PATH TESTING, DATA FLOW 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;

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.

Constructing a program graph from a given program is an easy process. It's illustrated here with the Pascal implementation of the Triangle program from Chapter 2. Line numbers refer to statements and statement fragments. There is an element of judgment here: sometimes it is convenient to keep a fragment (like a BEGIN) as a separate node, as at line 4. Other times is seems better to include this with another portion of a statement: the BEGIN at line 13 could really be merged with the THEN on line 12. We will see that 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, there might be several distinct program graphs, all of which reduce to a unique DD-Path graph.) We also need to decide whether to associate nodes with non-executable statements such as variable and type declarations: here we do not. with the potentially large number of execution paths in a program

In this program, there are 5 paths from node B to node F in the interior of the loop. If the loop may have up to 18 repetitions, there are some 4.77 trillion distinct program execution paths.

```
1. program triangle (input, output) ;
2. VAR a, b, c    : integer;
3.    IsATriangle : boolean;
4. BEGIN
5. writeln('Enter three integers which are sides of a triangle:');
6. readln (a,b,c);
7. writeln('Side A is ',a, 'Side B is ',b, 'side C is ',c);
8. IF (a < b + c) AND (b < a + c) AND (c < a + b)
9.    THEN IsATriangle :=TRUE
10.   ELSE IsATriangle := FALSE ;
11. IF IsATriangle
12.   THEN
13.   BEGIN
14.   IF (a = b) XOR (a = c) XOR (b = c) AND NOT((a=b) AND (a=c))
15.      THEN Writeln ('Triangle is Isosceles') ;
16.   IF (a = b) AND (b = c)
17.      THEN Writeln ('Triangle is Equilateral') ;
18.   IF (a <> b) AND (a <> c) AND (b <> c)
19.      THEN Writeln ('Triangle is Scalene') ;
20.   END
21.   ELSE WRITELN('Not a Triangle') ;
22. END.
```

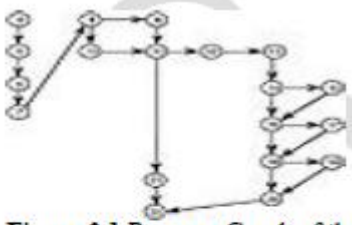

Figure 3.1 Program Graph of the Pascal Triangle Program

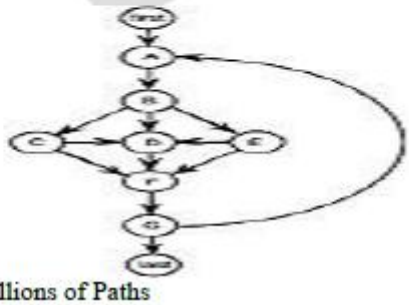

Figure 3.2 Trillions of Paths

DD-Paths

The best known form of structural testing is based on a construct known as a decision-to-decision path (DD-Path) [Miller 77]. 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. There are no internal branches 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 block structured languages (Pascal, Ada, C), 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.

Definition

A ***DD-Path*** is a chain 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 " ornmal 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 (non-trivial) chain.

Program Graph Nodes

	DD-Path Name	Case of Definition
4	first	1
5 - 8	A	5
9	B	4
10	C	4
11	D	3
12 - 14	E	5
15	F	4
16	G	3
17	H	4
18	I	3
19	J	4
20	K	3
21	L	4
22	last	2

Node 4 is a Case 1 DD-Path, we'll call it "first"; similarly, node 22 is a Case 2 DD-Path, and we'll call it "last". Nodes 5 through 8 are a Case 5 DD-Path. 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 went beyond node 8 to include nodes 9 and 10, these would both be 2-connected to the rest of the chain, but they are 1-connected to each other. If we stopped at node 7, we would violate the "maximal" criterion. Node 11 is a Case 3 DD-Path, which forces nodes 9 and 10 to be individual DD-Paths by case 4. Nodes 12 through 14 are a case 5 DD-Path by the same reasoning as for nodes 5 - 8. Nodes 14 through 20 correspond to a sequence of IF-THEN statements. Nodes 16 and 18 are both Case 3 DD-Paths, and this forces nodes 15, 17, and 19 to be Case 4 DD-Paths. Node 20 is a Case 3 DD-Path, and node 21 is a Case 4 DD-Path. All of this is summarized in Table 1, where the
DD-Path names correspond to the DD-Path 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 BEGIN

. . END blocks of computational statements, we would have longer DD-Paths, as we do in the commission problem.

