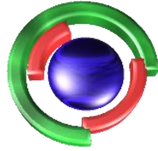
FUTURE VISION BIE

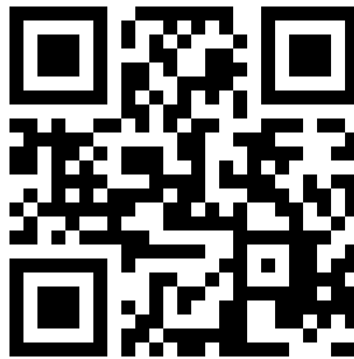
One Stop for All Study Materials
& Lab Programs



Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

**Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...**

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Software Testing

A Craftsman's Approach

Fourth Edition

https://hemanthrajhemu.github.io

7 Decision Table–Based Testing....117
7.1 Decision Tables....117
7.2 Decision Table Techniques....118
7.3 Test Cases for the Triangle Problem....122
7.4 Test Cases for the NextDate Function....123
7.4.1 First Try....123
7.4.2 Second Try....124
7.4.3 Third Try....126
7.5 Test Cases for the Commission Problem....127
7.6 Cause-and-Effect Graphing....128
7.7 Guidelines and Observations....130
References....131

8 Path Testing....133
8.1 Program Graphs....133
8.1.1 Style Choices for Program Graphs....133
8.2 DD-Paths....136
8.3 Test Coverage Metrics....138
8.3.1 Program Graph–Based Coverage Metrics....138
8.3.2 E.F. Miller's Coverage Metrics....139
8.3.2.1 Statement Testing....139
8.3.2.2 DD-Path Testing....140
8.3.2.3 Simple Loop Coverage....140
8.3.2.4 Predicate Outcome Testing....140
8.3.2.5 Dependent Pairs of DD-Paths....141
8.3.2.6 Complex Loop Coverage....141
8.3.2.7 Multiple Condition Coverage....142
8.3.2.8 "Statistically Significant" Coverage....142
8.3.2.9 All Possible Paths Coverage....142
8.3.3 A Closer Look at Compound Conditions....142
8.3.3.1 Boolean Expression (per Chilenski)....142
8.3.3.2 Condition (per Chilenski)....143
8.3.3.3 Coupled Conditions (per Chilenski)....143
8.3.3.4 Masking Conditions (per Chilenski)....144
8.3.3.5 Modified Condition Decision Coverage....144
8.3.4 Examples....145
8.3.4.1 Condition with Two Simple Conditions....145
8.3.4.2 Compound Condition from NextDate....146
8.3.4.3 Compound Condition from the Triangle Program....147
8.3.5 Test Coverage Analyzers....149
8.4 Basis Path Testing....149
8.4.1 McCabe's Basis Path Method....150
8.4.2 Observations on McCabe's Basis Path Method....152
8.4.3 Essential Complexity....154
8.5 Guidelines and Observations....156
References....158

https://hemanthrajhemu.github.io

9 Data Flow Testing 159
9.1 Define/Use Testing 160
9.1.1 Example 161
9.1.2 Du-paths for Stocks 164
9.1.3 Du-paths for Locks 164
9.1.4 Du-paths for totalLocks 168
9.1.5 Du-paths for Sales 169
9.1.6 Du-paths for Commission 170
9.1.7 Define/Use Test Coverage Metrics 170
9.1.8 Define/Use Testing for Object-Oriented Code 172
9.2 Slice-Based Testing 172
9.2.1 Example 175
9.2.2 Style and Technique 179
9.2.3 Slice Splicing 181
9.3 Program Slicing Tools 182
References 183

10 Retrospective on Unit Testing 185
10.1 The Test Method Pendulum 186
10.2 Traversing the Pendulum 188
10.3 Evaluating Test Methods 193
10.4 Insurance Premium Case Study 195
10.4.1 Specification-Based Testing 195
10.4.2 Code-Based Testing 199
10.4.2.1 Path-Based Testing 199
10.4.2.2 Data Flow Testing 200
10.4.2.3 Slice Testing 201
10.5 Guidelines 202
References 203

PART III BEYOND UNIT TESTING

11 Life Cycle–Based Testing 207
11.1 Traditional Waterfall Testing 207
11.1.1 Waterfall Testing 209
11.1.2 Pros and Cons of the Waterfall Model 209
11.2 Testing in Iterative Life Cycles 210
11.2.1 Waterfall Spin-Offs 210
11.2.2 Specification-Based Life Cycle Models 212
11.3 Agile Testing 214
11.3.1 Extreme Programming 215
11.3.2 Test-Driven Development 215
11.3.3 Scrum 216
11.4 Agile Model–Driven Development 218
11.4.1 Agile Model–Driven Development 218
11.4.2 Model–Driven Agile Development 218
References 219

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

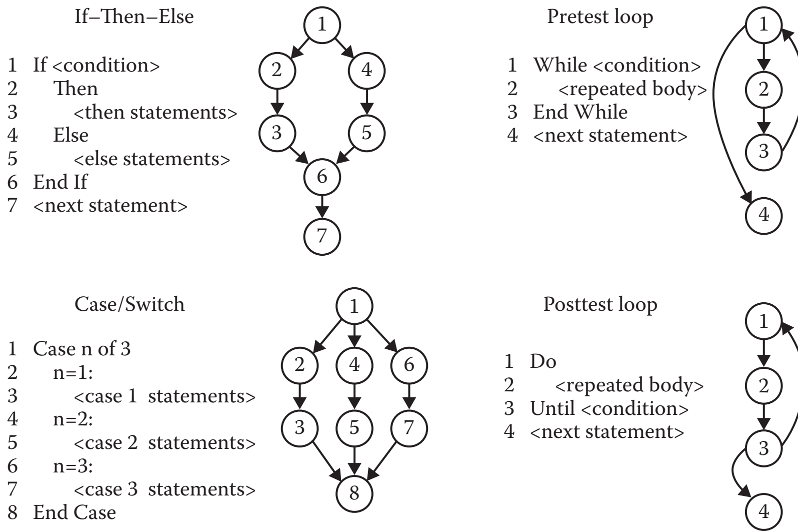


Figure 8.1 Program graphs of four structured programming constructs.

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

https://hemanthrajhemu.github.io

```
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
```

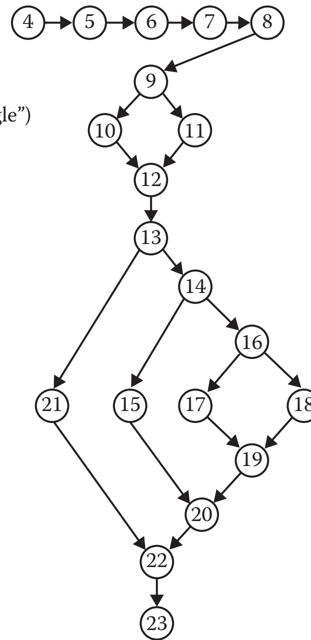


Figure 8.2 Program graph of triangle program.

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.

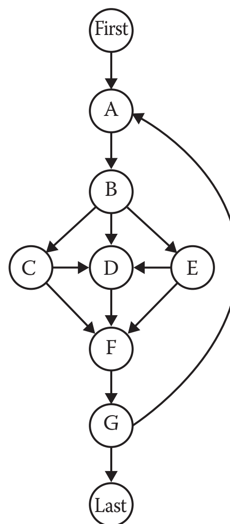


Figure 8.3 Trillions of paths.

https://hemanthrajhemu.github.io

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.

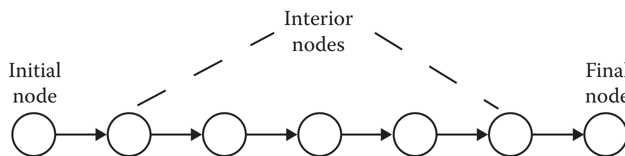


Figure 8.4 Chain of nodes in a directed graph.

https://hemanthrajhemu.github.io

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

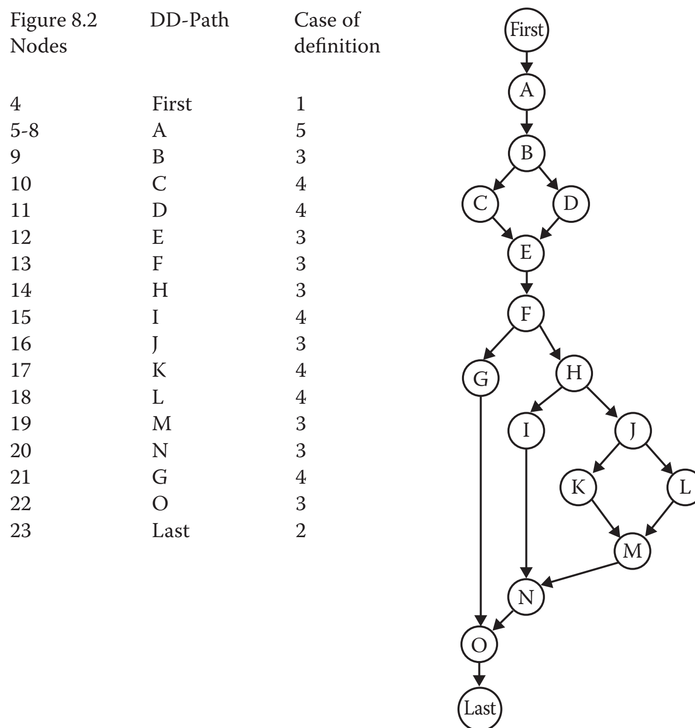


Figure 8.2 Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2

Figure 8.5 DD-path graph for triangle program.

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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,

https://hemanthrajhemu.github.io

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

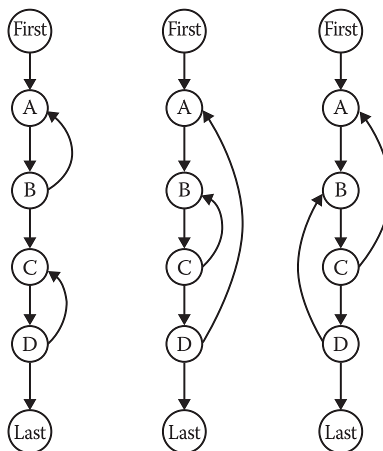


Figure 8.6 Concatenated, nested, and knotted loops.

https://hemanthrajhemu.github.io

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."

https://hemanthrajhemu.github.io

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 (($x \neq 0$) AND B)), the conditions ($x = 0$) and ($x \neq 0$) are strongly coupled.

