Chapter 8

Path Testing

The distinguishing characteristic of code-based testing methods is that, as the name implies, they are all based on the source code of the program tested, and not on the specification. Because of this absolute basis, code-based testing methods are very amenable to rigorous definitions, mathematical analysis, and useful 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.

8.1 Program Graphs

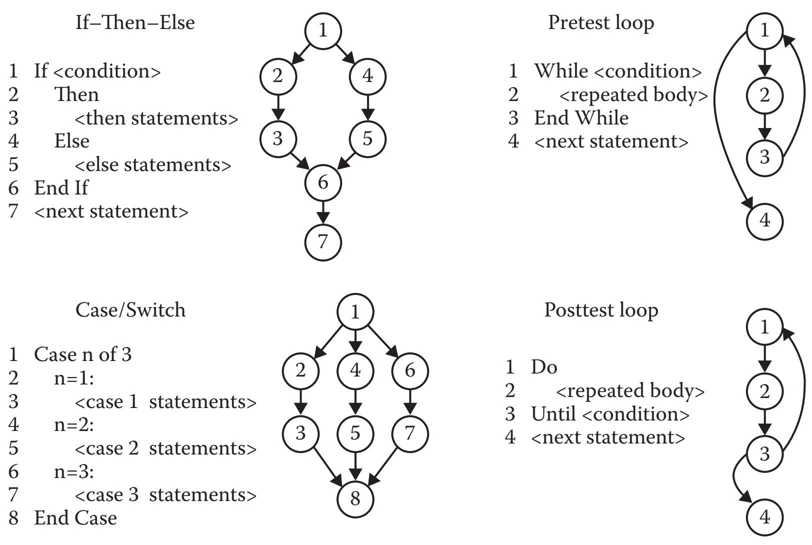
Definition

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* if and only if the statement fragment corresponding to node *j* can be executed immediately after the statement fragment corresponding to node *i*.

8.1.1 Style Choices for Program Graphs

Deriving a program graph from a given program is an easy process. It is illustrated here with four of the basic structured programming constructs (Figure 8.1), and also with our pseudocode implementation of the triangle program from Chapter 2. Line numbers refer to statements and statement fragments. An element of judgment can be used here: sometimes it is convenient to keep



If–Then–Else

```
1  If <condition>
2     Then
3        <then statements>
4     Else
5        <else statements>
6  End If
7  <next statement>
```

Pretest loop

```
1  While <condition>
2     <repeated body>
3  End While
4  <next statement>
```

Case/Switch

```
1  Case n of 3
2     n=1:
3        <case 1  statements>
4     n=2:
5        <case 2  statements>
6     n=3:
7        <case 3  statements>
8  End Case
```

Posttest loop

```
1  Do
2     <repeated body>
3  Until <condition>
4  <next statement>
```

Figure 8.1 Program graphs of four structured programming constructs.

a fragment as a separate node; other times it seems better to include this with another portion of a statement. For example, in Figure 8.2, line 14 could be split into two lines:

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

This latitude collapses onto a unique DD-path graph, so the differences introduced by differing judgments are moot. (A mathematician would make the point that, for a given program, several distinct program graphs might be used, all of which reduce to a unique DD-path graph.) We also need to decide whether to associate nodes with nonexecutable statements such as variable and type declarations; here we do not. A program graph of the second version of the triangle problem (see Chapter 2) is given in Figure 8.2.

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

There are detractors of path-based testing. Figure 8.3 is a graph of a simple (but unstructured!) program; it is typical of the kind of example detractors use to show the (practical) impossibility of completely testing even simple programs. (This example first appeared in Schach [1993].) In this program, five paths lead from node B to node F in the interior of the loop. If the loop may have up to 18 repetitions, some 4.77 trillion distinct program execution paths exist. (Actually, it

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

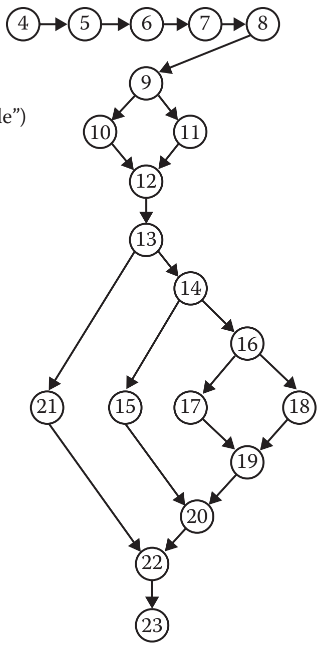


Figure 8.2 Program graph of triangle program.

is 4,768,371,582,030 paths.) The detractor's argument is a good example of the logical fallacy of extension—take a situation, extend it to an extreme, show that the extreme supports your point, and then apply it back to the original question. The detractors miss the point of code-based testing—later in this chapter, we will see how this enormous number can be reduced, with good reasons, to a more manageable size.

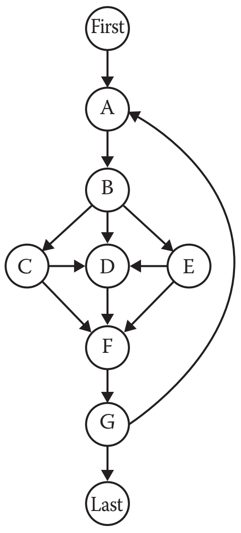


Figure 8.3 Trillions of paths.

8.2 DD-Paths

The best-known form of code-based testing is based on a construct known as a decision-to-decision path (DD-path) (Miller, 1977). The name refers to a sequence of statements that, in Miller's words, begins with the "outway" of a decision statement and ends with the "inway" of the next decision statement. No internal branches occur in such a sequence, so the corresponding code is like a row of dominoes lined up so that when the first falls, all the rest in the sequence fall. Miller's original definition works well for second-generation languages like FORTRAN II, because decision-making statements (such as arithmetic IFs and DO loops) use statement labels to refer to target statements. With modern languages (e.g., Pascal, Ada®, C, Visual Basic, Java), the notion of statement fragments resolves the difficulty of applying Miller's original definition. Otherwise, we end up with program graphs in which some statements are members of more than one DD-path. In the ISTQB literature, and also in Great Britain, the DD-path concept is known as a "linear code sequence and jump" and is abbreviated by the acronym LCSAJ. Same idea, longer name.