We can now define the DD-Path graph of a program.

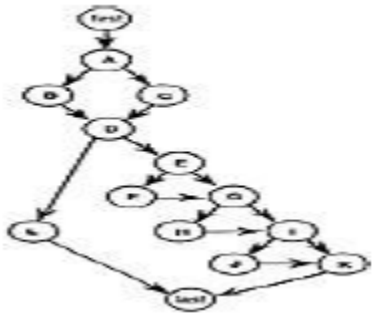

Figure 3.3 DD-Path Graph for the Triangle Program

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. 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.

Testers shouldn't be intimidated by this process — there are high quality commercial tools that 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's reasonable to make DD-Path graphs for programs up to about 100 source lines. Beyond that, most testers look for a tool.

Test Coverage Metrics

The *raison d'être* of DD-Paths is that they enable very precise descriptions of test coverage. Recall (from Chapter 8) that one of the fundamental limitations of functional 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.

There are several widely accepted test coverage metrics; most of those in Table 2 are due to the early work of E. F. Miller [Miller 77]. 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. The statement coverage metric (C_0) is still widely accepted: it is mandated by ANSI Standard 187B, and has been used successfully throughout IBM since the mid-1970s.

Table 2 Structural Test Coverage Metrics **Metric**

	Description of Coverage
C_0	Every statement
C_1	Every DD-Path (predicate outcome)
C_1^p	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

Metric Based Testing

The test coverage metrics in Table 2 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 in terms of the metrics in Table 2. 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 to be nodes. For the remainder of this section, the statement fragment formulation is "in effect".

Statement 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.

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), as opposed to just every node. 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 what else we might do to 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.

Dependent Pairs of DD-Paths

The C_d metric foreshadows the dataflow 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 9.4, B and D are such a pair, so are DD-Paths C and L. Simple DD- Path coverage might not exercise these dependencies, thus a deeper class of faults would not be revealed.

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.

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.

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.

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 re pesct 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) and which has 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 such that the vectors are independent of each other and they "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 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 "OK", we could hope that everything that can be expressed in terms of the basis is also "OK". In this section, we examine the early work of Thomas McCabe, who recognized this possibility in the mid-1970s.

McCabe's Basis Path Method

There is some confusion 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 het 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 9.6) to a strongly connected directed graph obtained by adding one edge from the sink to the source node . Adding an edge clearly affects value computed by the formula, but it shouldn't affect the number of circuits. Here's a way to resolve the apparent inconsistency:

$$V(G) = e - n + 2p$$
$$= 10 - 7 + 2(1) = 5,$$

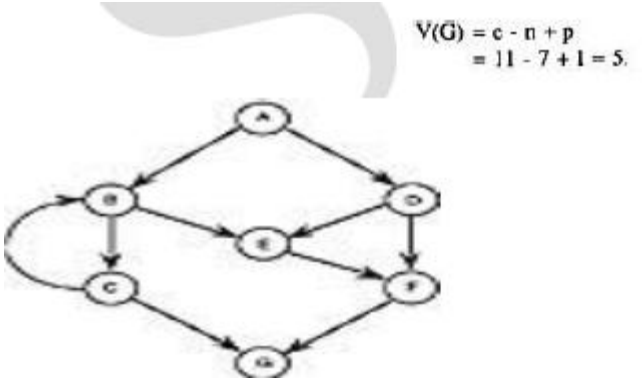

Figure 3.4 McCabe's Control Graph

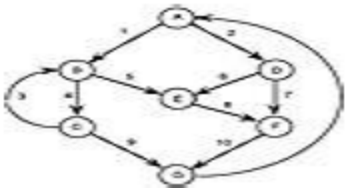

Figure 3.5 McCabe's Derived Strongly Connected Graph

there are five linearly independent circuits. If we now delete the added edge form 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

We can force this beginning 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 in which rows correspond to paths, andcolumns correspond to edges, as in Table 3. 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. Since edge 4 is traversed twice by path p2, that is the entry for the edge 4 column.