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

https://hemanthrajhemu.github.io

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 ($p \lor \sim p$) always evaluates to True, as does the condition ($a = a$). Similarly, ($p \land \sim p$) 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."

https://hemanthrajhemu.github.io

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
```

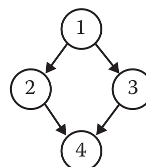


Figure 8.7 Compound condition and its program graph.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

```
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
```

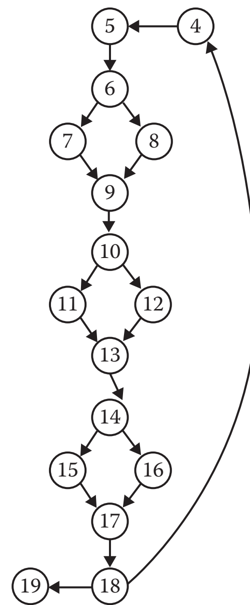


Figure 8.8 NextDate fragment and its program graph.

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.

https://hemanthrajhemu.github.io

```
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
```

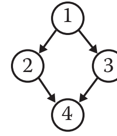


Figure 8.9 Triangle program fragment and its program graph.

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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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

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

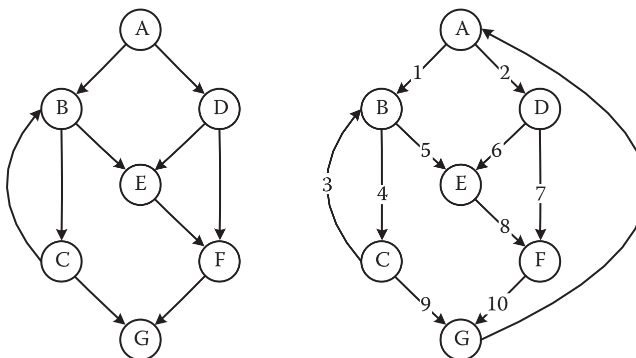


Figure 8.10 McCabe's control graph and derived strongly connected graph.

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

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;

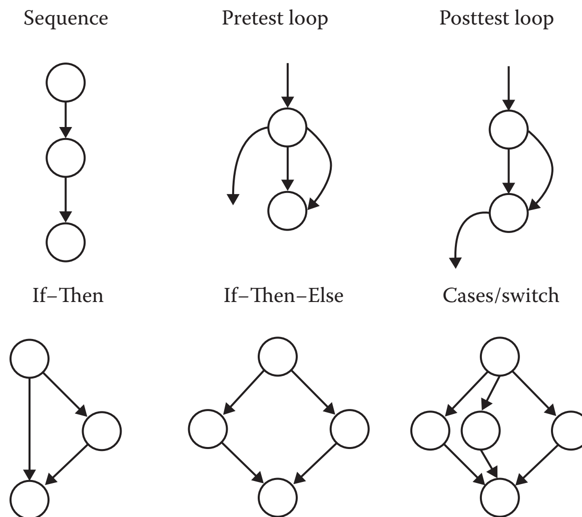


Figure 8.11 Structured programming constructs.

https://hemanthrajhemu.github.io

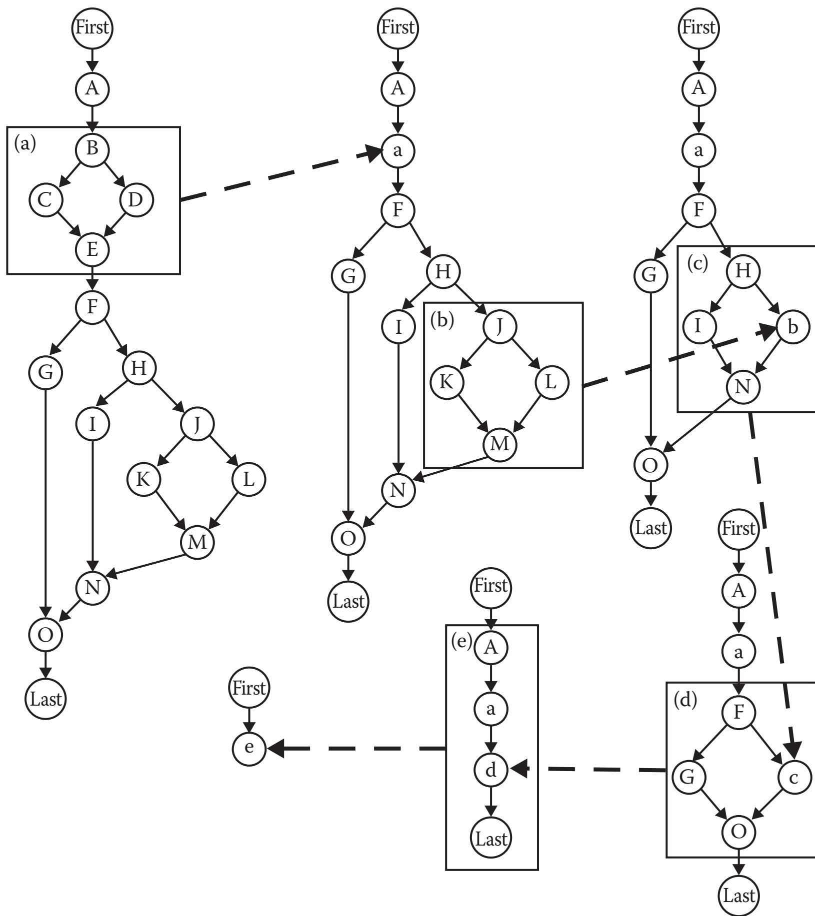


Figure 8.12 Condensing with respect to structured programming constructs.

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.

https://hemanthrajhemu.github.io

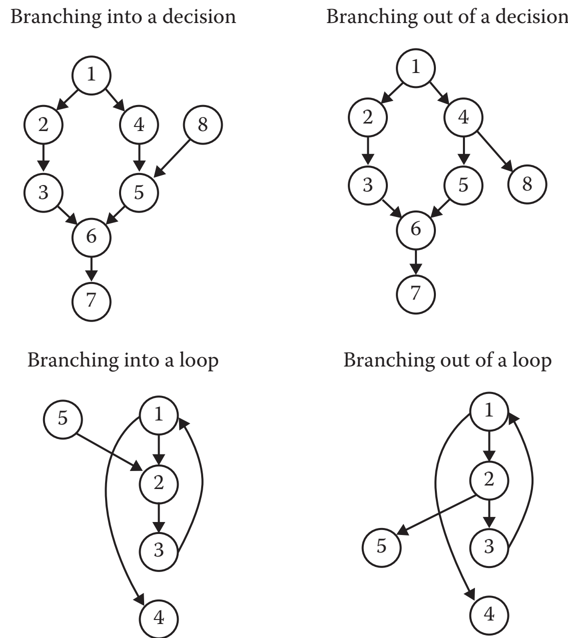


Figure 8.13 Violations of structured programming constructs.

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

https://hemanthrajhemu.github.io

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 $x + 0 = x$.

https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

Chapter 9

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. While dataflow and slice-based testing are cumbersome at the unit level; they are well suited for object-oriented code. 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. Also, both of these methods are difficult to perform manually, and unfortunately, few commercial tools exist to make life easier for the data flow and slicing testers. On the positive side, both techniques are helpful for coding and debugging.

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 and statement fragments) 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 before it is 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.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

9.1 Define/Use Testing

Much of the formalization of define/use testing was done in the early 1980s (Rapps and Weyuker, 1985); the definitions in this section are compatible with those in Clarke et al. (1989), which summarizes most define/use testing theory. This body of research is very compatible with the formulation we developed in Chapters 4 and 8. 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 $n \in G(P)$ is a *defining node* of the variable $v \in V$, written as DEF(v, n), if and only if the value of variable v is defined as 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 $n \in G(P)$ is a *usage node* of the variable $v \in V$, written as USE(v, n), if and only if the value of the variable v is used as 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) if and only if 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 an 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 $v \in V$, 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.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

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. One of the main values of du-paths is they identify points for variable "watches" and breakpoints when code is developed in an Integrated Development Environment. Figure 9.3 illustrates this very well later in the chapter.

9.1.1 Example

We will use the commission problem and its program graph to illustrate these definitions. The numbered pseudocode and its corresponding program graph are shown in Figure 9.1. This program computes the commission on the sales of the total numbers of locks, stocks, and barrels sold. The while loop is a classic 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.

Figure 9.2 shows the decision-to-decision path (DD-path) graph of the program graph in Figure 9.1. More compression exists in this DD-path graph because of the increased computation in the commission problem. Table 9.1 details the statement fragments associated with DD-paths. Some DD-paths (per the definition in Chapter 8) are combined to simplify the graph. We will need this figure later to help visualize the differences among DD-paths, du-paths, and program slices.

Table 9.2 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 9.1 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 is wrong, it can be helpful to include them. Take your pick. We will refer to the various paths as sequences of node numbers.

Tables 9.3 and 9.4 present some of the du-paths in the commission problem; they are named by their beginning and ending nodes (from Figure 9.1). The third column in Table 9.3 indicates whether the du-paths are definition clear. Some of the du-paths are trivial—for example, those for lockPrice, stockPrice, and barrelPrice. Others are more complex: the while loop (node sequence <14, 15, 16, 17, 18, 19, 20>) inputs and accumulated values for totalLocks, totalStocks, and totalBarrels. Table 9.3 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.)

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

