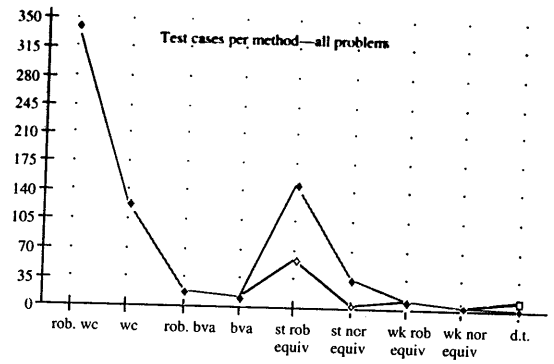

Figure 8.6 Test case trendline for all three problems.

of test cases for each of our examples in terms of the various methods. The domain-based numbers are identical, reflecting both the mechanical nature of the techniques and the formulas that describe the number of test cases generated by each method. The main differences are seen in strong equivalence class testing and decision table testing. Both of these reflect the logical complexity of the problems, so we would expect to see differences here. When we study structural testing (Chapters 9 and 10), we will see that these distinctions have an important implication for testing. The three graphs are superimposed on each other in Figure 8.6.

8.2 Testing Efficiency

If you look closely at these sets of test cases, you can get a feel for the fundamental limitation of functional testing: the twin possibilities of gaps of untested functionality and redundant tests. Consider the NextDate Problem, for example. The decision table (which took three tries to get it right) yields 13 test cases. We have confidence that these test cases are complete (in some sense) because the decision table is complete. On the other hand, worst-case boundary value analysis yielded 125 test cases. When we look closely at these, they are not very satisfactory: what do we expect to learn from cases 1–5? These test cases check the NextDate function for January 1 in five different years. The year has nothing to do with this part of the calendar, so we would expect that one of these would suffice. If we roughly estimate a "times ten" redundancy, we might expect a reduction to 25 test cases, quite compatible with the 22 from the decision table approach. Looking closer, we find a few cases for February, but none happen to hit February 28 or 29, and no interesting connection is made with leap years. Not only do we have a times ten redundancy, we also have some severe gaps around the end of February and leap years.

The strong equivalence class test cases move in the right direction: 36 test cases, of which 11 are impossible. Once again, the impossible test cases result from the presumed independence among the equivalence classes. All but six of

the decision table test cases (cases 2, 7, 12, 15, 17, and 18 of Table 8.6) map to corresponding strong equivalence class test cases (Table 8.5). Half of these deal with the 28th day of non-February months, so they are not very interesting. The remaining three are useful test cases, especially because they test possibilities that are missed by the strong equivalence class test cases. All this supports two conclusions: gaps occur in the functional test cases, and these gaps are reduced by using more sophisticated techniques.

Can we push this a little further by trying to quantify what we mean by testing efficiency? The intuitive notion is that a set of test cases is just right — that is, no gaps and no redundancy. We can develop various ratios of total number of test cases generated by method A to those generated by method B, or even ratios on a test-case basis. This is usually more trouble than it is worth, but sometimes management demands numbers even when they have little real meaning. We will revisit this in Chapter 11, after we complete our study of structural testing. The structural approaches support interesting (and useful) metrics, and these will provide a much better quantification of testing efficiency. Meanwhile, we can help recognize redundancy by annotating test cases with a brief purpose comment. When we see several test cases with the same purpose, we (correctly) sense redundancy. Detecting gaps is harder: If we can only use functional testing, the best we can do is compare the test cases that result from two methods. In general, the more sophisticated methods will help us recognize gaps, but nothing is guaranteed. We could develop excellent strong equivalence classes for a program, and then produce a klutzy decision table.

8.3 Testing Effectiveness

What we would really like to know about a set of test cases is how effective they are, but we need to clarify what "effective" means. The easy choice is to be dogmatic: mandate a method, use it to generate test cases, and then run the test cases. This is absolute, and conformity is measurable; so it can be used as a basis for contractual compliance. We can improve on this by relaxing a dogmatic mandate and require that testers choose "appropriate methods," using the guidelines given at the ends of various chapters here. We can gain another incremental improvement by devising appropriate hybrid methods, as we did with the Commission Problem in Chapter 5.

Structured testing techniques yield a second choice for test effectiveness. In Chapter 9, we will discuss the notion of program execution paths, which provide a good formulation of test effectiveness. We will be able to examine a set of test cases in terms of the execution paths traversed. When a particular path is traversed more than once, we might question the redundancy. Sometimes such redundancy can have a purpose, as we shall see in Chapter 10.

The best interpretation for testing effectiveness is (no great surprise) the most difficult. We would really like to know how effective a set of test cases is for finding faults present in a program. This is problematic for two reasons: first, it presumes we know all the faults in a program. Quite a circularity — if we did, we would take care of them. Because we do not know all the faults in a program, we could never know if the test cases from a given method revealed them. The

second reason is more theoretical: proving that a program is fault-free is equivalent to the famous halting problem of computer science, which is known to be impossible. The best we can do is to work backward from fault types. Given a particular kind of fault, we can choose testing methods (functional and structural) that are likely to reveal faults of that type. If we couple this with knowledge of the most likely kinds of faults, we end up with a pragmatic approach to testing effectiveness. This is improved if we track the kinds (and frequencies) of faults in the software we develop.

8.4 Guidelines

Here is one of my favorite testing stories. An inebriated man was crawling around on the sidewalk beneath a street light. When a policeman asked him what he was doing, he replied that he was looking for his car keys. "Did you lose them here?" the policeman asked. "No, I lost them in the parking lot, but the light is better here."

This little story contains an important message for testers: testing for faults that are not likely to be present is pointless. It is far more effective to have a good idea of the kinds of faults that are most likely (or most damaging) and then to select testing methods that are likely to reveal these faults.

Many times, we do not even have a feeling for the kinds of faults that may be prevalent. What then? The best we can do is use known attributes of the program to select methods that deal with the attributes — sort of a "punishment fits the crime" view. The attributes that are most helpful in choosing functional testing methods are:

Whether the variables represent physical or logical quantities
Whether dependencies exist among the variables
Whether single or multiple faults are assumed
Whether exception handling is prominent

Here is the beginning of an "expert system" on functional testing technique selection:

1. If the variables refer to physical quantities, domain testing and equivalence class testing are indicated.
2. If the variables are independent, domain testing and equivalence class testing are indicated.
3. If the variables are dependent, decision table testing is indicated.
4. If the single-fault assumption is warranted, boundary value analysis and robustness testing are indicated.
5. If the multiple-fault assumption is warranted, worst-case testing, robust worst-case testing, and decision table testing are indicated.
6. If the program contains significant exception handling, robustness testing and decision table testing are indicated.
7. If the variables refer to logical quantities, equivalence class testing and decision table testing are indicated.

Table 8.1 Appropriate Choices for Functional Testing

c1	Variables (P, physical; L, logical)	P	P	P	P	P	L	L	L	L	L
c2	Independent variables?	Y	Y	Y	Y	N	Y	Y	Y	Y	N
c3	Single-fault assumption?	Y	Y	N	N	–	Y	Y	N	N	–
c4	Exception handling?	Y	N	Y	N	–	Y	N	Y	N	–
a1	Boundary value analysis		x								
a2	Robustness testing	x									
a3	Worst-case testing				x						
a4	Robust worst case			x							
a5	Traditional equivalent class	x		x			x		x		
a6	Weak equivalent class	x	x				x	x			
a7	Strong equivalent class			x	x	x			x	x	x
a8	Decision table					x					x

Combinations of these may occur, therefore, the guidelines are summarized as a decision table in Table 8.1.

8.5 Case Study

Here is an example that lets us compare functional testing methods and apply the guidelines. A hypothetical insurance premium program computes the semi-annual car insurance premium based on two parameters: the policyholder's age and driving record:

Premium = BaseRate*ageMultiplier – safeDrivingReduction

The ageMultiplier is a function of the policyholder's age, and the safe driving reduction is given when the current points (assigned by traffic courts for moving violations) on the policyholder's driver's license are below an age-related cutoff. Policies are written for drivers in the age range of 16 to 100. Once a policyholder has 12 points, the driver's license is suspended (thus, no insurance is needed). The BaseRate changes from time to time; for this example, it is $500 for a semiannual premium.

Age Range	*Age Multiplier*	*Points Cutoff*	*Safe Driving Reduction*
16 ≤ age < 25	2.8	1	50
25 ≤ age < 35	1.8	3	50
35 ≤ age < 45	1.0	5	100
45 ≤ age < 60	0.8	7	150
60 ≤ age < 100	1.5	5	200

Worst-case boundary value testing, based on the input variables, age, and points, yields the following extreme values. The corresponding 25 test cases are shown in Figure 8.7.

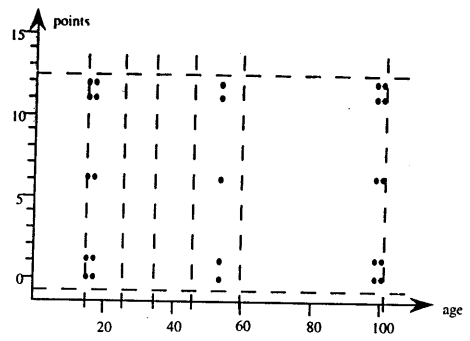

Figure 8.7 Worst-case boundary value test cases for the insurance premium program.

Variable	*Min*	*Min+*	*Nom.*	*Max-*	*Max*
Age	16	17	54	99	100
Points	0	1	6	11	12

I do not think anyone would be content with these test cases. There is too much of the problem statement missing. The various age cutoffs are not tested, nor are the point cutoffs. We could refine this by taking a closer look at the age ranges and the point ranges.

A1 = {age: $16 \leq age < 25$}
A2 = {age: $25 \leq age < 35$}
A3 = {age: $35 \leq age < 45$}
A4 = {age: $45 \leq age < 60$}
A5 = {age: $60 \leq age < 100$}
P1 = {points = 0, 1}
P2 = {points = 2, 3}
P3 = {points = 4, 5}
P4 = {points = 6, 7}
P5 = {points = 8, 9, 10, 11, 12}

Because these ranges meet at "endpoints," we would have the worst-case test values shown in Table 8.2. Notice that the discrete values of the point variable do not lend themselves to the min+ and max– convention in some cases.

If we drew the grid, we would get something like the one in Figure 8.8. Each vertical set (in which the age variable is held constant) has 13 points, and we have such a column for each value of the age variable (there are 21 of these); so there would be 273 worst-case boundary value test cases. We are clearly at a point of severe redundancy; time to move on to equivalence class testing.

The age sets A1–A5, and the points sets P1–P5 are natural choices for equivalence classes. The corresponding equivalence class test cases are shown in Figure 8.9; the open circles correspond to strong normal test cases, and the black circles are the weak normal test cases.

Table 8.2 Detailed Worst-Case Values

Variable	*Min*	*Min+*	*Nom.*	*Max-*	*Max*
Age	16	17	20	24	
Age	25	21	30	34	
Age	35	26	40	44	
Age	45	46	53	59	
Age	60	61	75	99	100
Points	0	n/a	n/a	n/a	1
Points	2	n/a	n/a	n/a	3
Points	4	n/a	n/a	n/a	5
Points	6	n/a	n/a	n/a	7
Points	8	9	10	11	12

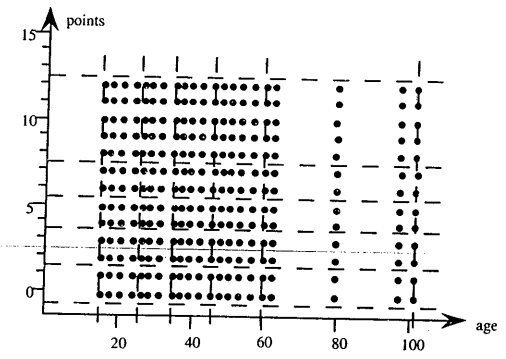

Figure 8.8 Detailed worst-case boundary value test cases for the insurance premium program.

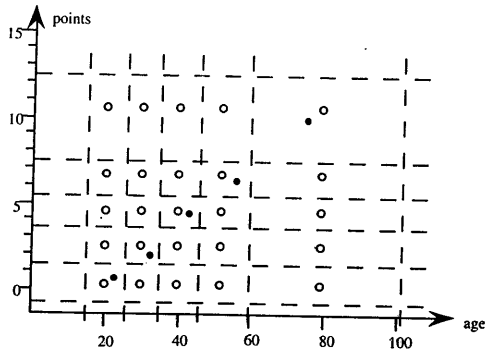

Figure 8.9 Weak and strong equivalence class test cases for the insurance premium program.

Table 8.3 Decision Table Test Cases for the Insurance Premium Program

Age is	*16–25*	*16–25*	*25–35*	*25–35*	*35–45*	*35–45*	*45–60*	*45–60*	*60–100*	*60–100*
Points	0	1–12	0–2	3–12	0–4	5–12	0–6	7–12	0–4	5–12
Age multiplier	2.8	2.8	1.8	1.8	1.8	1.8	0.8	0.8	1.5	1.5
Safe driving reduction	50	—	50	—	100	—	150	—	200	—

Equivalence class testing clearly reduces the redundancy problem, but there still seems to be some excess. Why test all the point classes P2–P5 for A1? Once the point threshhold is exceeded, the safe driving reduction is lost. We can address these dependencies with the extended entry decision table in Table 8.3.

The decision table test cases are shown in Figure 8.10.

Take a look at Figures 8.8 and 8.10; one is overkill and the other is inadequate. We need to find some happy compromise, and this is where the story about the drunkard looking for keys comes in. What are the error-prone aspects of the

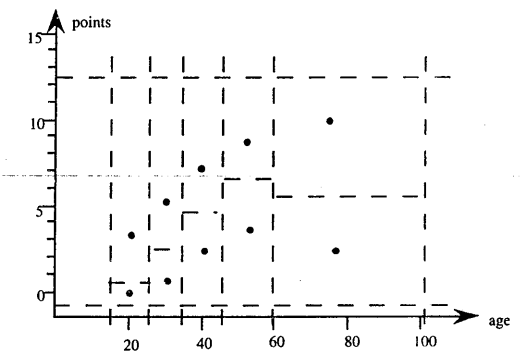

Figure 8.10 Decision table test cases for the insurance premium program.

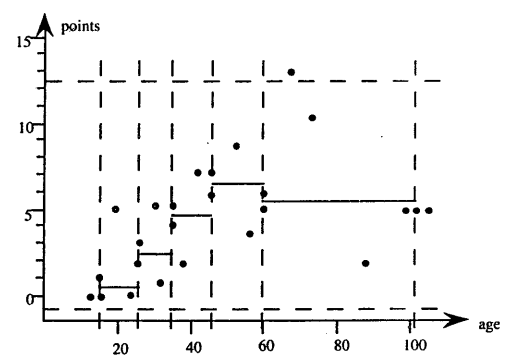

Figure 8.11 Final (hybrid) test cases for the insurance premium program.

insurance premium program? The endpoints of the age ranges appear to be a good place to start, and this puts us back in boundary value mode. Also, we have not considered ages under 16 and over 100, which suggests some element of robust boundary value thinking. Finally, we should probably check the values at which the safe driving reduction is lost, and maybe values of points over 12, when all insurance is lost. (Notice that the responses to these were not in the problem statement, but our testing analysis provokes us to think about them.) Maybe this should be called hybrid functional testing: it uses the advantages of all three forms in a blend that is determined by the nature of the application (shades of special value testing). Hybrid appears appropriate, because such selection is usually done to improve the stock.

STRUCTURAL TESTING

III

Chapter 9

Path Testing

The distinguishing characteristic of structural testing methods is that they are all based on the source code of the program 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 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; we repeat the improved definition from Chapter 4 here.

Definition

Given a program written in an imperative programming language, its program graph is a directed graph in which nodes are statement fragments, and edges represent flow of control. (A complete statement is a "default" statement fragment.)

If i and j are nodes in the program graph, an edge exists 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 is illustrated here with the 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 keep a fragment as a separate node, other times it seems better to include this with another portion of a statement. 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, 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.

```
1. Program triangle2 'Structured programming version of simpler specification
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
       'Step 1: Get Input
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)
       'Step 2: Is A Triangle?
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10.     Then IsATriangle = True
11.     Else IsATriangle = False
12 EndIf
       'Step 3: Determine Triangle Type
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("Not a Triangle")
22. EndIf
23. End triangle2
```

A program graph of this program is given in Figure 9.1. Examine it closely to find graphs of the structured programming constructs we discussed in Chapter 4.

Nodes 4 through 8 are a sequence, nodes 8 through 12 are an if-then-else construct, and nodes 15 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. Figure 9.2 is a graph of a simple (but unstructured!) program; it is typical of the kind of example used to show the impossibility of completely testing even simple programs (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.

9.1 DD-Paths

The best-known form of structural testing is based on a construct known as a decision-to-decision path (DD-Path) (Miller, 1977). The name refers to a sequence

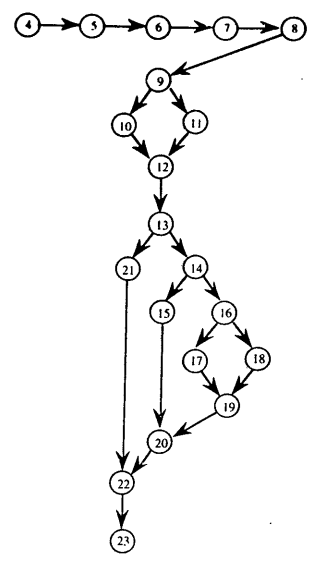

Figure 9.1 Program graph of the triangle program.

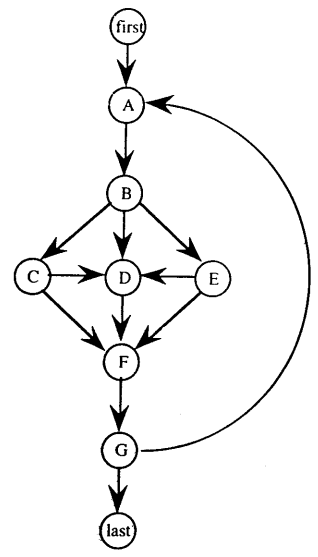

Figure 9.2 Trillions of paths.

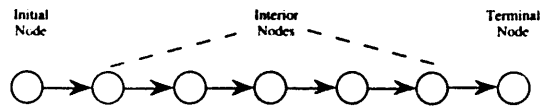

Figure 9.3 A chain of nodes in a directed graph.

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 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.

We will define DD-Paths in terms of paths of nodes in a directed graph. We might call these paths 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. 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 9.3. The length (number of edges) of the chain in Figure 9.3 is 6. We can have a degenerate case of a chain that is of length 0 — that is, a chain consisting of exactly one node and no edges.

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 "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.

This is a complex definition, so we will apply it to the program graph in Figure 9.1. Node 4 is a Case 1 DD-Path, we will call it "first"; similarly, node 22 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

Table 9.1 Types of DD-Paths in Figure 9.1

Program Graph Nodes	*DD-Path Name*	*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

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, and 18 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 Table 1, where the DD-Path names correspond to node names in the DD-Path graph in Figure 9.4.

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

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.

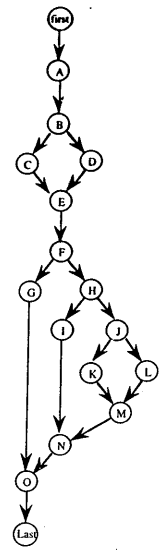

Figure 9.4 DD-Path graph for the triangle program.

In practice, it is reasonable to make DD-Path graphs for programs up to about 100 source lines. Beyond that, most testers look for a tool.

9.2 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.

Several widely accepted test coverage metrics are used; most of those in Table 9.2 are due to the early work of E.F. Miller (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. 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.

These coverage metrics form a lattice (see Chapter 10) 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. E. F. Miller observes that when DD-Path

Table 9.2 Structural Test Coverage Metrics

Metric	*Description of Coverage*
C_0	Every statement
C_1	Every DD-Path (predicate outcome)
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

coverage is attained by a set of test cases, roughly 85% of all faults are revealed (Miller, 1991).

9.2.1 Metric-Based Testing

The test coverage metrics in Table 9.2 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 9.2. 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 (which can be entire statements) to be nodes.

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 Problem (see Figure 9.1), 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 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 only 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 topic of Chapter 10 — 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, as 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. Instead of 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 is not swept under the DD-Path coverage rug.

Loop Coverage

The condensation graphs we studied in Chapter 4 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. To start, an amusing taxonomy of loops occurs (Beizer, 1983): concatenated, nested, and horrible, shown in Figure 9.5.

Concatenated loops are simply a sequence of disjoint loops, while nested loops are such that one is contained inside another. Horrible loops cannot occur when the structured programming precepts are followed. 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 Beizer's horrible loop. (Other sources define this as a knot — how appropriate.) The simple view of loop testing is that every loop involves a decision, and we need to test both outcomes of the decision: one is to traverse the loop, and the other is to exit (or not enter) the loop. This is carefully proved in Huang (1979). We can also take a modified boundary value approach, where the loop index is given its minimum, nominal, and maximum values (see Chapter 5). 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

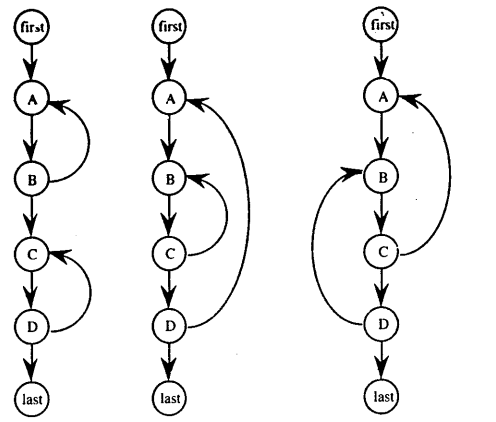

Figure 9.5 Concatenated, nested, and knotted loops.

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 working 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. If loops are knotted, it will be necessary to carefully analyze them in terms of the dataflow methods discussed in Chapter 10. As a preview, consider the infinite loop that could occur if one loop tampers with the value of the other loop's index.

9.2.2 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.

9.3 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 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.

9.3.1 McCabe's Basis Path Method

Figure 9.6 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.) Figure 9.7 shows the result of doing this; it also contains edge labels that are used in the discussion that follows.

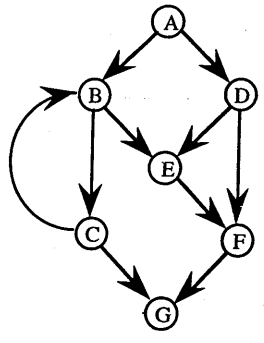

Figure 9.6 McCabe's control graph.

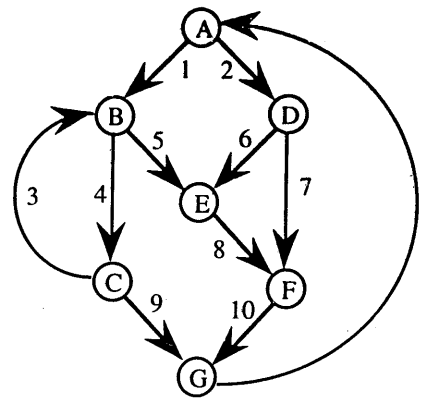

Figure 9.7 McCabe's derived strongly connected graph.

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 9.6) to a strongly connected, directed graph obtained by adding one edge from the sink to the source node (as in Figure 9.7). Adding an edge clearly affects value computed by the formula, but it should not affect the number of circuits. Here is a way to resolve the apparent inconsistency. The number of linearly independent paths from the source node to the sink node in Figure 9.6 is:

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

and the number of linearly independent circuits in the graph in Figure 9.7 is

$$V(G) = e - n + p$$
$$= 11 - 7 + 1 = 5,$$

The cyclomatic complexity of the strongly connected graph in Figure 9.7 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 9.3 Path/Edge Traversal

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

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 9.3. 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. 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 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 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; 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 9.3: this is not problematic, because a unique basis is not required.

Table 9.4 Basis Paths in Figure 9.4

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

9.3.2 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? Don't 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 9.4. 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 9.4). 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 5 basis paths; they are shown in Table 9.4.

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 respectively to equilateral and isosceles triangles. 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 functional 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; 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 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 8 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.

9.3.3 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; so 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 9.8.

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 Figures 9.9 and 9.10, which starts with the DD-Path graph of the pseudocode triangle program. The if-then-else construct involving nodes A, B, C, and D 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

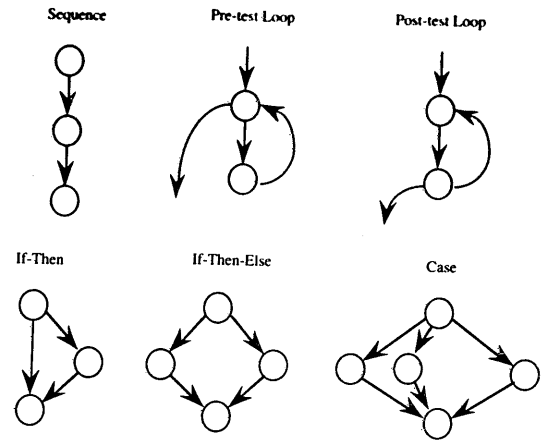

Figure 9.8 Structured programming constructs.

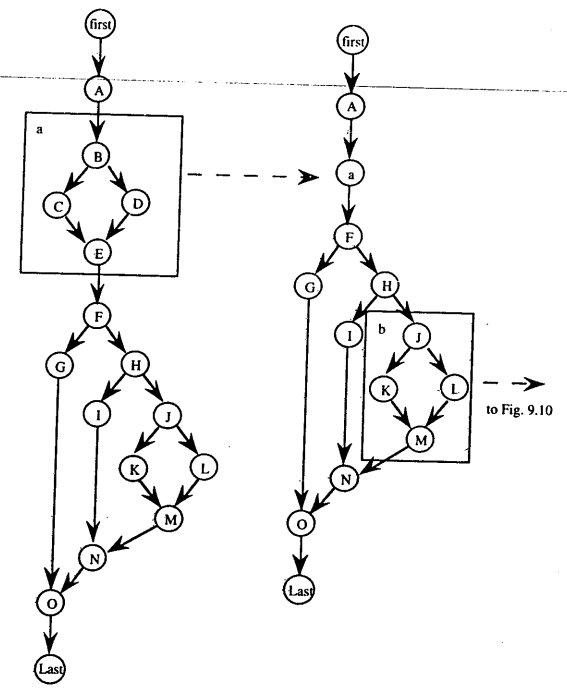

Figure 9.9 Condensing with respect to the structured programming constructs.

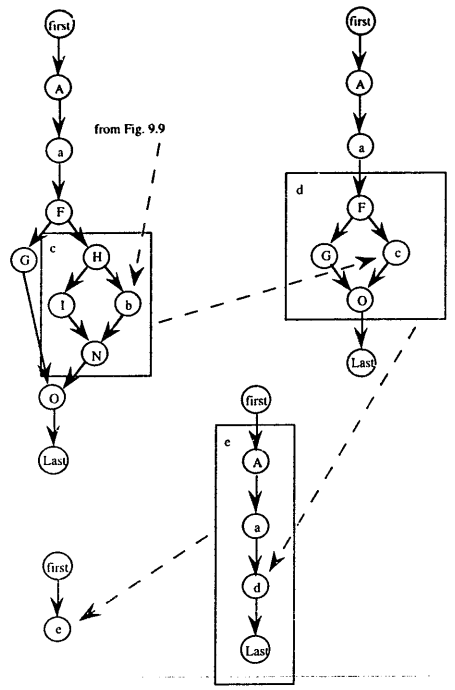

Figure 9.10 Condensing with respect to the structured programming constructs (continued).

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 9.6 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 went on to find elemental "unstructures" that violate the precepts of structured programming (McCabe, 1976). These are shown in Figure 9.11.