Table 3 Path/Edge Traversal path \ edges travesed

	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

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 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 should check the linear combinations of the two example paths. The addition and multiplication are performed on the column entries.

McCabe next develops an algorithmic procedure (called the “baseline method”) to determine a set of basis paths. The method begins with het 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, so 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: this is not problematic, because there is no requirement that a basis be unique.

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 there are two major soft spots in the McCabe view: one is that testing the set of basis paths is sufficient (it’s 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 backwards? Undo the most recent execution of p1? Don't do p1 next time? Mathematical sophistries like this are a real turn-off to practitioners looking for solutions to their very real problems.

We begin with a baseline path that corresponds to a scalene triangle, say with sides 3, 4, 5. This test case will traverse the path p1. Now if we flip the decision at node A, we get path p2. Continuing the procedure, we flip the decision at node D, 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 E. When we flip node E, we get the path p4. Next we flip node G to get p5. Finally, (we know we're done, because there are only 6 basis paths) we flip node I to get p6. This procedure yields the following basis paths:

p1: A-B-D-E-G-I-J-K-Last
p2: A-C-D-E-G-I-J-K-Last
p3: A-B-D-L-Last
p4: A-B-D-E-F-G-I-J-K-Last
p5: A-B-D-E-F-G-H-I-J-K-Last
p6: A-B-D-E-F-G-H-I-K-Last

Time for a reality check: if you follow paths p2, p3, p4, p5, and p6, you find that they are all infeasible. Path p2 is infeasible, because passing through node C means the sides are not a triangle, so none of the sequel decisions can be taken. Similarly, in p3, passing through node B means the sides do form a triangle, so node L cannot be traversed. The others are all infeasible because they involve cases where a triangle is of two types (e.g., isosceles and equilateral). The problem here is that there are several inherent dependencies in the triangle problem. One is that if three integers constitute sides of a triangle, they must be one of the three possibilities: equilateral, isosceles, or scalene. A second dependency is that the three possibilities are mutually exclusive: if one is true, the other two must be false.

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 functional testing, where we addressed data dependencies in the decision table. Here we are dealing with code level dependencies, and these are absolutely incompatible with the latent assumption that basis paths are independent. McCabe's procedure successfully identifies basis paths that are topologically independent, but 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 several rules:

If node B is traversed, then we must traverse nodes D and E.

If node C is traversed, then we must traverse nodes D and L.

If node E is traversed, then we must traverse one of nodes F, H, and J.

If node F is traversed, then we cannot traverse nodes H and J.

If node H is traversed, then we cannot traverse nodes F and J.

If node J is traversed, then we cannot traverse nodes F and I.

Taken together, these rules, in conjunction with McCabe's baseline method, will yield the following feasible basis path set:

fp1: A-C-D-L-Last	(Not a triangle)
fp2: A-B-D-E-F-G-I-K-Last	(Isosceles)
fp3: A-B-D-E-G-H-I-K-Last	(Equilateral)
fp4: A-B-D-E-G-I-J-K-Last	(Scalene)

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 82], which is just the cyclomatic complexity of yet another form of condensation graph. Recall that condensation graphs are a way of simplifying an existing graph; so far our simplifications have been based on removing either strong components or DD-Paths

Each of these "unstructures" contains three distinct paths, as opposed to the two paths present in the corresponding structured programming constructs, so one conclusion is that such violations increase cyclomatic complexity. The *piece d' resistance* of McCabe's analysis is that these unstructures cannot occur by themselves: if there is one in a program, there must be at least one more, so a program cannot be just slightly unstructured. Since these increase cyclomatic complexity, the minimum number of test cases is thereby increased. In the next chapterThe bottom line for testers is this: programs with high cyclomatic complexity require more testing. 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 that exceeds the guidelines, often the best choice is to eliminate the unstructures.