```
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 totalBarrels = 0
11 totalStocks = 0
12 totalBarrels = 0
13 Input(locks)
14 While NOT(locks = –1)  "locks = –1 signals 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 barrelsSales = 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
39     commission = 0.10 * sales
40   EndIf
41 EndIf
42 Output("Commission is $", commission)
43 End Commission
```

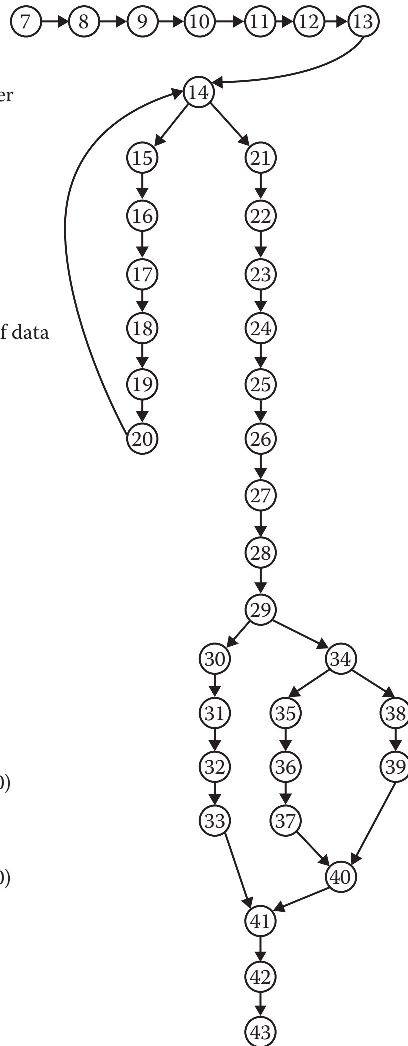


Figure 9.1 Commission problem and its program graph.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

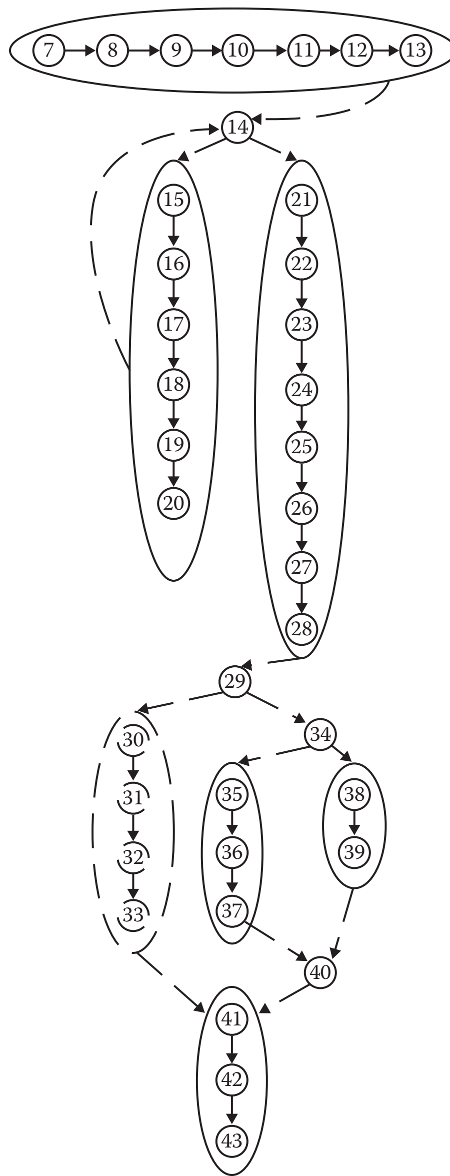


Figure 9.2 DD-path graph of commission problem pseudocode (in Figure 9.1).

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

9.1.2 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.

9.1.3 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; they are shown in Figure 9.3.

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

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

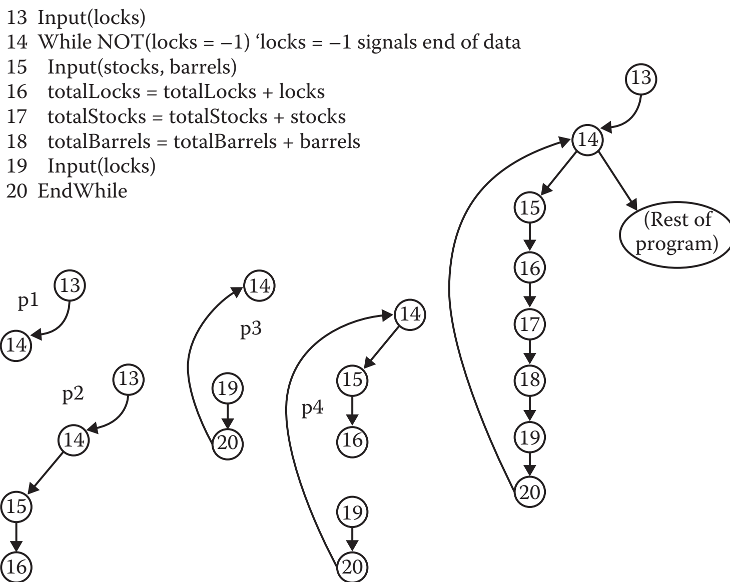


Figure 9.3 Du-paths for locks.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Table 9.1 DD-paths in Figure 9.1

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

Table 9.2 Define/Use Nodes for Variables in 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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Table 9.3 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	11, 25	No
totalStocks	17, 17	Yes
totalStocks	17, 22	No
totalStocks	17, 25	No
Locks	13, 14	Yes
Locks	13, 16	Yes
Locks	19, 14	Yes
Locks	19, 16	Yes
Sales	27, 28	Yes
Sales	27, 29	Yes
Sales	27, 33	Yes
Sales	27, 34	Yes
Sales	27, 37	Yes
Sales	27, 38	Yes

© 2010 Taylor & Francis Group, LLC
https://hemanthrajhemu.github.io

Table 9.4 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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

9.1.4 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, 14, 21>

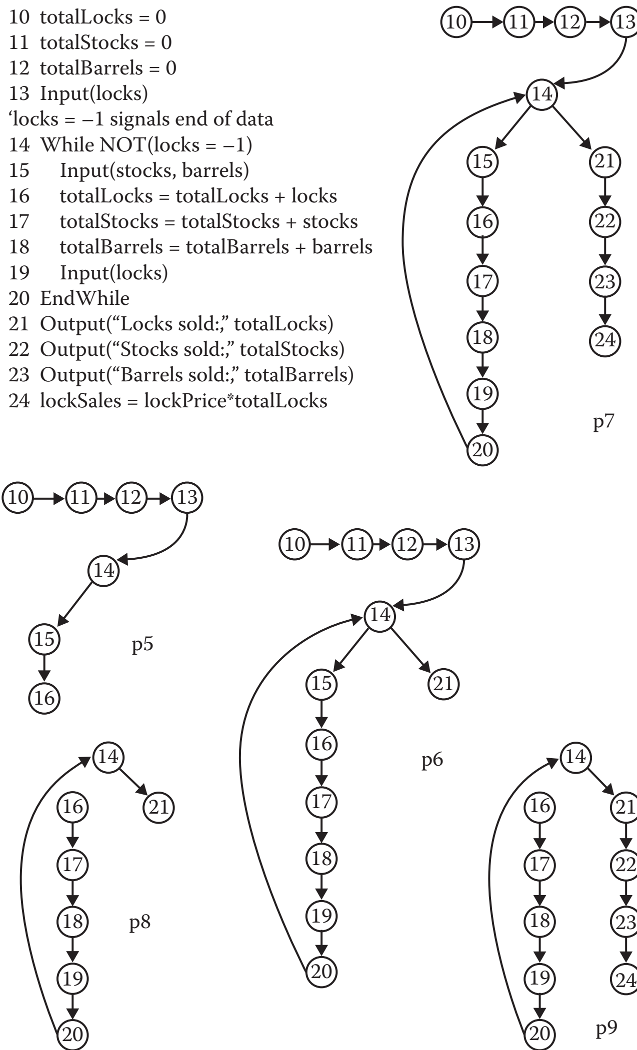


Figure 9.4 Du-paths for totalLocks.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

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

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

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

Du-path p7 is not definition clear because it includes node 16. Subpaths that begin with node 16 (an assignment statement) are interesting. The first, <16, 16>, seems degenerate. If we "expanded" it into machine code, we would be able to separate the define and usage portions. We will disallow these as du-paths. Technically, the usage on the right-hand side of the assignment refers to a value defined at node 10 (see path p5). The remaining two du-paths are both subpaths of p7:

p8 = <16, 17, 18, 19, 20, 14, 21>
p9 = <16, 17, 18, 19, 20, 14, 21, 22, 23, 24>

Both are definition clear, and both have the loop iteration problem we discussed before. The du-paths for totalLocks are shown in Figure 9.4.

9.1.5 Du-paths for Sales

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

p10 = <27, 28>
p11 = <27, 28, 29>
p12 = <27, 28, 29, 30, 31, 32, 33>

Notice that p12 is a definition-clear path with three usage nodes; it also contains paths p10 and p11. If we were testing with p12, we know we would also have covered the other two paths. We will revisit this toward the end of the chapter.

The IF, ELSE IF logic in statements 29 through 40 highlights an ambiguity in the original research. Two choices for du-paths begin with path p11: one choice is the path <27, 28, 29, 30, 31, 32, 33>, and the other is the path <27, 28, 29, 34>. The remaining du-paths for sales are

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

Note that the dynamic view is very compatible with the kind of thinking we used for DD-paths in Chapter 8.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

9.1.6 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, 39). Only one usage node is used: USE(commission, 41).

9.1.7 Define/Use Test Coverage Metrics

The whole point of analyzing a program with definition/use paths is to define a set of test coverage metrics known as the Rapps–Weyuker data flow metrics (Rapps and Weyuker, 1985). The first three of these are equivalent to three of E.F. Miller's metrics in Chapter 8: All-Paths, All-Edges, and All-Nodes. The others presume that define and usage nodes have been identified for all program variables, and that du-paths have been identified with respect to each variable. In the following definitions, T is a set of paths in the program graph $G(P)$ of a program P, with the set V of variables. It is not enough to take the cross product of the set of DEF nodes with the set of USE nodes for a variable to define du-paths. This mechanical approach can result in infeasible paths. In the next definitions, we assume that the define/use paths are all feasible.

Definition

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

Definition

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

Definition

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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Definition

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

Definition

The set T satisfies the *All-DU-paths criterion* for the program P if and only if for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every use of v and to the successor node of each USE(v, n), and that these paths are either single loop traversals or they are cycle free.

These test coverage metrics have several set-theory-based relationships, which are referred to as "subsumption" in Rapps and Weyuker (1985). These relationships are shown in Figure 9.5. We now have a more refined view of structural testing possibilities between the extremes of the (typically unattainable) All-Paths metric and the generally accepted minimum, All-Edges. What good is all this? Define/use testing provides a rigorous, systematic way to examine points at which faults may occur.

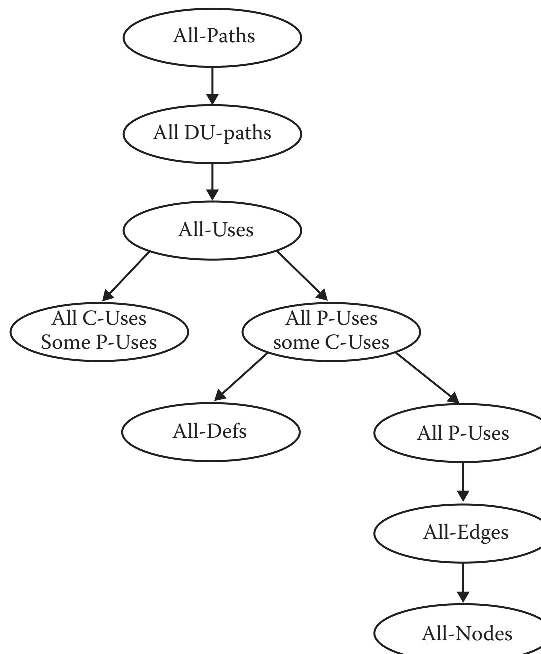


Figure 9.5 Rapps–Weyuker hierarchy of data flow coverage metrics.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

9.1.8 Define/Use Testing for Object-Oriented Code

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

9.2 Slice-Based Testing

Program slices have surfaced and submerged in software engineering literature since the early 1980s. They were proposed in Mark Weiser's dissertation in 1979 (Weiser, 1979), made more generally available in Weiser (1985), used as an approach to software maintenance in Gallagher and Lyle (1991), and more recently used to quantify functional cohesion in Bieman (1994). During the early 1990s, there was a flurry of published activity on slices, including a paper (Ball and Eick, 1994) describing a program to visualize program slices. This latter paper describes a tool used in industry. (Note that it took about 20 years to move a seminal idea into industrial practice.)

Part of the utility and versatility of program slices is due to the natural, intuitively clear intent of the concept. Informally, a program slice is a set of program statements that contributes to, or affects the value of, a variable at some point in a program. This notion of slice corresponds to other disciplines as well. We might study history in terms of slices—US history, European history, Russian history, Far East history, Roman history, and so on. The way such historical slices interact turns out to be very analogous to the way program slices interact.

We will start by growing our working definition of a program slice. We continue with the notation we used for define/use paths: a program P that has a program graph $G(P)$ and a set of program variables V. The first try refines the definition in Gallagher and Lyle (1991) to allow nodes in $P(G)$ to refer to statement fragments.

Definition

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

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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

The static/dynamic dichotomy is more complex. We borrow two terms from database technology to help explain the difference. In database parlance, we can refer to the intension and extensions of a database. The intension (it is unique) is the fundamental database structure, presumably expressed in a data modeling language. Populating a database creates an extension, and changes to a populated database all result in new extensions. With this in mind, a static backward slice $S(v, n)$ consists of all the statements in a program that determine the value of variable v at statement n, independent of values used in the statements. Dynamic slices refer to execution–time execution of portions of a static slice with specific values of all variables in $S(v, n)$. This is illustrated in Figures 9.6 and 9.7.

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

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 static, backward slice on the variable set V at statement fragment n*, written $S(V, n)$, is the set of node numbers of all statement fragments in P that contribute to the values of variables in V at statement fragment n.

The idea of program slicing is to separate a program into components that have some useful (functional) meaning. Another refinement is whether or not a program slice is executable. Adding all the data declaration statements and other syntactically necessary statements clearly increases the size of a slice, but the full version can be compiled and separately executed and tested. Further, such compilable slices can be “spliced” together (Gallagher and Lyle, 1991) as a bottom–up way to develop a program. As a test of clear diction, Gallagher and Lyle suggest the term “slice splicing.” In a sense, this is a precursor to agile programming. The alternative is to just consider program fragments, which we do here for space and clarity considerations. Eventually, we will develop a