We will define DD-paths in terms of paths of nodes in a program graph. In graph theory, these paths are called chains, where a chain is a path in which the initial and terminal nodes are distinct, and every interior node has indegree = 1 and outdegree = 1. (See Chapter 4 for a formal definition.) Notice that the initial node is 2-connected to every other node in the chain, and no instances of 1- or 3-connected nodes occur, as shown in Figure 8.4. The length (number of edges) of the chain in Figure 8.4 is 6.

Definition

A *DD-path* is a sequence of nodes in a program graph such that

Case 1: It consists of a single node with indeg = 0.
Case 2: It consists of a single node with outdeg = 0.
Case 3: It consists of a single node with indeg ≥ 2 or outdeg ≥ 2.
Case 4: It consists of a single node with indeg = 1 and outdeg = 1.
Case 5: It is a maximal chain of length ≥ 1.

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

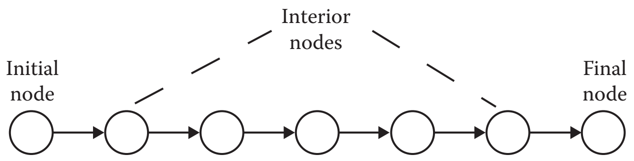


Figure 8.4 Chain of nodes in a directed graph.

Definition

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

This is a complex definition, so we will apply it to the program graph in Figure 8.2. Node 4 is a case 1 DD-path; we will call it "first." Similarly, node 23 is a case 2 DD-path, and we will call it "last." Nodes 5 through 8 are case 5 DD-paths. We know that node 8 is the last node in this DD-path because it is the last node that preserves the 2-connectedness property of the chain. If we go beyond node 8 to include node 9, we violate the indegree = outdegree = 1 criterion of a chain. If we stop at node 7, we violate the "maximal" criterion. Nodes 10, 11, 15, 17, 18, and 21 are case 4 DD-paths. Nodes 9, 12, 13, 14, 16, 19, 20, and 22 are case 3 DD-paths. Finally, node 23 is a case 2 DD-path. All this is summarized in Figure 8.5.

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

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

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

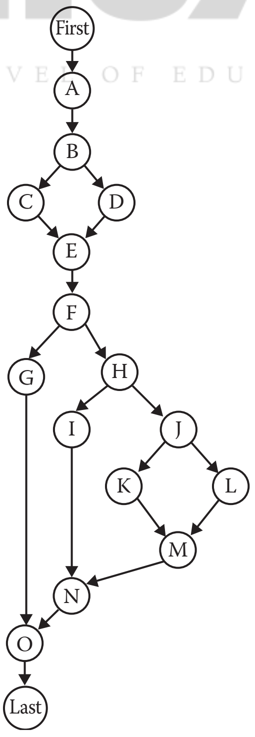


Figure 8.5 DD-path graph for triangle program.

DD-path graphs for programs up to about 100 source lines. Beyond that, most testers look for a tool.

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.

8.3 Test Coverage Metrics

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

8.3.1 Program Graph–Based Coverage Metrics

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

Definition

Given a set of test cases for a program, they constitute *node coverage* if, when executed on the program, every node in the program graph is traversed. Denote this level of coverage as G_{node}, where the G stands for program graph.

Since nodes correspond to statement fragments, this guarantees that every statement fragment is executed by some test case. If we are careful about defining statement fragment nodes, this also guarantees that statement fragments that are outcomes of a decision-making statement are executed.

Definition

Given a set of test cases for a program, they constitute *edge coverage* if, when executed on the program, every edge in the program graph is traversed. Denote this level of coverage as G_{edge}.

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

Definition

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

The G_{chain} coverage is the same as node coverage in the DD-path graph that corresponds to the given program graph. Since DD-paths are important in E.F. Miller's original formulation of test covers (defined in Section 8.3.2), we now have a clear connection between purely program graph constructs and Miller's test covers.

Definition

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

This coverage is open to severe limitations when there are loops in a program (as in Figure 8.3). E.F. Miller partially anticipated this when he postulated the C_2 metric for loop coverage. Referring back to Chapter 4, observe that every loop in a program graph represents a set of strongly (3-connected) nodes. To deal with the size implications of loops, we simply exercise every loop, and then form the condensation graph of the original program graph, which must be a directed acyclic graph.

8.3.2 E.F. Miller's Coverage Metrics

Several widely accepted test coverage metrics are used; most of those in Table 8.1 are due to the early work of Miller (1977). Having an organized view of the extent to which a program is tested makes it possible to sensibly manage the testing process. Most quality organizations now expect the C_1 metric (DD-path coverage) as the minimum acceptable level of test coverage.

These coverage metrics form a lattice (see Chapter 9 for a lattice of data flow coverage metrics) in which some are equivalent and some are implied by others. The importance of the lattice is that there are always fault types that can be revealed at one level and can escape detection by inferior levels of testing. Miller (1991) observes that when DD-path coverage is attained by a set of test cases, roughly 85% of all faults are revealed. The test coverage metrics in Table 8.1 tell us what to test but not how to test it. In this section, we take a closer look at techniques that exercise source code. We must keep an important distinction in mind: Miller's test coverage metrics are based on program graphs in which nodes are full statements, whereas our formulation allows statement fragments (which can be entire statements) to be nodes.

8.3.2.1 Statement Testing

Because our formulation of program graphs allows statement fragments to be individual nodes, Miller's C_0 metric is subsumed by our G_{node} metric.

Table 8.1 Miller's Test Coverage Metrics

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

Statement coverage is generally viewed as the bare minimum. If some statements have not been executed by the set of test cases, there is clearly a severe gap in the test coverage. Although less adequate than DD-path coverage, the statement coverage metric (C_0) is still widely accepted: it is mandated by ANSI (American National Standards Institute) Standard 187B and has been used successfully throughout IBM since the mid-1970s.

8.3.2.2 DD-Path Testing

When every DD-path is traversed (the C_1 metric), we know that each predicate outcome has been executed; this amounts to traversing every edge in the DD-path graph (or program graph). Therefore, the C_1 metric is exactly our G_{chain} metric.

