20}
S15: S(barrels, 18) = {12, 13, 14, 15, 19, 20}
S16: S(totalBarrels, 18) = {12, 13, 14, 15, 18, 19, 20}
S17: S(locks, 19) = {13, 14, 19, 20}

Slices 18, 19, and 20 are output statements, and none of the variables is defined; hence, the corresponding statements are not included in these slices.

*S*18: *S*(totalLocks, 21) = {10, 13, 14, 16, 19, 20}
*S*19: *S*(totalStocks, 22) = {11, 13, 14, 15, 17, 19, 20}
*S*20: *S*(totalBarrels, 23) = {12, 13, 14, 15, 18, 19, 20}

Slices 21 through 30 deal with the calculation of the variable sales. As an aside, we could simply write *S*30: *S*(sales, 27) = *S*23 ∪ *S*26 ∪ *S*29 ∪ {27}. This is more like the form that Weiser (1979) refers to in his dissertation—a natural way to think about program fragments. Gallagher and Lyle (1991) echo this as a thought pattern among maintenance programmers. This also leads to Gallager's "slice splicing" concept. Slice *S*23 computes the total lock sales, *S*25 the total stock sales, and *S*28 the total barrel sales. In a bottom–up way, these slices could be separately coded and tested, and later spliced together. "Splicing" is actually an apt metaphor—anyone who has ever spliced a twisted rope line knows that splicing involves carefully merging individual strands at just the right places. (See Figure 9.7 for the effect of looping on a slice.)

*S*21: *S*(lockPrice, 24) = {7}
*S*22: *S*(totalLocks, 24) = {10, 13, 14, 16, 19, 20}
*S*23: *S*(lockSales, 24) = {7, 10, 13, 14, 16, 19, 20, 24}
*S*24: *S*(stockPrice, 25) = {8}
*S*25: *S*(totalStocks, 25) = {11, 13, 14, 15, 17, 19, 20}
*S*26: *S*(stockSales, 25) = {8, 11, 13, 14, 15, 17, 19, 20, 25}
*S*27: *S*(barrelPrice, 26) = {9}
*S*28: *S*(totalBarrels, 26) = {12, 13, 14, 15, 18, 19, 20}
*S*29: *S*(barrelSales, 26) = {9, 12, 13, 14, 15, 18, 19, 20, 26}
*S*30: *S*(sales, 27) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
Slices 31 through 36 are identical. Slice *S*31 is an O-use of sales; the others are all C-uses. Since none of these changes the value of sales defined at *S*30, we only show one set of statement fragment numbers here.
*S*31: *S*(sales, 28) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

The last seven slices deal with the calculation of commission from the value of sales. This is literally where it all comes together.

*S*37: *S*(commission, 31) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31}
*S*38: *S*(commission, 32) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32}
*S*39: *S*(commission, 33) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33}
*S*40: *S*(commission, 36) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 35, 36}
*S*41: *S*(commission, 37) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 35, 36, 37}
*S*42: *S*(commission, 39) = { 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 38, 39}
*S*43: *S*(commission, 41) = { 7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 39}

Looking at slices as sets of fragment numbers (Figure 3.18) is correct in terms of our definition, but it is also helpful to see how slices are composed of sets of previous slices. We do this next, and show the final lattice in Figure 3.19.

S1: S(lockPrice, 7) = {7}
S2: S(stockPrice, 8) = {8}
S3: S(barrelPrice, 9) = {9}
S4: S(totalLocks, 10) = {10}
S5: S(totalStocks, 11) = {11}
S6: S(totalBarrels, 12) = {12}
S7: S(locks, 13) = {13}
S8: S(locks, 14) = S7 ∪ {14, 19, 20}
S9: S(stocks, 15) = S8 ∪ {15}
S10: S(barrels, 15) = S8
S11: S(locks, 16) = S8
S12: S(totalLocks, 16) = S4 ∪ S11 ∪ {16}
S13: S(stocks, 17) = S9 = {13, 14, 19, 20}

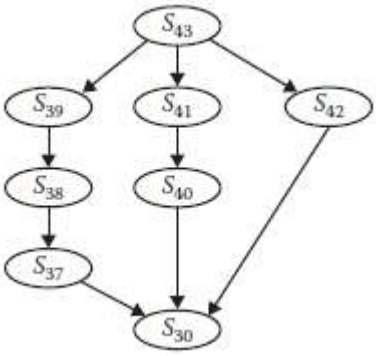

Figure 3.18 Partial lattice of slices on commission.