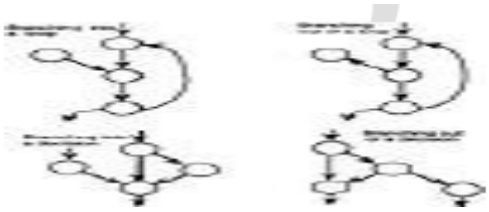

Figure 3.6 Violations of Structured Programming

Guidelines and Observations

In our study of functional testing, we observed that gaps and redundancies can both exist, and at the same time, cannot be recognized. The problem was that functional testing removes us "too far" from code. The path testing approaches to structural 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. In the next chapter, we look at dataflow based testing. These techniques move closer to the code, so the pendulum will swing back from the path analysis extreme.

McCabe 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" [McCabe 82]. He was referring to the DD-Path coverage metric (which is equivalent to the predicate outcome metric) and the cyclomatic complexity metric that requires at least the cyclomatic number of distinct program paths must be traversed. Basis path testing therefore gives us a lower bound on how much testing is necessary.

Path based testing also provides us with a set of metrics that act as cross checks on functional 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 (rain, mi n+, 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, there is an opportunity for real testing craftsmanship. The coverage metrics in Table 2 can operate in two ways: 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

cross check 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

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.

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.

```
1   program lock-stock_and_barrel
2   const
3       lock_price = 45.0;
4       stock_price = 30.0;
5       barrel_price 25.0;
6   type
7       STRING_30 = string[30]; {Salesman's Name}
8   var
9       locks, stocks, barrels, num_locks, num_stocks,
10      num_barrels, salesman_index, order_index : INTEGER;
11      sales, commission : REAL;
12      salesman :
STRING_30; 13
14    BEGIN {program lock_stock_and_barrel}
15   FOR salesman_index := 1 TO 4 DO
16    BEGIN
17      READLN(salesman);
18      WRITELN ('Salesman is ', salesman);
19     num_locks := 0;
20     num_stocks := 0;
21     num_barrels := 0;
22     READ(locks);
23     WHILE locks <> -1 DO
24      BEGIN
25        READLN (stocks, barrels);
26        num_locks := num_locks + locks;
27        num_stocks := num_stocks + stocks;
28        num_barrels := num_barrels + barrels;
29       READ(locks);
30      END; (WHILE locks)
31    READLN;
32    WRITELN('Sales for ',salesman);
33    WRITELN('Locks sold: ', num_locks);
34    WRITELN('Stocks sold: ', num_stocks);
35    WRITELN('Barrels sold: ', num_barrels);
36    sales := lock_price*num_locks + stock_price*num_stocks
          + barrel_price*num_barrels;
37    WRITELN('Total sales: ', sales:8:2);
```

```
38    WRITELN;
39    IF (sales > 1800.0) THEN
40     BEGIN
41       commission := 0.10 * 1000.0;
42       commission := commission + 0.15 * 800.0;
43       commission := commission + 0.20 * (sales-1800.0);
44     END;
45    ELSE IF (sales > 1000.0) THEN
46     BEGIN
47       commission := 0.10 * 1000.0;
48       commission := commission + 0.15*(sales - 1000.0);
49      END

50     ELSE commission := 0.10 * sales;
51      WRITELN('Commission is $',commission:6:2);
52   END; (FOR salesman)
53  END. {program lock_stock_and-barrel}
```

	Nodes
1	14
2	15-22
3	23
4	24-30

5	31-39
6	40-44
7	45
8	46-49
9	50
10	51,52
11	53

Du-paths p1 and p2 refer to the priming value of locks which is read at node 22: locks has a predicate use in the While statement (node 23), and if the condition is true (as in path p2), a computation use at statement 26. The other two du-paths start near the end of the While loop and occur when the loop repeats. If we "extended" paths p1 and p3 to include node 31,

p1' = <22, 23, 31>

p3' = <29, 30, 23, 31>

then the paths p1', p2, p3', and p4 form a very complete set of test cases for the While-loop: bypass the loop, begin the loop, repeat the loop, and exit the loop. All of these du-paths are definition-clear.