For if–then and if–then–else statements, this means that both the true and the false branches are covered (C_{1p} coverage). For CASE statements, each clause is covered. Beyond this, it is useful to ask how we might test a DD-path. Longer DD-paths generally represent complex computations, which we can rightly consider as individual functions. For such DD-paths, it may be appropriate to apply a number of functional tests, especially those for boundary and special values.

8.3.2.3 Simple Loop Coverage

The C_2 metric requires DD-path coverage (the C_1 metric) plus loop testing.

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). Notice that this is equivalent to the G_{edge} test coverage.

8.3.2.4 Predicate Outcome Testing

This level of testing requires that every outcome of a decision (predicate) must be exercised. Because our formulation of program graphs allows statement fragments to be individual nodes, Miller's C_{1p} metric is subsumed by our G_{edge} metric. Neither E.F. Miller's test covers nor the graph-based covers deal with decisions that are made on compound conditions. They are the subjects of Section 8.3.3.

8.3.2.5 Dependent Pairs of DD-Paths

Identification of dependencies must be made at the code level. This cannot be done just by considering program graphs. The C_d metric foreshadows the topic of Chapter 9—data flow testing. The most common dependency among pairs of DD-paths is the define/reference relationship, in which a variable is defined (receives a value) in one DD-path and is referenced in another DD-path. The importance of these dependencies is that they are closely related to the problem of infeasible paths. We have good examples of dependent pairs of DD-paths: in Figure 8.5, C and H are such a pair, as are DD-paths D and H. The variable IsATriangle is set to TRUE at node C, and FALSE at node D. Node H is the branch taken when IsATriangle is TRUE win the condition at node F. Any path containing nodes D and H is infeasible. Simple DD-path coverage might not exercise these dependencies; thus, a deeper class of faults would not be revealed.

8.3.2.6 Complex Loop Coverage

Miller's C_{ik} metric extends the loop coverage metric to include full paths from source to sink nodes that contain loops.

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, 1984): concatenated, nested, and horrible, shown in Figure 8.6.

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

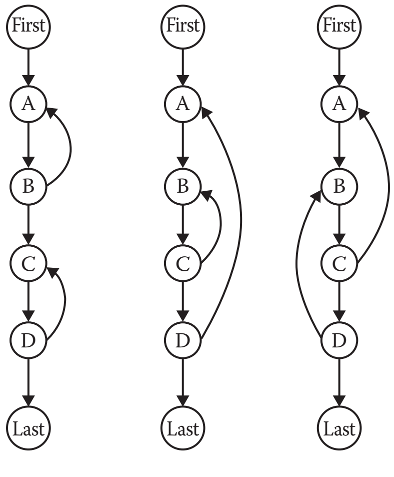


Figure 8.6 Concatenated, nested, and knotted loops.

are internal to other loops, the result is Beizer's knotted loop. 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 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 data flow methods discussed in Chapter 9. As a preview, consider the infinite loop that could occur if one loop tampers with the value of the other loop's index.

8.3.2.7 Multiple Condition Coverage

Miller's C_{MCC} metric addresses the question of testing decisions made by compound conditions. Look closely at the compound conditions in DD-paths B and H. 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 decision table; a compound condition of three simple conditions will have eight rules (see Table 8.2), 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 tradeoff: statement complexity versus path complexity. Multiple condition coverage assures that this complexity is not swept under the DD-path coverage rug. This metric has been refined to Modified Condition Decision Coverage (MCDC), defined in Section 8.3.3.

8.3.2.8 "Statistically Significant" Coverage

The C_{stat} metric is awkward—what constitutes a statistically significant set of full program paths? Maybe this refers to a comfort level on the part of the customer/user.

8.3.2.9 All Possible Paths Coverage

The subscript in Miller's C_{∞} metric says it all—this can be enormous for programs with loops, *a la* Figure 8.3. This can make sense for programs without loops, and also for programs for which loop testing reduces the program graph to its condensation graph.

8.3.3 A Closer Look at Compound Conditions

There is an excellent reference (Chilenski, 2001) that is 214 pages long and is available on the Web. The definitions in this subsection are derived from this reference. They will be related to the definitions in Sections 8.3.1 and 8.3.2.

8.3.3.1 Boolean Expression (per Chilenski)

"A *Boolean expression* evaluates to one of two possible (Boolean) outcomes traditionally known as False and True."

A Boolean expression may be a simple Boolean variable, or a compound expression containing one or more Boolean operators. Chilenski clarifies Boolean operators into four categories:

Operator Type	*Boolean Operators*
Unary (single operand)	NOT(~),
Binary (two operands)	AND(∧), OR(∨), XOR(⊕)
Short circuit operators	AND (AND–THEN), OR (OR–ELSE)
Relational operators	=, ≠, <, ≤, >, ≥

In mathematical logic, Boolean expressions are known as *logical expressions*, where a logical expression can be

1. A simple proposition that contains no logical connective
2. A compound proposition that contains at least one logical connective

Synonyms: *predicate, proposition, condition*.

In programming languages, Chilenski's Boolean expressions appear as conditions in decision making statements: If–Then, If–Then–Else, If–ElseIf, Case/Switch, For, While, and Until loops. This subsection is concerned with the testing needed for compound conditions. Compound conditions are shown as single nodes in a program graph; hence, the complexity they introduce is obscured.

8.3.3.2 Condition (per Chilenski)

"A *condition* is an operand of a Boolean operator (Boolean functions, objects and operators).

Generally this refers to the lowest level conditions (i.e., those operands that are not Boolean operators themselves), which are normally the leaves of an expression tree. Note that a condition is a Boolean (sub)expression."

In mathematical logic, Chilenski's conditions are known as simple, or atomic, propositions. Propositions can be simple or compound, where a compound proposition contains at least one logical connective. Propositions are also called predicates, the term that E.F. Miller uses.

8.3.3.3 Coupled Conditions (per Chilenski)

Two (or more) conditions are *coupled* if changing one also changes the other(s).

When conditions are coupled, it may not be possible to vary individual conditions, because the coupled condition(s) might also change. Chelinski notes that conditions can be strongly or weakly coupled. In a strongly coupled pair, changing one condition always changes the other. In a weakly coupled triplet, changing one condition may change one other coupled condition, but not the third one. Chelinski offers these examples:

In (((x = 0) AND A) OR ((x ≠ 0) AND B)), the conditions (x = 0) and (x ≠ 0) are strongly coupled.