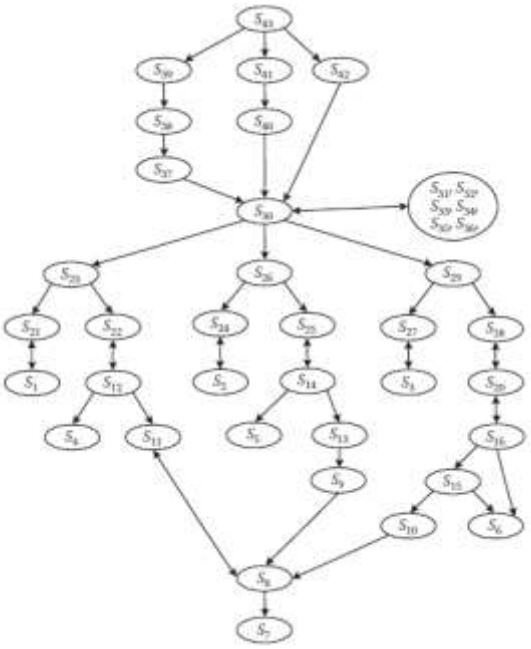

Figure 3.19 Full lattice on commission.

S14: S(totalStocks, 17) = S5 ∪ S13 ∪ {17}
S15: S(barrels, 18) = S6 ∪ S10
S16: S(totalBarrels, 18) = S6 ∪ S15 ∪ {18}

S18: S(totalLocks, 21) = S12
S19: S(totalStocks, 22) = S14
S20: S(totalBarrels, 23) = S16
S21: S(lockPrice, 24) = S1
S22: S(totalLocks, 24) = S12
S23: S(lockSales, 24) = S21 ∪ S22 ∪ {24}
S24: S(stockPrice, 25) = S2
S25: S(totalStocks, 25) = S14
S26: S(stockSales, 25) = S24 ∪ S25 ∪ {25}
S27: S(barrelPrice, 26) = S3
S28: S(totalBarrels, 26) = S20
S29: S(barrelSales, 26) = S27 ∪ S28 ∪ {26}
S30: S(sales, 27) = S23 ∪ S26 ∪ S29 ∪ {27}
S31: S(sales, 28) = S30
S32: S(sales, 29) = S30
S33: S(sales, 33) = S30
S34: S(sales, 34) = S30
S35: S(sales, 37) = S30
S36: S(sales, 39) = S30
S37: S(commission, 31) = S30 ∪ {29, 30, 31}
S38: S(commission, 32) = S37 ∪ {32}
S39: S(commission, 33) = S38 ∪ {33}
S40: S(commission, 36) = S30 ∪ {29, 34, 35, 36}
S41: S(commission, 37) = S40 ∪ {37}
S42: S(commission, 39) = S30 ∪ {29, 34, 38, 39}

S43: S(commission, 41) = S39 ∪ S41 ∪ S42

Several of the connections in Figure 9.9 are double-headed arrows indicating set equivalence. (Recall from Chapter 3 that if A U B and B U A, then $A = B$.) We can clean up Figure 3.19 by removing these, and thereby get a better lattice. The result of doing this is in Figure 3.20.

3.7.2 Style and Technique

When we analyze a program in terms of interesting slices, we can focus on parts of interest while disregarding unrelated parts. We could not do this with du-paths—they are sequences that include statements and variables that may not be of interest. Before discussing some analytic techniques, we will first look at "good style." We could have built these stylistic precepts into the definitions, but then the definitions are more restrictive than necessary.

1. Never make a slice $S(V, n)$ for which variables v of V do not appear in statement fragment n. This possibility is permitted by the definition of a slice, but it is bad practice. As an example, suppose we defined a slice on the locks variable at node 27. Defining such slices necessitates tracking the values of all variables at all points in the program.

2. Make slices on one variable. The set *V* in slice *S*(*V*, *n*) can contain several variables, and sometimes such slices are useful. The slice *S*(*V*, 27) where

V = {lockSales, stockSales, barrelSales}

contains all the elements of the slice *S*30: *S*(sales, 27) except statement 27.