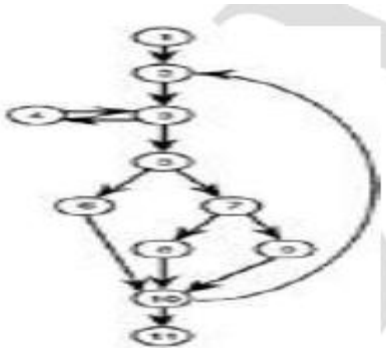

Figure 3.7 DD-Path Graph of the Commission Program

Table 2 Define/Use Information for locks, stocks, and num_locks Variable

	Defined at	Used at	Comment
locks	9		(to compiler)
locks	22		READ
locks		23	predicate use
locks		26	computation use
locks	29		READ
stocks	9		(to compiler)
stocks	25		READ
stocks		27	computation use
num_locks	9		(to compiler)
num_locks	19		assignment
num_locks	26		assignment
num_locks		26	computation use
num_locks		33	WRITE
num_locks		36	computation use

Table 3 Define/Use Information for Sales and Commission Variable

	Defined at	Used at	Comment
sales	11		(to compiler)
sales	36		assignment
sales		37	WRITE
sales		39	predicate use
sales		43	computation use
sales		45	predicate use
sales		48	computation use
sales		50	computation use
commission	11		(to compiler)
commission	41		assignment

commission	42		assignment
commission		42	computation use
commission	43		assignment
commission		43	computation use
commission	47		assignment
commission	48		assignment
commission		48	computation use
commission	50		assignment
commission		51	WRITE

The du-paths for num_locks will lead us to typical test cases for computations. With two defining nodes (DEF(num_locks, 19) and DEF(num_locks, 26)) and three usage nodes (USE(num_locks, 26), USE(num_locks, 33), USE(num_locks, 36)), we might expect six du-paths.

p6 = <19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33>

We have ignored the possible repetition of the While-loop. We could highlight this by noting that the subpath <26, 27, 28, 29, 30, 22, 23, 24, 25> might be traversed several times. Ignoring this for now, we still have a du-path that fails to be definition-clear. If there is a problem with the value of num_locks at node 33 (the WRITE statement), we should look at the intervening DEF(num_locks, 26) node.

The next path contains p6; we can show this by using a path name in place of its corresponding node sequence:

p7 = <19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36>
p7 = < p6, 34, 35, 36>

Du-path p7 is not definition-clear because it includes node 26.

Subpaths that begin with node 26 (an assignment statement) are interesting. The first, <26, 26>, 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 19, (see path p5). The remaining two du- paths are both subpaths of

side of the assignment refers to a value defined at node 19, (see path p5). The remaining two du- paths are both subpaths of p7:

p7: p8 = <26, 27, 28, 29, 30, 31, 32, 33>
p9 = <26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36>

Du-path Test Coverage Metrics

The whole point of analyzing a program as in the previous section is to define a set of test coverage metrics known as the Rapps-Weyuker data flow metrics [Rapps 85]. The first three of these are equivalent to three of E. F. Miller's metrics in Chapter 9: 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 (sub)paths in the program graph G(P) of a program P, with the set V of variables.

Definition

The set T satisfies the ***All-Defs*** criterion for the program P iff for every variable v definition clear (sub)paths from every defining node of v to a use of v.

V, T contains

Definition

The set T satisfies the ***All-Uses*** criterion for the program P iff for every variable v V, T contains definition-clear (sub)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 iff for every variable v V, T contains definition-clear (sub)paths from every defining node of v to every predicate use of v, and if a definition of v has no P-uses, there is a definition-clear path to at least one computation use.