In ((x = 1) OR (x = 2) OR (x = 3)), the three conditions are weakly coupled.

8.3.3.4 Masking Conditions (per Chilenski)

"The process *masking conditions* involves of setting the one operand of an operator to a value such that changing the other operand of that operator does not change the value of the operator.

Referring to Chapter 3.4.3, masking uses the Domination Laws. For an AND operator, masking of one operand can be achieved by holding the other operand False.

(X AND False = False AND X = False no matter what the value of X is.)

For an OR operator, masking of one operand can be achieved by holding the other operand True.

(X OR True = True OR X = True no matter what the value of X is.)."

8.3.3.5 Modified Condition Decision Coverage

MCDC is required for "Level A" software by testing standard DO-178B. MCDC has three variations: Masking MCDC, Unique-Cause MCDC, and Unique-Cause + Masking MCDC. These are explained in exhaustive detail in Chilenski (2001), which concludes that Masking MCDC, while demonstrably the weakest form of the three, is recommended for compliance with DO-178B. The definitions below are quoted from Chilenski.

Definition

MCDC requires

1. Every statement must be executed at least once.
2. Every program entry point and exit point must be invoked at least once.
3. All possible outcomes of every control statement are taken at least once.
4. Every nonconstant Boolean expression has been evaluated to both true and false outcomes.
5. Every nonconstant condition in a Boolean expression has been evaluated to both true and false outcomes.
6. Every nonconstant condition in a Boolean expression has been shown to independently affect the outcomes (of the expression).

The basic definition of MCDC needs some explanation. Control statements are those that make decisions, such as If statements, Case/Switch statements, and looping statements. In a program graph, control statements have an outdegree greater than 1. Constant Boolean expressions are those that always evaluate to the same end value. For example, the Boolean expression ($p \vee \sim p$) always evaluates to True, as does the condition ($a = a$). Similarly, ($p \wedge \sim p$) and ($a \neq a$) are constant expressions (that evaluate to False). In terms of program graphs, MCDC requirements 1 and 2 translate to node coverage, and MCDC requirements 3 and 4 translate to edge coverage. MCDC requirements 5 and 6 get to the complex part of MCDC testing. In the following, the three variations discussed by Chilenski are intended to clarify the meaning of point 6 of the general definition, namely, the exact meaning of "independence."

Definition (per Chilenski)

"*Unique-Cause MCDC* [requires] a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions."

Definition (per Chilenski)

"*Unique-Cause + Masking MCDC* [requires] a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed] for that condition only, i.e., all other (uncoupled) conditions will remain fixed."

Definition (per Chilenski)

"*Masking MCDC* allows masking for all conditions, coupled and uncoupled (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed] for that condition only (i.e., all other (uncoupled) conditions will remain fixed)."

Chilenski comments: "In the case of strongly coupled conditions, no coverage set is possible as DO-178B provides no guidance on how such conditions should be covered."

8.3.4 Examples

The examples in this section are directed at the variations of testing code with compound conditions.

8.3.4.1 Condition with Two Simple Conditions

Consider the program fragment in Figure 8.7. It is deceptively simple, with a cyclomatic complexity of 2.

The decision table (see Chapter 7) for the condition (a AND (b OR c)) is in Table 8.2. Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rules 3 and 4 provide decision coverage, as do rules 1 and 8. Condition coverage is attained by exercising a set of rules such that each condition is evaluated to both true and false. Test cases corresponding to rules 1 and 8 provide decision coverage, as do rules 4 and 5.

```
1. If (a AND (b OR c))
2.     Then  y = 1
3.     Else  y = 2
4. EndIf
```

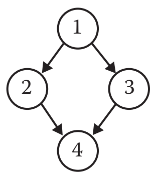


Figure 8.7 Compound condition and its program graph.

Table 8.2 Decision Table for Example Program Fragment

Conditions	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*	*Rule 8*
a	T	T	T	T	F	F	F	F
b	T	T	F	F	T	T	F	F
c	T	F	T	F	T	F	T	F
a AND (b OR c)	True	True	True	False	False	False	False	False
Actions								
y = 1	x	x	x	—	—	—	—	—
y = 2	—	—	—	x	x	x	x	x

To attain MCDC, each condition must be evaluated to both true and false while holding the other conditions constant, and the change must be visible at the outcome. Rules 1 and 5 toggle condition a; rules 2 and 4 toggle condition b; and rules 3 and 4 toggle condition c.

In the Chelinski (2001) paper (p. 9), it happens that the Boolean expression used is

(a AND (b OR c))

In its expanded form, (a AND b) OR (a AND c), the Boolean variable *a* cannot be subjected to unique cause MCDC testing because it appears in both AND expressions.

Given all the complexities here (see Chelinski [2001] for much, much more) the best practical solution is to just make a decision table of the actual code, and look for impossible rules. Any dependencies will typically generate an impossible rule.

8.3.4.2 Compound Condition from NextDate

In our continuing NextDate problem, suppose we have some code checking for valid inputs of the day, month, and year variables. A code fragment for this and its program graph are in Figure 8.8. Table 8.3 is a decision table for the NextDate code fragment. Since the day, month, and year variables are all independent, each can be either true or false. The cyclomatic complexity of the program graph in Figure 8.8 is 5.

Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rule 1 and any one of rules 2–8 provide decision coverage.

Multiple condition coverage requires exercising a set of rules such that each condition is evaluated to both True and False. The eight test cases corresponding to all eight rules are necessary to provide decision coverage.

To attain MCDC, each condition must be evaluated to both true and false while holding the other conditions constant, and the change must be visible at the outcome. Rules 1 and 2 toggle condition yearOK; rules 1 and 3 toggle condition monthOK, and rules 1 and 5 toggle condition dayOK.