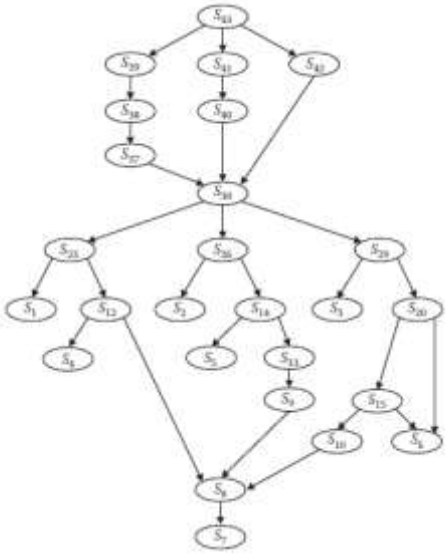

Figure 3.20 Simplified lattice on commission.

3. Make slices for all A-def nodes. When a variable is computed by an assignment statement, a slice on the variable at that statement will include (portions of) all du-paths of the variables used in the computation. Slice *S*30: *S*(sales, 27) is a good example of an A-def slice. Similarly for variables defined by input statements (I-def nodes), such as *S*10: *S*(barrels, 15).

4. There is not much reason to make slices on variables that occur in output statements. Slices on O-use variables can always be expressed as unions of slices on all the A-defs (and I-defs) of the O-use variable.

5. Make slices for P-use nodes. When a variable is used in a predicate, the slice on that variable at the decision statement shows how the predicate variable got its value. This is very useful in decision-intensive programs such as the triangle program and NextDate.

6. Consider making slices compilable. Nothing in the definition of a slice requires that the set of statements is compilable; however, if we make this choice, it means that a set of compiler directive and data declaration statements is a subset of every slice. If we added this same set of statements to all the slices we made for the commission program, our lattices remain undisturbed; however, each slice is separately compilable (and therefore executable). In Chapter 1, we suggested that good testing practices lead to better programming practices. Here, we have a good example. Think about developing programs in terms of compilable slices. If we did this, we could code a slice and immediately test it.

3.7.3 Guidelines and Observations

Dataflow testing is clearly indicated for programs that are computationally intensive. As a corollary, in control intensive programs, if control variables are computed (P-uses), dataflow testing is also indicated. The definitions we made for define/use paths and slices give us very precise ways to describe parts of a program that we would like to test.

There are academic tools that support these definitions, but they haven't migrated to the commercial marketplace. Some pieces are there; you can find programming language compilers that provide on-screen highlighting of slices, and most debugging tools let you "watch" certain variables as you step through a program execution. Here are some tidbits that may prove helpful to you, particularly when you have a difficult module to test.

1. Slices don't map nicely into test cases (because the other, non-related code is still in an executable path). On the other hand, they are a handy way to eliminate interaction among variables. Use the slice composition approach to re-develop difficult sections of code, and these slices before you splice (compose) them with other slices.

2. Relative complements of slices yield a "diagnostic" capability. The relative complement of a set B with respect to another set A is the set of all elements of A that are not elements of B. It is denoted as A -B. Consider the relative complement set S(commission, 48) - S(sales, 35):

S(commission, 48) = {3, 4, 5,36,18,19, 20, 23, 24, 25, 26, 27, 34, 38, 39, 40, 44,45,47}

S(sales, 35) = {3, 4, 5, 36, 18, 19, 20, 23, 24, 25, 26, 27}

S(commission, 48) - S(sales, 35) = {34, 38, 39, 40, 44,45,47}

If there is a problem with commission at line 48, we can divide the program into two parts, the computation of sales at line 34, and the computation of commission between lines 35 and 48. If sales is OK at line 34, the problem must lie in the relative complement; if not, the problem may be in either portion.

3. There is a many-to-many relationship between slices and DD-Paths: statements in one slice may be in several DD-Paths, and statements in one DD-Path may be in several slices. Well-chosen relative complements of slices can be identical to DD-Paths. For example, consider S(commission, 40) -S(commission, 37).

4. If you develop a lattice of slices, it's convenient to postulate a slice on the very first statement. This way, the lattice of slices always terminates in one root node. Show equal slices with a two-way arrow.

5. Slices exhibit define/reference information. Consider the following slices on num_locks:

S(num_locks, 17) = φ
S(num_locks, 24) = {17, 20, 27?}
S(num_locks, 31) = {17, 20, 24, 27}

S(num_locks, 34) = {17, 20, 24, 27}

S(num_locks, 17) is the first definition of num_locks.

S(num_locks, 24) - S(num_locks, 17) is a definition-clear, define reference path.
When slices are equal, the corresponding paths are definition-clear.