	Variable	Def Node	Use Node
1	locks	22	23

2	locks	22	26
3	locks	29	23
4	locks	29	26
5	stocks	25	27
6	barrels	25	28
7	num_locks	19	26
8	num_locks	19	33
9	num_locks	19	36
10	num_locks	26	33
11	num_locks	26	36
12	num_stocks	20	27
13	num_stocks	20	34
14	num_stocks	20	36
15	num_stocks	27	34
16	num_stocks	27	36
17	num_barrels	21	28
18	num_stocks	21	35
19	num_stocks	21	36
20	num_stocks	28	35
21	num_stocks	28	36
22	sales	36	37
23	sales	36	39
24	sales	36	43
25	sales	36	45

26	sales	36	48
27	sales	36	50
28	commission	41	42
29	commission	42	43
30	commission	43	51
31	commission	47	48
32	commission	48	51
33	commission	50	51

Definition

The set T satisfies the ***All-C-Uses /Some P-Uses*** criterion for the program P iff for every variable v V, T contains definition-clear (sub)paths from every defining node of v to every computation use of v, and if a definition of v has no C-uses, there is a definition-clear path to at least one predicate use.

Definition

The set T satisfies the ***All-DU-paths*** criterion for the program P iff for every variable v V, T contains definition-clear (sub)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.

Slice-Based Testing

Program slices have surfaced and submerged in software engineering literature since the early

1980s. They were originally proposed in [Weiser 85], used as an approach to software maintenance in [Gallagher 91], and most recently used to quantify functional cohesion in [Bieman 94]. Part of this versatility is due to the natural, intuitively clear intent of the program slice concept. Informally, a program slice is a set of program statements that contribute to, or affect a value for a variable at some point in the 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.

	locks	stocks	barrels	sales	commission	DD-Paths
1	5	5	5	500	50	1-5, 7, 9, 10, 11
2	15	15	15	1500	175	1-5, 7, 8, 10, 11
3	25	25	25	2500	360	1-5, 9, 10, 11

Table 6 Du-Path Coverage of Decision Table Functional Test Cases **Du-Path**

	Case 1	Case 2	Case 3
1	X	X	X
2	X	X	X
3	X	X	X
4	X	X	X
5	X	X	X
6	X	X	X
7	X	X	X

8			
9			
10	X	X	X
11	X	X	X
12	X	X	X
13	X	X	X
14			
15	X	X	X
16	X	X	X
17	X	X	X
18			
19			
20	X	X	X
21	X	X	X
22	X	X	X
23	X	X	X
24			X
25	X	X	
26		X	
27	X		
28			X
29			X
30			X
31		X	

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 statements in P that contribute to the values of variables in V.

Listing elements of a slice S(V,n) will be cumbersome, because the elements are program statement fragments. Since it is much simpler to list fragment numbers in P(G), we make the following trivial change (it keeps the set theory purists happy):

Definition

Given a program P, and a program graph G(P) in which statements and statement fragments are numbered, and a set V of variables in P, the ***slice on the variable set V at statement fragment n,***

written S(V,n), is the set node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n.

The idea of slices is to separate a program into components that have some useful meaning. First, we need to explain two parts of the definition. Here we mean "prior to" in the dynamic sense, so a slice captures the execution time behavior of a program with respect to the variable(s) in the slice. Eventually, we will develop a lattice (a directed, acyclic graph) of slices, in which nodes are slices, and edges correspond to the subset relationship.

The "contribute" part is more complex. In a sense, declarative statements (such as CONST and TYPE) have an effect on the value of a variable. A CONST definition sets a value that can never be changed by a definition node, and the difference between INTEGER and REAL variables can be a source of trouble. One resolution might be to simply exclude all non-executable statements. We will include CONST declarations in slices. The notion of contribution is partially clarified by the predicate (P-use) and computation (C-use) usage distinction of [Rapps 85], 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)

While we're at it, we identify two forms of definition nodes:

I-def defined by input

A-def defined by assignment

For now, presume 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. 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 modules, 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..