Since the three variables are truly independent, multiple condition coverage will be needed.

```
1  NextDate Fragment
2  Dim day, month, year As Integer
3  Dim dayOK, monthOK, yearOK As Boolean
4  Do
5      Input(day, month, year)
6      If 0 < day < 32
7          Then dayOK = True
8          Else dayOK = False
9      EndIf
10     If 0 < month < 13
11         Then monthOK = True
12         Else   monthOK = False
13     EndIf
14     If 1811 < year < 2013
15         Then yearOK = True
16         Else   yearOK = False
17     EndIf
18 Until (dayOK AND monthOK AND yearOK)
19 End Fragment
```

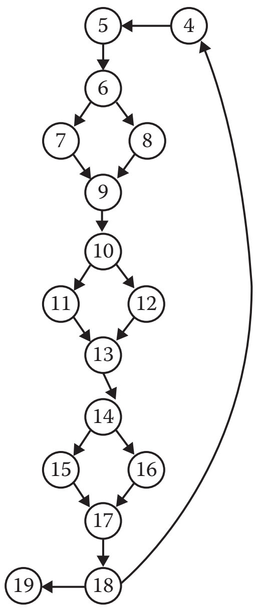


Figure 8.8 NextDate fragment and its program graph.

Table 8.3 Decision Table for NextDate Fragment

Conditions	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*	*Rule 8*
dayOK	T	T	T	T	F	F	F	F
monthOK	T	T	F	F	T	T	F	F
YearOK	T	F	T	F	T	F	T	F
The Until condition	True	False	False	False	False	False	False	False
Actions								
Leave the loop	x	—	—	—	—	—	—	—
Repeat the loop	—	x	x	x	x	x	x	x

8.3.4.3 Compound Condition from the Triangle Program

This example is included to show important differences between it and the first two examples. The code fragment in Figure 8.9 is the part of the triangle program that checks to see if the values of sides a, b, and c constitute a triangle. The test incorporates the definition that each side must be strictly less than the sum of the other two sides. Notice that the program graphs in Figures 8.7 and 8.9 are identical. The NextDate fragment and the triangle program fragment are both functions of three variables. The second difference is that a, b, and c in the triangle program are dependent, whereas dayOK, monthOK, and yearOK in the NextDate fragment are truly independent variables.

```
1.  If (a < b + c) AND (a < b + c) AND (a < b + c)
2.      Then  IsA Triangle = True
3.      Else   IsA Triangle = False
4.  EndIf
```

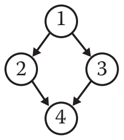


Figure 8.9 Triangle program fragment and its program graph.

The dependence among a, b, and c is the cause of the four impossible rules in the decision table for the fragment in Table 8.4; this is proved next.

Fact: It is numerically impossible to have two of the conditions false.

Proof (by contradiction): Assume any pair of conditions can both be true. Arbitrarily choosing the first two conditions that could both be true, we can write the two inequalities

$$a >= (b + c)$$

$$b >= (a + c)$$

Adding them together, we have

$$(a + b) >= (b + c) + (a + c)$$

and rearranging the right side, we have

$$(a + b) >= (a + b) + 2c$$

But a, b, and c are all > 0, so we have a contradiction. QED.

Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rules 1 and 2 provide decision coverage, as do rules 1 and 3, and rules 1 and 5. Rules, 4, 6, 7, and 8 cannot be used owing to their numerical impossibility.

Condition coverage is attained by exercising a set of rules such that each condition is evaluated to both true and false. Test cases corresponding to rules 1 and 2 toggle the (c < a + b) condition, rules 1 and 3 toggle the (b < a + c) condition, and 1 and 5 toggle the (a < b + c) condition.

MCDC is complicated by the numerical (and hence logical) impossibilities among the three conditions. The three pairs (rules 1 and 2, rules 1 and 3, and rules 1 and 5) constitute MCDC.

Table 8.4 Decision Table for Triangle Program Fragment

Conditions	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*	*Rule 8*
(a < b + c)	T	T	T	T	F	F	F	F
(b < a + c)	T	T	F	F	T	T	F	F
(c < a + b)	T	F	T	F	T	F	T	F
IsATriangle = True	x	—	—	—	—	—	—	—
IsATriangle = False	—	x	x	—	x	—	—	—
Impossible	—	—	—	x		x	x	x

In complex situations such as these examples, falling back on decision tables is an answer that will always work. Rewriting the compound condition with nested If logic, we will have (preserving the original statement numbers)

```
1.1    If (a < b + c)
1.2          Then If (b < a + c)
1.3                Then If (c < a + b)
2                        Then IsATriangle = True
3.1                      Else IsATriangle = False
3.2                     End If
3.3                Else IsATriangle = False
3.4              End If
3.5          Else IsATriangle = False
4      EndIf
```

This code fragment avoids the numerically impossible combinations of a, b, and c. There are four distinct paths through its program graph, and these correspond to rules 1, 2, 3, and 5 in the decision table.

8.3.5 Test Coverage Analyzers

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

8.4 Basis Path Testing

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

8.4.1 McCabe's Basis Path Method

Figure 8.10 is taken from McCabe (1982). It is a directed graph that we might take to be the program graph (or the DD-path graph) of some program. For the convenience of readers who have encountered this example elsewhere (McCabe, 1987; Perry, 1987), the original notation for nodes and edges is repeated here. (Notice that this is not a graph derived from a structured program: nodes B and C are a loop with two exits, and the edge from B to E is a branch into the if–then statement in nodes D, E, and F.) The program does have a single entry (A) and a single exit (G). McCabe based his view of testing on a major result from graph theory, which states that the cyclomatic number (see Chapter 4) of a strongly connected graph is the number of linearly independent circuits in the graph. (A circuit is similar to a chain: no internal loops or decisions occur, but the initial node is the terminal node. A circuit is a set of 3-connected nodes.)

We can always create a strongly connected graph by adding an edge from the (every) sink node to the (every) source node. (Notice that, if the single-entry, single-exit precept is violated, we greatly increase the cyclomatic number because we need to add edges from each sink node to each source node.) The right side of Figure 8.10 shows the result of doing this; it also contains edge labels that are used in the discussion that follows.