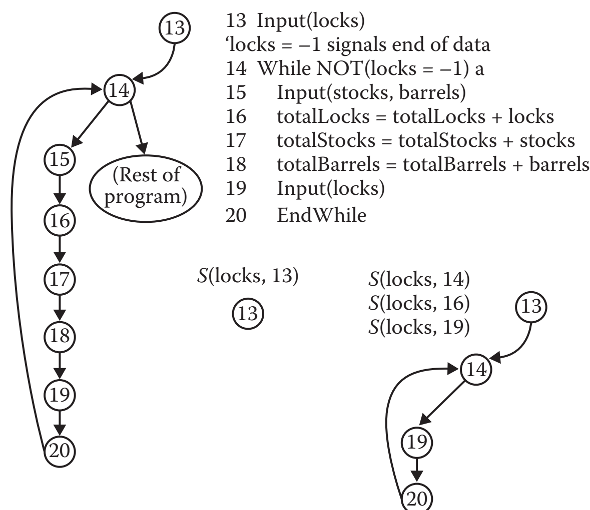


Figure 9.6 Selected slices on locks.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

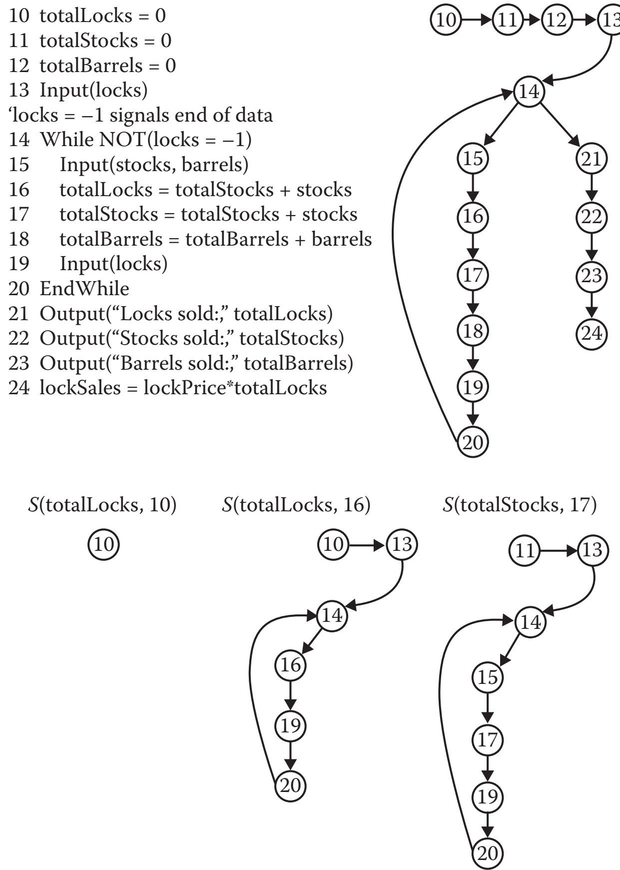


Figure 9.7 Selected slices in a loop.

lattice (a directed, acyclic graph) of static slices, in which nodes are slices and edges correspond to the subset relationship.

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

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

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

I-def	defined by input
A-def	defined by assignment

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

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

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

9.2.1 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 (nor in NextDate). In the following, except where specifically noted, we are speaking of static backward slices and we only include nodes corresponding to executable statement fragments. The examples refer to the source code for the commission problem in Figure 9.1. There are 42 "interesting" static backward slices in our example. They are named in Table 9.5. We will take a selective look at some interesting slices.

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

Table 9.5 Slices in Commission Problem

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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

S_1: S(lockPrice, 7) = {7}
S_2: S(stockPrice, 8) = {8}
S_3: S(barrelPrice, 9) = {9}
S_4: S(totalLocks, 10) = {10}
S_5: S(totalStocks, 11) = {11}
S_6: S(totalBarrels, 12) = {12}

Slices 7 through 17 focus on the sentinel controlled while loop in which the totals for locks, stocks, and barrels are accumulated. The locks variable has two uses in this loop: a P-use at fragment 14 and C-use at statement 16. It also has two defining nodes, at statements 13 and 19. The stocks and barrels variables have a defining node at 15, and computation uses at nodes 17 and 18, respectively. Notice the presence of all relevant statement fragments in slice 8. The slices on locks are shown in Figure 9.6.

S_7: S(locks, 13) = {13}
S_8: S(locks, 14) = {13, 14, 19, 20}
S_9: S(stocks, 15) = {13, 14, 15, 19, 20}
S_{10}: S(barrels, 15) = {13, 14, 15, 19, 20}
S_{11}: S(locks, 16) = {13, 14, 19, 20}
S_{12}: S(totalLocks, 16) = {10, 13, 14, 16, 19, 20}
S_{13}: S(stocks, 17) = {13, 14, 15, 19, 20}
S_{14}: S(totalStocks, 17) = {11, 13, 14, 15, 17, 19, 20}
S_{15}: S(barrels, 18) = {12, 13, 14, 15, 19, 20}
S_{16}: S(totalBarrels, 18) = {12, 13, 14, 15, 18, 19, 20}
S_{17}: S(locks, 19) = {13, 14, 19, 20}

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

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

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

S_{21}: S(lockPrice, 24) = {7}
S_{22}: S(totalLocks, 24) = {10, 13, 14, 16, 19, 20}
S_{23}: S(lockSales, 24) = {7, 10, 13, 14, 16, 19, 20, 24}
S_{24}: S(stockPrice, 25) = {8}

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

S_{25}: S(totalStocks, 25) = {11, 13, 14, 15, 17, 19, 20}
S_{26}: S(stockSales, 25) = {8, 11, 13, 14, 15, 17, 19, 20, 25}
S_{27}: S(barrelPrice, 26) = {9}
S_{28}: S(totalBarrels, 26) = {12, 13, 14, 15, 18, 19, 20}
S_{29}: S(barrelSales, 26) = {9, 12, 13, 14, 15, 18, 19, 20, 26}
S_{30}: S(sales, 27) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

Slices 31 through 36 are identical. Slice S_{31} is an O-use of sales; the others are all C-uses. Since none of these changes the value of sales defined at S_{30}, we only show one set of statement fragment numbers here.

S_{31}: S(sales, 28) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

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

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

Looking at slices as sets of fragment numbers (Figure 9.8) is correct in terms of our definition, but it is also helpful to see how slices are composed of sets of previous slices. We do this next, and show the final lattice in Figure 9.9.

S_1: S(lockPrice, 7) = {7}
S_2: S(stockPrice, 8) = {8}
S_3: S(barrelPrice, 9) = {9}
S_4: S(totalLocks, 10) = {10}
S_5: S(totalStocks, 11) = {11}
S_6: S(totalBarrels, 12) = {12}
S_7: S(locks, 13) = {13}
S_8: S(locks, 14) = $S_7 \cup$ {14, 19, 20}
S_9: S(stocks, 15) = $S_8 \cup$ {15}
S_{10}: S(barrels, 15) = S_8
S_{11}: S(locks, 16) = S_8
S_{12}: S(totalLocks, 16) = $S_4 \cup S_{11} \cup$ {16}
S_{13}: S(stocks, 17) = S_9 = {13, 14, 19, 20}

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

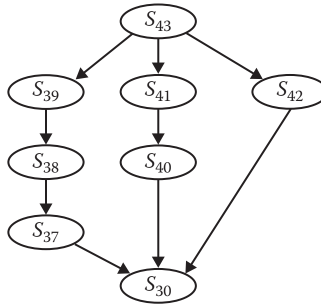


Figure 9.8 Partial lattice of slices on commission.

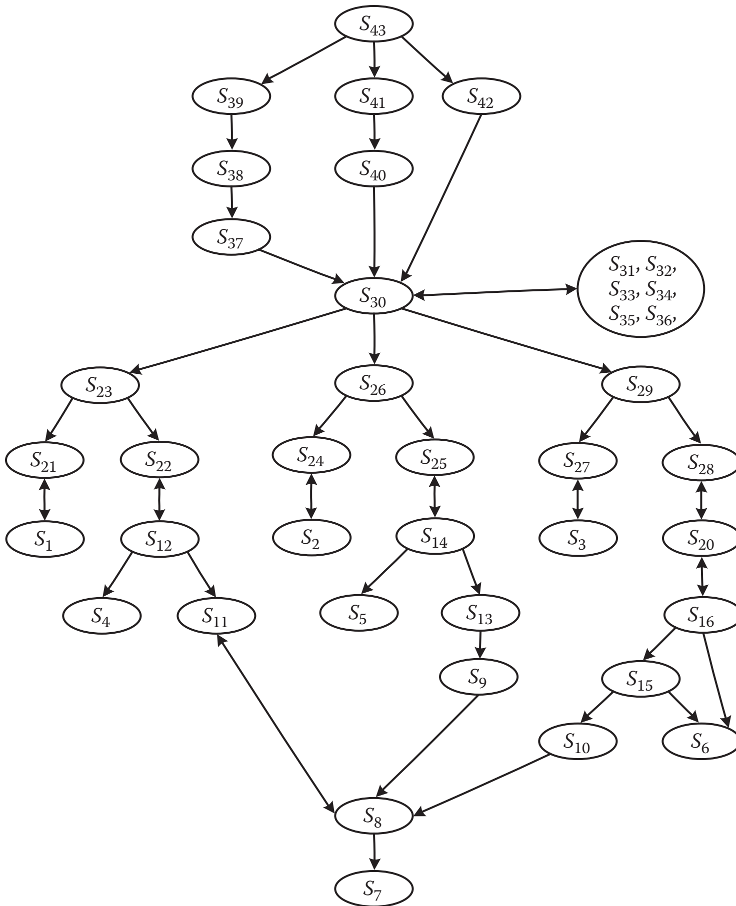


Figure 9.9 Full lattice on commission.

S_{14}: S(totalStocks, 17) $= S_5 \cup S_{13} \cup \{17\}$
S_{15}: S(barrels, 18) $= S_6 \cup S_{10}$
S_{16}: S(totalBarrels, 18) $= S_6 \cup S_{15} \cup \{18\}$
S_{18}: S(totalLocks, 21) $= S_{12}$

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

S_{19}: S(totalStocks, 22) = S_{14}
S_{20}: S(totalBarrels, 23) = S_{16}
S_{21}: S(lockPrice, 24) = S_1
S_{22}: S(totalLocks, 24) = S_{12}
S_{23}: S(lockSales, 24) = $S_{21} \cup S_{22} \cup \{24\}$
S_{24}: S(stockPrice, 25) = S_2
S_{25}: S(totalStocks, 25) = S_{14}
S_{26}: S(stockSales, 25) = $S_{24} \cup S_{25} \cup \{25\}$
S_{27}: S(barrelPrice, 26) = S_3
S_{28}: S(totalBarrels, 26) = S_{20}
S_{29}: S(barrelSales, 26) = $S_{27} \cup S_{28} \cup \{26\}$
S_{30}: S(sales, 27) = $S_{23} \cup S_{26} \cup S_{29} \cup \{27\}$
S_{31}: S(sales, 28) = S_{30}
S_{32}: S(sales, 29) = S_{30}
S_{33}: S(sales, 33) = S_{30}
S_{34}: S(sales, 34) = S_{30}
S_{35}: S(sales, 37) = S_{30}
S_{36}: S(sales, 39) = S_{30}
S_{37}: S(commission, 31) = $S_{30} \cup \{29, 30, 31\}$
S_{38}: S(commission, 32) = $S_{37} \cup \{32\}$
S_{39}: S(commission, 33) = $S_{38} \cup \{33\}$
S_{40}: S(commission, 36) = $S_{30} \cup \{29, 34, 35, 36\}$
S_{41}: S(commission, 37) = $S_{40} \cup \{37\}$
S_{42}: S(commission, 39) = $S_{30} \cup \{29, 34, 38, 39\}$
S_{43}: S(commission, 41) = $S_{39} \cup S_{41} \cup S_{42}$

Several of the connections in Figure 9.9 are double-headed arrows indicating set equivalence. (Recall from Chapter 3 that if $A \subseteq B$ and $B \subseteq A$, then $A = B$.) We can clean up Figure 9.9 by removing these, and thereby get a better lattice. The result of doing this is in Figure 9.10.

9.2.2 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 are more restrictive than necessary.

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

$$V = \{\text{lockSales, stockSales, barrelSales}\}$$

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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

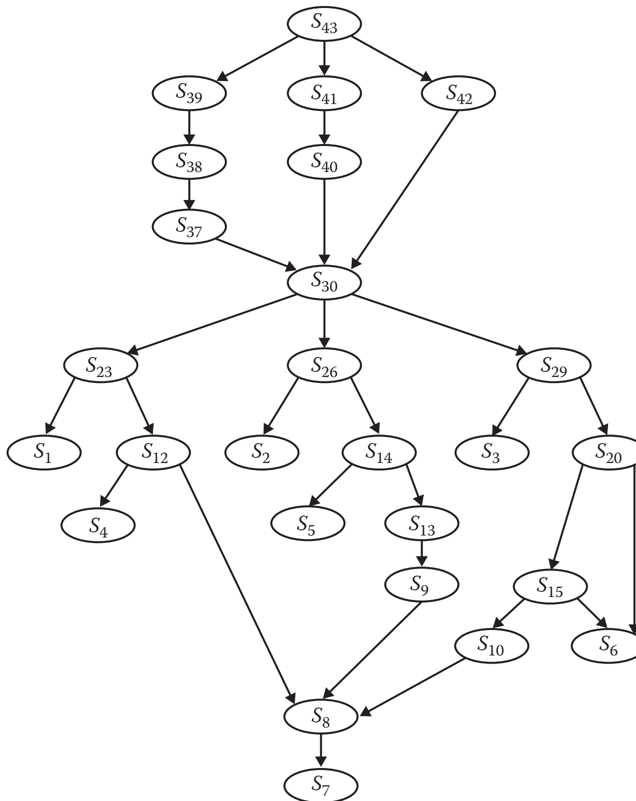


Figure 9.10 Simplified lattice on commission.

3. Make slices for all A-def nodes. When a variable is computed by an assignment statement, a slice on the variable at that statement will include (portions of) all du-paths of the variables used in the computation. Slice S_{30}: S(sales, 27) is a good example of an A-def slice. Similarly for variables defined by input statements (I-def nodes), such as S_{10}: S(barrels, 15).
4. There is not much reason to make slices on variables that occur in output statements. Slices on O-use variables can always be expressed as unions of slices on all the A-defs (and I-defs) of the O-use variable.
5. Make slices for P-use nodes. When a variable is used in a predicate, the slice on that variable at the decision statement shows how the predicate variable got its value. This is very useful in decision-intensive programs such as the triangle program and NextDate.
6. Consider making slices compilable. Nothing in the definition of a slice requires that the set of statements is compilable; however, if we make this choice, it means that a set of compiler directive and data declaration statements is a subset of every slice. If we added this same set of statements to all the slices we made for the commission program, our lattices remain undisturbed; however, each slice is separately compilable (and therefore executable). In Chapter 1, we suggested that good testing practices lead to better programming practices. Here, we have a good example. Think about developing programs in terms of compilable slices. If we did this, we could code a slice and immediately test it. We can then code and test other slices and merge them (sometimes called "slice splicing") into a fairly solid program. This is done in Section 9.2.3.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

9.2.3 Slice Splicing

The commission program is deliberately small, yet it suffices to illustrate the idea of "slice splicing." In Figures 9.11 through 9.14, the commission program is split into four slices. Statement fragment numbers and the program graphs are as they were in Figure 9.1. Slice 1 contains the input while loop controlled by the locks variable. This is a good starting point because both Slice 2 and Slice 3 use the loop to get input values for stocks and barrels, respectively. Slices 1, 2, and 3 each culminate in a value of sales, which is the starting point for Slice 4, which computes the commission bases on the value of sales.

This is overkill for this small example; however, the idea extends perfectly to larger programs. It also illustrates the basis for program comprehension needed in software maintenance. Slices allow the maintenance programmer to focus on the issues at hand and avoid the extraneous information that would be in du-paths.