Test Execution

Whereas test design, even when supported by tools, requires insight and ingenuity in similar measure to other facets of software design, test execution must be sufficiently automated for frequentreexecution without little human involvement. This chapter describes approaches for creating the run-time support for generating and managing test data, creating scaffolding for test execution, and automatically distinguishing between correct and incorrect test case executions.

3.8 From Test Case Specifications to Test Cases

If the test case specifications produced in test design already include concrete input values and expected results, as for example in the category-partition method, then producing a complete test case may be as simple as filling a template with those values. A more general test case specification (e.g., one that calls for "a sorted sequence, length greater than 2, with items in ascending order with no duplicates") may designate many possible concrete test cases, and it may be desirable to generate just one instance or many. There is no clear, sharp line between test case design and test case generation. A rule of thumb is that, while test case design involves judgment and creativity, test case generation should be a mechanical step. Automatic generation of concrete test cases from more abstract test case specifications reduces the impact of small interface changes in the course of development. Corresponding changes to the test suite are still required with each program change, but changes to test case specifications are likely to be smaller and more localized than changes to the concrete test cases.

3.8.1 Scaffolding

During much of development, only a portion of the full system is available for testing. In modern development methodologies, the partially developed system is likely to consist of one or more runnable programs and may even be considered a version or prototype of the final system from very early in construction, so it is possible at least to execute each new portion of the software as it is constructed, but the external interfaces of the evolving system may not be ideal for testing; often additional code must be added. For example, even if the actual subsystem for placing an order with a supplier is available and fully operational, it is probably not desirable to place a thousand supply orders each night as part of an automatic test run.

A common estimate is that half of the code developed in a software project is scaffolding of some kind, but the amount of scaffolding that must be constructed with a software project can vary widely, and depends both on the application domain and the architectural design and build plan, which can reduce cost by exposing appropriate interfaces and providing necessary functionality in a rational order.

The purposes of scaffolding are to provide controllability to execute test cases and observability to judge the outcome of test execution. Sometimes scaffolding is required to simply make a module executable, but even in incremental development with immediate integration of each module, scaffolding for controllability and observability may be required because the external interfaces of the system may not provide sufficient control to drive the module under test through test cases, or sufficient observability of the effect. It may be desirable to substitute a separate test "driver" program for the full system, in order to provide more direct control of an interface or to remove dependence on other subsystems.

3.8.2 Generic versus Specific Scaffolding

The simplest form of scaffolding is a driver program that runs a single, specific test case. If, for example, a test case specification calls for executing method calls in a particular sequence, this is easy to accomplish by writing the code to make the method calls in that sequence. Writing hundreds or thousands of such test-specific drivers, on the other hand, may be cumbersome and a disincentive to thorough testing. At the very least one will want to factor out some of the common driver code into reusable modules. Sometimes it is worthwhile to write more generic test drivers that essentially interpret test case specifications. At least some level of generic scaffolding support can be used across a fairly wide class of applications.

3.8.3 Test Oracles

It is little use to execute a test suite automatically if execution results must be manually inspected to apply a pass/fail criterion. Relying on human intervention to judge test outcomes is not merely expensive, but also unreliable. Even the most conscientious and hard-working person cannot maintain the level of attention required to identify one failure in a hundred program executions, little more one or ten thousand. That is a job for a computer. Software that applies a pass/fail criterion to a program execution is called a *test oracle*, often shortened to *oracle*. In addition to rapidly classifying a large number of test case executions, automated test oracles make it possible to classify behaviors that exceed human capacity in other ways, such as checking real-time response against latency requirements or dealing with voluminous output data in a machine-readable rather than human-readable form.

Capture-replay testing, a special case of this in which the predicted output or behavior is preserved from an earlier execution, is discussed in this chapter. A related approach is to capture the output of a trusted alternate version of the program under test. For example, one may produce output from a trusted implementation that is for some reason unsuited for production use; it may too slow or may depend on a component that is not available in the production environment.

It is not even necessary that the alternative implementation be *more* reliable than the program under test, as long as it is sufficiently different that the failures of the real and alternate version are likely to be independent, and both are sufficiently reliable that not too much time is wasted determining which one has failed a particular test case on which they disagree.