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 *pièce de resistance* of McCabe's analysis is that these unstructures 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 unstructures have interesting implications for dataflow testing.

The 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

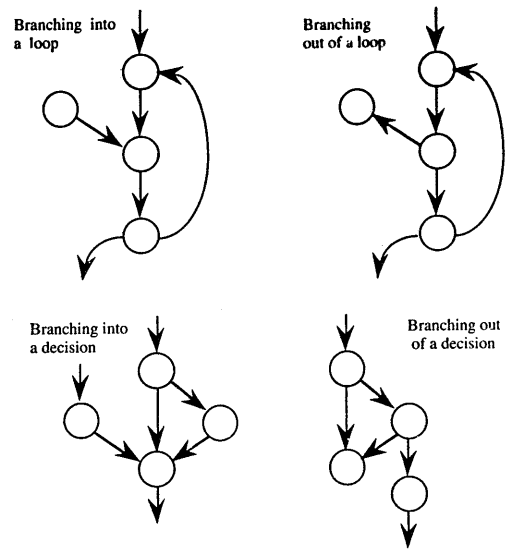

Figure 9.11 Violations of structured programming.

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.

9.4 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 the 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, 1982). He was referring to the DD-Path coverage metric (which is equivalent to the predicate outcome metric) and the cyclomatic complexity metric (which requires at least the cyclomatic number of distinct program paths must be traversed). 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 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

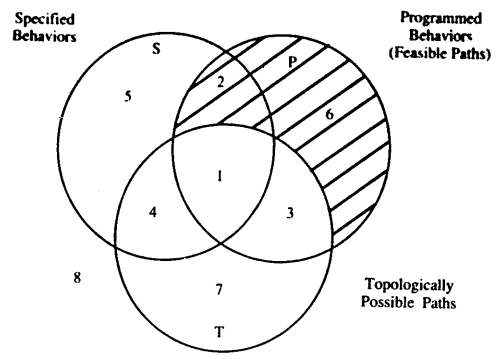

Figure 9.12 Feasible and topologically possible paths.

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. The coverage metrics in Table 9.2 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 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.

This is a good place to revisit the Venn diagram view of testing that we used in Chapter 1. Figure 9.12 shows the relationship among specified behaviors (set S), programmed behaviors (set P), and topologically feasible paths in a program (set T). As usual, region 1 is the most desirable — it contains specified behaviors that are implemented by feasible paths. By definition, every feasible path is topologically possible; so the shaded portion (regions 2 and 6) of the set P must be empty. Region 3 contains feasible paths that correspond to unspecified behaviors. Such extra functionality needs to be examined: if useful, the specification should be changed; otherwise, these feasible paths should be removed. Regions 4 and 7 contain the infeasible paths; of these, region 4 is problematic. Region 4 refers to specified behaviors that have almost been implemented — topologically possible yet infeasible program paths. This region very likely corresponds to coding errors, where changes are needed to make the paths feasible. Region 5 still corresponds to specified behaviors that have not been implemented. Path-based testing will never recognize this region. Finally, region 7 is a curiosity: unspecified, infeasible, yet topologically possible paths. Strictly speaking, no problem occurs here because infeasible paths cannot execute. If the corresponding code is incorrectly changed by a maintenance action (maybe by a programmer who does not fully understand the code), these could become feasible paths, as in region 3.

References

Huang, J.C., Detection of dataflow anomaly through program instrumentation, *IEEE Transactions on Software Engineering,* SE-5; pp. 226–236, 1979.

Schach, Stephen R., *Software Engineering,* 2nd ed., Richard D. Irwin, Inc., and Aksen Associates, Inc., 1993.

Miller, E.F., *Tutorial: Program Testing Techniques,* COMPSAC '77 IEEE Computer Society, 1977.

Miller, Edward F. Jr., Automated software testing: a technical perspective, *Amer. Programmer,* Vol. 4, No. 4, April 1991, pp. 38–43.

McCabe, Thomas J., A complexity metric, *IEEE Transactions on Software Engineering,* SE-2, 4, December 1976, pp. 308–320.

McCabe, Thomas J., Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, National Bureau of Standards (Now NIST), Special Publication 500-99, Washington, D.C., 1982.

McCabe, Thomas J., *Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric,* McCabe and Associates, Baltimore, 1987.

Perry, William E., *A Structured Approach to Systems Testing,* QED Information Systems, Inc., Wellesley, MA, 1987.

Exercises

1. Find the cyclomatic complexity of the graph in Figure 9.2.
2. Identify a set of basis paths for the graph in Figure 9.2.
3. Discuss McCabe's concept of "flipping" for nodes with outdegree ≥ 3.
4. Suppose we take Figure 9.2 as the DD-Path graph of some program. Develop sets of paths (which would be test cases) for the C_0, C_1, and C_2 metrics.
5. Develop multiple-condition coverage test cases for the pseudocode triangle program. (Pay attention to the dependency between statement fragments 14 and 16 with the expression (a = b) AND (b = c).)
6. Rewrite the program segment 14–20 such that the compound conditions are replaced by nested if-then-else statements. Compare the cyclomatic complexity of your program with that of the existing version.

```
14. 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
```

7. Look carefully at the original statement fragments 14–20. What happens with a test case (e.g., a = 3, b = 4, c = 3) in which a = c? The condition in Line 14 uses the transitivity of equality to eliminate the a = c condition. Is this a problem?
8. The whiteBox.exe program at the CRC Web site (www.crcpress.com) contains Visual Basic implementations of the triangle, NextDate, and commission problems. The output shows the DD-Path coverage of (sets of) test cases. Use it to experiment with various sets of test cases to determine DD-Path coverage of various functional techniques.
9. (For mathematicians only.) For a set V to be a vector space, two operations (addition and scalar multiplication) must be defined for elements in the set. In addition, the following criteria must hold for all vectors x, y, and z ∈ V, and for all scalars k, l, 0, and 1:
   a. if x, y ∈ V, the vector x + y ∈ V.
   b. x + y = y + x.
   c. (x + y) + z = x + (y + z).
   d. there is a vector 0 ∈ V such that x + 0 = x.
   e. for any x ∈ V, there is a vector -x ∈ V such that x + (-x) = 0.
   f. for any x ∈ V, the vector kx ∈ V.
   g. k(x + y) = kx + ky.
   h. (k + l)x = kx + lx.
   i. k(lx) = (kl)x.
   j. 1x = x.

   How many of these 10 criteria hold for the "vector space" of paths in a program?

Chapter 10

Data Flow Testing

Data flow testing is an unfortunate term because it suggests some connection with data flow diagrams; no connection exists. 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 other 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

Each of these anomalies can be recognized from the concordance of a program. Because the concordance information is compiler generated, these anomalies can be discovered by what is known as static analysis: finding faults in source code without executing it.

10.1 Define/Use Testing

Much of the formalization of define/use testing was done in the early 1980s (Rapps, 1985); the definitions in this section are compatible with those in Clarke (1989), which summarizes most 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. We also disallow edges from a node to itself. Paths, subpaths, and cycles are as they were in Chapter 4. The set of all paths in P is PATHS(P).

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 variable v 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 path with respect to a variable v (denoted du-path) is a 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 path.

Definition

A definition-clear path with respect to a variable v (denoted dc-path) is a definition-use path in PATHS(P) with initial and final nodes DEF (v, m) and USE (v, n) such that no other node in the 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.

10.1.1 Example

We will use the commission problem and its program graph to illustrate these definitions. The numbered pseudocode 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 the total numbers of locks, stocks, and barrels sold. The while-loop is a classical sentinel controlled loop in which a value of –1 for locks signifies the end of the sales data. The totals are accumulated as the data values 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 Commission (INPUT,OUTPUT)

2       Dim locks, stocks, barrels As Integer
3       Dim lockPrice, stockPrice, barrelPrice As Real
4       Dim totalLocks, totalStocks, totalBarrels As Integer
5       Dim lockSales, stockSales, barrelSales As Real
6       Dim sales, commission As Real

7       lockPrice = 45.0
8       stockPrice = 30.0
9       barrelPrice = 25.0
10      totalLocks = 0
11      totalStocks = 0
12      totalBarrels = 0

13      Input(locks)
14      While NOT(locks = -1)     'loop condition uses -1 to indicate end of data
15          Input(stocks, barrels)
16          totalLocks = totalLocks + locks
17          totalStocks = totalStocks + stocks
18          totalBarrels = totalBarrels + barrels
19          Input(locks)
20      EndWhile

21      Output("Locks sold: ", totalLocks)
22      Output("Stocks sold: ", totalStocks)
23      Output("Barrels sold: ", totalBarrels)
```

```
24    lockSales = lockPrice*totalLocks
25    stockSales = stockPrice*totalStocks
26    barrelSales = barrelPrice * totalBarrels
27    sales = lockSales + stockSales + barrelSales
28    Output("Total sales: ", sales)

29    If (sales > 1800.0)
30        Then
31            commission = 0.10 * 1000.0
32            commission = commission + 0.15 * 800.0
33        commission = commission + 0.20*(sales-1800.0)
34    Else If (sales > 1000.0)
35        Then
36                commission = 0.10 * 1000.0
37                commission = commission + 0.15*(sales-1000.0)
38            Else commission = 0.10 * sales
39            EndIf
40    EndIf

41    Output("Commission is $", commission)
42    End Commission
```

Figure 10.2 shows the decision-to-decision path (DD-Path) graph of the program graph in Figure 10.1. More compression exists in this DD-Path graph because of the increased computation in the commission problem. Table 10.1 details the statement fragments associated with DD-Paths.

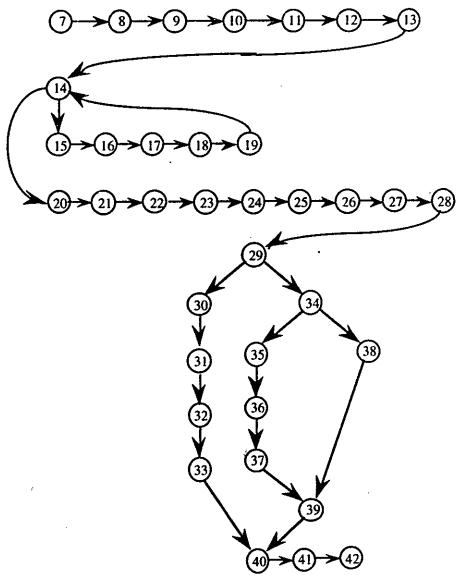

Figure 10.1 Program graph of the commission program.

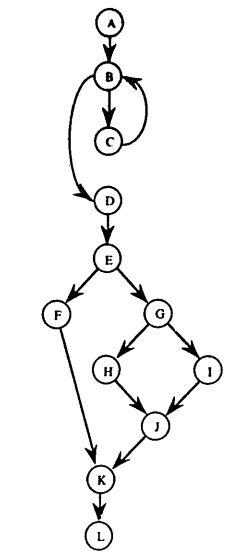

Figure 10.2 DD-Path graph of the commission program.

Table 10.2 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 10.1 to identify various definition-use and definition-clear paths. It is a judgment call whether nonexecutable 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

Table 10.1 DD-Paths in Figure 10.1

DD-Path	*Nodes*
A	7, 8, 9, 10, 11, 12, 13
B	14
C	15, 16, 17, 18, 19
D	20, 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 10.2 Define/Use Nodes for Variables in the Commission Problem

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

is wrong, it can be helpful to include them. Take your pick. We will refer to the various paths as sequences of node numbers.

Table 10.3 presents some of the du-paths in the commission problem; they are named by their beginning and ending nodes (from Figure 10.1). The third column in Table 10.3 indicates whether the du-paths are definition clear. Some

Table 10.3 Selected Define/Use Paths

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	17, 25	no
totalStocks	17, 17	yes
totalStocks	17, 22	no
totalStocks	17, 25	no
locks	13, 14	yes
locks	19, 14	yes
locks	13, 16	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 10.4 Define/Use Paths for Commission

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

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 accumulates values for totalLocks, totalStocks, and totalBarrels. Table 10.3 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 clear 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.)

10.1.2 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.

10.1.3 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:

p1 = <13, 14>
p2 = <13, 14, 15, 16>
p3 = <19, 20, 14>
p4 = <19, 20, 14, 15, 16>

Du-paths p1 and p2 refer to the priming value of locks, which is read at node 13: locks 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. If we "extended" paths p1 and p3 to include node 21,

p1' = <13, 14, 21>
p3' = <19, 20, 14, 21>

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 these du-paths are definition clear.

10.1.4 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, 21>

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, 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 11 (see path p5). The remaining two du-paths are both subpaths of p7:

p8 = <16, 17, 18, 19, 20, 21>
p9 = <16, 17, 18, 19, 20, 21, 22, 23, 24>

Both are definition clear, and both have the loop iteration problem we discussed before.

10.1.5 du-Paths for Sales

Only one defining node is used 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: the static choice is the path <27, 28, 29, 30, 31, 32, 33, 34>, and the dynamic choice is the path <27. 28, 29, 34>. Here, we will use the dynamic view, so the remaining du-paths for sales are:

p13 = <27, 28, 29, 34>
p14 = <27, 28, 29, 34, 35, 36, 37>
p15 = <27, 28. 29, 38>

Note that the dynamic view is very compatible with the kind of thinking we used for DD-Paths in Chapter 9.

10.1.6 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, 38). Only one usage node is used: USE(commission, 41).

10.1.7 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, 1985). 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 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, as discussed previously. 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 iff 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 iff 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 iff 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 iff 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.

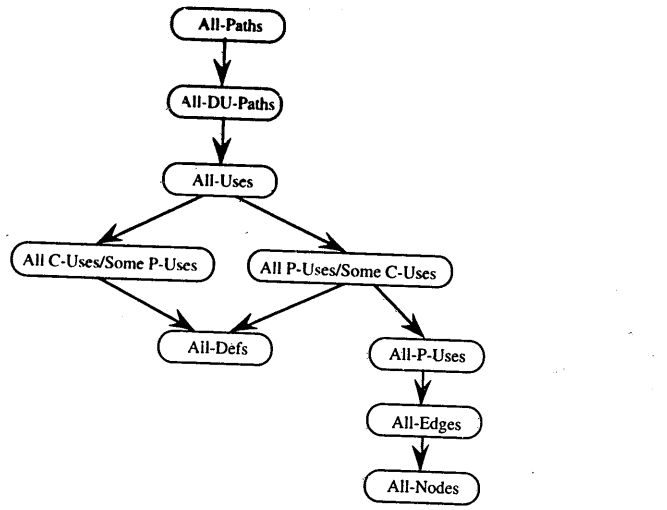

Figure 10.3 Rapps/Weyuker hierarchy of data flow coverage metrics.

Definition

The set T satisfies the All-DU-paths criterion for the program P iff 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 (1985). These relationships are shown in Figure 10.3.

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.

10.2 Slice-Based Testing

Program slices have surfaced and submerged in software engineering literature since the early 1980s. They were originally proposed in Weiser (1985), used as an approach to software maintenance in Gallagher (1991), and most recently used to quantify functional cohesion in Bieman (1994). 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 — U.S. 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 (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 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. It is much simpler to list fragment numbers in P(G), so 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 (functional) 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, data declaration statements have an effect on the value of a variable. For now, we simply exclude all nonexecutable statements. The notion of contribution is partially clarified by the predicate (P-use) and computation (C-use) usage distinction of Rapps (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)

While we are 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.

10.2.1 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 (or in NextDate). Follow these examples while looking at the source code for the commission problem that we used to analyze in terms of define-use paths.

Slices on the locks variable show why it is potentially fault-prone. It has a P-use at node 14 and a C-use at node 16 and has two definitions, the I-defs at nodes 13 and 19.

S_1: S(locks, 13) = {13}
S_2: S(locks, 14) = {13, 14, 19, 20}
S_3: S(locks, 16) = {13, 14, 16, 19, 20}
S_4: S(locks, 19) = {19}

The slices for stocks and barrels are boring. Both are short, definition clear paths contained entirely within a loop, so they are not affected by iterations of the loop. (Think of the loop body as a DD-Path.)

S_5: S(stocks, 15) = {13, 14, 15, 19, 20}
S_6: S(stocks, 17) = {13, 14, 15, 17, 19, 20}
S_7: S(barrels, 15) = {13, 14, 15, 19, 20}
S_8: S(barrels, 18) = {13, 14, 15, 18, 19, 20}

The next four slices illustrate how repetition appears in slices. Node 10 is an A-def for totalLocks, and node 16 contains both an A-def and a C-use. The remaining nodes in S_{10} (13, 14, 24, 19, and 20) pertain to the while-loop controlled by locks. Slices S_{10}, S_{11}, and S_{12} are equal because nodes 21 and 27 are, respectively, an O-use and a C-use of totalLocks.

S_9: S(totalLocks, 10) = {10}
S_{10}: S(totalLocks, 16) = {10, 13, 14, 16, 19, 20}
S_{11}: S(totalLocks, 21) = {10, 13, 14, 16, 19, 20}

The slices on totalStocks and totalBarrels are quite similar. They are initialized by A-defs at nodes 11 and 12 and then are redefined by A-defs at nodes 17 and 18. Again, the remaining nodes (13, 14, 19, and 20) pertain to the while-loop controlled by locks.

S_{12}: S(totalStocks, 11) = {11}
S_{13}: S(totalStocks, 17) = {11, 13, 14, 15, 17, 19, 20}
S_{14}: S(totalStocks, 22) = {11, 13, 14. 15, 17, 19, 20}
S_{15}: S(totalBarrels, 12) = {12}
S_{16}: S(totalBarrels, 18) = {12, 13, 14, 15, 18, 19, 20}
S_{17}: S(totalBarrels, 23) = {12, 13, 14, 15, 18, 19, 20}

The next six slices demonstrate our convention regarding values defined by assignment statements (A-defs).

S_{18}: S(lockPrice, 24) = {7}
S_{19}: S(stockPrice, 25) = {8}
S_{20}: S(barrelPrice, 26) = {9}
S_{21}: S(lockSales, 24) = {7, 10, 13, 14, 16, 19, 20, 24}
S_{22}: S(stockSales, 25) = {8, 11, 13, 14, 15, 17, 19, 20, 25}
S_{23}: S(barrelSales, 26) = {9, 12, 13, 14, 15, 18, 19, 20, 26}

The slices on sales and commission are the interesting ones. Only one defining node exists for sales, the A-def at node 27. The remaining slices on sales show the P-uses, C-uses, and the O-use in definition clear paths.

S_{24}: S(sales, 27) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S_{25}: S(sales, 28) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S_{26}: S(sales, 29) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S_{27}: S(sales, 33) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S_{28}: S(sales, 34) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S_{29}: S(sales, 37) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S_{30}: S(sales, 38) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

Think about slice S_{24} in terms of its "components," which are the slices on the C-use variables. We can write $S_{24} = S_{10} \cup S_{13} \cup S_{16} \cup S_{21} \cup S_{22} \cup S_{23}$. Notice how the formalism corresponds to our intuition: if the value of sales is wrong, we first look at how it is computed; if this is OK, we check how the components are computed.

Everything comes together (literally) with the slices on commission. Six A-def nodes are used for commission (corresponding to the six du-paths we identified earlier). Three computations of commission are controlled by P-uses of sales in the IF, ELSE IF logic. This yields three "paths" of slices that compute commission.

S_{31}: S(commission, 31) = {31}
S_{32}: S(commission, 32) = {31, 32}

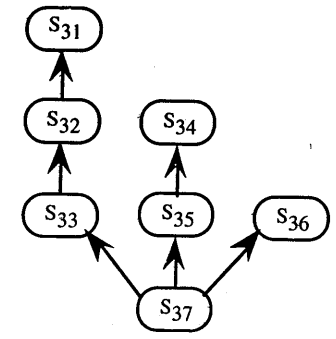

Figure 10.4 Lattice of slices on commission.

S_{33}: 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_{34}: S(commission, 36) = {36}
S_{35}: S(commission, 37) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 36, 37}
S_{36}: S(commission, 38) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 38}

Whichever computation is taken, all come together in the last slice.

S_{37}: 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, 38}

The slice information improves our insight. Look at the lattice in Figure 10.4; it is a directed acyclic graph in which slices are nodes, and an edge represents the proper subset relationship.

This lattice is drawn so that the position of the slice nodes roughly corresponds with their position in the source code. The definition clear paths <43, 51>, <48, 51>, and <50,51> correspond to the edges that show slices S_{36}, S_{38}, and S_{39} are subsets of slice S_{40}. Figure 10.5 shows a lattice of slices for the entire program. Some slices (those that are identical to others) have been deleted for clarity.

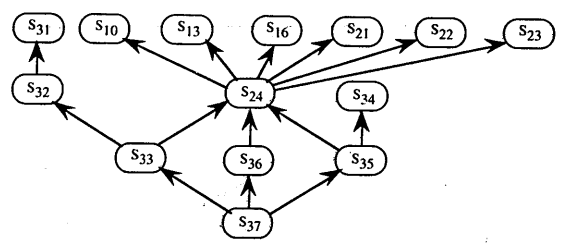

Figure 10.5 Lattice on sales and commission.

10.2.2 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 become even more cumbersome.

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, 36) where

   V = {num_locks, num_stocks, num_barrels}

   contains all the elements of the slice S({sales}, 36) except statement 36. These two slices are so similar, so 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 ({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 such as the triangle program and NextDate.
5. Slices on non-P-use usage nodes are not 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 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; but each slice is separately compilable (and therefore executable). In the 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. We can then code and test other slices and merge them (sometimes called "slice splicing") into a fairly solid program. Try coding the commission program this way.

10.3 Guidelines and Observations

Data flow testing is clearly indicated for programs that are computationally intensive. As a corollary, in control-intensive programs, if control variables are computed (P-uses), data flow 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. Academic tools can be used to support these definitions, but they have not 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, particularly when there is a difficult module to test.

1. Slices do not map nicely into test cases (because the other, nonrelated code is still in an executable path). On the other hand, they provide a handy way to eliminate interaction among variables. Use the slice composition approach to redevelop difficult sections of code, and test 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 a problem exists 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 okay at line 34, the problem must lie in the relative complement; if not, the problem may be in either portion.
3. A many-to-many relationship exists 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 is 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 totalLocks:

   S_9: S(totalLocks, 10) = {10}
   S_{10}: S(totalLocks, 16) = {10, 13, 14, 16, 19, 20}
   S_{11}: S(totalLocks, 21) = {10, 13, 14, 16, 19, 20}

When slices are equal, the corresponding paths are definition clear.

References

Clarke, Lori A. et al., A formal evaluation of data flow path selection criteria, *IEEE Transactions on Software Engineering,* Vol. SE-15, No. 11, pp. 1318–1332, November 1989.

Rapps, S. and Weyuker, E.J., Selecting software test data using data flow information, *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 4, pp. 367–375, April, 1985.

Exercises

1. Think about the static versus dynamic ambiguity of du-paths in terms of DD-Paths. As a start, what DD-Paths are found in the du-paths p12, p13, and p14 for sales?
2. Try to merge some of the DD-Path-based test coverage metrics into the Rapps/Weyuker hierarchy shown in Figure 10.2.
3. List the du-paths for the commission variable.
4. Express slice S_{37} as the union of other pertinent slices.
5. Find the following program slices:
   a. S(commission, 28)
   b. S(sales, 23)
   c. S(commission, 37), S(commission, 29), S(commission, 28)
   d. S(totalLocks, 22)
   e. S(totalStocks, 22)
   f. S(totalBarrels, 22)
6. Find the definition-clear paths (with respect to sales) from line 27 to 29, 33, 34, 38, 39.
7. Complete the lattice in Figure 10.5.
8. Our discussion of slices in this chapter has actually been about "backward slices" in the sense that we are always concerned with parts of a program that contribute to the value of a variable at a certain point in the program. We could also consider "forward slices" that refer to parts of the program where the variable is used. Compare and contrast forward slices with du-paths.

Chapter 11

Retrospective on Structural Testing

When should testing stop? Here are some possible answers:

1. When you run out of time.
2. When continued testing causes no new failures.
3. When continued testing reveals no new faults.
4. When you cannot think of any new test cases.
5. When you reach a point of diminishing returns.
6. When mandated coverage has been attained.
7. When all faults have been removed.

Unfortunately, the first answer is all too common, and the seventh cannot be guaranteed. This leaves the testing craftsperson somewhere in the middle. Software reliability models provide answers that support the second and third choices; both of these have been used with success in industry. The fourth choice is curious: if you have followed the precepts and guidelines we have been discussing, this is probably a good answer. On the other hand, if the reason is due to a lack of motivation, this choice is as unfortunate as the first. The point of diminishing returns choice has some appeal: It suggests that serious testing has continued, and the discovery of new faults has slowed dramatically. Continued testing becomes very expensive and may reveal no new faults. If the cost (or risk) of remaining faults can be determined, the trade-off is clear. (This is a big IF.) We are left with the coverage answer, and it is a pretty good one. In this chapter, we will see how using structural testing as a cross-check on functional testing yields powerful results. First, we demonstrate (by example) the gaps and redundancies problem of functional testing. Next, we develop some metrics of testing efficiency. Because these metrics are expressed in terms of structural coverage, we have an obvious answer to the gaps and redundancies question. Then the

Table 11.1 Paths in the Triangle Program

Path	*Node Sequence*	*Description*
p1	1–2–3–4–5–6–7–13–16–18–20	Equilateral
p2	1–3–5–6–7–13–16–18–19–15	Isosceles (b = c)
p3	1–3–5–6–7–13–16–18–19–12	Not a Triangle (b = c)
p4	1–3–4–5–7–13–16–17–15	Isosceles (a = c)
p5	1–3–4–5–7–13–16–17–12	Not a Triangle (a = c)
p6	1–2–3–5–7–13–14–15	Isosceles (a = b)
p7	1–2–3–5–7–13–14–12	Not a Triangle (a = b)
p8	1–3–5–7–8–12	Not a Triangle (a + b ≤ c)
p9	1–3–5–7–8–9–12	Not a Triangle (b + c ≤ a)
p10	1–3–5–7–8–9–10–12	Not a Triangle (a + c ≤ b)
p11	1–3–5–7–8–9–10–11	Scalene

question reduces to which coverage metric to use. The answer that is most common in industrial practice is DD-paths.

11.1 Gaps and Redundancies

The gaps and redundancies problem of functional testing is very prominent in the Triangle Problem. We use the (klutzy) traditional implementation here, mostly because it is the most frequently used in testing literature (Brown, 1975; Pressman, 1982). Recall that this implementation has exactly 11 feasible paths. They are given in Table 11.1. The path names will be used later; the node numbers are shown in Figure 11.1.

Now, suppose we use boundary value testing to define test cases. We will do this for both the nominal and worst-case formulations. Table 11.2 shows the test cases generated using the nominal boundary value form of functional testing. The last column shows the path (in Figure 11.1) taken by the test case.

The following paths are covered: p1, p2, p3, p4, p5, p6, p7; and paths p8, p9, p10, p11 are missed. Now, suppose we use a more powerful functional testing technique, worst-case boundary value testing. We saw, in Chapter 5, that this yields 125 test cases; they are summarized here in Table 11.3 so you can see the extent of the redundant path coverage. Taken together, the 125 test cases provide full path coverage, but the redundancy is onerous.

11.2 Metrics for Method Evaluation

Having convinced ourselves that the functional methods are indeed open to the twin problems of gaps and redundancies, we can develop some metrics that relate the effectiveness of a functional technique with the achievement of a structural metric. Functional testing techniques always result in a set of test cases, and the structural metric is always expressed in terms of something countable, such as the number of program paths, the number of decision-to-decision paths (DD-Paths), or the number of slices.

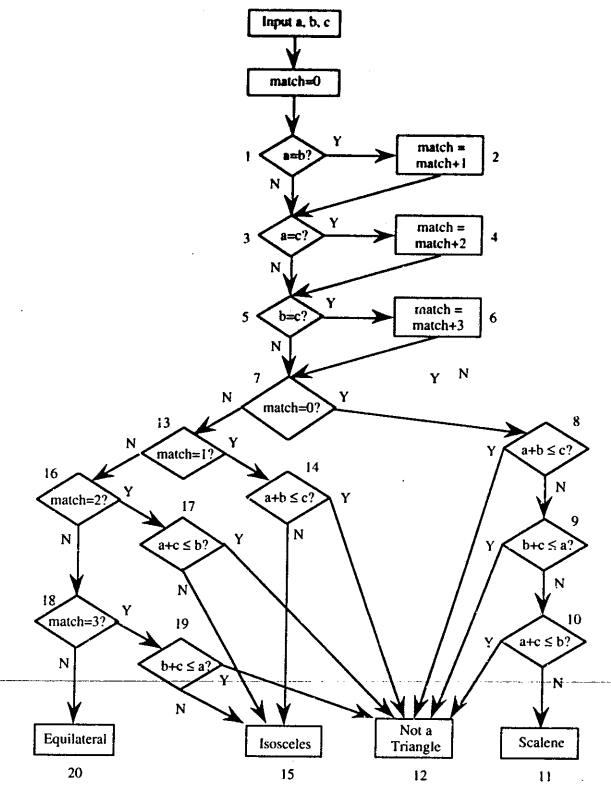

Figure 11.1 Traditional triangle program graph.

Table 11.2 Path Coverage of Nominal Values

Case	*a*	*b*	*c*	*Expected Output*	*Path*
1	100	100	1	Isosceles	p6
2	100	100	2	Isosceles	p6
3	100	100	100	Equilateral	p1
4	100	100	199	Isosceles	p6
5	100	100	200	Not a Triangle	p7
6	100	1	100	isosceles	p4
7	100	2	100	Isosceles	p4
8	100	100	100	Equilateral	p1
9	100	199	100	Isosceles	p4
10	100	200	100	Not a Triangle	p5
11	1	100	100	Isosceles	p2
12	2	100	100	Isosceles	p2
13	100	100	100	Equilateral	p1
14	199	100	100	Isosceles	p2
15	200	100	100	Not a Triangle	p3

Table 11.3 Path Coverage of Worst-Case Values

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11
Nominal	3	3	1	3	1	3	1	0	0	0	0
Worst-case	5	12	6	11	6	12	7	17	18	19	12

In the following definitions, we assume that a functional testing technique M generates m test cases, and that these test cases are tracked with respect to a structural metric S that identifies s elements in the unit under test. When the m test cases are executed, they traverse n of the s structural elements.

Definition

The coverage of a methodology M with respect to a metric S is ratio of n to s. We denote it as C(M,S).

Definition

The redundancy of a methodology M with respect to a metric S is ratio of m to s. We denote it as R(M,S).

Definition

The net redundancy of a methodology M with respect to a metric S is ratio of m to n. We denote it as NR(M,S).

We interpret these metrics as follows: the coverage metric, C(M,S), deals with gaps. When this value is less than 1, there are gaps in the coverage with respect to the metric. Notice that, when C(M,S) = 1, algebra forces R(M,S) = NR(M,S). The redundancy metric is obvious — the bigger it is, the greater the redundancy. Net redundancy is more useful — it refers to things actually traversed, not to the total space of things to be traversed. Taken together, these three metrics give a quantitative way to evaluate the effectiveness of any functional testing method (except special value testing) with respect to a structural metric. This is only half the battle, however. What we really would like is to know how effective test cases are with respect to kinds of faults. Unfortunately, information such as this simply is not available. We can come close by selecting structural metrics with respect to the kinds of faults we anticipate (or maybe faults we most fear). See the guidelines near the ends of Chapters 9 and 10 for specific advice.

In general, the more sophisticated structural metrics result in more elements (the quantity s), hence a given functional methodology will tend to become less effective when evaluated in terms of more rigorous structural metrics. This is intuitively appealing, and it is borne out by our examples. These metrics are devised such that the best possible value is 1. Table 11.4 summarizes the application of these definitions to the data in Table 11.3. The structural metric was program paths, and there were 11 of these. (It makes no sense to consider infeasible paths, because a test case can never traverse an infeasible path.) Table 11.5 contains similar results obtained from the commission problem.

Table 11.4 Metrics for the Triangle Program

Method	m	n	s	C(M, S) = n/s	R(M, S) = m/s	NR(M, S) = m/n
Nominal	15	7	11	0.64	1.36	2.14
Worst-case	125	11	11	1.00	11.36	11.36
Goal	s	s	s	1.00	1.00	1.00

Table 11.5 Metrics for the Commission Problem

Method	m	n	s	C(M, S) = n/s	R(M, S) = m/s
Output bva	25	11	11	1	2.27
Decision table	3	11	11	1	0.27
DD-Path	25	11	11	1	2.27
DU-Path	25	33	33	1	0.76
Slice	25	40	40	1	0.63

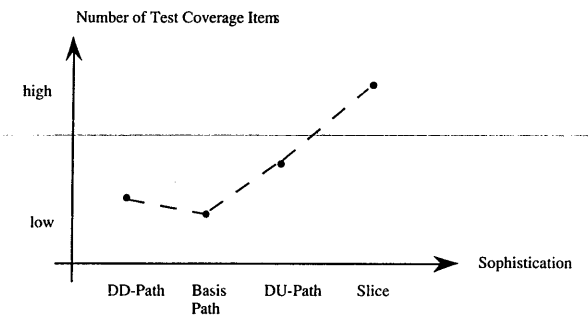

Figure 11.2 Trend of test coverage items (s).

Figures 11.2 and 11.3 show, respectively, the trend lines for the number of test coverage items (s in the definitions) and the effort to identify them as functions of structural testing methods. We no longer have the pleasing trade-off that we had for functional testing methods. Instead, these graphs illustrate the importance of choosing an appropriate structural coverage metric. We will see this when we revisit the case study from Chapter 8.

11.3 Case Study Revisited

Here, we continue our case study using the hypothetical insurance premium program example from the retrospective on functional testing (Chapter 8). The pseudocode implementation is minimal in the sense that it does very little error checking. The program graph of this implementation is in Figure 11.4.

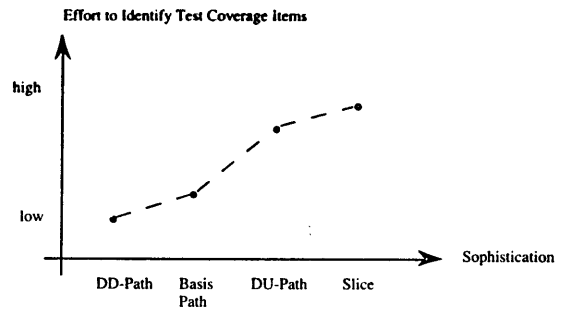

Figure 11.3 Trend of test method effort.

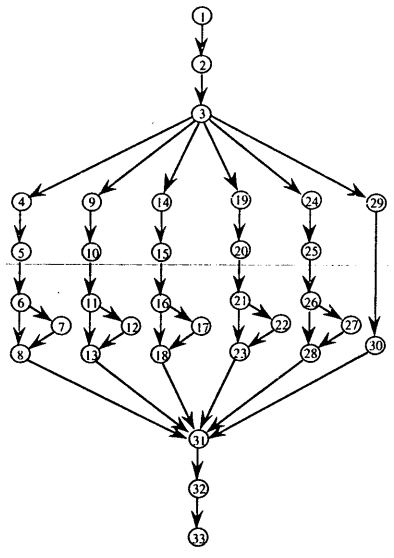

Figure 11.4 Program graph of the insurance premium program.

Pseudo-code for the Insurance Premium Program