```
 1  Program Slice1 (INPUT,OUTPUT)
 2  Dim locks As Integer
 3  Dim lockPrice As Real
 4  Dim totalLocks As Integer
 5  Dim lockSales As Real
 6  Dim sales As Real
 7  lockPrice = 45.0
13  Input(locks)
'locks = –1 signals end of data
14  While NOT(locks = –1)
16  totalLocks = totalLocks + locks
19  Input(locks)
20  EndWhile
21  Output("Locks sold:", totalLocks)
24  lockSales = lockPrice*totalLocks
27  sales = lockSales
28  Output("Total sales: ", sales)
```

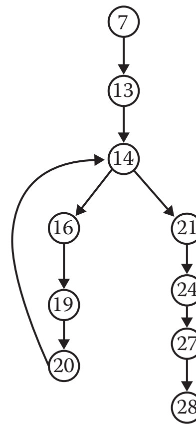


Figure 9.11 Slice 1.

```
 1  Program Slice2 (INPUT,OUTPUT)
 2  Dim locks, stocks As Integer
 3  Dim stockPrice As Real
 4  Dim totalStocks As Integer
 5  Dim stockSales As Real
 6  Dim sales As Real
 8  stockPrice = 30.0
11  totalStocks = 0
13  Input(locks)
'locks = –1 signals end of data
14  While NOT(locks = –1)
15    Input(stocks)
17    totalStocks = totalStocks + stocks
19    Input(locks)
20  EndWhile
22  Output("Stocks sold:," totalStocks)
25  stockSales = stockPrice*totalStockd
27  sales = stockSales
28  Output("Total sales:," sales)
```

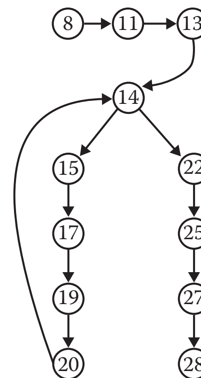


Figure 9.12 Slice 2.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

```
 1  Program Slice3 (INPUT,OUTPUT)
 2  Dim locks, barrels As integer
 3  Dim barrelPrice As Real
 4  Dim totalBarrels As Integer
 5  Dim barrelSales As Real
 6  Dim sales As Real
 9  barrelPrice = 25.0
12  totalBarrels = 0
13  Input(locks)
'locks = –1 signals end of data
14  While NOT(locks = –1)
15    Input(barrels)
18    totalBarrels = totalBarrels + barrels
19    Input(locks)
20  EndWhile
23  Output("Barrels sold:," totalBarrels)
26  barrelSales = barrelsPrice * totalBarrles
27  sales = barrelSales
28  Output("Total sales:," sales)
```

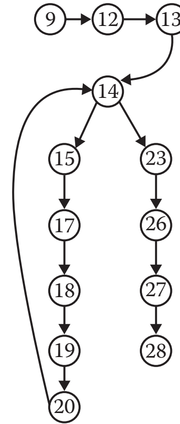


Figure 9.13 Slice 3.