Some confusion exists in the literature about the correct formula for cyclomatic complexity. Some sources give the formula as $V(G) = e - n + p$, while others use the formula $V(G) = e - n + 2p$; everyone agrees that e is the number of edges, n is the number of nodes, and p is the number of connected regions. The confusion apparently comes from the transformation of an arbitrary directed graph (such as the one in Figure 8.10, left side) to a strongly connected, directed graph obtained by adding one edge from the sink to the source node (as in Figure 8.10, right side). Adding an edge clearly affects value computed by the formula, but it should not affect the number of circuits. Counting or not counting the added edge accounts for the change to the coefficient of p, the number of connected regions. Since p is usually 1, adding the extra edge means we move from $2p$ to p. Here is a way to resolve the apparent inconsistency. The number of linearly independent paths from the source node to the sink node of the graph on the left side of Figure 8.10 is

$$\begin{aligned} V(G) &= e - n + 2p \\ &= 10 - 7 + 2(1) = 5 \end{aligned}$$

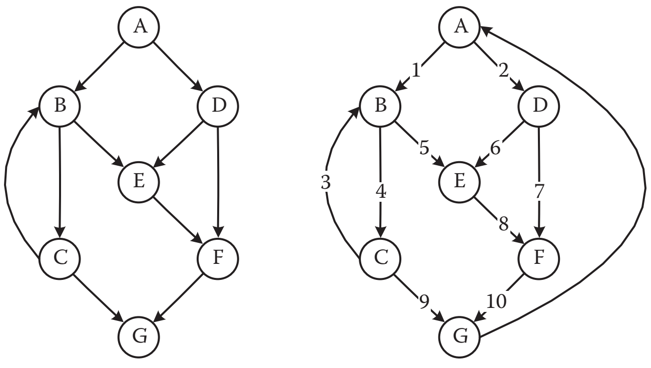


Figure 8.10 McCabe's control graph and derived strongly connected graph.

The number of linearly independent circuits of the graph on the right side of the graph in Figure 8.10 is

$$\begin{aligned} V(G) &= e - n + p \\ &= 11 - 7 + 1 = 5 \end{aligned}$$

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

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

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

We can check the independence of paths p1 – p5 by examining the first five rows of this incidence matrix. The bold entries show edges that appear in exactly one path, so paths p2 – p5 must

Table 8.5 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

Table 8.6 Basis Paths in Figure 8.5

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

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

McCabe next develops an algorithmic procedure (called the baseline method) to determine a set of basis paths. The method begins with the selection of a baseline path, which should correspond to some "normal case" program execution. This can be somewhat arbitrary; McCabe advises choosing a path with as many decision nodes as possible. Next, the baseline path is retraced, and in turn each decision is "flipped"; that is, when a node of outdegree ≥ 2 is reached, a different edge must be taken. Here we follow McCabe's example, in which he first postulates the path through nodes A, B, C, B, E, F, G as the baseline. (This was expressed in terms of paths p1 – p5 earlier.) The first decision node (outdegree ≥ 2) in this path is node A; thus, for the next basis path, we traverse edge 2 instead of edge 1. We get the path A, D, E, F, G, where we retrace nodes E, F, G in path 1 to be as minimally different as possible. For the next path, we can follow the second path, and take the other decision outcome of node D, which gives us the path A, D, F, G. Now, only decision nodes B and C have not been flipped; doing so yields the last two basis paths, A, B, E, F, G and A, B, C, G. Notice that this set of basis paths is distinct from the one in Table 8.6: this is not problematic because a unique basis is not required.

8.4.2 Observations on McCabe's Basis Path Method

If you had trouble following some of the discussion on basis paths and sums and products of these, you may have felt a haunting skepticism—something along the lines of, "Here's another academic oversimplification of a real-world problem." Rightly so, because two major soft spots occur in the McCabe view: one is that testing the set of basis paths is sufficient (it is not), and the other has to do with the yoga-like contortions we went through to make program paths look like a vector space. McCabe's example that the path A, B, C, B, C, B, C, G is the linear combination 2p2 – p1 is very unsatisfactory. What does the 2p2 part mean? Execute path p2 twice? (Yes, according to the math.) Even worse, what does the –p1 part mean? Execute path p1 backward? Undo the most recent execution of p1? Do not do p1 next time? Mathematical sophistries like this are a real turnoff to practitioners looking for solutions to their very real problems. To get a better understanding of these problems, we will go back to the triangle program example.

Start with the DD-path graph of the triangle program in Figure 8.5. 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 8.5). Now, if we flip the decision at node B, we get path p2. Continuing the procedure, we flip the decision at node F, which yields the path p3. Now, we continue to flip decision nodes in the baseline path p1; the next node with outdegree = 2 is node H. When we flip node H, we get the path p4. Next, we flip node J to get p5. We know we are done because there are only five basis paths; they are shown in Table 8.5.

Time for a reality check: if you follow paths p2 and p3, you find that they are both infeasible. Path p2 is infeasible because passing through node D means the sides are not a triangle; so the outcome of the decision at node F must be node G. Similarly, in p3, passing through node C means the sides do form a triangle; so node G cannot be traversed. Paths p4 and p5 are both feasible and correspond to equilateral and isosceles triangles, respectively. Notice that we do not have a basis path for the NotATriangle case.

Recall that dependencies in the input data domain caused difficulties for boundary value testing and that we resolved these by going to decision table-based specification-based testing, where we addressed data dependencies in the decision table. Here, we are dealing with code-level dependencies, which are absolutely incompatible with the latent assumption that basis paths are independent. McCabe's procedure successfully identifies basis paths that are topologically independent; however, when these contradict semantic dependencies, topologically possible paths are seen to be logically infeasible. One solution to this problem is to always require that flipping a decision results in a semantically feasible path. Another is to reason about logical dependencies. If we think about this problem, we can identify two rules:

If node C is traversed, then we must traverse node H.
If node D is traversed, then we must traverse node G.

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

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

The triangle problem is atypical in that no loops occur. The program has only eight topologically possible paths; and of these, only the four basis paths listed above are feasible. Thus, for this special case, we arrive at the same test cases as we did with special value testing and output range testing.

For a more positive observation, basis path coverage guarantees DD-path coverage: the process of flipping decisions guarantees that every decision outcome is traversed, which is the same as DD-path coverage. We see this by example from the incidence matrix description of basis paths and in our triangle program feasible basis paths. We could push this a step further and observe that the set of DD-paths acts like a basis because any program path can be expressed as a linear combination of DD-paths.