```
    Dim driverAge, points As Integer
    Dim baseRate, premium As Real

1.  Input(baseRate, driverAge, points)
2.  premium = 0
3.  Select Case driverAge
4.  Case 1: 16<= driverAge  < 20
5.      ageMultiplier = 2.8
6.      If points < 1 Then
7.          safeDrivingReduction = 50
8.      End If
```



```
9.  Case 2: 20<= driverAge  < 25
10.     ageMultiplier = 1.8
11.     If points < 3 Then
12.         safeDrivingReduction = 50
13.     End If
14. Case 3: 25<= driverAge  < 45
15.     ageMultiplier = 1#
16.     If points < 5 Then
17.         safeDrivingReduction = 100
18.     End If
19. Case 4: 45<= driverAge  < 60
20.     ageMultiplier = 0.8
21.     If points < 7 Then
22.         safeDrivingReduction = 150
23.     End If
24. Case 5: 60<= driverAge  < 120
25.     ageMultiplier = 1.5
26.     If points < 5 Then
27.         safeDrivingReduction = 200
28.     End If
29. Case 6: Else
30.         Output("Driver age out of range")
31. End Select
32. premium = baseRate * ageMultiplier - safeDrivingReduction
33. Output(premium)
```

The cyclomatic complexity of the program graph of the insurance premium program is V(G) = 11, and exactly 11 feasible program execution paths exist. They are listed in Table 11.6.

If you take the time to follow the pseudocode for the various sets of functional test cases in Chapter 8, you will find the results shown in Table 11.7.

Now, we can see some of the insights gained from structural testing. For one thing, the problem of gaps and redundancies is obvious. Only the test cases from the hybrid approach yield complete path coverage. It is instructive to compare the results of these 25 test cases with the other two methods yielding the same number of test cases. The 25 boundary value test cases only cover six of the

Table 11.6 Paths in the Insurance Premium Program

Path	*Node Sequence*
p1	1 - 2 - 3 - 4 - 5 - 6 - 8 - 31 - 32 - 33
p2	1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 31 - 32 - 33
p3	1 - 2 - 3 - 9 - 10 - 11 - 13 - 31 - 32 - 33
p4	1 - 2 - 3 - 9 - 10 - 11 - 12 - 13 - 31 - 32 - 33
p5	1 - 2 - 3 - 14 - 15- 16 - 18 - 31 - 32 - 33
p6	1 - 2 - 3 - 14 - 15- 16 - 17 - 18 - 31 - 32 - 33
p7	1 - 2 - 3 - 19 - 20 - 21 - 23 - 31 - 32 - 33
p8	1 - 2 - 3 - 19 - 20 - 21 - 22 - 23 - 31 - 32 - 33
p9	1 - 2 - 3 - 24 - 25 - 26 - 28 - 31 - 32 - 33
p10	1 - 2 - 3 - 24 - 25 - 26 - 27 - 28 - 31 - 32 - 33
p11	1 - 2 - 3 - 29 - 30 - 31 - 32 - 33

Table 11.7 Path Coverage of Functional Methods in the Insurance Premium Program

Figure	Method	Test Cases	Paths Covered
8.7	Boundary value	25	p1, p2, p7, p8, p9, p10
8.8	Worst-case boundary value	273	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10
8.9	Weak equivalence class	5	p2, p4, p6, p8, p9
8.9	Strong equivalence class	25	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10
8.10	Decision table	10	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10
8.11	Hybrid	25	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11

feasible execution paths, while the 25 strong equivalence classes test cases cover 10 of the feasible execution paths. The next difference is in the coverage of the conditions in the case statement. Each predicate is a compound condition of the form a < = x < b. The only methods that yield test cases that exercise these extreme values are the worst-case boundary value (273) test cases and the hybrid (25) test cases. Quite a difference!

11.3.1 Path-Based Testing

Because the program graph is acyclic, only a finite number of paths exist — in this case, 11. The best choice is simply to have test cases that exercise each path. This automatically constitutes both statement and DD-Path coverage. The compound case predicates indicate multiple-condition coverage; this is accomplished only with the worst-case boundary test cases and the hybrid test cases. The remaining path-based coverage metrics are not applicable.

11.3.2 Data Flow Testing

Data flow testing for this problem is boring. The driverAge, points, and safeDrivingReduction variables all occur in six definition clear du-paths. The "uses" for driverAge and points are both predicate uses. Recall from Chapter 10 that the all-paths criterion implies all the lower data flow coverages.

11.3.3 Slice Testing

Slice testing does not provide much insight either. Four slices are of interest (the EndIF statements are not listed):

S(safeDrivingReduction, 32) = {1, 3, 4, 6, 7, 9, 11, 12 14, 16, 17, 19, 21, 22, 24, 26, 27, 31}
S(ageMultiplier, 32) = {1, 3, 4, 5, 9, 10, 14, 15, 19, 20, 24, 25, 31}
S(baseRate, 32) = {1}
S(Premium, 31) = {2}

The union of these slices (plus the EndIf statements) is the whole program. The only insight we might get from slice-based testing is that, if a failure occurred at line 32, the slices on safeDrivingReduction and ageMultiplier separate the program into two disjoint pieces, and that would simplify fault isolation.

References

Pressman, Roger S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 1982.

Brown, J.R. and Lipov, M., Testing for Software Reliability, *Proceedings of the International Symposium on Reliable Software*, Los Angeles, pp. 518–527, April 1975.

Exercises

1. Repeat the gaps and redundancies analysis for the Triangle Problem using the structured implementation in Chapter 2 and its DD-Path graph in Chapter 9.
2. Compute the coverage, redundancy, and net redundancy metrics for your study in Exercise 1.
3. The pseudocode for the Insurance Premium Program does not check for driver ages under 16 or (unlikely) over 120. The Else clause (Case 6) will catch these, but the output message is not very specific. Also, the output statement (33) is not affected by the driver age checks. Which functional testing techniques will reveal this fault? Which structural testing coverage, if not met, will reveal this fault?

IV

INTEGRATION AND SYSTEM TESTING

Chapter 12

Levels of Testing

In this chapter, we build a context for Part IV, in which we examine integration and system testing for traditional software. Our immediate goal is to identify what we mean by these levels of testing. We took a simplistic view in Chapter 1, where we identified three levels (unit, integration, and system) in terms of symmetries in the waterfall model of software development. This view has been relatively successful for decades; however, the advent of alternative life cycle models mandates a second look at these views of testing. We begin with the traditional waterfall model, mostly because it has enormous acceptance and similar expressive power. To ground our discussion in something concrete, we switch to the automated teller machine example.

In Parts IV and V, we also make a major shift in our thinking. We are more concerned with how to represent the item tested, because the representation may limit our ability to identify test cases. Take a look at the papers presented at the leading conferences (professional or academic) on software testing — you will find nearly as many presentations on specification models and techniques as on testing techniques.

12.1 Traditional View of Testing Levels

The traditional model of software development is the waterfall model, which is drawn as a V in Figure 12.1 to emphasize the basic levels of testing. In this view, information produced in one of the development phases constitutes the basis for test case identification at that level. Nothing controversial here: we certainly would hope that system test cases are somehow correlated with the requirements specification, and that unit test cases are derived from the detailed design of the unit. Two observations: a clear presumption of functional testing is used here, and an implied bottom-up testing order is used. Here, "bottom up" refers to levels of abstraction; in Chapter 13, bottom up also refers to a choice of orders in which units are integrated (and tested).

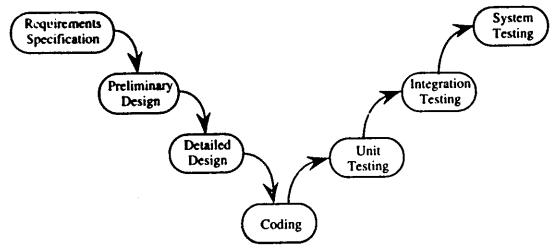

Figure 12.1 The waterfall life cycle.

Of the three traditional levels of testing (unit, integration, and system), unit testing is best understood. The testing theory and techniques we worked through in Parts II and III are directly applicable to unit testing. System testing is understood better than integration testing, but both need clarification. The bottom-up approach sheds some insight: test the individual components, and then integrate these into subsystems until the entire system is tested. System testing should be something that the customer (or user) understands, and it often borders on customer acceptance testing. Generally, system testing is functional instead of structural; this is mostly due to the lack of higher level structural notations.

The waterfall model is closely associated with top-down development and design by functional decomposition. The end result of preliminary design is a functional decomposition of the entire system into a tree-like structure of functional components. Figure 12.2 contains a partial functional decomposition of our automated teller machine (ATM) system. With this decomposition, top-down integration would begin with the main program, checking the calls to the three next level procedures (Terminal I/O, ManageSessions, and ConductTransactions). Following the tree, the ManageSessions procedure would be tested, and then the CardEntry, PIN Entry, and SelectTransaction procedures. In each case, the actual code for lower-level units is replaced by a stub, which is a throw-away piece of code that takes the place of the actual code. Bottom-up integration would be the opposite sequence, starting with the CardEntry, PIN Entry, and SelectTransaction procedures, and working up toward the main program. In bottom-up integration, units at higher levels are replaced by drivers (another form of throw-away code) that emulate the procedure calls. The "big bang" approach simply puts all the units together at once, with no stubs or drivers. Whichever approach is taken, the goal of traditional integration testing is to integrate previously tested units with respect to the functional decomposition tree. Although this describes integration testing as a process, dis-

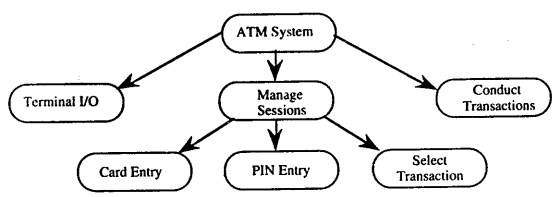

Figure 12.2 Partial functional decomposition of the ATM system.

cussions of this type offer little information about the methods or techniques. Before addressing these (real) issues, we need to see if the alternative life cycle models have any consequences for integration testing.

12.2 Alternative Life Cycle Models

Since the early 1980s, practitioners have devised alternatives in response to shortcomings of the traditional waterfall model of software development (Agresti, 1986). Common to all of these alternatives is the shift away from the functional decomposition to an emphasis on composition. Decomposition is a perfect fit both to the top-down progression of the waterfall model and to the bottom-up testing order. One of the major weaknesses of waterfall development cited by Agresti (1986) is the overreliance on this whole paradigm. Functional decomposition can only be well done when the system is completely understood, and it promotes analysis to the near exclusion of synthesis. The result is a very long separation between requirements specification and a completed system: and during this interval, no opportunity is available for feedback from the customer. Composition, on the other hand, is closer to the way people work: start with something known and understood, then add to it gradually, and maybe remove undesired portions. A very nice analogy can be applied to positive and negative sculpture. In negative sculpture, work proceeds by removing unwanted material, as in the mathematician's view of sculpting Michelangelo's David: start with a piece of marble, and simply chip away all non-David. Positive sculpture is often done with a medium like wax. The central shape is approximated, and then wax is either added or removed until the desired shape is attained. Think about the consequences of a mistake: with negative sculpture, the whole work must be thrown away and restarted. (A museum in Florence, Italy, contains half a dozen such false starts to David.) With positive sculpture, the erroneous part is simply removed and replaced. The centrality of composition in the alternative models has a major implication for integration testing.

12.2.1 Waterfall Spin-Offs

Three mainline derivatives of the waterfall model are used: incremental development, evolutionary development, and the spiral model (Boehm, 1988). Each of these involves a series of increments or builds as shown in Figure 12.3. Within a build, the normal waterfall phases from detailed design through testing occur with one important difference: system testing is split into two steps — regression and progression testing.

It is important to keep preliminary design as an integral phase, rather than to try to amortize such high-level design across a series of builds. (To do so usually results in unfortunate consequences of design choices made during the early builds that are regrettable in later builds.) Because preliminary design remains a separate step, we conclude that integration testing is unaffected in the spin-off models. The main impact of the series of builds is that regression testing becomes necessary. The goal of regression testing is to ensure that things that worked correctly in the previous build still work with the newly added code. Progression testing assumes that regression testing was successful and that the new function-

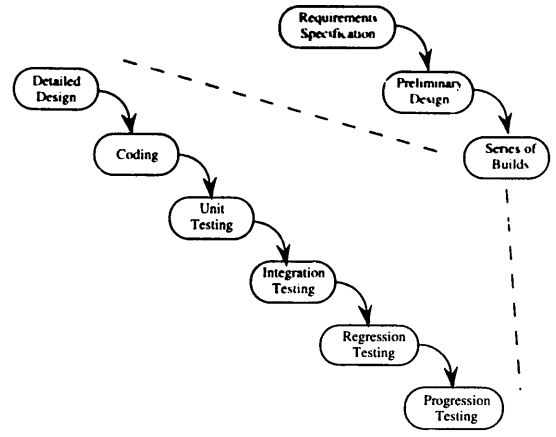

Figure 12.3 Life cycle with a build sequence.

ality can be tested. (We like to think that the addition of new code represents progress, not a regression.) Regression testing is an absolute necessity in a series of builds because of the well-known ripple effect of changes to an existing system. (The industrial average is that one change in five introduces a new fault.)

The differences among the three spin-off models are due to how the builds are identified. In incremental development, the motivation for separate builds is usually to level off the staff profile. With pure waterfall development, there can be a huge bulge of personnel for the phases from detailed design through unit testing. Most organizations cannot support such rapid staff fluctuations, so the system is divided into builds that can be supported by existing personnel. In evolutionary development, the presumption of a build sequence is still male, but only the first build is defined. Based on that, later builds are identified, usually in response to priorities set by the customer/user, so the system evolves to meet the changing needs of the user. The spiral model is a combination of rapid prototyping and evolutionary development, in which a build is defined first in terms of rapid prototyping and then is subjected to a go/no-go decision based on technology-related risk factors. From this, we see that keeping preliminary design as an integral step is difficult for the evolutionary and spiral models. To the extent that this cannot be maintained as an integral activity, integration testing is negatively affected. System testing is not affected.

Because a build is a set of deliverable end user functionality, one advantage common to all these spin-off models is that all yield earlier synthesis. This also results in earlier customer feedback, so two of the deficiencies of waterfall development are mitigated.

12.2.2 Specification-Based Life Cycle Models

Two other variations are responses to the "complete understanding" problem. (Recall that functional decomposition is successful only when the system is

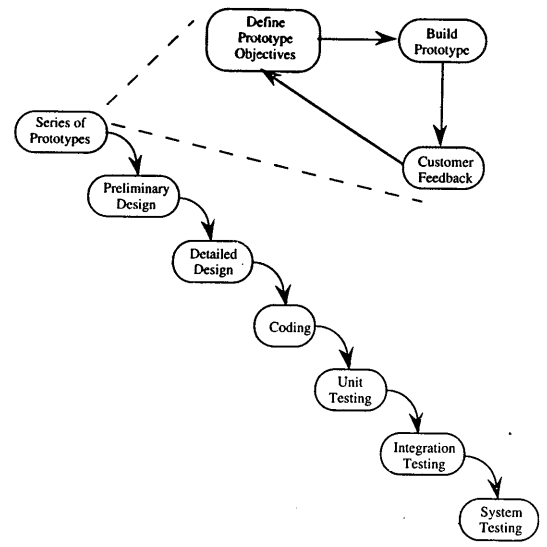

Figure 12.4 Rapid prototyping life cycle.

completely understood.) When systems are not fully understood (by either the customer or the developer), functional decomposition is perilous at best. The rapid prototyping life cycle (Figure 12.4) deals with this by drastically reducing the specification-to-customer feedback loop to produce very early synthesis. Rather than build a final system, a quick and dirty prototype is built and then used to elicit customer feedback. Depending on the feedback, more prototyping cycles may occur. Once the developer and the customer agree that a prototype represents the desired system, the developer goes ahead and builds to a correct specification. At this point, any of the waterfall spin-offs might also be used.

Rapid prototyping has no implications for integration testing; it has very interesting implications for system testing. Where are the requirements? Is the last prototype the specification? How are system test cases traced back to the prototype? One good answer to questions such as these is to use the prototyping cycles as information-gathering activities, and then produce a requirements specification in a more traditional manner. Another possibility is to capture what the customer does with the prototypes, define these as scenarios that are important to the customer, and then use these as system test cases. The main contribution of rapid prototyping is that it brings the operational (or behavioral) viewpoint to the requirements specification phase. Usually, requirements specification techniques emphasize the structure of a system, not its behavior. This is unfortunate, because most customers do not care about the structure, and they do care about the behavior.

Executable specifications (Figure 12.5) are an extension of the rapid prototyping concept. With this approach, the requirements are specified in an executable format (such as finite state machines, StateCharts, or Petri nets). The customer

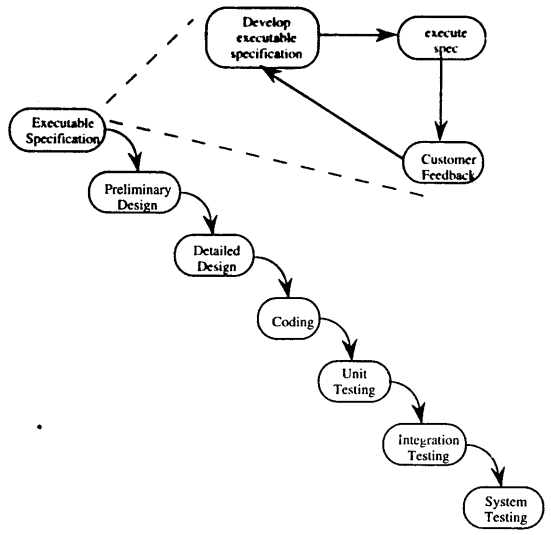

Figure 12.5 Executable specification.

then executes the specification to observe the intended system behavior and provides feedback as in the rapid prototyping model.

Once again, this life cycle has no implications for integration testing. One big difference is that the requirements specification document is explicit, as opposed to a prototype. More important, it is often a mechanical process to derive system test cases from an executable specification. We will see this in Chapter 15. Although more work is required to develop an executable specification, this is partially offset by the reduced effort to generate system test cases. Here is another important distinction: when system testing is based on an executable specification, we have an interesting form of structural testing at the system level.

12.3 The SATM System

In Part IV, we will relate our discussion to a higher-level example, the simple automatic teller machine (SATM) system. The version developed here is a revision of that found in Topper (1993); it is built around the 15 screens shown in Figure 12.6. This is a greatly reduced system; commercial ATM systems have hundreds of screens and numerous time-outs. The SATM terminal is sketched in Figure 12.7; in addition to the display screen, the terminal includes function buttons B1, B2, and B3, a digit keypad with a cancel key, slots for printer receipts and ATM cards, and doors for deposits and cash withdrawals.

The SATM system is described here with a traditional, structured analysis approach in Figures 12.8 and 12.9. The models are not complete, but they contain sufficient detail to illustrate the testing techniques under discussion. The structured analysis approach to requirements specification is still widely used. It enjoys extensive CASE tool support as well as commercial training and is described in

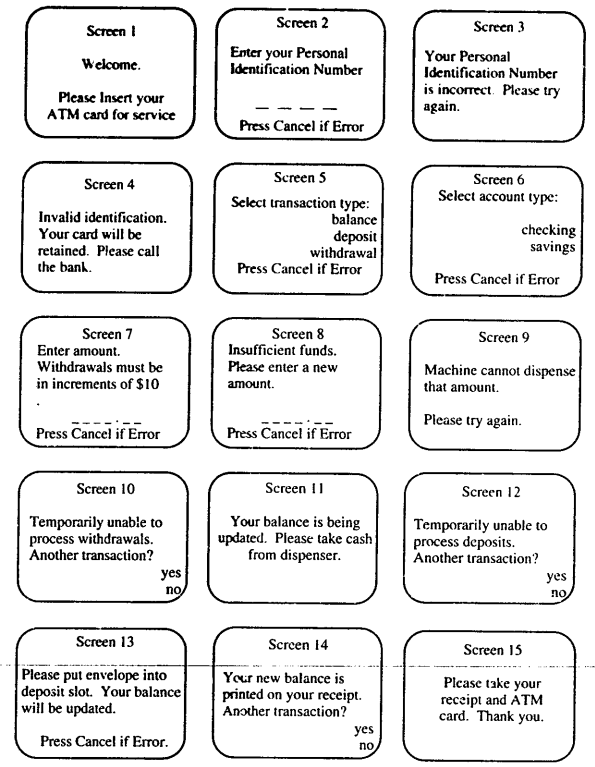

Figure 12.6 Screens for the SATM system.

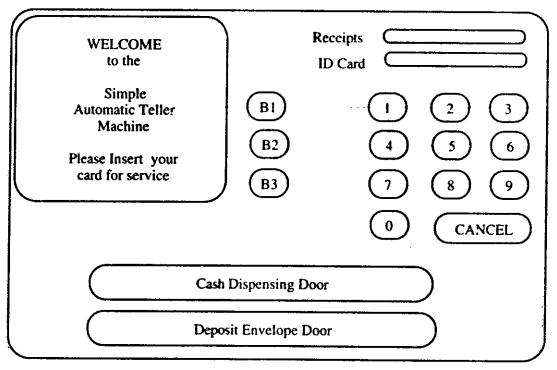

Figure 12.7 The SATM terminal.

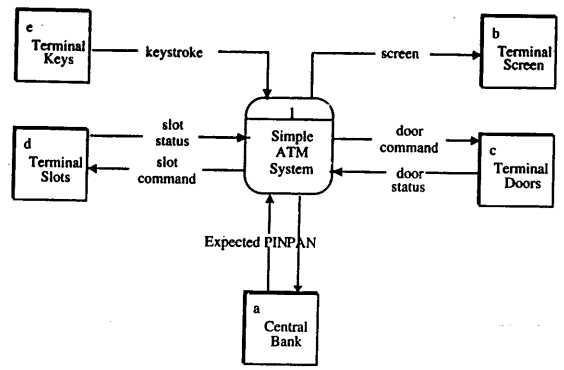

Figure 12.8 Context diagram of the SATM system.

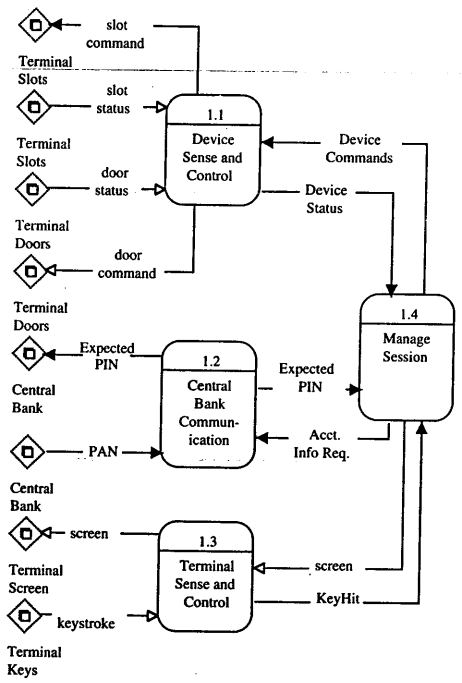

Figure 12.9 Level 1 data flow diagram of the SATM system.

numerous texts. The technique is based on three complementary models: function, data, and control. Here, we use data flow diagrams for the functional model, the entity/relationship model for data, and finite state machine models for the control aspect of the SATM system. The functional and data models were drawn with the Deft CASE tool from Sybase Inc. That tool identifies external devices (such as the terminal doors) with lower-case letters. Elements of the functional decomposition are identified with numbers (such as 1.5 for the Validate Card function). The open and filled arrowheads on flow arrows signify whether the flow item is simple or compound. The portions of the SATM system shown here pertain generally to the personal identification number (PIN) verification portion of the system.

The Deft CASE tool distinguishes between simple and compound flows, where compound flows may be decomposed into other flows, which may be compound. The graphic appearance of this choice is that simple flows have filled arrowheads, while compound flows have open arrowheads.

As an example, the compound flow screen has the following decomposition:

```
screen is comprised of
    screen1     welcome
    screen2     enter PIN
    screen3     wrong PIN
    screen4     PIN failed, card retained
    screen5     select trans type
    screen6     select account type
    screen7     enter amount
    screen8     insufficient funds
    screen9     cannot dispense that amount
    screen10    cannot process withdrawals
    screen11    take your cash
    screen12    cannot process deposits
    screen13    put dep envelop in slot
    screen14    another transaction?
    screen15    Thanks; take card and receipt
```

Figure 12.10 is an (incomplete) entity/relationship diagram of the major data structures in the SATM system: customers, accounts, terminals, and transactions. Good data modeling practice dictates postulating an entity for each portion of the system that is described by data that is retained (and used by functional components). The system needs customer data including each customer's identification and personal account number (PAN); these are encoded into the magnetic strip on the customer's ATM card. We would also want to know information about a customer's accounts, including the account numbers, the balances, the type of account (savings or checking), and the Personal Identification Number (PIN) of the account. At this point, we might ask why the PIN is not associated with the customer and the PAN with an account. Some design has crept into the specification: if the data were as questioned, a person's ATM card could be used by anyone; as it is, the present separation predisposes a security checking procedure. Part of the E/R model describes relationships among the entities: a customer HAS account(s), a customer conducts transaction(s) in a SESSION, and, independent of customer information, transaction(s) OCCUR at an ATM terminal. The single and double arrowheads signify the singularity or plurality of these relationships: one customer may have several accounts and may conduct none or several

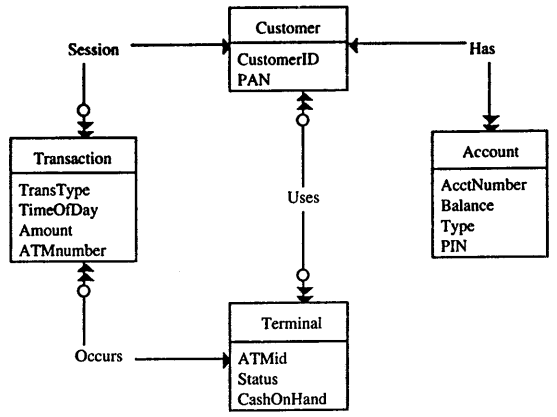

Figure 12.10 Entity/relationship model of the SATM system.

transactions. Many transactions may occur at a terminal, but one transaction never occurs at multiple terminals.

The data flow diagrams and the entity/relationship model contain information that is primarily structural. This is problematic for testers because test cases are concerned with behavior, not with structure. As a supplement, the functional and data information are linked by a control model; here, we use a finite state machine. Control models represent the point at which structure and behavior intersect; as such, they are of special utility to testers.

The upper-level finite state machine in Figure 12.11 divides the system into states that correspond to stages of customer usage. Other choices are possible; for instance, we might choose states to be screens displayed (this turns out to be a poor choice). Finite state machines can be hierarchically decomposed in much the same way as data flow diagrams can. The decomposition of the Await PIN state is shown in Figure 12.12. In both figures, state transitions are caused either by events at the ATM terminal (such as a keystroke) or by data conditions (such as the recognition that a PIN is correct). When a transition occurs, a corresponding action may also occur. We choose to use screen displays as such actions; this choice will prove to be very handy when we develop system-level test cases.

The function, data, and control models are the basis for design activities in the waterfall model (and its spin-offs). During design, some of the original decisions may be revised based on additional insights and more detailed requirements (such as performance or reliability goals). The end result is a functional decomposition such as the partial one shown in the structure chart in Figure 12.13. Notice that the original first-level decomposition into eight subsystems is no longer visible: the functionality has been reallocated among four logical components. Choices such as these are the essence of design, and design is beyond the scope of this book. In practice, testers often have to live with the results of poor design choices.

If we only use a structure chart to guide integration testing, we miss the fact that some (typically lower level) functions are used in more than one place. Here,

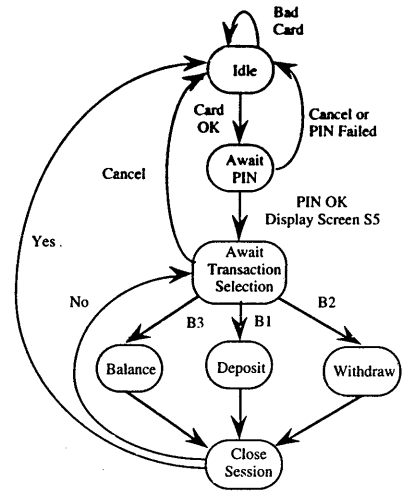

Figure 12.11 Upper-level SATM finite state machine.

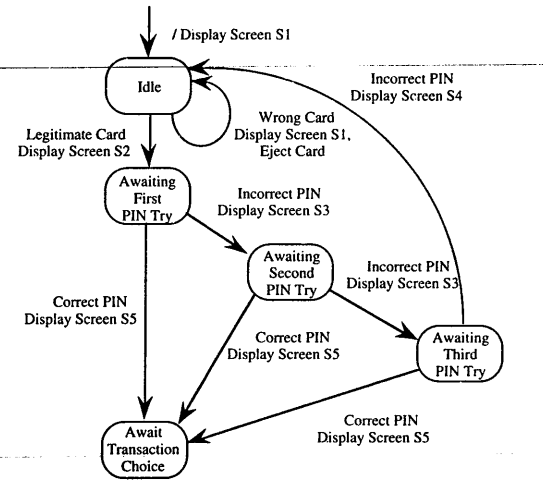

Figure 12.12 PIN entry finite state machine.

for example, the ScreenDriver function is used by several other modules, but it only appears once in the functional decomposition. In the next chapter, we will see that a "call graph" is a much better basis for integration test case identification. We can develop the beginnings of such a call graph from a more detailed view of portions of the system. To support this, we need a numbered decomposition and a more detailed view of two of the components.

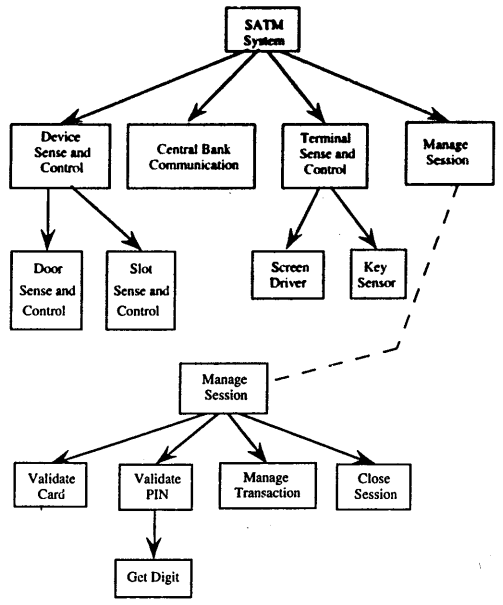

Figure 12.13 A decomposition tree for the SATM system.

Here is the functional decomposition carried further in outline form: the numbering scheme preserves the levels of the components in Figure 12.13.

1 SATM System
1.1 Device Sense & Control
1.1.1 Door Sense & Control
1.1.1.1 Get Door Status
1.1.1.2 Control Door
1.1.1.3 Dispense Cash
1.1.2 Slot Sense & Control
1.1.2.1 WatchCardSlot
1.1.2.2 Get Deposit Slot Status
1.1.2.3 Control Card Roller
1.1.2.3 Control Envelope Roller
1.1.2.5 Read Card Strip
1.2 Central Bank Comm.
1.2.1 Get PIN for PAN
1.2.2 Get Account Status
1.2.3 Post Daily Transactions
1.3 Terminal Sense & Control
1.3.1 Screen Driver
1.3.2 Key Sensor
1.4 Manage Session
1.4.1 Validate Card
1.4.2 Validate PIN
1.4.2.1 GetPIN

1.4.3 Close Session
1.4.3.1 New Transaction Request
1.4.3.2 Print Receipt
1.4.3.3 Post Transaction Local
1.4.4 Manage Transaction
1.4.4.1 Get Transaction Type
1.4.4.2 Get Account Type
1.4.4.3 Report Balance
1.4.4.4 Process Deposit
1.4.4.5 Process Withdrawal

As part of the specification and design process, each functional component is normally expanded to show its inputs, outputs, and mechanism. We do this with pseudocode (or PDL, for program design language) for three modules. The main program description follows the finite state machine description given in Figure 12.11. States in that diagram are implemented with a CASE statement.

```
Main Program
State = AwaitCard
Case State
Case 1: AwaitCard
            ScreenDriver(1, null)
            WatchCardSlot(CardSlotStatus)
            Do While CardSlotStatus is Idle
                WatchCardSlot(CardSlotStatus)
            End While
            ControlCardRoller(accept)
            ValidateCard(CardOK, PAN)
            If CardOK
                Then State = AwaitPIN
                Else    ControlCardRoller(eject)
            EndIf
            State = AwaitCard
 Case 2: AwaitPIN
            ValidatePIN(PINok, PAN)
            If PINok
                Then    ScreenDriver(2, null)
                        State = AwaitTrans
                Else    ScreenDriver(4, null)
            EndIf
            State = AwaitCard
 Case 3: AwaitTrans
            ManageTransaction
            State = CloseSession
 Case 4: CloseSession
            If NewTransactionRequest
                Then    State = AwaitTrans
                Else    PrintReceipt
            EndIf
            PostTransactionLocal
            CloseSession
            ControlCardRoller(eject)
            State = AwaitCard
 End Case (State)
 End. (Main program SATM)
```

The ValidatePIN procedure is based on the finite state machine shown in Figure 12.12, in which states refer to the number of PIN entry attempts.

```
Procedure ValidatePIN(PINok, PAN)
GetPINforPAN(PAN, ExpectedPIN)
Try = First
Case Try of
Case 1: First
        ScreenDriver(2, null)
        GetPIN(EnteredPIN)
        If EnteredPIN = ExpectedPIN
            Then   PINok = True
            Else   ScreenDriver(3, null)
        EndIf
        Try = Second
Case 2: Second
        ScreenDriver(2, null)
        GetPIN(EnteredPIN)
        If EnteredPIN = ExpectedPIN
            Then   PINok = True
            Else   ScreenDriver(3, null)
        EndIf
        Try = Third
Case 3: Third
        ScreenDriver(2, null)
        GetPIN(EnteredPIN)
        If EnteredPIN = ExpectedPIN
            Then   PINok = True
            Else   ScreenDriver(4, null)
                   PINok = False
        EndIf
EndCase (Try)
End.   (Procedure ValidatePIN)
```

The GetPIN procedure is based on another finite state machine in which states refer to the number of digits received; and in any state, either another digit key can be touched or the cancel key can be touched. Instead of another CASE statement implementation, the "states" are collapsed into iterations of a while-loop.

```
Procedure GetPIN(EnteredPIN, CancelHit)
Local Data: DigitKeys = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

CancelHit = False
EnteredPIN = null string
DigitsRcvd=0
Do While NOT(DigitsRcvd=4 OR CancelHit)
    KeySensor(KeyHit)
    If KeyHit IN DigitKeys
        Then
            EnteredPIN = EnteredPIN + KeyHit
            INCREMENT(DigitsRcvd)
            If DigitsRcvd=1 Then ScreenDriver(2,'X---')
            If DigitsRcvd=2 Then ScreenDriver(2,'XX--')
            If DigitsRcvd=3 Then ScreenDriver(2,'XXX-')
            If DigitsRcvd=4 Then ScreenDriver(2,'XXXX')
```



```
        Else
            CancelHit = True
        EndIf
End While
End.   (Procedure GetPIN)
```

If we follow the pseudocode in these three modules, we can identify the "uses" relationship among the modules in the functional decomposition. In Chapter 13, we shall see how this provides useful insights into integration testing.

Module	*Uses Modules*
SATM Main	WatchCardSlot
	Control Card Roller
	Screen Driver
	Validate Card
	Validate PIN
	Manage Transaction
	New Transaction Request
ValidatePIN	GetPINforPAN
	GetPIN
	Screen Driver
GetPIN	KeySensor
	Screen Driver

Notice that the "uses" information is not readily apparent in the functional decomposition. This information is developed (and extensively revised) during the more detailed phases of the design process. We will revisit this in Chapter 13.

12.4 Separating Integration and System Testing

We are almost in a position to make a clear distinction between integration and system testing. We need this distinction to avoid gaps and redundancies across levels of testing, to clarify appropriate goals for these levels, and to understand how to identify test cases at different levels. This whole discussion is facilitated by a concept essential to all levels of testing: the notion of a "thread." A thread is a construct that refers to execution time behavior. When we test a system, we use test cases to select (and execute) threads. We can speak of levels of threads: system threads describe system-level behavior, integration threads correspond to integration-level behavior, and unit threads correspond to unit-level behavior. Many authors use the term, but few define it; and of those who do, the offered definitions are not very helpful. For now, we take "thread" to be a primitive term, much like function and data. We will be very specific when we define threads in Chapter 14. In the next two chapters, we shall see that threads are most often recognized in terms of the way systems are described and developed. For example, we might think of a thread as a path through a finite state machine description

of a system, or we might think of a thread as something that is determined by a data context and a sequence of port-level input events, such as those in the context diagram of the SATM system. We could also think of a thread as a sequence of source statements, or as a sequence of machine instructions. The point is, threads are a generic concept, and they exist independently of how a system is described and developed.

We have already observed the structural versus behavioral dichotomy; here, we shall find that both of these views help us separate integration and system testing. The structural view reflects both the process by which a system is built and the techniques used to build it. We certainly expect that test cases at various levels can be traced back to developmental information. Although this is necessary, it fails to be sufficient: We will finally make our desired separation in terms of behavioral constructs.

12.4.1 Structural Insights

Everyone agrees that some distinction must be made, and that integration testing is at a more detailed level than system testing. There is also general agreement that integration testing can safely assume that the units have been separately tested, and that, taken individually, the units function correctly. One common view, therefore, is that integration testing is concerned with the interfaces among the units.

One possibility is to fall back on the symmetries in the waterfall life cycle model and say that integration testing is concerned with preliminary design information, while system testing is at the level of the requirements specification. This is a popular academic view, but it begs an important question: How do we discriminate between specification and preliminary design? The pat academic answer to this is the what versus how dichotomy: The requirements specification defines what, and the preliminary design describes how. Although this sounds good at first, it does not stand up well in practice. Some scholars argue that even the choice of a requirements specification technique is a design choice

The life cycle approach is echoed by designers who often take a "Don't Tread on Me" view of a requirements specification: a requirements specification should neither predispose nor preclude a design option. With this view, when information in a specification is so detailed that it "steps on the designer's toes," the specification is too detailed. This sounds good, but it still does not yield an operational way to separate integration and system testing.

The models used in the development process provide some clues. If we follow the definition of the SATM system, we could first postulate that system testing should make sure that all 15 display screens have been generated (an output domain-based, functional view of system testing). The entity/relationship model also helps: The one-to-one and one-to-many relationships help us understand how much testing must be done. The control model (in this case, a hierarchy of finite state machines) is the most helpful. We can postulate system test cases in terms of paths through the finite state machine(s); doing this yields a system-level analog of structural testing. The functional models (data flow diagrams and structure charts) move in the direction of levels because both express a functional decomposition. Even with this, we cannot look at a structure chart and identify

where system testing ends and integration testing starts. The best we can do with structural information is identify the extremes. For instance, the following threads are all clearly at the system level:

1. Insertion of an invalid card (this is probably the "shortest" system thread)
2. Insertion of a valid card, followed by three failed PIN entry attempts
3. Insertion of a valid card, a correct PIN entry attempt, followed by a balance inquiry
4. Insertion of a valid card, a correct PIN entry attempt, followed by a deposit
5. Insertion of a valid card, a correct PIN entry attempt, followed by a withdrawal
6. Insertion of a valid card, a correct PIN entry attempt, followed by an attempt to withdraw more cash than the account balance

We can also identify some integration-level threads. Go back to the pseudocode descriptions of ValidatePIN and GetPIN. ValidatePIN calls GetPIN, and GetPIN waits for KeySensor to report when a key is touched. If a digit is touched, GetPIN echoes an "X" to the display screen; but if the cancel key is touched, GetPIN terminates, and ValidatePIN considers another PIN entry attempt. We could push still lower and consider keystroke sequences such as two or three digits followed by cancel keystroke.

12.4.2 Behavioral Insights

Here is a pragmatic, explicit distinction that has worked well in industrial applications. Think about a system in terms of its port boundary, which is the location of system-level inputs and outputs. Every system has a port boundary; the port boundary of the SATM system includes the digit keypad, the function buttons, the screen, the deposit and withdrawal doors, the card and receipt slots, and so on. Each of these devices can be thought of as a port, and events occur at system ports. The port input and output events are visible to the customer, and the customer very often understands system behavior in terms of sequences of port events. Given this, we mandate that system port events are the "primitives" of a system test case; that is, a system test case (or equivalently, a system thread) is expressed as an interleaved sequence of port input and port output events. This fits our understanding of a test case, in which we specify preconditions, inputs, outputs, and postconditions. With this mandate, we can always recognize a level violation: If a test case (thread) ever requires an input (or an output) that is not visible at the port boundary, the test case cannot be a system-level test case (thread). Notice that this is clear, recognizable, and enforceable. We will refine this in Chapter 14 when we discuss threads of system behavior.

Threads support a highly analytical view of testing. Unit-level threads, for example, are sequences of source statements that execute (feasible paths). Integration-level threads can be thought of as sequences of unit-level threads, where we are concerned not with the "internals" of unit threads, but the interaction among them. Finally, system-level threads can be interpreted as sequences of integration-level threads. We will also be able to describe the interaction among

system-level threads. To end on a pun, the definitions of the next two chapters will tie these threads together.

References

Agresti, W.W., *New Paradigms for Software Development*, IEEE Computer Society Press, Washington, D.C., 1986.

Boehm, B.W., A spiral model for software development and enhancement, *IEEE Computer*, Vol. 21, No. 6, IEEE Computer Society Press, Washington, D.C., May 1988, pp. 61–72.

Topper, Andrew et al., *Structured Methods: Merging Models, Techniques, and CASE*, McGraw-Hill, New York, 1993.

Chapter 13

Integration Testing

In September 1999, the Mars Climate Orbiter mission failed after successfully traveling 416 million miles in 41 weeks. It disappeared just as it was to begin orbiting Mars. The fault should have been revealed by integration testing: Lockheed Martin Astronautics used acceleration data in English units (pounds), while the Jet Propulsion Laboratory did its calculations with metric units (newtons). NASA announced a $50,000 project to discover how this could have happened (Fordahl, 1999). They should have read this chapter.

Craftspersons are recognized by two essential characteristics: they have a deep knowledge of the tools of their trade, and they have a similar knowledge of the medium in which they work so that they understand their tools in terms of how they work with the medium. In Parts II and III, we focused on the tools (techniques) available to the testing craftsperson. Our goal there was to understand testing techniques in terms of their advantages and limitations with respect to particular types of software. Here, we shift our emphasis to the medium, with the goal of improving the testing craftsperson's judgment through a better understanding of the medium. We make a deliberate separation here: this chapter and the next address testing for software that has been defined, designed, and developed with the traditional models for function, data, control, and structure. Testing for object-oriented software is deferred to Part V. We continue our refinement of the simple automated teller machine (SATM) system in this chapter and use it to illustrate three distinct approaches to integration testing. For each approach, we begin with its basis and then discuss various techniques that use the base information. To continue the craftsperson metaphor, we emphasize the advantages and limitations of each integration testing technique.

13.1 A Closer Look at the SATM System

In Chapter 12, we described the SATM system in terms of its output screens (Figure 12.6), the actual terminal (Figure 12.7), its context and partial data flow diagrams

(Figures 12.8 and 12.9), an entity/relationship model of its data (Figure 12.10), finite state machines describing some of its behavior (Figures 12.11 and 12.12), and a partial functional decomposition (Figure 12.13). We also developed a pseudocode description of the main program and two units, ValidatePIN and GetPIN.

We begin here by expanding the functional decomposition that was started in Figure 12.12; the numbering scheme preserves the levels of the components in that figure. For easier reference, each component that appears in our analysis is given a new (shorter) number; these numbers are given in Table 13.1. (The only reason for this is to make the figures more readable.)

Table 13.1 SATM Units and Abbreviated Names

Unit Number	*Level Number*	*Unit Name*
1	1	SATM System
A	1.1	Device Sense & Control
D	1.1.1	Door Sense & Control
2	1.1.1.1	Get Door Status
3	1.1.1.2	Control Door
4	1.1.1.3	Dispense Cash
E	1.1.2	Slot Sense & Control
5	1.1.2.1	WatchCardSlot
6	1.1.2.2	Get Deposit Slot Status
7	1.1.2.3	Control Card Roller
8	1.1.2.3	Control Envelope Roller
9	1.1.2.5	Read Card Strip
10	1.2	Central Bank Comm.
11	1.2.1	Get PIN for PAN
12	1.2.2	Get Account Status
13	1.2.3	Post Daily Transactions
B	1.3	Terminal Sense & Control
14	1.3.1	Screen Driver
15	1.3.2	Key Sensor
C	1.4	Manage Session
16	1.4.1	Validate Card
17	1.4.2	Validate PIN
18	1.4.2.1	GetPIN
F	1.4.3	Close Session
19	1.4.3.1	New Transaction Request
20	1.4.3.2	Print Receipt
21	1.4.3.3	Post Transaction Local
22	1.4.4	Manage Transaction
23	1.4.4.1	Get Transaction Type
24	1.4.4.2	Get Account Type
25	1.4.4.3	Report Balance
26	1.4.4.4	Process Deposit
27	1.4.4.5	Process Withdrawal

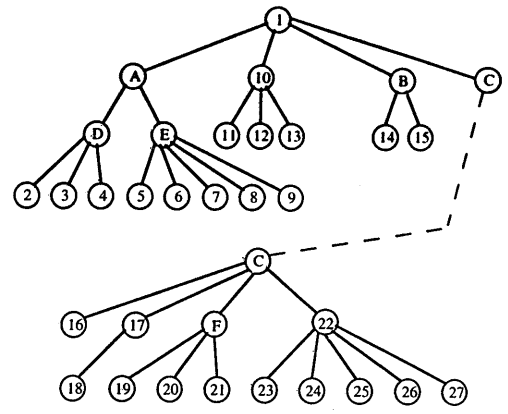

Figure 13.1 SATM functional decomposition tree.

The decomposition in Table 13.1 is pictured as a decomposition tree in Figure 13.1. This decomposition is the basis for the usual view of integration testing. It is important to remember that such a decomposition is primarily a packaging partition of the system. As software design moves into more detail, the added information lets us refine the functional decomposition tree into a unit calling graph. The unit calling graph is the directed graph in which nodes are program units and edges correspond to program calls; that is, if unit A calls unit B, a directed edge runs from node A to node B. We began the development of the call graph for the SATM system in Chapter 12 when we examined the calls made by the main program and the ValidatePIN and GetPIN modules. That information is captured in the adjacency matrix given in Table 13.2. This matrix was created with a spreadsheet; this turns out to be a handy tool for testers.

The SATM call graph is shown in Figure 13.2 Some of the hierarchy is obscured to reduce the confusion in the drawing. One thing should be quite obvious: drawings of call graphs do not scale up well. Both the drawings and the adjacency matrix provide insights to the tester. Nodes with high degree will be important to integration testing, and paths from the main program (node 1) to the sink nodes can be used to identify contents of builds for an incremental development.

13.2 Decomposition-Based Integration

Most textbook discussions of integration testing only consider integration testing based on the functional decomposition of the system tested. These approaches are all based on the functional decomposition, expressed either as a tree (Figure 13.1) or in textual form. These discussions inevitably center on the order in which modules are to be integrated. Four choices are available: from the top of the tree downward (top down), from the bottom of the tree upward (bottom up), some combination of these (sandwich), or most graphically, none of these (the big bang). All these integration orders presume that the units have been separately

Table 13.2 Adjacency Matrix for the SATM Call Graph

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1				X		X							X													
2																										
3																										
4																										
5																										
6																										
7																										
8																										
9																										
10								X																		
11																										
12									X																	
13																										
14																										
15									X	X																
16									X																	
17										X																
18											X															
19									X																	
20																										
21																										
22																										
23																										
24																										
25																										
26			X		X		X					X														
27			X		X		X					X														

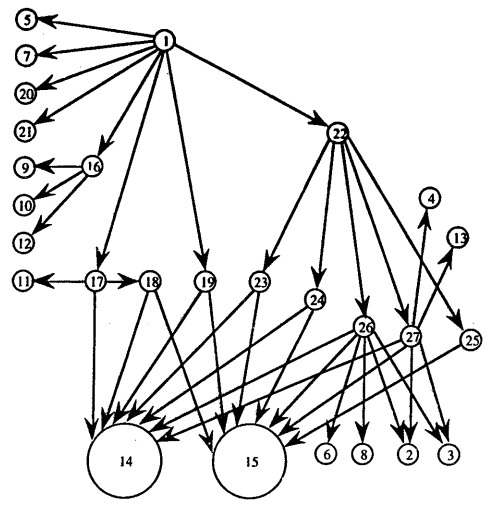

Figure 13.2 SATM call graph.

tested, thus, the goal of decomposition-based integration is to test the interfaces among separately tested units.

We can dispense with the big bang approach most easily: in this view of integration, all the units are compiled together and tested at once. The drawback to this is that when (not if!) a failure is observed, few clues are available to help isolate the location(s) of the fault. (Recall the distinction we made in Chapter 1 between faults and failures.)

13.2.1 Top-down Integration

Top-down integration begins with the main program (the root of the tree). Any lower level unit that is called by the main program appears as a "stub," where stubs are pieces of throw-away code that emulate a called unit. If we performed top-down integration testing for the SATM system, the first step would be to develop stubs for all the units called by the main program — WatchCardSlot, Control Card Roller, Screen Driver, Validate Card, Validate PIN, Manage Transaction, and New Transaction Request. Generally, testers have to develop the stubs, and some imagination is required. Here are two examples of stubs:

```
Procedure GetPINforPAN(PAN, ExpectedPIN) STUB
If PAN = '1123' Then PIN := '8876'
If PAN = '1234' Then PIN := '8765'
If PAN = '8746' Then PIN := '1253'
End

Procedure KeySensor(KeyHit) STUB
data: KeyStrokes STACK OF '8'. '8', '7', 'cancel'
KeyHit = POP (KeyStrokes)
End
```

In the stub for GetPINforPAN, the tester replicates a table look-up with just a few values that will appear in test cases. In the stub for KeySensor, the tester must devise a sequence of port events that can occur once each time the KeySensor procedure is called. (Here, we provided the keystrokes to partially enter the PIN '8876,' but the user hit the cancel button before the fourth digit.) In practice, the effort to develop stubs is usually quite significant. There is good reason to consider stub code as part of the software development and maintain it under configuration management.

Once all the stubs for SATM main have been provided, we test the main program as if it were a stand-alone unit. We could apply any of the appropriate functional and structural techniques and look for faults. When we are convinced that the main program logic is correct, we gradually replace stubs with the actual code. Figure 13.3 shows part of the top-down integration testing sequence for the SATM decomposition in Figure 13.1. At the uppermost level, we would have stubs for the four components in the first-level decomposition. There would be four integration sessions; in each, one component would be actual (previously unit tested) code, and the other three would be stubs. Top-down integration follows a breadth-first traversal of the functional decomposition tree. Two additional integration levels are shown in Figure 13.3.

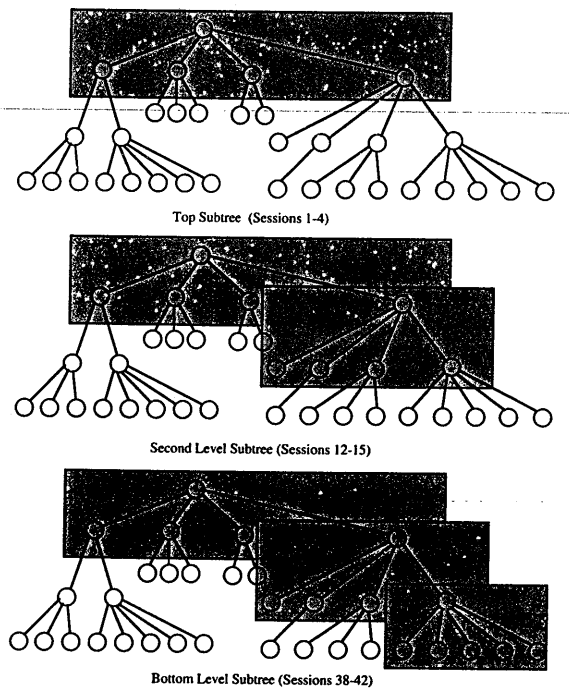

Figure 13.3 Top-down integration.

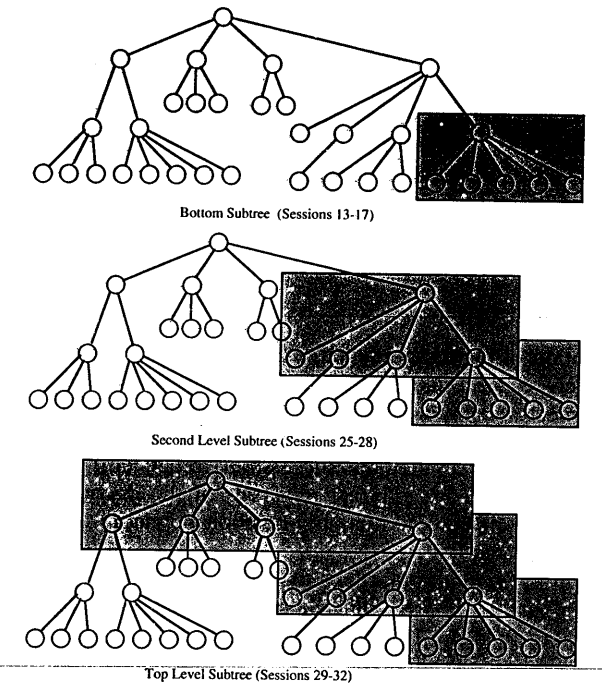

Figure 13.4 Bottom-up integration.

Even this can be problematic. Would we replace all the stubs at once? If we did, we would have a "small bang" for units with a high outdegree. If we replace one stub at a time, we retest the main program once for each replaced stub. This means that, for the SATM main program example here, we would repeat its integration test eight times (once for each replaced stub, and once with all the stubs).

13.2.2 Bottom-up Integration

Bottom-up integration is a "mirror image" to the top-down order, with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree. (See Figure 13.4.) In bottom-up integration, we start with the leaves of the decomposition tree (units like ControlDoor and DispenseCash), and test them with specially coded drivers. Less throw-away code exists in drivers than there is in stubs. Recall we had one stub for each child node in the decomposition tree. Most systems have a fairly high fan-out near the leaves; so in the bottom-up integration order, we will not have as many drivers. This is partially offset by the fact that the driver modules will be more complicated.

13.2.3 Sandwich Integration

Sandwich integration is a combination of top-down and bottom-up integration. If we think about it in terms of the decomposition tree, we are really only doing big bang integration on a subtree (see Figure 13.5). There will be less stub and driver development effort, but this will be offset to some extent by the added difficulty of fault isolation that is a consequence of big bang integration. (We could probably discuss the size of a sandwich, from dainty finger sandwiches to Dagwood-style sandwiches, but not now.)

13.2.4 Pros and Cons

With the exception of big bang integration, the decomposition-based approaches are all intuitively clear. Build with tested components. Whenever a failure is observed, the most recently added unit is suspected. Integration testing progress is easily tracked against the decomposition tree. (If the tree is small, it is a nice touch to shade in nodes as they are successfully integrated.) The top-down and bottom-up terms suggest breadth-first traversals of the decomposition tree, but this is not mandatory. (We could use full-height sandwiches to test the tree in a depth-first manner.)

One of the most frequent objections to functional decomposition and waterfall development is that both are artificial, and both serve the needs of project management more than the needs of software developers. This holds true also for decomposition-based testing. The whole mechanism is that units are integrated with respect to structure; this presumes that correct behavior follows from individually correct units and correct interfaces. (Practitioners know better.) The development effort for stubs or drivers is another drawback to these approaches, and this is compounded by the retesting effort. Here is a formula that computes the number of integration test sessions for a given decomposition tree (a test session is one set of tests for a specific configuration actual code and stubs):

$$\text{Sessions} = \text{nodes} - \text{leaves} + \text{edges}$$

The SATM system has 42 integration testing sessions, which means 42 separate sets of integration test cases.

For top-down integration, (nodes – 1) stubs are needed, and for bottom-up integration, (nodes – leaves) drivers are needed. For the SATM system, this is 32 stubs and 10 drivers.

13.3 Call Graph-Based Integration

One of the drawbacks of decomposition-based integration is that the basis is the functional decomposition tree. If we use the call graph instead, we mitigate this deficiency; we also move in the direction of structural testing. We are in a position to enjoy the investment we made in the discussion of graph theory. Because the call graph is a directed graph, why not use it the way we used program graphs?

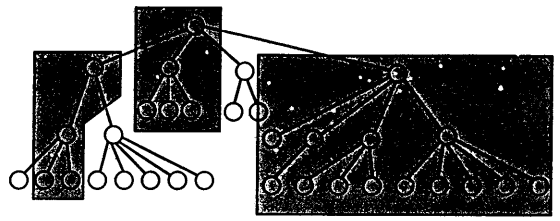

Figure 13.5 Sandwich integration.

This leads us to two new approaches to integration testing: we will refer to them as pair-wise integration and neighborhood integration.

13.3.1 Pair-Wise Integration

The idea behind pair-wise integration is to eliminate the stub/driver development effort. Instead of developing stubs and/or drivers, why not use the actual code? At first, this sounds like big bang integration, but we restrict a session to only a pair of units in the call graph. The end result is that we have one integration test session for each edge in the call graph (40 for the SATM call graph in Figure 13.2). This is not much of a reduction in sessions from either top-down or bottom-up (42 sessions), but it is a drastic reduction in stub/driver development. Four pair-wise integration sessions are shown in Figure 13.6.

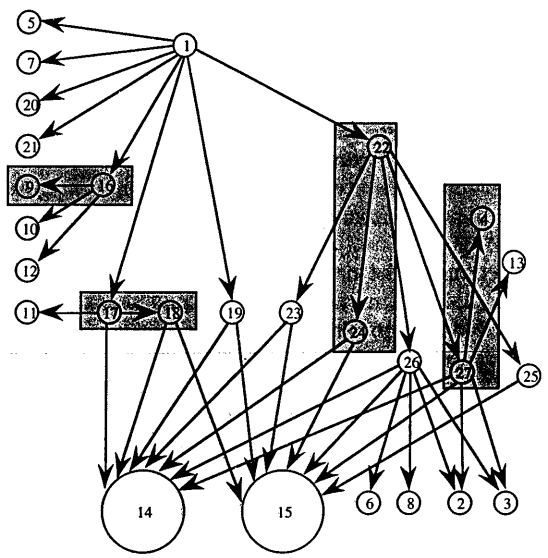

Figure 13.6 Pair-wise integration.

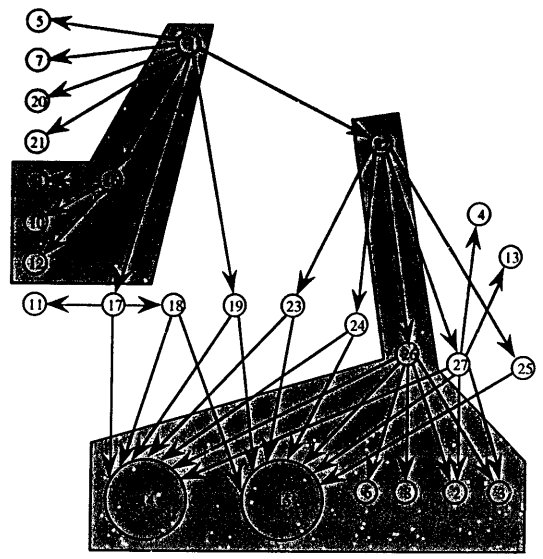

Figure 13.7 Neighborhood integration.

13.3.2 Neighborhood Integration

We can let the mathematics carry us still further by borrowing the notion of a neighborhood from topology. (This is not too much of a stretch — graph theory is a branch of topology.) The neighborhood of a node in a graph is the set of nodes that are one edge away from the given node. In a directed graph, this includes all the immediate predecessor nodes and all the immediate successor nodes (notice that these correspond to the set of stubs and drivers of the node). The neighborhoods for nodes 16 and 26 are shown in Figure 13.7. The 11 neighborhoods for the SATM example (based on the call graph in Figure 13.2) are listed in Table 13.3.

We can always compute the number of neighborhoods for a given call graph. Each interior node will have one neighborhood, plus one extra in case leaf nodes are connected directly to the root node. (An interior node has a nonzero indegree and a nonzero outdegree.) We have:

Interior nodes = nodes – (source nodes + sink nodes)
Neighborhoods = interior nodes + source nodes

which combine to:

Neighborhoods = nodes – sink nodes

Neighborhood integration yields a drastic reduction in the number of integration test sessions (down to 11 from 40), and it avoids stub and driver development. The end result is that neighborhoods are essentially the sandwiches that we slipped past in the previous section. (It is slightly different, because the base information for neighborhoods is the call graph, not the decomposition tree.) What they share with sandwich integration is more significant: neighborhood integration testing has the fault isolation difficulties of "medium bang" integration.

Table 13.3 SATM Neighborhoods

Node	Predecessors	Successors
16	1	9, 10, 12
17	1	11, 14, 18
18	17	14, 15
19	1	14, 15
23	22	14, 15
24	22	14, 15
26	22	14, 15, 6, 8, 2, 3
27	22	14, 15, 2, 3, 4, 13
25	22	15
22	1	23, 24, 26, 27, 25
1	n/a	5, 7, 2, 21, 16, 17, 19, 22

13.3.3 Pros and Cons

The call graph-based integration techniques move away from a purely structural basis toward a behavioral basis, thus, the underlying assumption is an improvement. These techniques also eliminate the stub/driver development effort. In addition to these advantages, call graph-based integration matches well with developments characterized by builds and composition. For example, sequences of neighborhoods can be used to define builds. Alternatively, we could allow adjacent neighborhoods to merge (into villages?) and provide an orderly, composition-based growth path. All this supports the use of neighborhood-based integration for systems developed by life cycles in which composition dominates.

The biggest drawback to call graph-based integration testing is the fault isolation problem, especially for large neighborhoods. A more subtle but closely related problem occurs. What happens if (when) a fault is found in a node (unit) that appears in several neighborhoods? (For example, the screen driver unit appears in 7 of the 11 neighborhoods.) Obviously, we resolve the fault; but this means changing the unit's code in some way, which in turn means that all the previously tested neighborhoods that contain the changed node need to be retested.

Finally, a fundamental uncertainty exists in any structural form of testing: the presumption that units integrated with respect to structural information will exhibit correct behavior. We know where we are going: we want system-level threads of behavior to be correct. When integration testing based on call graph information is complete, we still have quite a leap to get to system-level threads. We resolve this by changing the basis from call graph information to special forms of paths.

13.4 Path-Based Integration

Much of the progress in the development of mathematics comes from an elegant pattern: have a clear idea of where you want to go, and then define the concepts that take you there. We do this here for path-based integration testing, but first we need to motivate the definitions.

We already know that the combination of structural and functional testing is highly desirable at the unit level; it would be nice to have a similar capability for integration (and system) testing. We also know that we want to express system testing in terms of behavioral threads. Lastly, we revise our goal for integration testing: instead of testing interfaces among separately developed and tested units, we focus on interactions among these units. ("Co-functioning" might be a good term.) Interfaces are structural; interaction is behavioral.

When a unit executes, some path of source statements is traversed. Suppose that a call goes to another unit along such a path: at that point, control is passed from the calling unit to the called unit, where some other path of source statements is traversed. We cleverly ignored this situation in Part III, because this is a better place to address the question. Two possibilities are available: abandon the single-entry, single-exit precept and treat such calls as an exit followed by an entry, or suppress the call statement because control eventually returns to the calling unit anyway. The suppression choice works well for unit testing, but it is antithetical to integration testing.

13.4.1 New and Extended Concepts

To get where we need to go, we need to refine some of the program graph concepts. As before, these refer to programs written in an imperative language. We allow statement fragments to be a complete statement, and statement fragments are nodes in the program graph.

Definition

A source node in a program is a statement fragment at which program execution begins or resumes.

The first executable statement in a unit is clearly a source node. Source nodes also occur immediately after nodes that transfer control to other units.

Definition

A sink node in a unit is a statement fragment at which program execution terminates.

The final executable statement in a program is clearly a sink node; so are statements that transfer control to other units.

Definition

A module execution path is a sequence of statements that begins with a source node and ends with a sink node, with no intervening sink nodes.

The effect of the definitions so far is that program graphs now have multiple source and sink nodes. This would greatly increase the complexity of unit testing, but integration testing presumes unit testing is complete.

Definition

A message is a programming language mechanism by which one unit transfers control to another unit.

Depending on the programming language, messages can be interpreted as subroutine invocations, procedure calls, and function references. We follow the convention that the unit that receives a message (the message destination) always eventually returns control to the message source. Messages can pass data to other units. We can finally make the definitions for path-based integration testing. Our goal is to have an integration testing analog of DD-Paths.

Definition

An MM-Path is an interleaved sequence of module execution paths and messages.

The basic idea of an MM-Path is that we can now describe sequences of module execution paths that include transfers of control among separate units. These transfers are by messages, therefore, MM-Paths always represent feasible execution paths, and these paths cross unit boundaries. We can find MM-Paths in an extended program graph in which nodes are module execution paths and edges are messages. The hypothetical example in Figure 13.8 shows an MM-Path (the dark line) in which module A calls module B, which in turn calls module C. Notice that, for traditional (procedural) software, MM-Paths will always begin (and end) in the main program.

In module A, nodes 1 and 5 are source nodes, and nodes 4 and 6 are sink nodes. Similarly, in module B, nodes 1 and 3 are source nodes, and nodes 2 and 4 are sink nodes. Module C has a single source node, 1, and a single sink node, 4. Seven module execution paths are shown in Figure 13.3:

MEP(A,1) = <1, 2, 3, 6>
MEP(A,2) = <1, 2, 4>
MEP(A,3) = <5, 6>
MEP(B,1) = <1, 2>
MEP(B,1) = <3, 4>
MEP(C,1) = <1, 2, 4, 5>
MEP(C,2) = <1, 3, 4, 5>

We can now define an integration testing analog of the DD-Path graph that serves unit testing so effectively.

Definition

Given a set of units, their MM-Path graph is the directed graph in which nodes are module execution paths and edges correspond to messages and returns from one unit to another.

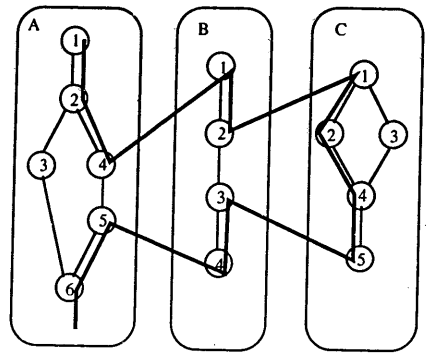

Figure 13.8 MM-Path across three units.

Notice that MM-Path graphs are defined with respect to a set of units. This directly supports composition of units and composition-based integration testing. We can even compose down to the level of individual module execution paths, but that is probably more detailed than necessary.

Figure 13.9 shows the MM-Path graph for the example in Figure 13.8. The solid arrows indicate messages; the corresponding returns are indicated by dotted arrows. We should consider the relationships among module execution paths, program paths, DD-Paths, and MM-Paths. A program path is a sequence of DD-Paths, and an MM-Path is a sequence of module execution paths. Unfortunately, there is no simple relationship between DD-Paths and module execution paths. Either might be contained in the other, but more likely, they partially overlap. Because MM-Paths implement a function that transcends unit boundaries, we do have one relationship: consider the intersection of an MM-Path with a unit. The module execution paths in such an intersection are an analog of a slice with respect to the (MM-Path) function. Stated another way, the module execution paths in such an intersection are the restriction of the function to the unit in which they occur.

The MM-Path definition needs some practical guidelines. How long ("deep" might be better) is an MM-Path? Two observable behavioral criteria put endpoints

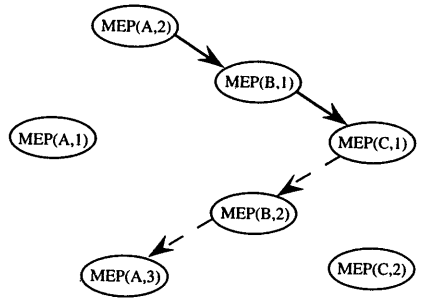

Figure 13.9 MM-Path graph derived from Figure 13.8.

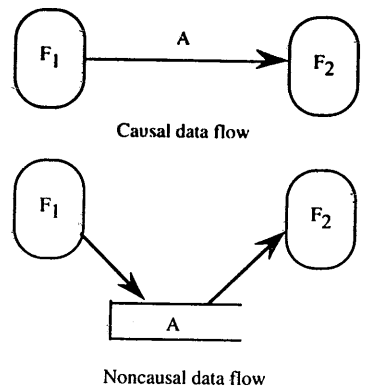

Figure 13.10 Data quiescence.

on MM-Paths: message and data quiescence. Message quiescence occurs when a unit that sends no messages is reached (like module C in Figure 13.8).

Data quiescence occurs when a sequence of processing culminates in the creation of stored data that is not immediately used. In the ValidateCard unit, the account balance is obtained, but it is not used until after a successful PIN entry. Figure 13.10 shows how data quiescence appears in a traditional data flow diagram. Points of quiescence are natural endpoints for an MM-Path.

13.4.2 MM-Paths in the SATM System

The pseudocode descriptions developed in Chapter 12 are repeated for convenient reference; statement fragments are numbered as we did to construct program graphs. Also, the messages are numbered as comments. We will use these to describe selected MM-Paths. The arguments to ScreenDriver refer to the screens as numbered in Figure 12.6. Procedure GetPIN is really a stub that is designed to respond to a correct digit event sequence for ExpectedPIN = 1234.

```
1.  Main Program
2.  State = AwaitCard
3.  Case State
4.  Case 1: AwaitCard
5.          ScreenDriver(1, null)                  msg 1
6.          WatchCardSlot(CardSlotStatus)          msg 2
7.          Do While CardSlotStatus is Idle
8.              WatchCardSlot(CardSlotStatus)      msg 3
9.          End While
10.         ControlCardRoller(accept)              msg 4
11.         ValidateCard(CardOK, PAN)              msg 5
12.         If CardOK
13.             Then State = AwaitPIN
14.             Else ControlCardRoller(eject)      msg 6
15.         EndIf
16.         State = AwaitCard
17. Case 2: AwaitPIN
18.         ValidatePIN(PINok, PAN)                msg 7
```

```
19.              If PINok
20.                  Then   ScreenDriver(2, null)        msg 8
21.                         State = AwaitTrans
22.                  Else   ScreenDriver(4, null)        msg 9
23.              EndIf
24.              State = AwaitCard
25. Case 3: AwaitTrans
26.              ManageTransaction                       msg 10
27.              State = CloseSession
28. Case 4: CloseSession
29.              If NewTransactionRequest
30.                  Then   State = AwaitTrans
31.                  Else   PrintReceipt                 msg 11
32.              EndIf
33.              PostTransactionLocal                    msg 12
34.              CloseSession                            msg 13
35.              ControlCardRoller(eject)                msg 14
36.              State = AwaitCard
37. End Case (State)
38. End.     (Main program SATM)

39. Procedure ValidatePIN(PINok, PAN)
40. GetPINforPAN(PAN, ExpectedPIN)                       msg 15
41. Try = First
42. Case Try of
43. Case 1: First
44.          ScreenDriver(2, null)                       msg 16
45.          GetPIN(EnteredPIN)                          msg 17
46.          If EnteredPIN = ExpectedPIN
47.              Then   PINok = True
48.              Else   ScreenDriver(3, null)            msg 18
49.                     Try = Second
50.          EndIf
51. Case 2: Second
52.          ScreenDriver(2, null)                       msg 19
53.          GetPIN(EnteredPIN)                          msg 20
54.          If EnteredPIN = ExpectedPIN
55.              Then   PINok = True
56.              Else   ScreenDriver(3, null)            msg 21
57.          EndIf
58.          Try = Third
59. Case 3: Third
60.          ScreenDriver(2, null)                       msg 22
61.          GetPIN(EnteredPIN)                          msg 23
62.          If EnteredPIN = ExpectedPIN
63.              Then   PINok = True
64.              Else   ScreenDriver(4, null)            msg 24
65.                     PINok = False
66.          EndIf
67. EndCase (Try)
68. End.         (Procedure ValidatePIN)

69. Procedure GetPIN(EnteredPIN, CancelHit)
70. Local Data: DigitKeys = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
71. CancelHit = False
72. EnteredPIN = null string
73. DigitsRcvd=0
74. Do While NOT(DigitsRcvd=4 OR CancelHit)
75.      KeySensor(KeyHit)                               msg 25
```



```
76.      If KeyHit IN DigitKeys
77.          Then
78.              EnteredPIN = EnteredPIN + KeyHit
79.              INCREMENT(DigitsRcvd)
80.              If digitsRcvd = 1
81.                  Then ScreenDirver (2, 'X---')       msg 26
82.              EndIf
83.              If digitsRcvd = 2
84.                  Then ScreenDirver (2, 'XX--')       msg 27
85.              EndIf
86.              If digitsRcvd = 3
87.                  Then ScreenDirver (2, 'XXX-')       msg 28
88.              EndIf
89.              If digitsRcvd = 4
90.                  Then ScreenDirver (2, 'XXXX')       msg 29
91.              EndIf
92.          Else
93.              CancelHit = True
94.      EndIf
95. End While
96. End.     (Procedure GetPIN)
```

SATM Main contains 16 source nodes. All except node 1 are where a procedure/function call returns control: 1, 6, 7, 9, 11, 12, 15, 19, 21, 23, 27, 30, 32, 34, 35, and 36. SATM Main contains 16 sink nodes. As with the source nodes, most of these are at procedure/function calls: 5, 6, 8, 10, 11, 14, 17, 18, 20, 22, 26, 31, 33, 34, 35, and 38. Notice that when two sequential procedure calls are used, a statement can be both a sink and a source node. Most of the module execution paths in SATM Main are very short; this pattern is due to the high density of messages to other units.

Only one nontrivial module execution path is contained in the first 17 lines of SATM Main: <1, 2, 3, 4>. Procedure calls, such as <5>, <6>, <8>, <10>, <11> and <14> are trivial in the sense that not much happens in SATM Main. Other very short module execution paths are associated with the control structures — for example, <9, 7>, <9, 10>, and <15, 16>.

Here is the MM-Path for a correct PIN entry on the first try. The module execution paths are described by giving the name of the unit followed by the sequence of the statement fragment numbers. Figure 13.11 illustrates the sequential nature of an MM-Path using a UML-style sequence diagram.