```
 1  Program Slice4 (INPUT,OUTPUT)
 6  Dim sales, commission As Real
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
39      commission = 0.10 * sales
40    EndIf
41  EndIf
42  Output("Commission is $," commission)
43  End Commission
```

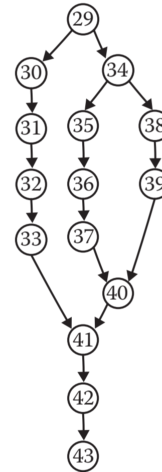


Figure 9.14 Slice 4.

9.3 Program Slicing Tools

Any reader who has gone carefully through the preceding section will agree that program slicing is not a viable manual approach. I hesitate assigning a slicing exercise to my university students because the actual learning is only marginal in terms of the time spent with good tools; however, program slicing has its place. There are a few program slicing tools; most are academic or experimental, but there are a very few commercial tools. (See Hoffner [1995] for a dated comparison.)

The more elaborate tools feature interprocedural slicing, something clearly useful for large systems. Much of the market uses program slicing to improve the program comprehension that maintenance programmers need. One, JSlice, will be appropriate for object-oriented software. Table 9.6 summarizes a few program slicing tools.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Table 9.6 Selected Program Slicing Tools

Tool/Product	*Language*	*Static/Dynamic?*
Kamkar	Pascal	Dynamic
Spyder	ANSI C	Dynamic
Unravel	ANSI C	Static
CodeSonar®	C, C++	Static
Indus/Kaveri	Java	Static
JSlice	Java	Dynamic
SeeSlice	C	Dynamic

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 9.5.
3. List the du-paths for the commission variable.
4. 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.

References

Bieman, J.M. and Ott, L.M., Measuring functional cohesion, *IEEE Transactions on Software Engineering*, Vol. SE-20, No. 8, August 1994, pp. 644–657.

Ball, T. and Eick, S.G., Visualizing program slices, *Proceedings of the 1994 IEEE Symposium on Visual Languages*, St. Louis, MO, October 1994, pp. 288–295.

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

Gallagher, K.B. and Lyle, J.R., Using program slicing in software maintenance, *IEEE Transactions on Software Engineering*, Vol. SE-17, No. 8, August 1991, pp. 751–761.

Hoffner, T., *Evaluation and Comparison of Program Slicing Tools,* Technical Report, Dept. of Computer and Information Science, Linkoping University, Sweden, 1995.

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

Weiser, M., *Program Slices*: *Formal Psychological and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, University of Michigan, Ann Arbor, MI. 1979.

Weiser, M.D., Program slicing, *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, April 1988, pp. 352–357.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Chapter 10

Retrospective on Unit Testing

When should unit 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 take a broad brush look at the unit testing methods we studied. Metaphorically, this is pictured as a pendulum that swings between extremes. Next, we follow one swing of the pendulum from the most abstract form of code-based testing through strongly semantic-based methods, and then back toward the very abstract shades of specification-based testing. We do this tour with the triangle program. After that, some recommendations for both forms of unit testing, followed by another case study—this time of an automobile insurance example.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

10.1 The Test Method Pendulum

As with many things in life, there is a pendulum that swings between two extremes. The test method pendulum swings between two extremes of low semantic content—from strictly topological to purely functional. As testing methods move away from the extremes and toward the center, they become, at once, both more effective and more difficult (see Figure 10.1).

On the code-based side, path-based testing relies on the connectivity of a program graph—the semantic meaning of the nodes is lost. A program graph is a purely topological abstraction of the code, and is nearly devoid of code meaning—only the control flow remains. This gives rise to program paths that can never be recognized as infeasible by automated means. Moving to data flow testing, the kinds of dependencies that typically create infeasible paths can often be detected. Finally, when viewed in terms of slices, we arrive as close as we can to the semantic meaning of the code.

On the specification-based side, testing based only on boundary values of the variables is vulnerable to severe gaps and redundancies, neither of which can be known in purely specification-based testing. Equivalence class testing uses the "similar treatment" idea to identify classes, and in doing so, uses more of the semantic meaning of the specification. Finally, decision table testing uses both necessary and impossible combinations of conditions, derived from the specification, to deal with complex logical considerations.

On both sides of the testing pendulum, test case identification becomes easier as we move toward the extremes. It also becomes less effective. As testing techniques move toward higher semantic meaning, they become more difficult to automate—and more effective. Hmmm … could it be that, when he wrote "The Pit and the Pendulum," Edgar Allan Poe was actually thinking about testing as a pit, and methods as a pendulum? You decide. Meanwhile, these ideas are approximated in Figure 10.2.

These graphs need some elaboration. Starting with program graph testing, notice that the nodes contain absolutely no semantic information about the statement fragments—and the edges just describe whether one fragment can be executed after a predecessor fragment. Paths in a

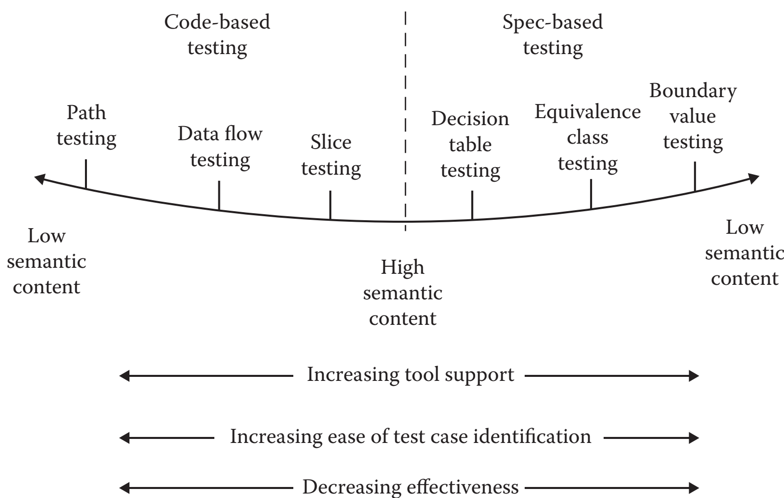


Figure 10.1 Test method pendulum.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

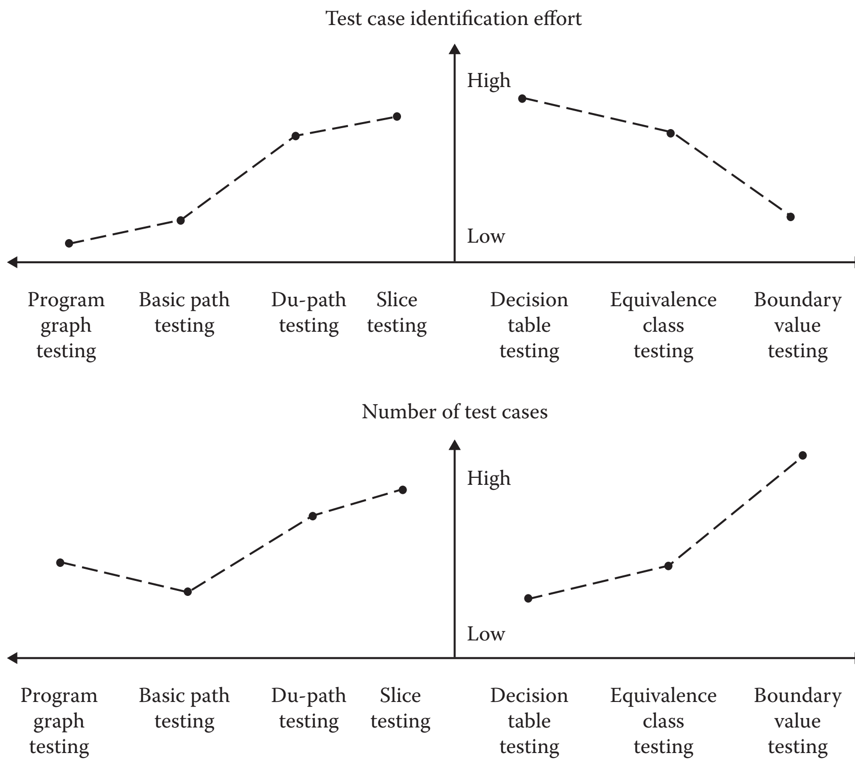


Figure 10.2 Effort and efficacy of unit test methods.

program graph are all topologically possible—in fact, they can be generated mathematically with Warshall's algorithm. The problem is that the set of topologically possible paths includes both feasible and infeasible paths, as we discussed in Chapter 8. Moving in the direction of McCabe's basis path testing adds a little semantic content. The recommended starting point is a mainline path that represents common unit functionality. The basis path method runs into trouble after that due to the heuristic of simply "flipping" decisions as they are encountered on the starting point path. This also leads to the possibility of infeasible paths. When testing moves to the define/use domain, we use more semantic meaning. We follow where values of variables are defined and later used. The distinction between du-paths and definition-clear du-paths gives the tester even more semantic information. Finally, backward slices do two things, they eliminate unwanted detail, and thereby focus attention exactly where it is needed—all the statements affecting the value of a variable at a given point in a program. The program slicing literature contains extensive discussions of automatic slicing algorithms that are beyond the scope of this book.

On the specification-based side, the various forms of boundary value testing are shown as the most abstract. All test cases are derived from properties of the input space with absolutely no consideration about how the values are used in the unit code. When we move to equivalence class testing, the prime factor that determines a class is the "similar treatment" principle. Clearly, this moves in the direction of semantic meaning. Moving from equivalence class testing to decision table testing is usually done for two reasons: the presence of dependencies among the variables and the possibility of impossible combinations.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

The lower half of Figure 10.2 shows that, for specification-based testing, there is a true trade-off between test case creation effort and test case execution time. If the testing is automated, as in a jUnit environment, this is not a penalty. On the code-based side, as methods get more sophisticated, they concurrently generate more test cases.

The bottom line to this discussion is that the combination of specification-based and code-based methods depends on the nature of the unit being tested, and this is where testers can exhibit craftsmanship.

10.2 Traversing the Pendulum

We will use the triangle program to explore some of the lessons of the testing pendulum. We begin with the FORTRAN-like version so popular in the early literature. We use this implementation here, mostly because it is the most frequently used in testing literature (Brown and Lipov, 1975; Pressman, 1982). The flowchart from Chapter 2 is repeated here in Figure 10.3, and transformed to a directed graph in Figure 10.4. The numbers of the flowchart symbols are preserved as node numbers in the corresponding directed acyclic graph in Figure 10.4.

We can begin to see some of the difficulties when we base testing on a program graph. There are 80 topologically possible paths in Figure 10.4 (and also in Figure 10.3), but only 11 of these are feasible; they are listed in Table 10.1. Since this is at one abstract end of the testing pendulum, we cannot expect any automated help to separate feasible from infeasible paths. Much of the infeasibility is due to the Match variable. Its intent was to reduce the number of decisions. The boxes incrementing the Match variable depend on tests of equality among the three pairs of sides. Of the eight paths from box 1 to box 7, the logically possible values of Match are 0, 1, 2, 3, and 6. The three impossible paths correspond to exactly two pairs of sides being equal, which, by transitivity, is impossible. There are 13 decisions in the flowchart, so the goal of reducing decisions was missed anyway.