8.4.3 Essential Complexity

Part of McCabe's work on cyclomatic complexity does more to improve programming than testing. In this section, we take a quick look at this elegant blend of graph theory, structured programming, and the implications these have for testing. This whole package centers on the notion of essential complexity (McCabe, 1982), which is only the cyclomatic complexity of yet another form of condensation graph. Recall that condensation graphs are a way of simplifying an existing graph; thus far, our simplifications have been based on removing either strong components or DD-paths. Here, we condense around the structured programming constructs, which are repeated as Figure 8.11.

The basic idea is to look for the graph of one of the structured programming constructs, collapse it into a single node, and repeat until no more structured programming constructs can be found. This process is followed in Figure 8.12, which starts with the DD-path graph of the pseudocode triangle program. The if–then–else construct involving nodes B, C, D, and E is condensed into node a, and then the three if–then constructs are condensed onto nodes b, c, and d. The remaining if–then–else (which corresponds to the IF IsATriangle statement) is condensed into node e, resulting in a condensed graph with cyclomatic complexity $V(G) = 1$. In general, when a program is well structured (i.e., is composed solely of the structured programming constructs), it can always be reduced to a graph with one path.

The graph in Figure 8.10 cannot be reduced in this way (try it!). The loop with nodes B and C cannot be condensed because of the edge from B to E. Similarly, nodes D, E, and F look like an if–then construct, but the edge from B to E violates the structure. McCabe (1976) went on to find elemental "unstructures" that violate the precepts of structured programming. These are shown in Figure 8.13. Each of these violations contains three distinct paths, as opposed to the two paths present in the corresponding structured programming constructs;

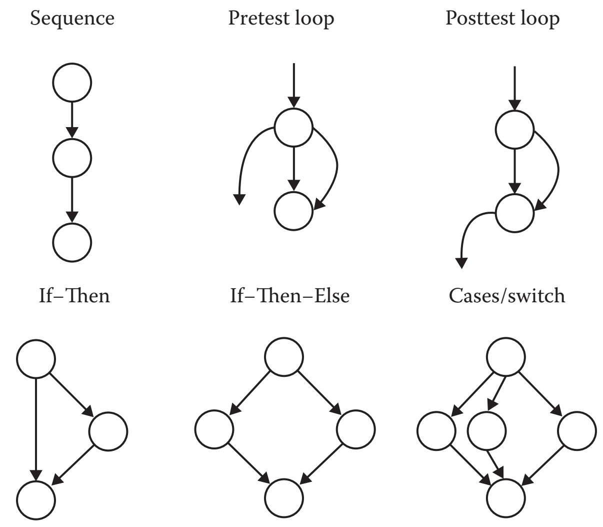


Figure 8.11 Structured programming constructs.

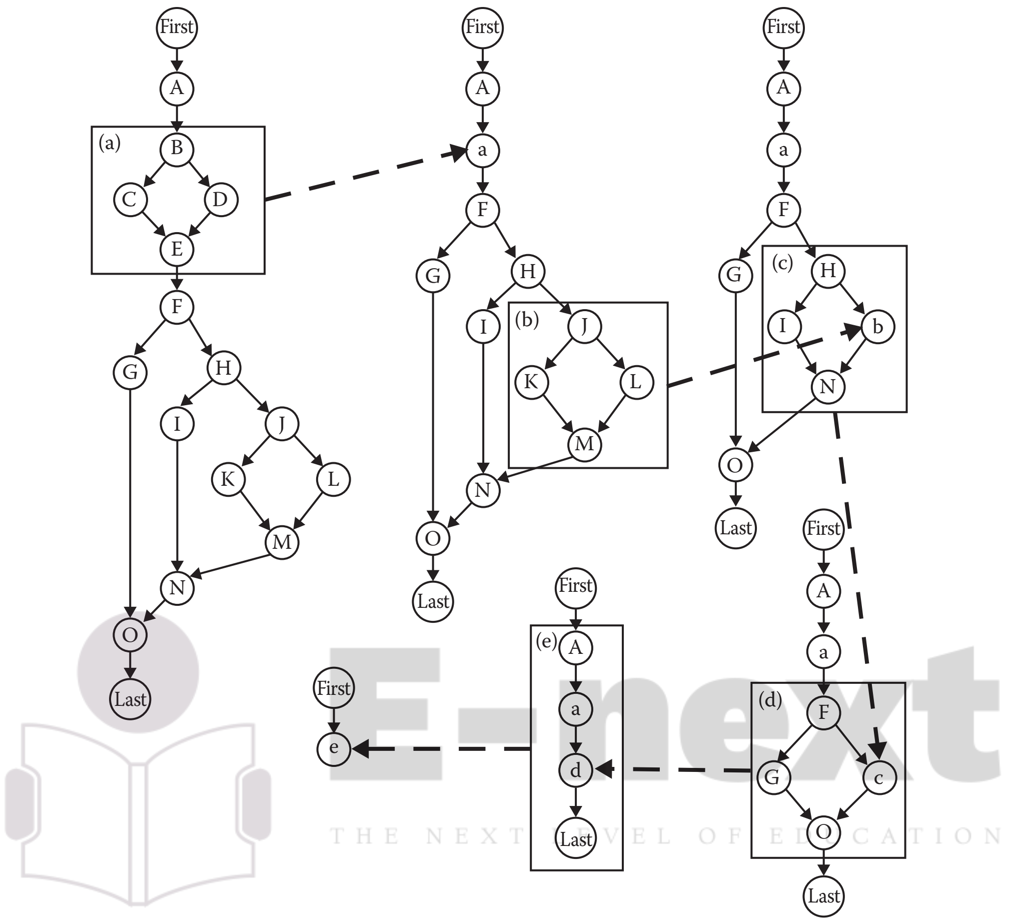


Figure 8.12 Condensing with respect to structured programming constructs.

so one conclusion is that such violations increase cyclomatic complexity. The *pièce de resistance* of McCabe's analysis is that these violations cannot occur by themselves: if one occurs in a program, there must be at least one more, so a program cannot be only slightly unstructured. Because these increase cyclomatic complexity, the minimum number of test cases is thereby increased. In the next chapter, we will see that the violations have interesting implications for data flow testing.

The bottom line for testers is this: programs with high cyclomatic complexity require more testing. Of the organizations that use the cyclomatic complexity metric, most set some guideline for maximum acceptable complexity; $V(G) = 10$ is a common choice. What happens if a unit has a higher complexity? Two possibilities: either simplify the unit or plan to do more testing. If the unit is well structured, its essential complexity is 1; so it can be simplified easily. If the unit has an essential complexity greater than 1, often the best choice is to eliminate the violations.