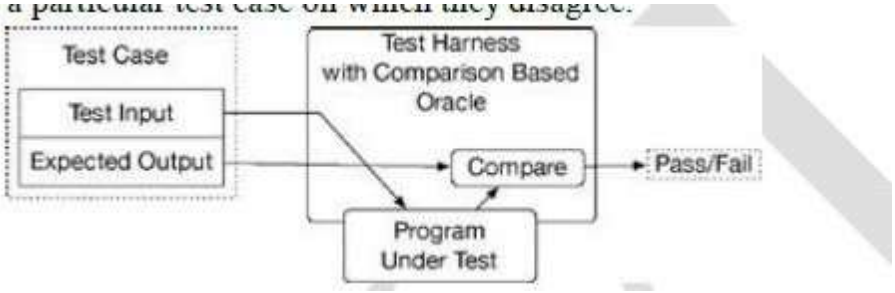

Figure 3.21: A test harness with a comparison-based test oracle processes test cases consisting of (program input, predicted output) pairs.

independently compute the route to ascertain that it is in fact a valid route that starts at *A* and ends at *B*. Oracles that check results without reference to a predicted output are often partial, in the sense that they can detect some violations of the actual specification but not others. They check necessary but not sufficient conditions for correctness. For example, if the specification calls for finding the optimum bus route according to some metric, partial oracle a validity check is only a partial oracle because it does not check optimality. Similarly, checking that a sort routine produces sorted output is simple and cheap, but it is only a partial oracle because the output is also required to be a permutation of the input. A cheap partial oracle that can be used for a large number of test cases is often combined with a more expensive comparison-based ora oracle that can be used with a smaller set of test cases for which predicted output has been obtained.

Ideally, a single expression of a specification would serve both as a work assignment and as a source from which useful test oracles were automatically derived. Specifications are often incomplete, and their informality typically makes automatic derivation of test oracles impossible. The idea is nonetheless a powerful one, and wherever formal or semiformal specifications (including design models) are available, it is worth- while to consider whether test oracles can be derived from them.

3.8.4 Self-Checks as Oracles

A program or module specification describes *all* correct program behaviors, so an oracle based on a specification need not be paired with a particular test case.

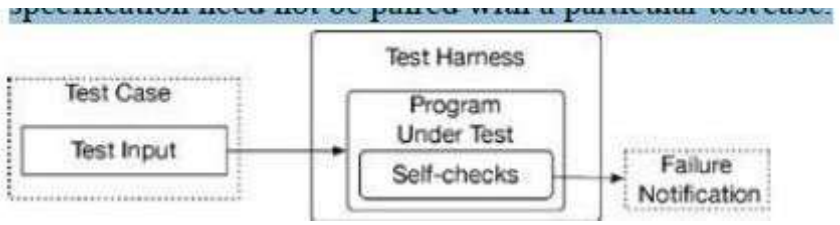

Figure 3.22: When self-checks are embedded in the program, test cases need not include predicted outputs.

Self-check assertions may be left in the production version of a system, where they provide much better diagnostic information than the uncontrolled application crash the customer may otherwise report. If this is not acceptable - for instance, if the cost of a runtime assertion check is too high - most tools for assertion processing also provide controls for activating and deactivating assertions. It is generally considered good design practice to make assertions and self-checks be free of side- effects on program state. Side-effect free assertions are essential when assertions may be deactivated, because otherwise suppressing assertion checking can introduce program failures that appear only when one is *not* testing. Self-checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specifications, rather than overall program behavior. Devising program assertions that correspond in a nuartal way to specifications (formal or informal) poses two main challenges: bridging the gap between concrete execution values and abstractions used in specification, and dealing in a reasonable way with quantification over collections of values.

$$(|< \boldsymbol{K}, \boldsymbol{V} > \in \emptyset(\boldsymbol{dict})|)$$
$$\boldsymbol{O} = \boldsymbol{dict}.\boldsymbol{get}(\boldsymbol{k})$$
$$(|\boldsymbol{O} = \boldsymbol{V}|)$$

φ is an abstraction function that constructs the abstract model type (sets of key, value pairs) from the concrete data structure. φ is a logical association that need not be implemented when reasoning about program correctness. To create a test oracle, it is useful to have an actual implementation of φ. For this example, we might implement a special observer method that creates a simple textual representation of the set of (key, value) pairs. Assertions used as test oracles can then correspond

directly to the specification. Besides simplifying implementation of oracles by implementing this mapping once and using it in several assertions, structuring test oracles to mirror a correctness argument is rewarded when a later change to the program invalidates some part of that argument

In addition to an abstraction function, reasoning about the correctness of internal structures usually involves structural invariants, that is, properties of the data structure that are preserved by all operations. Structural invariants are good candidates for self checks implemented as assertions. They pertain directly to the concrete data structure implementation, and can be implemented within the

module that encapsulates that data structure. For example, if a dictionary structure is implementedas a red-black tree or an AVL tree, the balance property is an invariant of the structure that can be checked by an assertion within the module.