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 ba dpractice. 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, 36) where

V= { num_locks, num_stocks, num_barrels }

contains all the elements of the slice S({sales}, 36) except the CONST declarations and statement 36. Since these two slices are so similar, why define the one in terms of C-uses?

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({sales}, 36) is a good example of an A-def slice.

4. 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 like the Triangle program and NextDate.

5. Slices on non-P-use usage nodes aren't very interesting. We discussed C-use slices in point 2,

where we saw they were very redundant with the A-def slice. 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. Slices on I-use and O-use variables are useful during debugging, but if they are mandated for all testing, the test effort is dramatically increased.

6. Consider making slices compilable. Nothing in the definition of a slice requires that the set of

statements is compilable, but if we make this choice, it means that a set of compiler directive and declarative statements is a subset of every slice. As a

{22, 23, 24, 29, 30}, contains the statements

enxample, the slice S_5, which is S(locks, 23) =

```
22 READ(locks);
23 WHILE locks <> -1 DO
24 BEGIN
29 READ(locks);
30 END; {WHILE locks}
```

If we add statements 1-14 and 53, we have the compilable slice shown here:

```
1 program lock_stock_and_barrel
2 const
3 lock_price = 45.0;
4 stock_price = 30.0;
5 barrel_price = 25.0;
6 type
7 STRING_30 = string[30]; {Salesman's Name}
8 var
9 locks, stocks, barrels, num_locks, num_stocks,
10 num_barrels, salesman_index, order_index : INTEGER;
11 sales, commission : REAL;
12 salesman : STRING_30; 13
```

```
14 BEGIN {program lock_stock_and_barrel}
22 READ(locks);
23 WHILE locks <> -1 DO
24 BEGIN
29 READ(locks);
30 END; {WHILE locks}

53 END. {program lock_stock_and_barrel}
```

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.

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.

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.

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.

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.

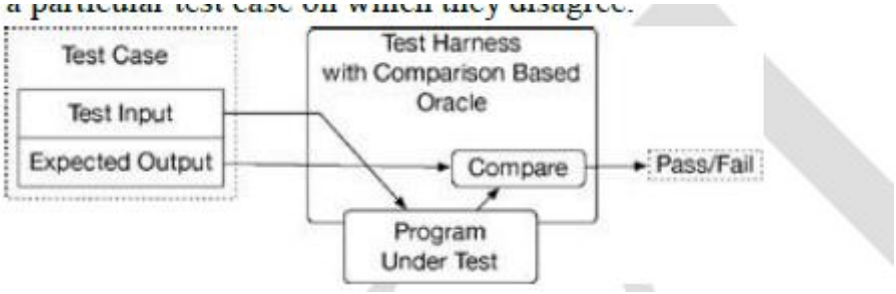

Figure 3.8: A test harness with a comparison-based test oracle processes test cases consisting of (program input, predicted output) pairs.

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.

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.

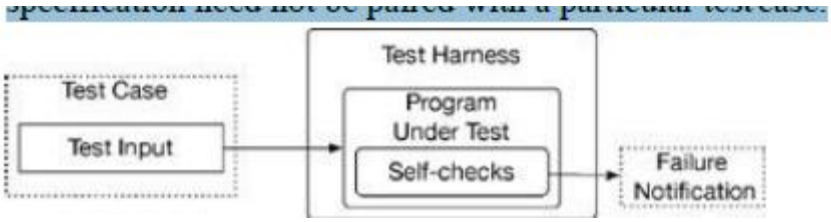

Figure 3.8: When self-checks are embedded in the program, test cases need not include predicted outputs.

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.

$$(|\langle k, v\rangle \in \phi(dict)|)$$
$$o = dict.get(k)$$
$$(|o = v|)$$

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 postcondition 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.

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. Distinguishing 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