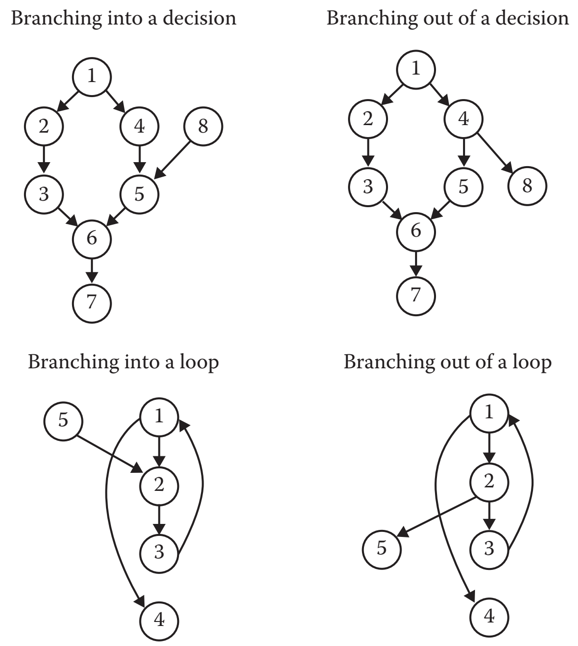


Figure 8.13 Violations of structured programming constructs.

8.5 Guidelines and Observations

In our study of specification-based testing, we observed that gaps and redundancies can both exist and, at the same time, cannot be recognized. The problem was that specification-based testing removes us too far from the code. The path testing approaches to code-based testing represent the case where the pendulum has swung too far the other way: moving from code to directed graph representations and program path formulations obscures important information that is present in the code, in particular the distinction between feasible and infeasible paths. Also, no form of code-based testing can reveal missing functionality that is specified in the requirements. In the next chapter, we look at data flow-based testing. These techniques move closer to the code, so the pendulum will swing back from the path analysis extreme.

McCabe (1982) was partly right when he observed, “It is important to understand that these are purely criteria that measure the quality of testing, and not a procedure to identify test cases.” He was referring to the DD-path coverage metric and his basis path heuristic based on cyclomatic complexity metric. Basis path testing therefore gives us a lower boundary on how much testing is necessary.

Path-based testing also provides us with a set of metrics that act as crosschecks on specification-based testing. We can use these metrics to resolve the gaps and redundancies question. When we find that the same program path is traversed by several functional test cases, we suspect that this redundancy is not revealing new faults. When we fail to attain DD-path coverage, we know that there are gaps in the functional test cases. As an example, suppose we have a program that contains extensive error handling, and we test it with boundary value test cases (min, min+, nom, max–, and max). Because these are all permissible values, DD-paths corresponding to the error-handling code will not be traversed. If we add test cases derived from robustness testing or traditional equivalence class testing, the DD-path coverage will improve. Beyond this rather obvious use of coverage metrics, an opportunity exists for real testing craftsmanship. Any of the coverage metrics in Section 8.3 can operate in two ways: either as a blanket-mandated standard (e.g., all units shall be tested to attain full DD-path coverage) or as a mechanism to selectively test portions of code more rigorously than others. We might choose multiple-condition coverage for modules with complex logic, while those with extensive iteration might be tested in terms of the loop coverage techniques. This is probably the best view of structural testing: use the properties of the source code to identify appropriate coverage metrics, and then use these as a crosscheck on functional test cases. When the desired coverage is not attained, follow interesting paths to identify additional (special value) test cases.

EXERCISES

1. Find the cyclomatic complexity of the graph in Figure 8.3.
2. Identify a set of basis paths for the graph in Figure 8.3.
3. Discuss McCabe's concept of "flipping" for nodes with outdegree ≥ 3.
4. Suppose we take Figure 8.3 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 codeBasedTesting.xls Excel spreadsheet at the CRC website (www.crcpress.com/product/isbn/9781466560680) contains instrumented VBA implementations of the triangle, NextDate, and commission problems that you may have analyzed with the specBasedTesting.xls spreadsheet. The output shows the DD-path coverage of individual test cases and an indication of any faults revealed by a failing test case. Experiment with various sets of test cases to see if you can devise a set of test cases that has full DD-path coverage yet does not reveal the known faults.
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 $z \in V$, and for all scalars k, l, 0, and 1:
   a. If $x, y \in V$, the vector $x + y \in V$.
   b. $x + y = y + x$.
   c. $(x + y) + z = x + (y + z)$.
   d. There is a vector $0 \in V$ such that $x + 0 = x$.

e. For any $x \in V$, there is a vector $-x \in V$ such that $x + (-x) = 0$.
f. For any $x \in V$, the vector $kx \in V$, where k is a scalar constant.
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?

References

Beizer, B., *Software Testing Techniques*, Van Nostrand, New York, 1984.
Chilenski, J.J., *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*, DOT/FAA/AR-01/18, April 2001.
http://www.faa.gov/about/office_org/headquarters_offices/ang/offices/tc/library/ (see actlibrary.tc.faa.gov).
Huang, J.C., Detection of dataflow anomaly through program instrumentation, *IEEE Transactions on Software Engineering*, Vol. SE-5, 1979, pp. 226–236.
Miller, E.F. Jr., *Tutorial: Program Testing Techniques*, COMPSAC '77, IEEE Computer Society, 1977.
Miller, E.F. Jr., Automated software testing: a technical perspective, *American Programmer*, Vol. 4, No. 4, April 1991, pp. 38–43.
McCabe, T. J., A complexity metric, *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308–320.
McCabe, T.J., *Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, National Bureau of Standards (Now NIST)*, Special Publication 500-99, Washington, DC, 1982.
McCabe, T.J., *Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*, McCabe and Associates, Baltimore, 1987.
Perry, W.E., *A Structured Approach to Systems Testing*, QED Information Systems, Inc., Wellesley, MA, 1987.
Schach, S.R., *Software Engineering*, 2nd ed., Richard D. Irwin, Inc. and Aksen Associates, Inc., Homewood, IL, 1993.