1 **package** org.eclipse.jdt.internal.ui.text;
2 **import** java.text.CharacterIterator;
3 **import** org.eclipse.jface.text.Assert; 4 /**
*5 *A <code>CharSequence</code> based implementation of*
*6 * <code>CharacterIterator</code>. 7 * @since 3.0*
8 */
9 **public class** SequenceCharacterIterator **implements** CharacterIterator { 13 ...
14 **private void** invariant() {
15 Assert.isTrue(fIndex >= fFirst);
16 Assert.isTrue(fIndex <= fLast); 17 }

49 ...
50 **public** SequenceCharacterIterator(CharSequence sequence, **int** first, **int** last)
51 **throws** IllegalArgumentException {
52 **if** (sequence == null)
53 **throw new** NullPointerException();

```
54          if (first < 0 || first > last)
55              throw new IllegalArgumentException();
56          if (last > sequence.length())
57              throw new IllegalArgumentException();
58          fSequence= sequence;
59          fFirst= first;
60          fLast= last;
61          fIndex= first;
62          invariant();
63      }
143 ...
144     public char setIndex(int position) {
145         if (position >= getBeginIndex() && position <= getEndIndex())
146             fIndex= position;
147         else
148             throw new IllegalArgumentException();
149
150         invariant();
151         return current();
152     }
263 ...
264 }|
```

There is a natural tension between expressiveness that makes it easier to write and understand specifications, and limits on expressiveness to obtain efficient implementations. It is not much of a stretch to say that programming languages are just formal specification languages in which expressiveness has been purposely limited to ensure that specifications can be executed with predictable and satisfactory performance.

An important way in which specifications used for human communication and reasoning about programs are more expressive and less constrained than programming languages is that they freely quantify over collections of values. For example, a specification of database consistency might state that account identifiers are unique; that is, *for all* account records in the database, there *does not exist* another account record with the same identifier.

The problem of quantification over large sets of values is a variation on the basic problem of program testing, which is that we cannot exhaustively check all program behaviors. Instead, we select a tiny fraction of possible program behaviors or inputs as representatives. The same tactic is applicable to quantification in specifications. If we cannot fully evaluate the specified property, we can at least select some elements to check (though at present we know of no program assertion packages that

support sampling of quantifiers). For example, although we cannot afford to enumerate all possible paths between two points in a large map, we may be able to compare to a sample of other paths found by the same procedure. program may use ghost variables to track entry and exit of threads from a critical section. The post condition of an in-place sort operation will state that the new value is sorted and a permutation of the input value. This permutation relation refers to both the "before" and "after" values of the object to be sorted. A run-time assertion system must manage ghost variables and retained "before" values and must ensure that they have no side-effects outside assertion checking.

It may seem unreasonable for a program specification to quantify over an infinite collection, but in fact it can arise quite naturally when quantifiers are combined with negation. If we say "there is no integer greater than 1 that divides *k* evenly," we have combined negation with "there exists" to form a statement logically equivalent to universal ("for all") quantification over the integers. We may be clever enough to realize that it suffices to check integers between 2 and $\sqrt{k}$, but that is no longer a direct translation of the specification statement.

3.8.5 Capture and Replay

Sometimes it is difficult to either devise a precise description of expected behavior or adequately characterize correct behavior for effective self-checks. For example, while many properties of a program with a graphical interface may be specified in a manner suitable for comparison-based or self-check oracles, some properties are likely to require a person to interact with the program and judge its behavior. If one cannot completely avoid human involvement in test case execution, one can at least avoid unnecessary repetition of this cost and opportunity for error. The principle is simple. The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged (by the human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated retesting. The savings from automated retesting with a captured log depends on how many build- and-test cycles we can continue to use it in, before it is invalidated by some change to the program.

inguishing between significant and insignificant variations from predicted behavior, in order to prolong the effective lifetime of a captured log, is a major challenge for capture/replay testing. Capturing events at a more abstract level suppresses insignificant changes. For example, if we log only the actual pixels of windows and menus, then changing even a typeface or background color can invalidate an entire suite of execution logs. Mapping from concrete state to an abstract model of interaction sequences is sometimes possible but is generally quite limited. A more fruitful approach is capturing input and output behavior at multiple levels of abstraction within the implementation