What suggests that this is a FORTRAN-like implementation is that, in the early days of FORTRAN programming, memory was expensive and computers were relatively slow. On the basis of the flowchart, a good FORTRAN programmer would compute the sums of pairs (a + b, a + c, and b + c) only once and use these again in the decisions checking the triangle inequality (decisions 8, 9, 10, 14, 17, and 19).

Moving on to basis path testing, we have another problem. The program graph in Figure 10.4 has a cyclomatic complexity of 14. McCabe's basis path method would ask us to find 14 test cases, but there are only 11 feasible paths. Again, this view is too far removed from the semantic meaning of the code to help.

Data flow testing will give us some valuable insights. Consider du-paths on the Match variable. It has four definition nodes, three computation uses, and four predicate uses, so there are 28 possible du-paths. Looking at the definition-clear paths will be a good start to data flow testing. The sides, a, b, and c, have one definition node and nine use nodes. All nine du-paths on these variables will be definition clear. That means very little can happen to these variables unless there is an input problem.

Testing using backward static slices would be a good idea. Although no variable for this appears in the original flowchart, we can postulate a variable, triangleType, that has the four string values shown in boxes 11, 12, 15, and 20. The first slice to test would be *S*(triangleType, 11), which represents the only way to have a scalene triangle. We could test it with three test cases: (a, b, c) = (3, 4, 5), = (4, 5, 3), and = (5, 3, 4). These triplets let each variable take on all three possibilities,

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

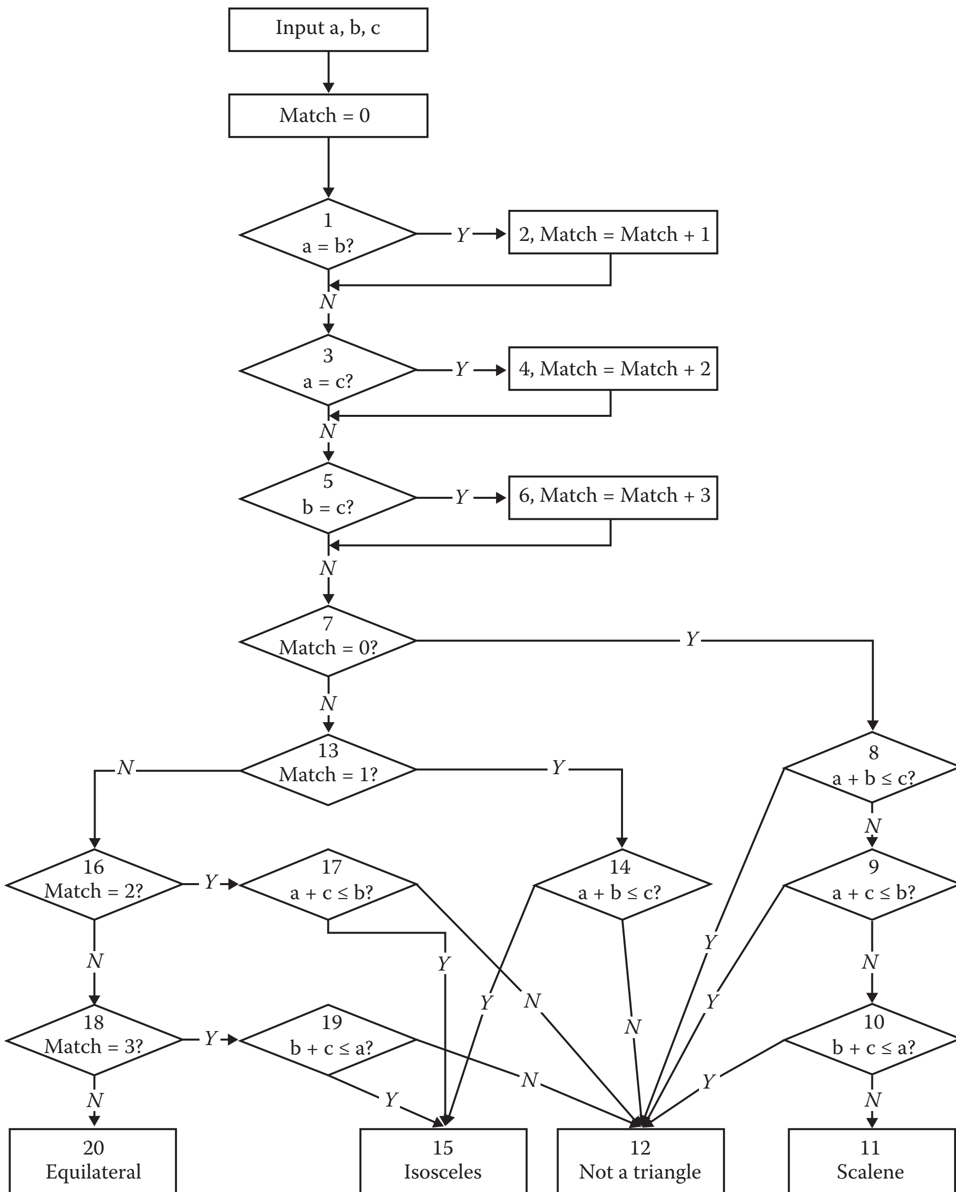


Figure 10.3 Flowchart of FORTRAN-like triangle program.

a little like a Sudoku puzzle. Similar comments apply to the slice *S*(triangleType, 20) where the expected value of triangleType is “Equilateral.” Here we would only need one test case, maybe foreshadowing equivalence class testing. The last slices to test would be for isosceles triangles, and then six ways for a, b, and c to fail to constitute sides of a triangle.

Notice that, in the pendulum swing from the very abstract program graphs to the semantically rich slice-based testing, the testing is improved. We can expect the same on the specification-based side.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

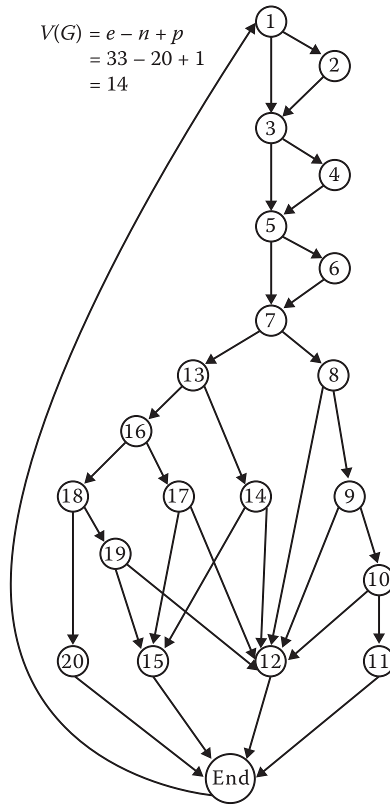


Figure 10.4 Directed graph of the FORTRAN-like triangle program.

Table 10.1 Feasible Paths in FORTRAN-Like 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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Suppose we use boundary value testing to define test cases. We will do this for both the basic and worst-case formulations. Table 10.2 shows the test cases generated using the nominal boundary value form of functional testing. The last column shows the path (from Table 10.1) taken by the test case.

The following paths are covered: p1, p2, p3, p4, p5, p6, p7; 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 10.3 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.

The next step in the pendulum progression is equivalence class testing. For the triangle problem, equivalence classes on the individual variables are pointless. Instead, we can make equivalence

Table 10.2 Path Coverage of Nominal Boundary 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 10.3 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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Table 10.4 Decision Table for FORTRAN-Like Triangle Program

c1. *Match* =	*0*				*1*				*2*			
c2. a + b < c?	T	F!	F!	F	T	F!	F!	F	T	F!	F!	F
c3. a + c < b?	F!	T	F!	F	F!	T	F!	F	F!	T	F!	F
c4. b + c < a?	F!	F!	T	F	F!	F!	T	F	F!	F!	T	F
a1. Scalene				×								
a2. Not a triangle	×	×	×		×	×	×		×	×	×	
a3. Isosceles							×					×
a4. Equilateral												
a5. Impossible												

c1. *Match* =	*3*				*4*	*5*	*6*			
c2. a + b < c?	T	F!	F!	F	—	—	T	F!	F!	F
c3. a + c < b?	F!	T	F!	F	—	—	F!	T	F!	F
c4. b + c < a?	F!	F!	T	F	—	—	F!	F!	T	F
a1. Scalene										
a2. Not a triangle	×	×	×				×	×	×	
a3. Isosceles				×						
a4. Equilateral										×
a5. Impossible					×	×				

classes on the types of triangles, and the six ways that the variables a, b, and c can fail to be sides of a triangle. In Chapter 6 (Section 6.4), we ended up with these equivalence classes:

D1 = {<a, b, c>: a = b = c}
D2 = {<a, b, c>: a = b, a ≠ c}
D3 = {<a, b, c>: a = c, a ≠ b}
D4 = {<a, b, c>: b = c, a ≠ b}
D5 = {<a, b, c>: a ≠ b, a ≠ c, b ≠ c}
D6 = {<a, b, c>: a > b + c}
D7 = {<a, b, c>: b > a + c}
D8 = {<a, b, c>: c > a + b}
D9 = {<a, b, c>: a = b + c}
D10 = {<a, b, c>: b = a + c}
D11 = {<a, b, c>: c = a + b}

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Since these are equivalence classes, we will have just 11 test cases, and we know we will have full coverage of the 11 feasible paths in Figure 10.3.

The last step is to see if decision tables will add anything to the equivalence class test cases. They do not, but they can provide some insight into the decisions in the FORTRAN-like flowchart. In the decision table in Table 10.4, first notice that the condition on Match is an extended entry. Although it is topologically possible to have Match = 4 and Match = 5, these values are logically impossible. Conditions c2, c3, and c4 are exactly those used in the flowchart. We use the *F!* (must be false) notation to denote the impossibility of more than one of these conditions to be true. Also, note that there is no point in developing conditions on the individual variables a, b, and c. To conclude the traversal of the pendulum, decision table–based testing did not add much, but it did highlight why some cases are impossible.

10.3 Evaluating Test Methods

Evaluating a test method reduces to ways to evaluate how effective is a set of test cases generated by a test method, 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; we will have an example of this in Section 10.4.

Structured testing techniques yield a second choice for test effectiveness. We can use 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. Mutation testing, the subject of Chapter 21, is an interesting way to assess the utility of a set of test cases.

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 (specification-based and code-based) 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.

By now, we have convinced ourselves that the specification-based methods are indeed open to the twin problems of gaps and redundancies; we can develop some metrics that relate the effectiveness of a specification-based technique with the achievement of a code-based metric. Specification-based testing techniques always result in a set of test cases, and a code-based 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.

In the following definitions, we assume that a specification-based testing technique M generates m test cases, and that these test cases are tracked with respect to a code-based metric S that

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

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 the 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 the 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 the ratio of m to n. We denote it as $\mathrm{NR}(M,S)$.

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 $R(M,S) = \mathrm{NR}(M,S)$. 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 specification-based testing method (except special value testing) with respect to a code-based 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 code-based metrics with respect to the kinds of faults we anticipate (or maybe faults we most fear). See the guidelines near the end of this chapter for specific advice.

In general, the more sophisticated code-based 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 code-based 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 10.5 uses test case data from our earlier chapters to apply these metrics to the triangle program.
Table 10.6 repeats this analysis for the commission problem.

Table 10.5 Metrics for 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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Table 10.6 Metrics for 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

10.4 Insurance Premium Case Study

Here is an example that lets us compare both specification-based and code-based testing methods and apply the guidelines. A hypothetical insurance premium program computes the semiannual 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 exceeds 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. The data for the insurance premium program are in Table 10.7.

10.4.1 Specification-Based Testing

Worst-case boundary value testing, based on the input variables, age, and points, yields the following extreme values of the age and points variables (Table 10.8). The corresponding 25 test cases are shown in Figure 10.5.