```
Main (1, 2, 3, 17, 18)
msg 7
ValidatePIN (39, 40)
msg 15
GetPINforPAN (no pseudo-code given)
ValidatePIN (41, 42, 43, 44)
msg 16
ScreenDriver (no pseudo-code given)
ValidatePIN (45)
msg 17
GetPIN(69, 70, 71, 72, 73, 74, 75)
msg 25
KeySensor (no pseudo-code given)     'first digit
GetPIN (76, 77, 78, 79, 80, 81)
```

```
 msg 26
ScreenDriver (no pseudo-code given)
GetPIN (82, 83, 85, 86, 88, 89, 91, 94, 95, 74, 75)
msg 25
KeySensor (no pseudo-code given) )    'second digit
GetPIN (76, 77, 78, 79, 80, 82, 83, 84)
 msg 27
ScreenDriver (no pseudo-code given)
GetPIN (85, 86, 88, 89, 91, 94, 95, 74, 75)
msg 25
KeySensor (no pseudo-code given) )    'third digit
GetPIN (76, 77, 78, 79, 80, 82, 83, 85, 86, 87)
 msg 28
ScreenDriver (no pseudo-code given)
GetPIN (88, 89, 91, 94, 95, 74, 75)
msg 25
KeySensor (no pseudo-code given) )    'fourth digit
GetPIN (76, 77, 78, 79, 80, 82, 83, 85, 86, 88, 89, 90)
 msg 29
ScreenDriver (no pseudo-code given)
GetPIN (91, 94, 95, 74, 96)
ValidatePIN (46, 47, 50, 67, 68)
Main(19)
```

13.4.4 MM-Path Complexity

If you compare the MM-Paths in Figures 13.8 and 13.11, it is intuitively clear that the latter is more complex than the former. Their directed graphs are shown together in Figure 13.12. The multiplicity of edges (e.g., between ScreenDriver and GetPIN) preserves the message connections, and the double-headed arrows capture the sending and return of a message (with less clutter). Because these

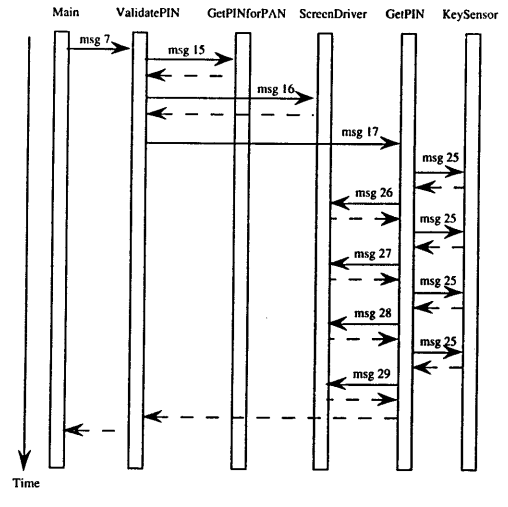

Figure 13.11 UML sequence diagram of the sample MM-Path.

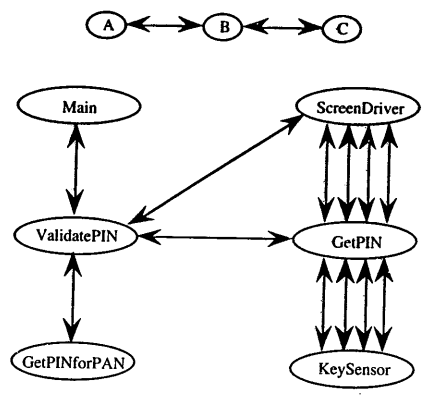

Figure 13.12 MM-Path directed graphs.

are strongly connected directed graphs, we can "blindly" compute their cyclomatic complexities; recall the formula is:

$$V(G) = e - n + 2p$$

where p is the number of strongly connected regions. For structured procedural code, we will always have p = 1, so the formula reduces to V(G) = e – n +2. The results, respectively, are V(G) = 3 and V(G) = 20.

This seems reasonable. The second graph is clearly more complex than the first; and if we remove some of the message traffic, say between the GetPIN and KeySensor nodes, both the intuitive and the computed complexity will be reduced. The role of "2p" in the formula is annoying. It acts like an offset because it will always be exactly 2. This suggests simply dropping it. If we were to do this, the simplest MM-Path, in which unit A calls unit B and B returns, would have a complexity of 0. Worse yet, a stand-alone unit would have a negative complexity of -1. Some other possibilities are suggested in the exercises.

13.4.5 Pros and Cons

MM-Paths are a hybrid of functional and structural testing. They are functional in the sense that they represent actions with inputs and outputs. As such, all the functional testing techniques are potentially applicable. The structural side comes from how they are identified, particularly the MM-Path graph. The net result is that the cross-check of the functional and structural approaches is consolidated into the constructs for path-based integration testing. We therefore avoid the pitfall of structural testing; and, at the same time, integration testing gains a fairly seamless junction with system testing. Path-based integration testing works equally well for software developed in the traditional waterfall process or with one of the composition-based alternative life cycle models. We will revisit these concepts again in Chapter 18; there we will see that the concepts are equally applicable to object-oriented software testing. The most important advantage of path-based integration

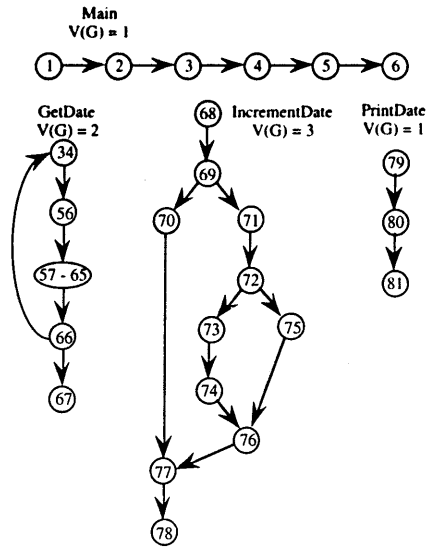

Figure 13.13 Main and first level units.

testing is that it is closely coupled with actual system behavior, instead of the structural motivations of decomposition and call graph-based integration.

The advantages of path-based integration come at a price: more effort is needed to identify the MM-Paths. This effort is probably offset by the elimination of stub and driver development.

13.5 Case Study

Our now familiar NextDate is rewritten here as a main program with a functional decomposition into procedures and functions. This "integration version" is a slight extension: there is (limited) added validity checking for months, days, and years. The pseudocode grows from 50 statements to 81. Figures 13.13 and 13.14 show the program graphs of units in the integration version of NextDate. The functional decomposition is shown in Figure 13.15, and the Call Graph is shown in Figure 13.16.

```
1.  Main integrationNextDate
        Type    Date
                Month As Integer
                Day As Integer
                Year As Integer
        EndType
        Dim today As Date
        Dim tomorrow As Date
2.      GetDate(today)                          'msg1
3.      PrintDate(today)                        'msg2
```


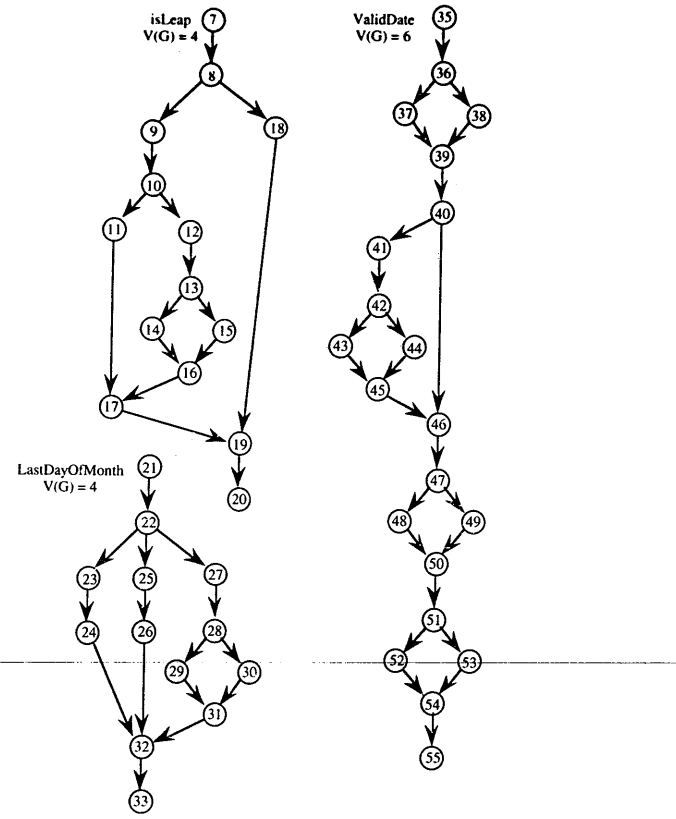

Figure 13.14 Lower level units.

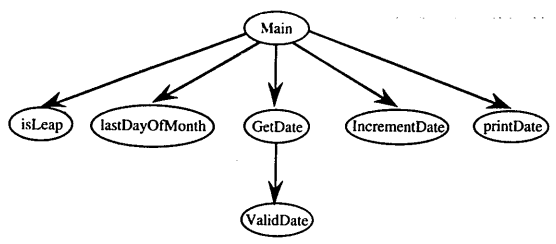

Figure 13.15 Functional decomposition of integration version.

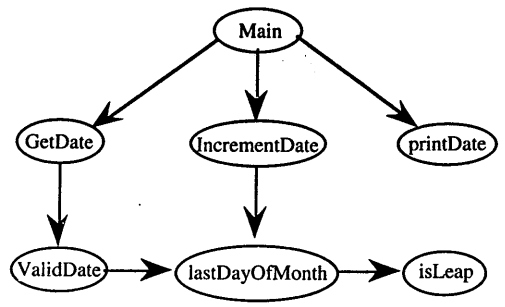

Figure 13.16 Call graph of integration version.

```
4.      tomorrow = IncrementDate(today)                    'msg3
5.      PrintDate(tomorrow)                                    'msg4
6.  End Main

7.  Function isLeap(year)   Boolean
8.      If (year divisible by 4)
9.          Then
10.             If (year is NOT divisible by 100)
11.                 Then isLeap = True
12.                 Else
13.                     If (year is divisible by 400)
14.                         Then isLeap = True
15.                         Else isLeap = False
16.                     EndIf
17.             EndIf
18.         Else isLeap = False
19.     EndIf
20. End (Function isLeap)

21. Function lastDayOfMonth(month, year)     Integer
22.     Case month Of
23.         Case 1: 1, 3, 5, 7, 8, 10, 12
24.             lastDayOfMonth = 31
25.         Case 2: 4, 6, 9, 11
26.             lastDayOfMonth = 30
27.         Case 3: 2
28.             If (isLeap(year))                              'msg5
29.                 Then lastDayOfMonth = 29
30.                 Else lastDayOfMonth = 28
31.             EndIf
32.     EndCase
33. End (Function lastDayOfMonth)

34. Function GetDate(aDate)         Date
        dim aDate As Date
35.     Function ValidDate(aDate)  Boolean     'within scope of GetDate
```



```
        dim aDate As Date
        dim dayOK, monthOK, yearOK As Boolean

36.     If ((aDate.Month > 0) AND (aDate.Month <=12)
37.         Then monthOK = True
38.         Else monthOK = False
39.     EndIf

40.     If (monthOK)
41.         Then
42.             If ((aDate.Day > 0) AND
                        (aDate.Day <= lastDayOfMonth(aDate.Month, aDate.Year))'msg6
43.                 Then dayOK = True
44.                 Else dayOK = False
45.             EndIf
46.         EndIf

47.  If ((aDate.Year > 1811) AND (aDate.Year <= 2012)
48.         Then yearOK = True
49.         Else yearOK = False
50.         EndIf

51.  If (monthOK AND dayOK AND yearOK)
52.         Then ValidDate = True
53.         Else ValidDate = False
54.     EndIf
55.     End (Function ValidDate)
    ' GetDate body begins here
56.   Do
57.     Output("enter a month")
58.     Input(aDate.Month)
59.     Output("enter a day")
60.     Input(aDate.Day)
61.     Output("enter a year")
62.     Input(aDate.Year)
63.     GetDate.Month = aDate.Month
64.     GetDate.Day = aDate.Day
65.     GetDate.Year = aDate.Year
66.   Until (ValidDate(aDate))                              'msg7
67.   End (Function GetDate)

68. Function IncrementDate(aDate)     Date
69.   If (aDate.Day < lastDayOfMonth(aDate.Month))    'msg8
70.     Then aDate.Day = aDate.Day + 1
71.     Else aDate.Day = 1
72.       If (aDate.Month = 12)
73.         Then aDate.Month = 1
74.           aDate.Year = aDate.Year + 1
75.         Else aDate.Month = aDate.Month + 1
76.       EndIf
77.   EndIf
78. End (IncrementDate)
```

```
79. Procedure PrintDate(aDate)
80.     Output( "Day is ", aDate.Month, "/", aDate.Day, "/", aDate.Year)
81. End (PrintDate)
```

13.5.1 Decomposition Based Integration

The isLeap and lastDayOfMonth Functions are in the first level of decomposition because they must be available to both GetDate and IncrementDate. (We could move isLeap to be contained within the scope of lastDayOfMonth.) Pairwise integration based on the decomposition in Figure 13.15 is problematic; the isLeap and lastDayOfMonth Functions are never directly called by the Main program, so these integration sessions would be empty. Bottom-up pairwise integration starting with isLeap, then lastDayOfMonth, ValidDate, and GetDate would be useful. The pairs involving Main and GetDate, IncrementDate, and PrintDate are all useful (but short) sessions. Building stubs for ValidDate and lastDayOfMonth would be easy.

13.5.2 Call Graph Based Integration

Pairwise integration based on the Call Graph in Figure 13.16 is an improvement over that for the decomposition based pairwise integration. Obviously there are no empty integration sessions because edges refer to actual unit references. There is still the problem of stubs. Sandwich integration is appropriate because this example is so small. In fact, it lends itself to a build sequence. Build 1 could contain Main and PrintDate. Build 2 could contain Main, IncrementDate, lastDayOfMonth, and IncrementDate in addition to the already present PrintDate. Finally, Build 3 would add the remaining units, GetDate and ValidDate.

Neighborhood integration based on the Call Graph would likely proceed with the neighborhoods of ValidDate and lastDayOfMonth. Next, we could integrate the neighborhoods of GetDate and IncrementDate. Finally, we would integrate the neighborhood of Main. Notice that these neighborhoods form a build sequence.

13.5.3 MM-Path Based Integration

Because the program is data-driven, all MM-Paths begin in and return to the main program. Here is the first MM-Path for May 27, 2002 (there are others when the main program calls Print Date and Increment Date).