Table 10.7 Data for Insurance Premium Problem

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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Table 10.8 Data Boundaries for Insurance Premium Problem

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

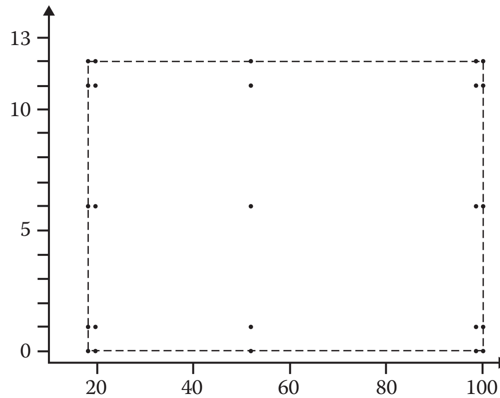


Figure 10.5 Worst-case boundary value test cases for insurance premium problem.

Nobody should 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 classes based on the age ranges.

A1 = {age: 16 ≤ age < 25}
A2 = {age: 25 ≤ age < 35}
A3 = {age: 35 ≤ age < 45}
A4 = {age: 45 ≤ age < 60}
A5 = {age: 60 ≤ age < 100}

Here are the age-dependent classes on license points.

P1(A1) = {points = 0, 1}, {points = 2, 3, …, 12}
P2(A2) = {points = 0, 1, 2, 3}, {points = 4, 5, …, 12}
P3(A3) = {points = 0, 1, 2, 3, 4, 5}, {points = 6, 7, …, 12}
P4(A4) = {points = 0, 1, 2, 3, 4, 5, 6, 7}, {points = 8, 9, 10, 11, 12}
P5(A5) = {points = 0, 1, 2, 3, 4, 5}, {points = 6, 7, …, 12}

One added complexity is that the point ranges are dependent on the age of the policyholder and also overlap. Both of these constraints are shown in Figure 10.6. The dashed lines show the age-dependent equivalence classes. A set of worst-case boundary value test cases is shown only for class A4 and its two related point classes are given in Figure 10.6. Because these ranges meet at "endpoints," we would have the worst-case test values shown in Table 10.9. Notice that the discrete

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

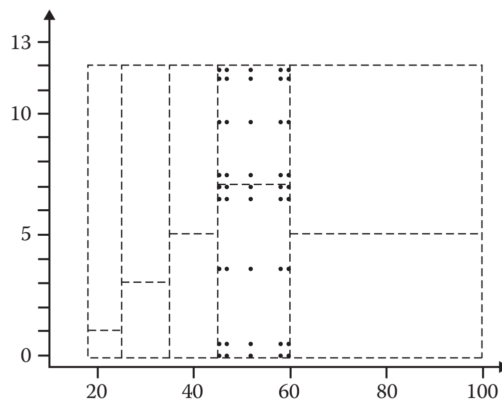


Figure 10.6 Detailed worst-case boundary value test cases for one age class.

Table 10.9 Detailed Worst-Case Values

Variable	*Min*	*Min+*	*Nom.*	*Max–*	*Max*
Age	16	17	20	24	
Age	25	26	30	34	
Age	35	36	40	44	
Age	45	46	53	59	
Age	60	61	75	99	100
Points(A1)	0	n/a	n/a	n/a	1
Points(A1)	2	3	7	11	12
Points(A2)	0	1	n/a	2	3
Points(A2)	4	5	8	11	12
Points(A3)	0	1	3	4	5
Points(A3)	6	7	9	11	12
Points(A4)	0	1	4	6	7
Points(A4)	8	9	10	11	12
Points(A5)	0	1	3	4	5
Points(A5)	6	7	9	11	12

values of the point variable do not lend themselves to the min+ and max– convention in some cases. These are the variable values that lead to 103 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 weak normal equivalence class test cases are shown in Figure 10.7. Since the point

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

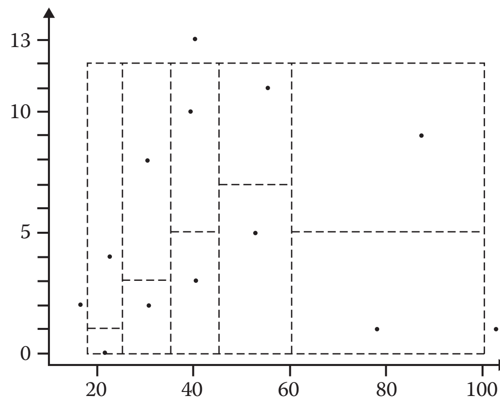


Figure 10.7 Weak and robust normal equivalence class test cases for insurance premium program.

classes are not independent, we cannot do the usual cross product. Weak robust cases are of some value because we would expect different outputs for drivers with age less than 16, and points in excess of 12. The additional weak robust test cases are shown as open circles in Figure 10.7.

The next step is to see if a decision table approach might help. Table 10.10 is a decision table based on the age equivalence classes. The decision table test cases are almost the same as those shown in Figure 10.7; the only weak robust test case missing in the decision table is that for points exceeding 12.

What are the error-prone aspects of the 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. We can imagine many complaints from policyholders whose premium did not reflect a recent borderline birthday. Incidentally, this would be a good example of risk-based testing. Dealing with such complaints would be costly. Also, we should consider ages under 16 and over 100. Finally, we should probably check the values at which the safe driving reduction is lost, and maybe values

Table 10.10 Insurance Premium Decision Table

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Age is	<16	16–24		25–34		35–44		45–59		60–100		>100
Points below cutoff?	—	T	F	T	F	T	F	T	F	T	F	—
ageMultiplier = 2.8		×	×									
ageMultiplier = 1.8				×	×							
ageMultiplier = 1.0						×	×					
ageMultiplier = 0.8								×	×			
ageMultiplier = 1.5										×	×	
Safe driving discount		×		×		×		×		×		
No policy allowed	×											×

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

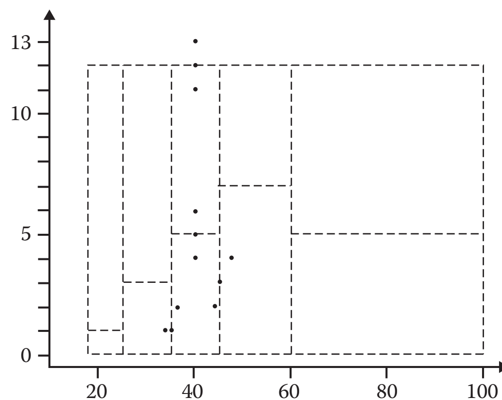


Figure 10.8 Hybrid test cases for the 35 to 45 age class.

of points over 12, when all insurance is lost. All of this is shown in Figure 10.7. (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.

To blend boundary value testing with weak robust equivalence class testing, note that the age class borders are helpful. Testing the max–, max, and max+ values of one age class automatically moves us into the next age class, so there is a slight economy. Figure 10.8 shows the hybrid test cases for the age range 35–45 in the insurance premium problem.

10.4.2 Code-Based Testing

Our analysis thus far has been entirely specification based. To be complete, we really need the code. It will answer questions such as whether or not the age variable is an integer (our assumption thus far) or not. There is no question that the points variable is an integer. The pseudocode implementation is minimal in the sense that it does very little error checking. The pseudocode and its program graph are in Figure 10.9. 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.

10.4.2.1 Path-Based Testing

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 10.11. If you follow the pseudocode for the various sets of functional test cases in Chapter 5, you will find the results shown in Table 10.12. 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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

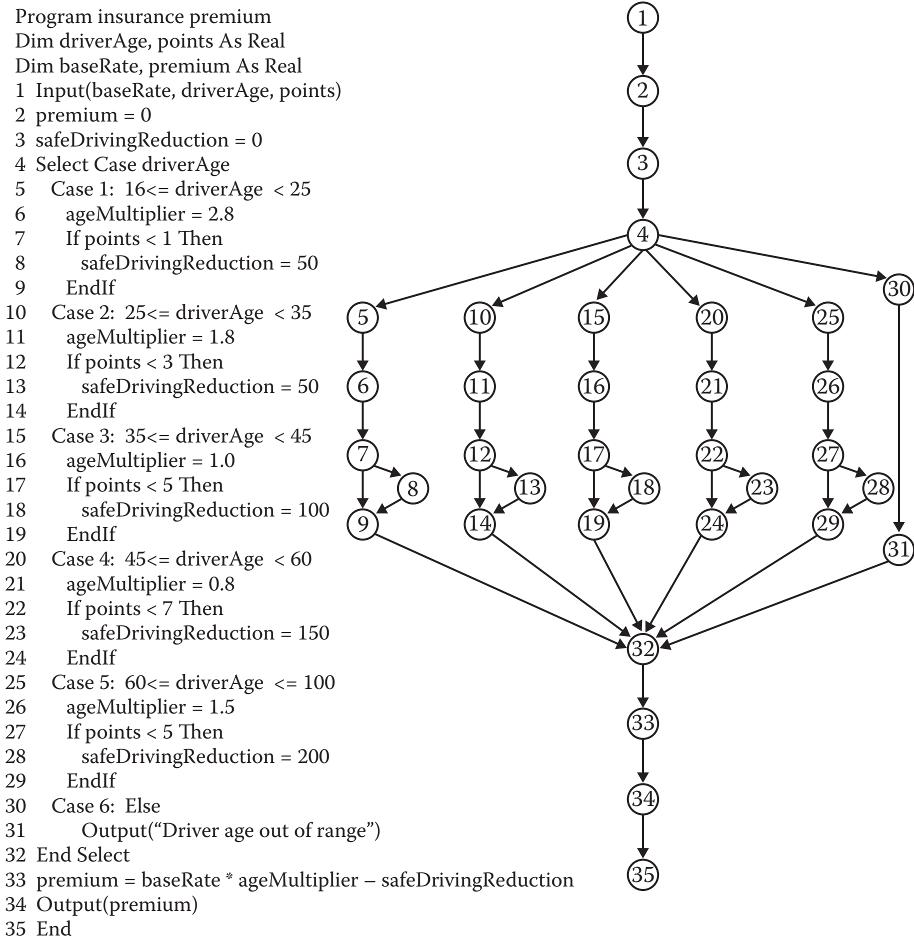


Figure 10.9 Insurance premium pseudocode and program graph.

cases only cover six of the feasible execution paths, while the 25 weak normal 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 $a \leq x < b$. The only methods that yield test cases that exercise these extreme values are the worst-case boundary value (103) test cases and the hybrid (32) test cases. Incidentally, the McCabe baseline method will yield 11 of the 12 decision table test cases.

10.4.2.2 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 9 that the All-Paths criterion implies all the lower data flow covers.

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

Table 10.11 Paths in Insurance Premium Program

Path	*Node Sequence*
p1	1–2–3–4–5–6–7–9–32–33–34–35
p2	1–2–3–4–5–6–7–8–9–32–33–34–35
p3	1–2–3–4–10–11–12–14–32–33–34–35
p4	1–2–3–4–10–11–12–13–14–32–33–34–35
p5	1–2–3–4–15–16–17–19–32–33–34–35
p6	1–2–3–4–15–16–17–18–19–32–33–34–35
p7	1–2–3–4–20–21–22–24–32–33–34–35
p8	1–2–3–4–20–21–22–23–24–32–33–34–35
p9	1–2–3–4–25–26–27–29–32–33–34–35
p10	1–2–3–4–25–26–27–28–29–32–33–34–35
p11	1–2–3–4–30–31–32–33–34–35

Table 10.12 Path Coverage of Functional Methods in Insurance Premium Program

Figure	*Method*	*Test Cases*	*Paths Covered*
10.5	Boundary value	25	p1, p2, p7, p8, p9, p10
10.6	Worst-case boundary value	103	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10
10.7	Weak normal equivalence class	10	P1, p2, p3, p4, p5, p6, p7, p8, p9
10.7	Robust normal equivalence class	12	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11
10.7	Decision table	12	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11
10.8	Hybrid specification-based (extended to all age classes)	32	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11

10.4.2.3 Slice Testing

Slice testing does not provide much insight either. Four slices are of interest:

S(safeDrivingReduction, 33) = {1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 17, 18, 19, 20, 22, 23, 24, 25, 27, 28, 29, 32}
S(ageMultiplier, 33) = {1, 2, 3, 4, 5, 6, 10, 11, 15, 16, 20, 21, 25, 26, 32}
S(baseRate, 33) = {1}
S(Premium, 33) = {1, 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, 28, 29, 32}

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io

The union of these slices is the whole program. The only insight we might get from slice-based testing is that, if a failure occurred at line 33, the slices on safeDrivingReduction and ageMultiplier separate the program into two disjoint pieces, and that would simplify fault isolation.

10.5 Guidelines

Here is one of my favorite testing stories. An inebriated man was crawling around on the sidewalk beneath a streetlight. 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 specification-based 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" to help with this:

1. If the variables refer to physical quantities, boundary value testing and equivalence class testing are indicated.
2. If the variables are independent, boundary value 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.

Combinations of these may occur; therefore, the guidelines are summarized as a decision table in Table 10.13.

© 2010 Taylor & Francis Group, LLC
https://hemanthrajhemu.github.io

Table 10.13 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		×								
a2	Robustness testing	×									
a3	Worst-case testing				×						
a4	Robust worst case			×							
a5	Weak robust equivalence class	×		×			×		×		
a6	Weak normal equivalence class	×	×				×	×			
a7	Strong normal equivalence class			×	×	×			×	×	×
a8	Decision table					×					×

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 8.
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 100. 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?

References

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

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

© 2010 Taylor & Francis Group, LLC

https://hemanthrajhemu.github.io