```
Main (1, 2)
    msg1
    GetDate (34, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66)
        msg7
        validDate (35, 36, 37, 39, 40, 41, 42))
            msg6
            lastDayOfMonth (21, 22, 23, 24, 32, 33)'point of message quiessence
        ValidDate (43, 45, 46, 47, 48, 50, 51, 52, 54, 55)
    GetDate (67
Main (3)
```


Notice that the statement fragment sequences (Figures 13.13 and 13.14) identify full paths from source to sink nodes. This is direct at the point of message quiessence; at the other units, the pair of node sequences must be concatenated to get the full source to sink path. We are now in a strong position to describe how many MM-Paths are sufficient: the set of MM-Paths should cover all source-to-sink paths in the set of units. When loops are present, condensation graphs will reduce result in directed acyclic graphs, thereby resolving the problem of potentially infinite (or excessively large) number of paths.

Reference

Fordahl, Matthew, Elementary Mistake Doomed Mars Probe, The Associated Press, Oct. 1, 1999; also, www.fas.org/mars/991001/~mars01.htm.

Exercises

1. Find the source and sink nodes in ValidatePIN and in GetPIN.
2. Find the module execution paths in ValidatePIN.
3. Here are some other possible complexity metrics for MM-Paths:

   V(G) = e – n
   V(G) = 0.5e – n + 2
   sum of the outdegrees of the nodes
   sum of the nodes plus the sum of the edges

   Make up some examples, try these out, and see if they have any explanatory value.

Chapter 14

System Testing

Of the three levels of testing, the system level is closest to everyday experience. We test many things: a used car before we buy it, an online network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations — not with respect to a specification or a standard. Consequently, the goal is not to find faults, but to demonstrate performance. Because of this, we tend to approach system testing from a functional standpoint instead of from a structural one. Because it is so intuitively familiar, system testing in practice tends to be less formal than it might be; and this is compounded by the reduced testing interval that usually remains before a delivery deadline.

The craftsperson metaphor continues to serve us. We need a better understanding of the medium; as we said in Chapter 12, we will view system testing in terms of threads of system-level behavior. We begin with a new construct — an atomic system function — and further elaboration on the thread concept, highlighting some of the practical problems of thread-based system testing. System testing is closely coupled with requirements specification, therefore, we will discuss how to find threads in common notations. All this leads to an orderly thread-based system testing strategy that exploits the symbiosis between functional and structural testing. We will apply the strategy to our simple automated teller machine (SATM) system.

14.1 Threads

Threads are hard to define; in fact, some published definitions are counterproductive, misleading, or wrong. It is possible to simply treat threads as a primitive concept that needs no formal definition. For now, we will use examples to develop a "shared vision." Here are several views of a thread:

A scenario of normal usage
A system-level test case

A stimulus/response pair
Behavior that results from a sequence of system-level inputs
An interleaved sequence of port input and output events
A sequence of transitions in a state machine description of the system
An interleaved sequence of object messages and method executions
A sequence of machine instructions
A sequence of source instructions
A sequence of MM-paths
A sequence of atomic system functions

Threads have distinct levels. A unit-level thread is usefully understood as an execution-time path of source instructions or, alternatively, as a path of DD-Paths. An integration-level thread is an MM-Path — that is, an alternating sequence of module executions and messages. If we continue this pattern, a system-level thread is a sequence of atomic system functions (to be defined shortly). Because atomic system functions have port events as their inputs and outputs, a sequence of atomic system functions implies an interleaved sequence of port input and output events. The end result is that threads provide a unifying view of our three levels of testing. Unit testing tests individual functions; integration testing examines interactions among units; and system testing examines interactions among atomic system functions. In this chapter, we focus on system-level threads and answer some fundamental questions, such as, "How big is a thread? Where do we find them? How do we test them?"

14.1.1 Thread Possibilities

Defining the endpoints of a system-level thread is a bit awkward. We motivate a tidy, graph theory-based definition by working backward from where we want to go with threads. Here are four candidate threads in our SATM system:

Entry of a digit
Entry of a personal identification number (PIN)
A simple transaction: ATM Card Entry, PIN Entry, select transaction type (deposit, withdraw), present account details (checking or savings, amount), conduct the operation, and report the results
An ATM session containing two or more simple transactions

Digit entry is a good example of a minimal atomic system function. It begins with a port input event (the digit keystroke) and ends with a port output event (the screen digit echo), so it qualifies as a stimulus/response pair. If you go back to our example in Chapter 13, you will see that this atomic system function (ASF) is a subpath of the sample MM-Path that we listed in great detail. This level of granularity is too fine for the purposes of system testing. We saw this to be an appropriate level for integration testing.

The second candidate, PIN Entry, is a good example of an upper limit to integration testing and, at the same time, a starting point of system testing. PIN Entry is a good example of an atomic system function. It is also a good example of a family of stimulus/response pairs (system-level behavior that is initiated by a port input event, traverses some programmed logic, and terminates in one of several possible responses (port output events). As we saw in Chapter 13, PIN Entry entails a sequence of system-level inputs and outputs.

1. A screen requesting PIN digits
2. An interleaved sequence of digit keystrokes and screen responses
3. The possibility of cancellation by the customer before the full PIN is entered
4. A system disposition: A customer has three chances to enter the correct PIN. Once a correct PIN has been entered, the user sees a screen requesting the transaction type; otherwise, a screen advises the customer that the ATM card will not be returned, and no access to ATM functions is provided.

This is clearly in the domain of system-level testing, and several stimulus/response pairs are evident. Other examples of ASFs include Card Entry, Transaction Selection, Provision of Transaction Details, Transaction Reporting, and Session Termination. Each of these is maximal in an integration testing sense and minimal in a system testing sense. That is, we would not want to integration test something larger than an ASF; at the same time, we would not want to system test anything smaller.

The third candidate, the simple transaction, has a sense of "end-to-end" completion. A customer could never execute PIN Entry alone (a Card Entry is needed), but the simple transaction is commonly executed. This is a good example of a system-level thread; note that it involves the interaction of several ASFs.

The last possibility (the session) is actually a sequence of threads. This is also properly a part of system testing; at this level, we are interested in the interactions among threads. Unfortunately, most system testing efforts never reach the level of thread interaction. (More on this in Chapter 15.)

14.1.2 Thread Definitions

We simplify our discussion by defining a new term that helps us get to our desired goal.

Definition

An atomic system function (ASF) is an action that is observable at the system level in terms of port input and output events.

In an event-driven system, ASFs are separated by points of event quiescence; these occur when a system is (nearly) idle, waiting for a port input event to trigger further processing. Event quiescence has an interesting Petri net insight. In a traditional Petri net, deadlock ocurs when no transition is enabled. In an event-driven Petri net, event quiescence is similar to deadlock; but an input event can bring new life to the net. The SATM system exhibits event quiescence in several places: one is the tight loop at the beginning of SATM Main, where the system has displayed the welcome screen and is waiting for a card to be entered into the card slot. Event quiescence is a system-level property; there is an analog at the integration level — message quiescence.

The notion of event quiescence does for ASFs what message quiescence does for MM-Paths: it provides a natural endpoint. An ASF begins with a port input event, traverses parts of one or more MM-Paths, and terminates with a port output event. When viewed from the system level, no compelling reason exists to decompose an ASF into lower levels of detail (hence the atomicity). In the SATM system, digit entry is a good example of an ASF — so are card entry, cash dispensing, and session closing. PIN Entry is probably too big; perhaps we should call it a molecular system function.

Atomic system functions represent the seam between integration and system testing. They are the largest item to be tested by integration testing and the smallest item for system testing. We can test an ASF at both levels. Again, the digit entry ASF is a good example (see the MM-Path example in Chapter 13). During system testing, the port input event is a physical key press that is detected by KeySensor and sent to GetPIN as a string variable. (Notice that KeySensor performs the physical-to-logical transition.) GetPIN determines whether the digit key or the cancel key was pressed and responds accordingly. (Notice that button presses are ignored.) The ASF terminates with either screen 2 or 4 displayed. Instead of requiring system keystrokes and visible screen displays, we could use a driver to provide these and test the digit entry ASF via integration testing.

Interesting ASFs are included in ValidatePIN. This unit controls all screen displays relevant to the PIN entry process. It begins with the display of screen 2 (which asks the customer to enter the PIN). Next, GetPIN is called, and the system is event quiescent until a keystroke occurs. These keystrokes initiate the GetDigit ASFs we just discussed. Here, we find a curious integration fault. Notice that screen 2 is displayed in two places: by the Then clauses in the while-loop in GetPIN and by the first statements in each Case clause in ValidatePIN. We could fix this by removing the screen displays from GetPIN and simply returning the string (e.g., 'X —') to be displayed.

Referring to the pseudocode example in Chapter 13, four ASFs are included in statements 75 through 93: each begins with KeySensor observing a port input event (a keystroke) and ends with a closely knit family of port output events (the calls to ScreenDriver with different PIN echoes). We could name these four ASFs GetDigit1, GetDigit2, GetDigit3, and GetDigit4. They are slightly different because the later ones include the earlier If statements. (This module might be reworked so that the while-loop repeated a single ASF.)

This portion of the SATM system also illustrates the difference between unit and integration testing. When GetPIN is unit tested, its inputs come from KeySensor (which acts like an input statement). The input space of GetPIN contains the digits 0 through 9 and the cancel key. (These would likely be treated as string or character data.) We could add inputs for the function keys B1, B2, and B3; if we did, traditional equivalence class testing would be a good choice. The function we test is whether GetDigit reconstructs the keystrokes into a digit string and whether the Boolean indication for the cancel key is correct.

Definition

Given a system defined in terms of atomic system functions, the ASF Graph of the system is the directed graph in which nodes are ASFs and edges represent sequential flow.

Definition

A source ASF is an atomic system function that appears as a source node in the ASF graph of a system; similarly, a sink ASF is an atomic system function that appears as a sink node in the ASF graph.

In the SATM system, the Card Entry ASF is a source ASF, and the session termination ASF is a sink ASF. Notice that intermediary ASFs could never be tested at the system level by themselves — they need the predecessor ASFs to "get there."

Definition

A system thread is a path from a source ASF to a sink ASF in the ASF graph of a system.

Definition

Given a system defined in terms of system threads, the thread graph of the system is the directed graph in which nodes are system threads and edges represent sequential execution of individual threads.

This set of definitions provides a coherent set of increasingly broader views of threads, starting with very short threads (within a unit) and ending with interactions among system-level threads. We can use these views much like the ocular on a microscope, switching among them to see different levels of granularity. Having these concepts is only part of the problem; supporting them is another. We next take a tester's view of requirements specification to see how to identify threads.

14.2 Basis Concepts for Requirements Specification

Recall the notion of a basis of a vector space: a set of independent elements from which all the elements in the space can be generated. Instead of anticipating all the variations in scores of requirements specification methods, notations, and techniques, we will discuss system testing with respect to a basis set of requirements specification constructs: data, actions, devices, events, and threads (Jorgensen, 1989). Every system can be expressed in terms of these five fundamental concepts (and every requirements specification technique is some combination of these). We examine these fundamental concepts here to see how they support the tester's process of thread identification.

14.2.1 Data

When a system is described in terms of its data, the focus is on the information used and created by the system. We describe data in terms of variables, data structures, fields, records, data stores, and files. Entity/relationship models are the most common choice at the highest level, and some form of a regular expression (e.g., Jackson diagrams or data structure diagrams) is used at a more detailed

level. The data-centered view is also the starting point for several flavors of object-oriented analysis. Data refers to information that is either initialized, stored, updated, or (possibly) destroyed. In the SATM system, initial data describe the various accounts (PANs) and their PINs, and each account has a data structure with information such as the account balance. As ATM transactions occur, the results are kept as created data and used in the daily posting of terminal data to the central bank. For many systems, the data-centered view dominates. These systems are often developed in terms of CRUD actions (Create, Retrieve, Update, Delete). We could describe the transaction portion of the SATM system in this way, but it would not work well for the user interface portion.

Sometimes threads can be identified directly from the data model. Relationships between data entities can be one-to-one, one-to-many, many-to-one, or many-to-many; these distinctions all have implications for threads that process the data. For example, if bank customers can have several accounts, each account needs a unique PIN. If several people can access the same account, they need ATM cards with identical PANs. We can also find initial data (such as PAN, ExpectedPIN pairs) that are read but never written. Such read-only data must be part of the system initialization process. If not, there must be threads that create such data. Read-only data is therefore an indicator of source ASFs.

14.2.2 Actions

Action-centered modeling is still a common requirements specification form. This is an historical outgrowth of the action-centered nature of imperative programming languages. Actions have inputs and outputs, and these can be either data or port events. Here are some methodology-specific synonyms for actions: transform, data transform, control transform, process, activity, task, method, and service. Actions can also be decomposed into lower level actions, as we saw with the data flow diagrams in Chapter 12. The input/output view of actions is exactly the basis of functional testing, and the decomposition (and eventual implementation) of actions is the basis of structural testing.

14.2.3 Devices

Every system has port devices; these are the sources and destinations of system-level inputs and outputs (port events). The slight distinction between ports and port devices is sometimes helpful to testers. Technically, a port is the point at which an I/O device is attached to a system, as in serial and parallel ports, network ports, and telephone ports. Physical actions (keystrokes and light emissions from a screen) occur on port devices, and these are translated from physical to logical (or logical to physical). In the absence of actual port devices, much of system testing can be accomplished by "moving the port boundary inward" to the logical instances of port events. From now on, we will just use the term "port" to refer to port devices. The ports in the SATM system include the digit and cancel keys, the function keys, the display screen, the deposit and withdrawal doors, the card and receipt slots, and several less obvious devices, such as the rollers that move cards and deposit envelopes into the machine, the cash dispenser, the receipt printer, and so on.

Thinking about the ports helps the tester define both the input space that functional system testing needs; similarly, the output devices provide output-based functional test information. (For example, we would like to have enough threads to generate all 15 SATM screens.)

14.2.4 Events

Events are somewhat schizophrenic: they have some characteristics of data and some of actions. An event is a system-level input (or output) that occurs on a port device. Similar to data, events can be inputs to or outputs of actions. Events can be discrete (such as SATM keystrokes) or they can be continuous (such as temperature, altitude, or pressure). Discrete events necessarily have a time duration, and this can be a critical factor in real-time systems. We might picture input events as destructive read-out data, but it is a stretch to imagine output events as destructive write operations.

Events are like actions in the sense that they are the translation point between real-world physical events and internal logical manifestations of these. Port input events are physical-to-logical translations; and, symmetrically, port output events are logical-to-physical translations. System testers should focus on the physical side of events, not the logical side (the focus of integration testers). Situations occur where the context of present data values changes the logical meaning of physical events. In the SATM system, for example, the port input event of depressing button B1 means "Balance" when screen 5 is displayed, "checking" when screen 6 is displayed, and "yes" when screens 10, 11, and 14 are displayed. We refer to such situations as "context-sensitive port events," and we would expect to test such events in each context.

14.2.5 Threads

Unfortunately for testers, threads are the least frequently used of the five fundamental constructs. Because we test threads, it usually falls to the tester to find them in the interactions among the data, events, and actions. About the only place that threads appear per se in a requirements specification is when rapid prototyping is used in conjunction with a scenario recorder. It is easy to find threads in control models, as we will soon see. The problem with this is that control models are just that — they are models, not the reality of a system.

14.2.6 Relationships among Basis Concepts

Figure 14.1 is an entity/relationship (E/R) model of our basis concepts. Notice that all relationships are many-to-many: Data and Events are generalized into an entity; the two relationships to the Action entity are for inputs and outputs. The same event can occur on several ports, and typically many events occur on a single port. Finally, an action can occur in several threads, and a thread is composed of several actions. This diagram demonstrates some of the difficulty of system testing. Testers must use events and threads to ensure that all the many-to-many relationships among the five basis concepts are correct.

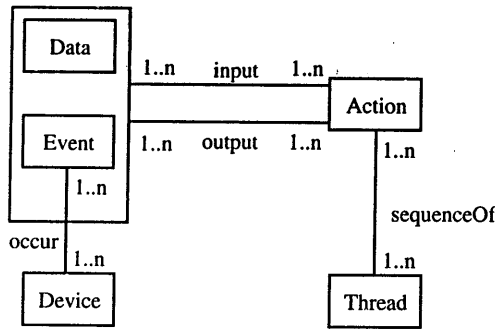

Figure 14.1 E/R model of basis concepts.

14.2.7 Modeling with the Basis Concepts

All flavors of requirements specification develop models of a system in terms of the basis concepts. Figure 14.2 shows three fundamental forms of requirements specification models: structural, contextual, and behavioral. Structural models are used for development; these express the functional decomposition, data decomposition, and the interfaces among components. Contextual models are often the starting point of structural modeling. They emphasize system devices and, to a lesser extent, actions, and threads very indirectly. The models of behavior (also called control models) are where four of the five basis constructs come together. Selection of an appropriate control model is the essence of requirements specification: models that are too weak cannot express important system behaviors, while models that are too powerful typically obscure interesting behaviors. As a general rule, decision tables are a good choice only for computational systems; finite state machines are good for menu-driven systems; and Petri nets are the model of choice for concurrent systems. Here, we use finite state machines for the SATM system, and in Chapter 15, we will use Petri nets to analyze thread interaction.

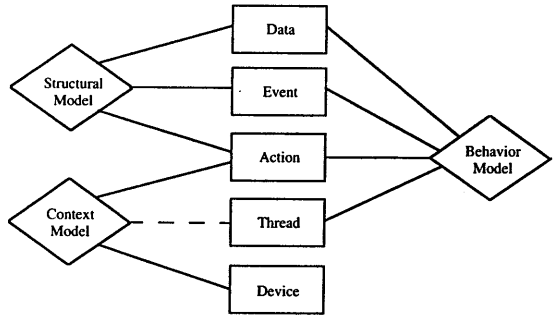

Figure 14.2 Modeling relationships among basic constructs.

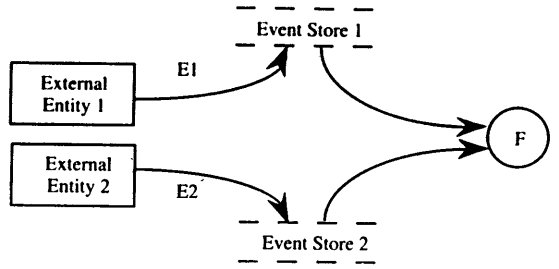

Figure 14.3 Event partitioning view of function F.

We must make an important distinction between a system itself (reality) and models of a system. Consider a system in which some function F cannot occur until two prerequisite events E1 and E2 have occurred, and that they can occur in either order. We could use the notion of event partitioning to model this situation. The result would be a diagram like that in Figure 14.3.

In the event partitioning view, events E1 and E2 occur on their respective external devices. When they occur, they are held in their respective event stores. (An event store acts like a destructive read operation.) When both events have occurred, function F gets its prerequisite information from the event stores. Notice that we cannot tell from the model which event occurs first; we only know that both must occur.

We could also model the system as a finite state machine (FSM) in Figure 14.4, in which states record which event has occurred. The state machine view explicitly shows the two orders of the events.

Both models express the same prerequisites for the function F, and neither is the reality of the system. Of these two models, the state machine is more useful to the tester, because paths are instantly convertible to threads.

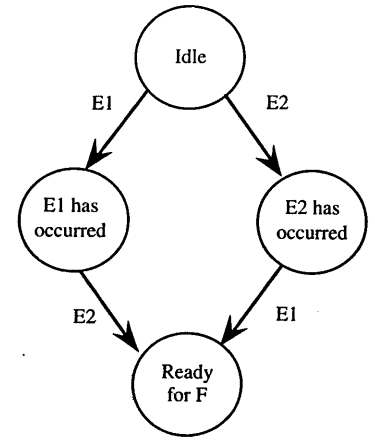

Figure 14.4 FSM for function F.

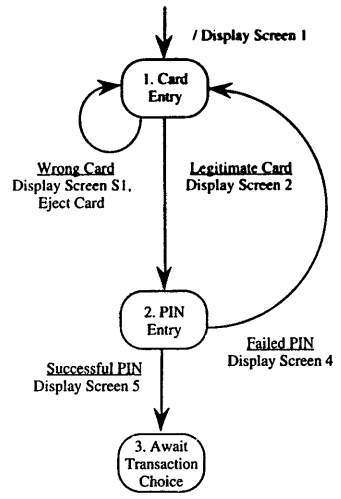

Figure 14.5 Top-level SATM state machine.

14.3 Finding Threads

The finite state machine models of the SATM system are the best place to look for system testing threads. We will start with a hierarchy of state machines; the upper level is shown in Figure 14.5. At this level, states correspond to stages of processing, and transitions are caused by logical (instead of port) events. The Card Entry "state," for example, would be decomposed into lower levels that deal with details like jammed cards, cards that are upside down, stuck card rollers, and checking the card against the list of cards for which service is offered. Once the details of a macro-state are tested, we use an easy thread to get to the next macro-state.

The PIN Entry state is decomposed into the more detailed view in Figure 14.6, which is a slight revision of the version in Chapter 12. The adjacent states are shown because they are sources and destinations of transitions from the PIN Entry portion. At this level, we focus on the PIN Retry mechanism; all of the output events are true port events, but the input events are still logical events. The states and edges are numbered for reference later when we discuss test coverage.

To start the thread identification process, we first list the port events shown on the state transitions; they appear in Table 14.1. We skipped the eject card event because it is not really part of the PIN Entry component.

Notice that Correct PIN and Incorrect PIN are really compound port input events. We cannot actually enter an entire PIN — we enter digits, and at any point, we might hit the cancel key. These more detailed possibilities are shown in Figure 14.7. A truly paranoid tester might decompose the digit port input event into the actual choices (0-pressed, 1-pressed,..., 9-pressed), but this should have been tested at a lower level. The port events in the PIN Try finite state machine are in Table 14.2.

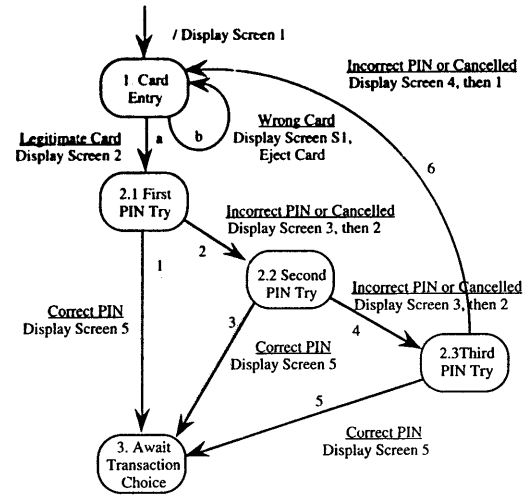

Figure 14.6 PIN Entry finite state machine.

The "x" in the state names in the PIN Try machine refers to which try (first, second, or third) is passing through the machine.

In addition to the true port events in the PIN Try finite state machine, there are three logical output events (Correct PIN, Incorrect PIN, and Canceled); these correspond exactly to the higher level events in Figure 14.6.

The hierarchy of finite state machines multiplies the number of threads. There are 156 distinct paths from the First PIN Try state to the Await Transaction Choice or Card Entry states in Figure 14.6. Of these, 31 correspond to eventually correct PIN entries (1 on the first try, 5 on the second try, and 25 on the third try); the other 125 paths correspond to those with incorrect digits or with cancel keystrokes. This is a fairly typical ratio. The input portion of systems, especially interactive systems, usually has a large number of threads to deal with input errors and exceptions.

It is good form to reach a state machine in which transitions are caused by actual port input events, and the actions on transitions are port output events. If we have such a finite state machine, generating system test cases for these threads is a mechanical process — simply follow a path of transitions and note the port inputs and outputs as they occur along the path. This interleaved sequence is

Table 14.1 Events in the PIN Entry Finite State Machine

Port Input Events	*Port Output Events*
Legitimate card	Display screen 1
Wrong card	Display screen 2
Correct PIN	Display screen 3
Incorrect PIN	Display screen 4
Canceled	Display screen 5

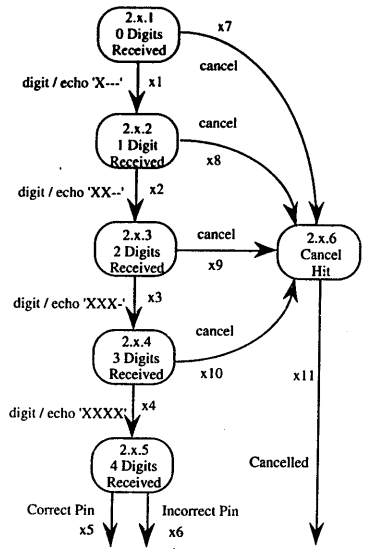

Figure 14.7 PIN Try finite state machine.

Table 14.2 Port Events in the PIN Try Finite State Machine

Port Input Events	*Port Output Events*
Digit	echo 'X- - -'
Cancel	echo 'XX- -'
	echo 'XXX-'
	echo 'XXXX'

performed by the test executor (person or program). Tables 14.3 and 14.4 follow two paths through the hierarchic state machines. Table 14.3 corresponds to a thread in which a PIN is correctly entered on the first try. Table 14.4 corresponds to a thread in which a PIN is incorrectly entered on the first try, cancels after the third digit on the second try, and gets it right on the third try. To make the test case explicit, we assume a precondition that the expected PIN is 1234.

The event in parentheses in the last row of Table 14.3 is the logical event that "bumps up" to the parent state machine and causes a transition there to the Await Transaction Choice state.

If you look closely at Tables 14.3 and 14.4, you will see that the bottom third of Table 14.4 is exactly Table 14.3; thus, a thread can be a subset of another thread.

14.4 Structural Strategies for Thread Testing

Although generating thread test cases is easy, deciding which ones to actually use is more complex. (If you have an automatic test executor, this is not a

Table 14.3 Port Event Sequence for Correct PIN on First Try

Port Input Event	*Port Output Event*
	Screen 2 displayed with '- - - -'
1 pressed	
	Screen 2 displayed with 'X- - -'
2 pressed	
	Screen 2 displayed with 'XX- -'
3 pressed	
	Screen 2 displayed with 'XXX-'
4 pressed	
	Screen 2 displayed with 'XXXX'
(Correct PIN)	Screen 5 displayed

Table 14.4 Port Event Sequence for Correct PIN on Third Try

Port Input Event	*Port Output Event*
	Screen 2 displayed with '- - - -'
1 pressed	
	Screen 2 displayed with 'X- - -'
2 pressed	
	Screen 2 displayed with 'XX- -'
3 pressed	
	Screen 2 displayed with 'XXX-'
5 pressed	
	Screen 2 displayed with 'XXXX'
(Incorrect PIN)	Screen 3 displayed
(Second try)	Screen 2 displayed with '- - - -'
1 pressed	
	Screen 2 displayed with 'X- - -'
2 pressed	
	Screen 2 displayed with 'XX- -'
3 pressed	
	Screen 2 displayed with 'XXX-'
Cancel key pressed	
(End of second try)	Screen 3 displayed
	Screen 2 displayed with '- - - -'
1 pressed	
	Screen 2 displayed with 'X- - -'
2 pressed	
	Screen 2 displayed with 'XX- -'
3 pressed	
	Screen 2 displayed with 'XXX-'
4 pressed	
	Screen 2 displayed with 'XXXX'
(Correct PIN)	Screen 5 displayed

Table 14.5 Thread Paths in the PIN Try FSM

Input Event Sequence	*Path of Transitions*
1234	x1, x2, x3, x4, x5
1235	x1, x2, x3, x4, x6
C	x7, x11
1C	x1, x8, x11
12C	x1, x2, x9, x11
123C	x1, x2, x3, x10, x11

problem.) We have the same path explosion problem at the system level that we had at the unit level. Just as we did there, we can use the directed graph insights to make an intelligent choice of threads to test.

14.4.1 Bottom-up Threads

When we organize state machines in a hierarchy, we can work from the bottom up. Six paths are used in the PIN Try state machine. If we traverse these six, we test for three things: correct recognition and echo of entered digits, response to the cancel keystroke, and matching expected and entered PINs. These paths are described in Table 14.5 as sequences of the transitions in Figure 14.7. A thread that traverses the path is described in terms of its input keystrokes, thus the input sequence 1234 corresponds to the thread described in more detail in Table 14.3 (the cancel keystroke is indicated with a C).

Once this portion is tested, we can go up a level to the PIN Entry machine, where four paths are used. These four are concerned with the three-try mechanism and the sequence of screens presented to the user. In Table 14.6, the paths in the PIN Entry state machine (Figure 14.6) are named as transition sequences.

These threads were identified with the goal of path traversal in mind. Recall from our discussion of structural testing that these goals can be misleading. The assumption is that path traversal uncovers faults, and traversing a variety of paths reduces redundancy. The last path in Table 14.6 illustrates how structural goals can be counterproductive. Hitting the cancel key three times does indeed cause the three-try mechanism to fail and returns the system to the Card Entry state; but it seems like a degenerate thread. A more serious flaw occurs with these threads: we could not really execute them alone because of the hierarchic state machines. What really happens with the 1235 input sequence in Table 14.5? It traverses an interesting path in the PIN Try machine; and then it "returns" to the

Table 14.6 Thread Paths in the PIN Entry FSM

Input Event Sequence	*Path of Transitions*
1234	1
12351234	2, 3
1235C1234	2,4,5
CCC	2, 4, 6

PIN Entry machine where it is seen as a logical event (incorrect PIN), which causes a transition to state 2.2 (Second PIN Try). If no additional keystrokes occur, this machine would remain in state 2.2. We show how to overcome such situations next.

14.4.2 Node and Edge Coverage Metrics

Because the finite state machines are directed graphs, we can use the same test coverage metrics that we applied at the unit level. The hierarchic relationship means that the upper-level machine must treat the lower machine as a procedure that is entered and returned. (Actually, we need to do this for one more level to get to true threads that begin with the Card Entry state.) The two obvious choices are node coverage and edge coverage. Table 14.7 is extended from Table 14.4 to show the node and edge coverage of the three-try thread.

Table 14.7 Node and Edge Traversal of a Thread

Port Input Event	*Port Output Event*	*Nodes*	*Edges*
	Screen 2 displayed with '- - - -'	2.1	a
1 pressed		2.1.1	
	Screen 2 displayed with 'X- - -'		x1
2 pressed		2.1.2	
	Screen 2 displayed with 'XX- -'		x2
3 pressed		2.1.3	
	Screen 2 displayed with 'XXX-'		x3
5 pressed		2.1.4	
	Screen 2 displayed with 'XXXX'		x4
(Incorrect PIN)	Screen 3 displayed	2.1.5, 3	x6, 2
(Second try)	Screen 2 displayed with '- - - -'	2.2	
1 pressed		2.2.1	
	Screen 2 displayed with 'X- - -'		x1
2 pressed		2.2.2	
	Screen 2 displayed with 'XX- -'		x2
3 pressed		2.2.3	
	Screen 2 displayed with 'XXX-'		x3
Cancel pressed		2.2.4	x10
(End of second try)	Screen 3 displayed	2.2.6	x11
	Screen 2 displayed with '- - - -'	2.3	4
1 pressed		2.3.1	
	Screen 2 displayed with 'X- - -'		x1
2 pressed		2.3.2	
	Screen 2 displayed with 'XX- -'		x2
3 pressed		2.3.3	
	Screen 2 displayed with 'XXX-'		x3
4 pressed		2.3.4	
	Screen 2 displayed with 'XXXX'		x4
(Correct PIN)	Screen 5 displayed	2.3.5, 3	x5, 5

Table 14.8 Thread/State Incidence

Input Events	*2.1*	*2.x.1*	*2.x.2*	*2.x.3*	*2.x.4*	*2.x.5*	*2.2.6*	*2.2*	*2.3*	*3*	*1*
1234	x	x	x	x	x	x				x	
12351234	x	x	x	x	x	x		x		x	
C1234	x	x	x	x	x	x	x	x		x	
1C12C1234	x	x	x	x			x	x	x	x	
123C1C1C	x	x	x	x	x		x	x	x		x

Table 14.9 Thread/Transition Incidence

Input Events	*x1*	*x2*	*x3*	*x4*	*x5*	*x6*	*x7*	*x8*	*x9*	*x10*	*x11*	*1*	*2*	*3*	*4*	*5*	*6*
1234	x	x	x	x	x							x					
12351234	x	x	x	x	x	x							x	x			
C1234	x	x	x	x	x		x				x		x	x			
1C12C1234	x	x	x	x	x			x	x		x		x		x	x	
123C1C1C	x	x	x					x		x	x		x		x		x

Node (state) coverage is analogous to statement coverage at the unit level — it is the bare minimum. In the PIN Entry example, we can attain node coverage without ever executing a thread with a correct PIN. If you examine Table 14.8, you will see that two threads (initiated by C1234 and 123C1C1C) traverse all the states in both machines.

Edge (state transition) coverage is a more acceptable standard. If the state machines are "well formed" (transitions in terms of port events), edge coverage also guarantees port event coverage. The threads in Table 14.9 were picked in a structural way to guarantee that the less traveled edges (those caused by cancel keystrokes) are traversed.

14.5 Functional Strategies for Thread Testing

The finite state machine-based approaches to thread identification are clearly useful, but what if no behavioral model exists for a system to be tested? The testing craftsperson has two choices: develop a behavioral model or resort to the system-level analogs of functional testing. Recall that when functional test cases are identified, we use information from the input and output spaces as well as the function itself. We describe functional threads here in terms of coverage metrics that are derived from three of the basis concepts (events, ports, and data).

14.5.1 Event-Based Thread Testing

Consider the space of port input events. Five port input thread coverage metrics are of interest. Attaining these levels of system test coverage requires a set of threads such that:

PI1: each port input event occurs
PI2: common sequences of port input events occur
PI3: each port input event occurs in every "relevant" data context
PI4: for a given context, all "inappropriate" input events occur
PI5: for a given context, all possible input events occur

The PI1 metric is a bare minimum and is inadequate for most systems. PI2 coverage is the most common, and it corresponds to the intuitive view of system testing because it deals with "normal use." It is difficult to quantify, however. What is a common sequence of input events? What is an uncommon one?

The last three metrics are defined in terms of a "context." The best view of a context is that it is a point of event quiescence. In the SATM system, screen displays occur at the points of event quiescence. The PI3 metric deals with context-sensitive port input events. These are physical input events that have logical meanings determined by the context within which they occur. In the SATM system, for example, a keystroke on the B1 function button occurs in five separate contexts (screens displayed) and has three different meanings. The key to this metric is that it is driven by an event in all of its contexts. The PI4 and PI5 metrics are converses: they start with a context and seek a variety of events. The PI4 metric is often used on an informal basis by testers who try to break a system. At a given context, they want to supply unanticipated input events just to see what happens. In the SATM system, for example, what happens if a function button is depressed during the PIN Entry stage? The appropriate events are the digit and cancel keystrokes. The inappropriate input events are the keystrokes on the B1, B2, and B3 buttons.

This is partially a specification problem: we are discussing the difference between prescribed behavior (things that should happen) and proscribed behavior (things that should not happen). Most requirements specifications have a hard time only describing prescribed behavior; it is usually testers who find proscribed behavior. The designer who maintains my local ATM system told me that once someone inserted a fish sandwich in the deposit envelope slot. (Apparently they thought it was a waste receptacle.) At any rate, no one at the bank ever anticipated insertion of a fish sandwich as a port input event. The PI4 and PI5 metrics are usually very effective, but they raise one curious difficulty. How does the tester know what the expected response should be to a proscribed input? Are they simply ignored? Should there be an output warning message? Usually, this is left to the tester's intuition. If time permits, this is a powerful point of feedback to requirements specification. It is also a highly desirable focus for either rapid prototyping or executable specifications.

We can also define two coverage metrics based on port output events:

PO1: each port output event occurs
PO2: each port output event occurs for each cause

PO1 coverage is an acceptable minimum. It is particularly effective when a system has a rich variety of output messages for error conditions. (The SATM system does not.) PO2 coverage is a good goal, but it is hard to quantify; we will revisit this in Chapter 15 when we examine thread interaction. For now, note

that PO2 coverage refers to threads that interact with respect to a port output event. Usually, a given output event only has a small number of causes. In the SATM system, screen 10 might be displayed for three reasons: the terminal might be out of cash, it may be impossible to make a connection with the central bank to get the account balance, or the withdrawal door might be jammed. In practice, some of the most difficult faults found in field trouble reports are those in which an output occurs for an unsuspected cause. Here is one example: My local ATM system (not the SATM) has a screen that informs me that "Your daily withdrawal limit has been reached." This screen should occur when I attempt to withdraw more than $300 in one day. When I see this screen, I used to assume that my wife has made a major withdrawal (thread interaction), so I request a lesser amount. I found out that the ATM also produces this screen when the amount of cash in the dispenser is low. Instead of providing a lot of cash to the first users, the central bank prefers to provide less cash to more users.

14.5.2 Port-Based Thread Testing

Port-based testing is a useful complement to event-based testing. With port-based testing, we ask, for each port, what events can occur at that port. We then seek threads that exercise input ports and output ports with respect to the event lists for each port. (This presumes such event lists have been specified; some requirements specification techniques mandate such lists.) Port-based testing is particularly useful for systems in which the port devices come from external suppliers. The main reason for port-based testing can be seen in the E/R model of the basis constructs (Figure 14.1). The many-to-many relationship between devices and events should be exercised in both directions. Event-based testing covers the one-to-many relationship from events to ports; and, conversely, port-based testing covers the one-to-many relationship from ports to events. The SATM system fails us at this point — no SATM event occurs at more than one port.

14.5.3 Data-Based Thread Testing

Port- and event-based testing work well for systems that are primarily event driven. Such systems are sometimes called "reactive" systems because they react to stimuli (port input events), and often the reaction is in the form of port output events. Reactive systems have two important characteristics: they are long-running (as opposed to the short burst of computation we see in a payroll program) and they maintain a relationship with their environment. Typically, event-driven, reactive systems do not have a very interesting data model (as we see with the SATM system), so data model-based threads are not particularly useful. So, what about conventional systems that are data driven? These systems, described as "static" in Topper (1993), are transformational (instead of reactive); they support transactions on a database. When these systems are specified, the E/R model is dominant and is therefore a fertile source of system testing threads. To attach our discussion to something familiar, we use the E/R model of a simple library system. See Figure 14.8 from Topper (1993).

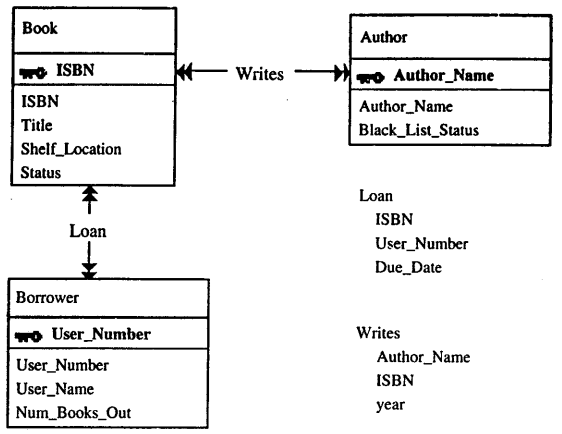

Figure 14.8 E/R model of a library.

Here are some typical transactions in the library system:

1. Add a book to the library.
2. Delete a book from the library.
3. Add a borrower to the library.
4. Delete a borrower from the library.
5. Loan a book to a borrower.
6. Process the return of a book from a borrower.

These transactions are all mainline threads; in fact, they represent families of threads. For example, suppose the book loan transaction is attempted for a borrower whose current number of checked-out books is at the lending limit (a nice boundary value example). We might also try to return a book that was never owned by the library. Here is one more: Suppose we delete a borrower that has some unreturned books. All are interesting threads to test, and all are at the system level.

We can identify each of these examples, and many more, by close attention to the information in the entity/relationship model. As we did with event-based testing, we describe sets of threads in terms of data-based coverage metrics. These refer to relationships for an important reason. Information in relationships is generally populated by system-level threads, whereas that in the entities is usually handled at the unit level. (When E/R modeling is the starting point of object-oriented analysis, this is enforced by encapsulation.)

DM1: Exercise the cardinality of every relationship
DM2: Exercise the participation of every relationship
DM3: Exercise the functional dependencies among relationships

Cardinality refers to the four possibilities of relationship that we discussed in Chapter 3: one-to-one, one-to-many, many-to-one, and many-to-many. In the library example, both the loan and the writes relationships are many-to-many, meaning that one author can write many books, and one book can have many

Table 14.10 SATM Test Data

PAN	*Expected PIN*	*Checking Balance*	*Savings Balance*
100	1234	$1000.00	$800.00
200	4567	$100.00	$90.00
300	6789	$25.00	$20.00

authors; and that one book can be loaned to many borrowers (in sequence), and one borrower can borrow many books. Each of these possibilities results in a useful system testing thread.

Participation refers to whether every instance of an entity participates in a relationship. In the writes relationship, both the Book and the Author entities have mandatory participation (we cannot have a book with no authors, or an author of no books). In some modeling techniques, participation is expressed in terms of numerical limits; the Author entity, for example, might be expressed as "at least 1 and at most 12." When such information is available, it leads directly to obvious boundary value system test threads.

Sometimes, transactions determine explicit logical connections among relationships; these are known as functional dependencies. For example, we cannot loan a book that is not possessed by the library, and we would not delete a book that is out on loan. Also, we would not delete a borrower who still has some books checked out. These kinds of dependencies are reduced when the database is normalized; but they still exist, and they lead to interesting system test threads.

14.6 SATM Test Threads

If we apply the discussion of this chapter to the SATM system, we get a set of threads that constitutes a thorough system-level test. We develop such a set of threads here in terms of an overall state model in which states correspond to key atomic system functions. The macro-level states are: Card Entry, PIN Entry, Transaction Request (and processing), and Session Management. The stated order is the testing order, because these stages are in prerequisite order. (We cannot enter a PIN until successful card entry, we cannot request a transaction until successful PIN entry, and so on.) We also need some precondition data that defines some actual accounts with PANs, Expected PINs, and account balances. These are given in Table 14.10. Two less obvious preconditions are that the ATM terminal is initially displaying screen 1 and the total cash available to the withdrawal dispenser is $500 (in $10 notes).

We will express threads in tables in which pairs of rows correspond to port inputs and expected port outputs at each of the four major stages. We start with three basic threads, one for each transaction type (balance inquiry, deposit, and withdrawal).

Thread 1 (Balance)	*Card Entry (PAN)*	*PIN Entry*	*Transaction Request*	*Session Management*
Port Inputs	100	1234	B1, B1	B2
Port Outputs	Screen 2	Screen 5	Screen 6, screen 14, $1000.00	Screen 15, eject card, screen 1

In thread 1, a valid card with PAN = 100 is entered, which causes screen 2 to be displayed. The PIN digits 1234 are entered; and because they match the expected PIN for the PAN, screen 5 inviting a transaction selection is displayed. When button B1 is touched the first time (requesting a balance inquiry), screen 6 asks which account is displayed. When B1 is pressed the second time (checking), screen 14 is displayed and the checking account balance ($1000.00) is printed on the receipt. When B2 is pushed, screen 15 is displayed, the receipt is printed, the ATM card is ejected, and then screen 1 is displayed.

Thread 2 is a deposit to checking: same PAN and PIN, but B2 is touched when screen 5 is displayed and B1 is touched when screen 6 is displayed. The amount 25.00 is entered when screen 7 is displayed, and then screen 13 is displayed. The deposit door opens and the deposit envelope is placed in the deposit slot. Screen 14 is displayed; and when B2 is pushed, screen 15 is displayed, the receipt showing the new checking account balance of $1025.00 is printed, the ATM card is ejected, and then screen 1 is displayed.

Thread 2 (Deposit)	*Card Entry (PAN)*	*PIN Entry*	*Transaction Request*	*Session Management*
Port Inputs	100	1234	B2, B1, 25.00 insert envelope	B2
Port Outputs	Screen 2	Screen 5	Screen 6, screen 7, screen 13 dep. door opens screen 14 $1025.00	Screen 15, eject card, screen 1

Thread 3 is a withdrawal from savings: again the same PAN and PIN, but B3 is touched when screen 5 is displayed, and B2 is touched when screen 6 is displayed. The amount 30.00 is entered when screen 7 is displayed, and then screen 11 is displayed. The withdrawal door opens and three $10 notes are dispensed. Screen 14 is displayed, and when B2 is pushed, screen 15 is displayed, the receipt showing the new savings account balance of $770.00 is printed, the ATM card is ejected, and then screen 1 is displayed.

Thread 3 (Withdrawal)	*Card Entry (PAN)*	*PIN Entry*	*Transaction Request*	*Session Management*
Port Inputs	100	1234	B3, B2, 30.00	B2
Port Outputs	Screen 2	Screen 5	Screen 6, screen 7, screen 11, withdrawal door opens, 3 $10 notes screen 14 $770.00	Screen 15, eject card, screen 1

A few of these detailed descriptions are needed to show the pattern; the remaining threads are described in terms of input and output events that are the objective of the test thread.

Thread 4 is the shortest thread in the SATM system; it consists of an invalid card, which is immediately rejected.

Thread 4	*Card Entry (PAN)*	*PIN Entry*	*Transaction Request*	*Session Management*
Port Inputs	400			
Port Outputs	Eject card screen 1			

Following the macro-states along thread 1, we next perform variations on PIN Entry. We get four new threads from Table 14.9, which yield edge coverage in the PIN Entry finite state machines.

Thread 5 (Balance)	*Card Entry (PAN)*	*PIN Entry*	*Transaction Request*	*Session Management*
Port Inputs	100	12351234	As in thread 1	
Port Outputs	Screen 2	Screens 3, 2, 5		

Thread 6 (Balance)				
Port Inputs	100	C1234	As in thread 1	
Port Outputs	Screen 2	Screens 3, 2, 5		

Thread 7 (Balance)				
Port Inputs	100	1C12C1234	As in thread 1	
Port Outputs	Screen 2	Screens 3, 2, 3, 2, 5		

Thread 8 (Balance)				
Port Inputs	100	123C1C1C		
Port Outputs	Screen 2	Screens 3, 2, 3, 2, 4, 1		

Moving to the Transaction Request stage, variations exist with respect to the type of transaction (balance, deposit, or withdraw), the account (checking or savings), and several that deal with the amount requested. Threads 1, 2, and 3 cover the type and account variations, so we focus on the amount-driven threads. Thread 9 rejects the attempt to withdraw an amount not in $10 increments, Thread 10 rejects the attempt to withdraw more than the account balance, and thread 11 rejects the attempt to withdraw more cash than the dispenser contains.

Thread 9 (Withdrawal)	*Card Entry (PAN)*	*PIN Entry*	*Transaction Request*	*Session Management*
Port Inputs	100	1234	B3, B2, 15.00 Cancel	B2
Port Outputs	Screen 2	Screen 5	Screens 6, 7, 9, 7	Screen 15, eject card, screen 1

Thread 10 (Withdrawal)				
Port Inputs	300	6789	B3, B2, 50.00 Cancel	B2
Port Outputs	Screen 2	Screen 5	Screens 6, 7, 8	Screen 15, eject card, screen 1

Thread 11 (Withdrawal)				
Port Inputs	100	1234	B3, B2, 510.00 Cancel	B2
Port Outputs	Screen 2	Screen 5	Screens 6, 7, 10	Screen 15, eject card, screen 1

Having exercised the transaction processing portion, we proceed to the session management stage, where we test the multiple transaction option.

Thread 12 (Balance)	*Card Entry (PAN)*	*PIN Entry*	*Transaction Request*	*Session Management*
Port Inputs	100	1234	B1, B1	B1, Cancel
Port Outputs	Screen 2	Screen 5	Screen 6, screen 14 $1000.00	Screen 15, screen 5 screen 15, eject card, screen 1

At this point, the threads provide coverage of all output screens except for screen 12, which informs the user that deposits cannot be processed. Causing this condition is problematic (perhaps we should place a fish sandwich in the deposit envelope slot). This is an example of a thread selected by a precondition that is a hardware failure. We simply give it a thread name here; it is thread 13. Next, we develop threads 14 through 22 to exercise context-sensitive input events. They are shown in Table 14.11; notice that some of the first 13 threads exercise context sensitivity.

These 22 threads comprise a reasonable test of the portion of the SATM system that we have specified. Of course, certain aspects are untested; one good example

Table 14.11 Threads for Context-Sensitive Input Events

Thread	*Keystroke*	*Screen*	*Logical Meaning*
6	Cancel	2	PIN Entry error
14	Cancel	5	Transaction selection error
15	Cancel	6	Account selection error
16	Cancel	7	Amount selection error
17	Cancel	8	Amount selection error
18	Cancel	13	Deposit envelope not ready
1	B1	5	Balance
1	B1	6	Checking
19	B1	10	Yes (a nonwithdrawal transaction)
20	B1	12	Yes (a nondeposit transaction)
12	B1	14	Yes (another transaction)
2	B2	5	Deposit
3	B2	6	Savings
21	B2	10	No (no additional transaction)
22	B2	12	No (no additional transaction)
1	B2	14	No (no additional transaction)

involves the balance of an account. Consider two threads — one that deposits $40 to an account, and a second that withdraws $80 — and suppose that the balance obtained form the central bank at the Card Entry stage is $50. Two possibilities exist: one is to use the central bank balance, record all transactions, and then resolve these when the daily posting occurs. The other is to maintain a running local balance, which is what would be shown on a balance inquiry transaction. If the central bank balance is used, the withdrawal transaction is rejected; but if the local balance is used, it is processed. This detail was not addressed in our specification; we will revisit this when we discuss thread interaction in Chapter 15.

Another prominent untested portion of the SATM system is the Amount Entry process that occurs in screens 7 and 8. The possibility of a cancel keystroke at any point during amount entry produces a multiplicity greater than that of PIN Entry. A more subtle (and therefore more interesting) test for Amount Entry can be used. What actually happens when we enter an amount? To be specific, suppose we wish to enter $40.00. We expect an echo after each digit keystroke, but in which position does the echo occur? Two obvious solutions are available: always require six digits to be entered (so we would enter '004000') or use the high-order digits first and shift left as successive digits are entered, as shown in Figure 14.9. Most ATM systems use the shift approach, and this raises the subtle point: How does the ATM system know when all amount digits have been entered? The ATM system clearly cannot predict that the deposit amount is $40.00 instead of $400.00 or $4000.000 because no "enter" key is used to signify when the last digit has been entered. The reason for this digression is that this is a good example of the kind of detail discovered by testers that is often missing from a requirements specification. (Such details would likely be found with either Rapid Prototyping or using an executable specification.)

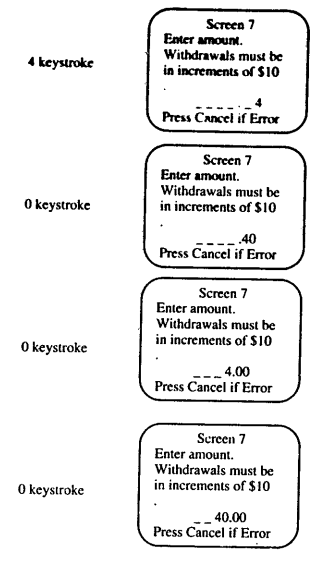

Figure 14.9 Digit echoes with left shifts.

14.7 System Testing Guidelines

If we disallow compound sessions (more than one transaction) and if we disregard the multiplicity due to Amount Entry possibilities, there are 435 distinct threads per valid account in the SATM system. Factor in the effects of compound sessions and the Amount Entry possibilities, and tens of thousands of threads are possible for the SATM system. We end this chapter with three strategies to deal with the thread explosion problem.

14.7.1 Pseudostructural System Testing

When we studied unit testing, we saw that the combination of functional and structural testing yields a desirable cross-check. We have something similar with system-level threads: we defined ten system-level, functional coverage metrics in Section 14.5, and two graph-based metrics (node and edge coverage) in Section 14.4. We can use the graph-based metrics as a cross-check on the functional threads in much the same way that we used DD-Paths at the unit level to identify gaps and redundancies in functional test cases. We can only claim pseudostructural testing (Jorgensen, 1994) because the node and edge coverage metrics are defined in terms of a control model of a system and are not derived directly from the system implementation. (Recall that we started out with a concern over the distinction between reality and models of reality.) In general, behavioral models are only approximations of a system's reality, which is why we could decompose our models down to several levels of detail. If we made a true structural model,

its size and complexity would make it too cumbersome to use. The big weakness of pseudostructural metrics is that the underlying model may be a poor choice. The three most common behavioral models (decision tables, finite state machines, and Petri nets) are appropriate, respectively, to transformational, interactive, and concurrent systems.

Decision tables and finite state machines are good choices for ASF testing. If an ASF is described using a decision table, conditions typically include port input events, and actions are port output events. We can then devise test cases that cover every condition, every action, or, most completely, every rule. As we saw for finite state machine models, test cases can cover every state, every transition, or every path.

Thread testing based on decision tables is cumbersome. We might describe threads as sequences of rules from different decision tables, but this becomes very messy to track in terms of coverage. We need finite state machines as a minimum, and if any form of interaction occurs, Petri nets are a better choice. There, we can devise thread tests that cover every place, every transition, and every sequence of transitions.

14.7.2 Operational Profiles

In its most general form, Zipf's Law holds that 80% of the activities occur in 20% of the space. Activities and space can be interpreted in numerous ways: people with messy desks hardly ever use most of their desktop clutter; programmers seldom use more than 20% of the features of their favorite programming language; and Shakespeare (whose writings contain an enormous vocabulary) uses a small fraction of his vocabulary most of the time. Zipf's Law applies to software (and testing) in several ways. The most useful interpretation for testers is that the space consists of all possible threads, and activities are thread executions (or traversals). Thus, for a system with many threads, 80% of the execution traverses only 20% of the threads.

Recall that a failure occurs when a fault is executed. The whole idea of testing is to execute test cases such that, when a failure occurs, the presence of a fault is revealed. We can make an important distinction: the distribution of faults in a system is only indirectly related to the reliability of the system. The simplest view of system reliability is the probability that no failure occurs during a specific time interval. (Notice that no mention is even made of faults, the number of faults, or fault density.) If the only faults are "in the corners" on threads that are seldom traversed, the overall reliability is higher than if the same number of faults were on "high-traffic" threads. The idea of operational profiles is to determine the execution frequencies of various threads and to use this information to select threads for system testing. Particularly when test time is limited (usually), operational profiles maximize the probability of finding faults by inducing failures in the most frequently traversed threads.

One way to determine the operational profile of a system is to use a decision tree. This works particularly well when system behavior is modeled in hierarchic state machines, as we did with the SATM system. For any state, we find (or estimate) the probability of each outgoing transition (the sum of these must be 1). When a state is decomposed into a lower level, the probabilities at the lower

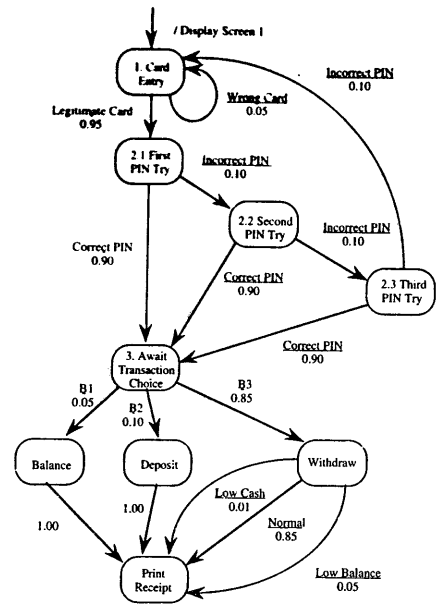

Figure 14.10 Transition probabilities for the SATM system.

level become "split edges" at the upper level. Figure 14.10 shows the result of this with hypothetical transition probabilities. Given the transition probabilities, the overall probability of a thread is simply the product of the transition probabilities along the thread. Table 14.12 shows this calculation for the most and least frequent threads.

Operational profiles provide a feeling for the traffic mix of a delivered system. This is helpful for reasons other than only optimizing system testing. These profiles can also be used in conjunction with simulators to get an early indication of execution time performance and system transaction capacity. Many times, customers are a good source of traffic mix information, so this approach to system testing is often well received simply because it makes an attempt to replicate the reality of a delivered system.

Table 14.12 Thread Probabilities

Common Thread	*Probabilities*	*Rare Thread*	*Probabilities*
Legitimate Card	0.95	Legitimate Card	0.95
PIN OK 1st try	0.90	Bad PIN 1st try	0.10
Withdraw	0.85	Bad PIN 2nd try	0.10
Normal	0.85	PIN OK 3rd try	0.90
		Withdraw	0.85
		Low Cash	0.01
	0.6177375		0.00072675

14.7.3 Progression vs. Regression Testing

When we discussed software development life cycles in Chapter 12, we mentioned that the use of builds induces the need for regression testing. When build 2 is added to build 1, we test the new material in build 2, and we also retest build 1 to see that the new material has no deleterious effect on build 1 contents. (The industrial average for such "ripple effect" is that 20% of changes to an existing system induce new faults in the system.) If a project has several builds, regression testing implies a significant repetition of testing, especially for the early builds. We can reduce this by concentrating on the difference between progression and regression testing.

The most common approach to regression testing is to simply repeat the system tests. We can refine this (and drastically reduce the effort) by choosing test threads with respect to the goals of regression and progression testing. With progression testing, we are testing new territory, so we expect a higher failure rate than with regression testing. Another difference is that because we expect to find more faults with progression testing, we need to be able to locate the faults. This requires test cases with a diagnostic capability — that is, tests that can fail only in a few ways. For thread-based testing, progression testing should use shorter threads that can fail only in a few ways. These threads might be ordered as we did with the SATM thread test set, such that longer threads are built up from shorter (and previously tested) threads.

We have lower expectations of failure with regression testing, and we are less concerned with fault isolation. Taken together, this means regression testing should use longer threads, ones that can fail in several ways. If we think in terms of coverage, both progression and regression testing will have thorough coverage, but the density is different. State and transition coverage matrices (like Tables 14.8 and 14.9) will be sparse for progression testing threads and dense for regression testing threads. This is somewhat antithetical to the use of operational profiles. As a rule, "good" regression testing threads will have low operational frequencies, and progression testing threads will have high operational frequencies.

References

Topper, Andrew et al., *Structured Methods: Merging Models, Techniques, and CASE,* McGraw-Hill, New York, 1993.

Jorgensen, Paul C., An operational common denominator to the structured real-time methods, *Proceedings of the Fifth Structured Techniques Association (STA-5) Conference,* Chicago, May 11, 1989.

Jorgensen, Paul C., System testing with pseudo-structures, *Amer. Programmer,* Vol. 7, No. 4, pp. 29–34, April 1994.

Exercises

1. One of the problems of system testing, particularly interactive systems, is to anticipate all the strange things the user might do. What happens in the SATM system if a customer enters three digits of a PIN and then leaves?

2. To remain "in control" of abnormal user behavior (the behavior is abnormal, not the user), the SATM system might introduce a timer with a 30-second time-out. When no port input event occurs for 30 seconds, the SATM system asks if the user needs more time. The user can answer yes or no. Devise a new screen and identify port events that would implement such a time-out event.
3. Suppose you add this time-out feature to the SATM system. What regression testing would you perform?
4. Make an additional refinement to the PIN Try finite state machine (Figure 14.6) to implement your time-out mechanism, then revise the thread test case in Table 14.3.
5. The text asserts that "the B1 function button occurs in five separate contexts (screens displayed) and has three different meanings." Examine the 15 screens (points of event quiescence) and decide whether a B1 keystroke has three or five different logical meanings.
6. Does it make sense to use test coverage metrics in conjunction with operational profiles? Discuss this.

Chapter 15

Interaction Testing

Faults and failures due to interaction are the bane of testers. Their subtleties make them difficult to recognize and even more difficult to reveal by testing. These are deep faults, ones that remain in a system even after extensive thread testing. Unfortunately, faults of interaction most frequently occur as failures in delivered systems that have been in use for some time. Typically, they have a very low probability of execution, and they occur only after a large number of threads have been executed. Most of this chapter is devoted to describing forms of interaction, not to testing them. As such, it is really more concerned with requirements specification than with testing. The connection is important: knowing how to specify interactions is the first step in detecting and testing for them. This chapter is also a somewhat philosophical and mildly mathematical discussion of faults and failures of interaction; we cannot hope to test something if we do not understand it. We begin with an important addition to our five basic constructs and use this to develop a taxonomy of types of interaction. Next we develop a simple extension to conventional Petri nets that reflects the basic constructs, and then we illustrate the whole discussion with the simple automated teller machine (SATM) and Saturn windshield wiper systems, and sometimes with examples from telephone systems. We conclude by applying the taxonomy to an important application type: client–server systems.

15.1 Context of Interaction

Part of the difficulty of specifying and testing interactions is that they are so common. Think of all the things that interact in everyday life: people, automobile drivers, regulations, chemical compounds, and abstractions, to name just a few. We are concerned with interactions in software-controlled systems (particularly the unexpected ones), so we start by restricting our discussion to interactions among our basic system constructs: actions, data, events, ports, and threads.

One way to establish a context for interaction is to view it as a relationship among the five constructs. If we did this, we would find that the relation

InteractsWith is a reflexive relationship on each entity (data interact with data, actions with other actions, and so on). It also is a binary relationship between data and events, data and threads, and events and threads. The data modeling approach is not a dead end, however. Whenever a data model contains such pervasive relationships, that is a clue that an important entity is missing. If we add some tangible reality to our fairly abstract constructs, we get a more useful framework for our study of interaction. The missing element is location, and location has two components: time and position. Data modeling provides another choice: we can treat location as a sixth basic entity, or as an attribute of the other five. We choose the attribute approach here.

What does it mean for location (time and position) to be an attribute of any of the five basic constructs? This is really a shortcoming of nearly all requirements, specification notations, and techniques. (This is probably also the reason that interactions are seldom recognized and tested.) Information about location is usually created when a system is implemented. Sometimes location is mandated as a requirement — when this happens, the requirement is actually a forced implementation choice. We first clarify the meaning of the components of location: time and position.

We can take two views of time: as an instant or as a duration. The instantaneous view lets us describe when something happens — it is a point when time is an axis. The duration view is an interval on the time axis. When we think about durations, we usually are interested in the length of the time interval, not the endpoints (the start and finish times). Both views are useful. Because threads execute, they have a duration; they also have points in time when they execute. Similar observations apply to events. Often, events have very short durations, and this is problematic if the duration is so short that the event is not recognized by the system.

The position aspect is easier. We could take a very tangible, physical view of position and describe it in terms of some coordinate system. Position can be a three-dimensional Cartesian coordinate system with respect to some origin, or it could be a longitude–latitude–elevation geographic point. For most systems, it is more helpful to slightly abstract position into processor residence. Taken together, time and position tell the tester when and where something happens, and this is essential to understanding interactions.

Before we develop our taxonomy, we need some ground rules about threads and processors. For now, a processor is something that executes threads or a device where events occur.

1. Because threads execute, they have a strictly positive time duration. We usually speak of the execution time of a thread, but we might also be interested in when a thread occurs (executes). Actions are degenerate cases of threads, therefore, actions also have durations.
2. In a single processor, two threads cannot execute simultaneously. This resembles a fundamental precept of physics: no two bodies may occupy the same space at the same time. Sometimes threads appear to be simultaneous, as in time-sharing on a single processor; in fact, time-shared threads are interleaved. Even though threads cannot execute simultaneously on a single processor, events can be simultaneous. (This is really problematic for testers.)
3. Events have a strictly positive time duration. When we consider events to be actions that execute on port devices, this reduces to the first ground rule.
4. Two (or more) input events can occur simultaneously, but an event cannot occurs simultaneously in two (or more) processors. This is immediately clear if we consider port devices to be separate processors.
5. In a single processor, two output events cannot begin simultaneously. This is a direct consequence of output events being caused by thread executions. We need both the instantaneous and duration views of time to fully explain this ground rule. Suppose two output events are such that the duration of one is much greater than the duration of the other. The durations may overlap (because they occur on separate devices), but the start times cannot be identical, as shown in Figure 15.1. An example of this occurs in the SATM system, when a thread causes screen 15 to be displayed and then ejects the ATM card. The screen is still displayed when the card eject event occurs. (This may be a fine distinction; we could also say that port devices are separate processors, and that port output events are really a form of interprocessor communication.)
6. A thread cannot span more than one processor. This convention helps in the definition of threads. By confining a thread to a single processor, we create a natural endpoint for threads; this also results in more simple threads instead of fewer complex threads. In a multiprocessing setting, this choice also results in another form of quiescence — transprocessor quiescence.

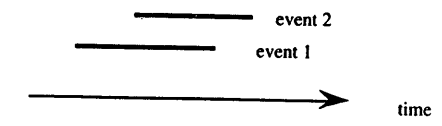

Figure 15.1 Overlapping events.

Taken together, these six ground rules force what we might call "sane behavior" onto the interactions in the taxonomy we define in Section 15.2.

15.2 A Taxonomy of Interactions

The two aspects of location, time and position, form the starting point of a useful taxonomy of interaction. Certain interactions are completely independent of time; for example, two data items that interact exhibit their interaction regardless of time. Certain time-dependent interactions also occur, such as when something is a prerequisite for something else. We will refer to time-independent interactions as static and time-dependent interactions as dynamic. We can refine the static/dynamic dichotomy with the distinction between single and multiple processors. These two considerations yield a two-dimensional plane (as shown in Figure 15.2) with four basic types of interactions:

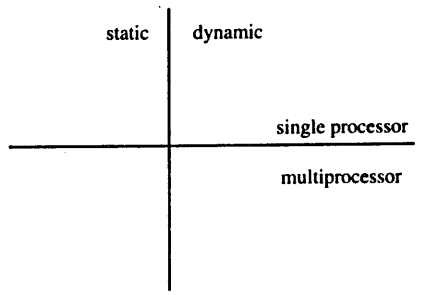

Figure 15.2 Types of interactions.

Static interactions in a single processor
Static interactions in multiple processors
Dynamic interactions in a single processor
Dynamic interactions in multiple processors

We next refine these basic four types using the notion of duration. Threads and events have durations (because they execute), thus, they cannot be static. Data, on the other hand, is static, but we need to be careful here. Consider two examples, the triangle type that corresponds to the triplet (5, 5, 5) of sides, and the balance of a checking account. The triangle type is always equilateral — time will never change geometry, but the balance of a bank account is likely to change in time. If it does, the change is due to the execution of some thread, and this will be a key consideration.

15.2.1 Static Interactions in a Single Processor

Of the five basic constructs, only two have no duration — ports and data. Ports are physical devices, therefore, we can view them as separate processors and thereby simplify our discussion. Port devices interact in physical ways, such as space and power consumption, but this is usually not important to testers. Data items interact in logical ways (as opposed to physical), and these are important to testers. In an informal way, we often speak of corrupt data and of maintaining the integrity of a database. We sometimes get a bit more precise and speak of incompatible or even inconsistent data. We can be very specific if we borrow some terms from Aristotle. (We finally have a chance to use the propositional logic discussed in Chapter 3.) In the following definitions, let p and q be propositions about data items. As examples, we might take p and q to be:

p: AccountBalance = $10.00
q: Sales < $1800.00

Definition

Propositions p and q are:

Contraries if they cannot both be true
Sub-contraries if they cannot both be false
Contradictories if exactly one is true
q is a sub-altern of p if the truth of p guarantees the truth of q

These relationships are known to logicians as the "square of opposition," which is shown in Figure 15.3, where p, q, r, and s are all propositions.

Aristotelian logic seems arcane for software testers, but here are some situations that are exactly characterized by data interactions in the square of opposition:

When the precondition for a thread is a conjunction of data propositions, contrary or contradictory data values will prevent thread execution.
Context-sensitive port input events usually involve contradictory (or at least, contrary) data.
Case statement clauses are contradictories.
Rules in a decision table are contradictories.

Static interactions in a single processor are exactly analogous to combinatorial circuits; they are also well represented by decision tables and unmarked event-driven Petri nets (EDPNs). Features in telephone systems are good examples of interaction (Zave, 1993). One example is the logical conflict between calling party identification service and unlisted directory numbers. With calling party identification, the directory number of the source of a telephone call is provided to the called party. A conflict occurs when a party with an unlisted directory number makes a call to a party with calling party identification. Which takes precedence — the calling party's desire for privacy or the called party's right to know who is placing an incoming call? These two features are contraries: they cannot both be satisfied, but they could both be waived. Call waiting service and data line conditioning comprise another example of contrary features. When a business (or home computing enthusiast) pays for a specially conditioned data line, calls on that line are frequently used for the transmission of formatted binary data. If such a line also has call waiting service, if a call is made to the line that is already in use, a call waiting tone is superimposed onto the preexisting connection. If the connection had been transmitting data, the transmission would be corrupted by the call waiting tone. In this case, the resolution is easier. The customer disables the call waiting service before making data transmission calls.

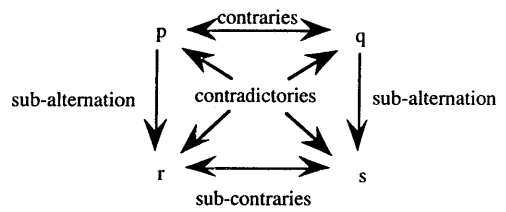

Figure 15.3 The square of opposition.