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.

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.

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.)

```
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
```

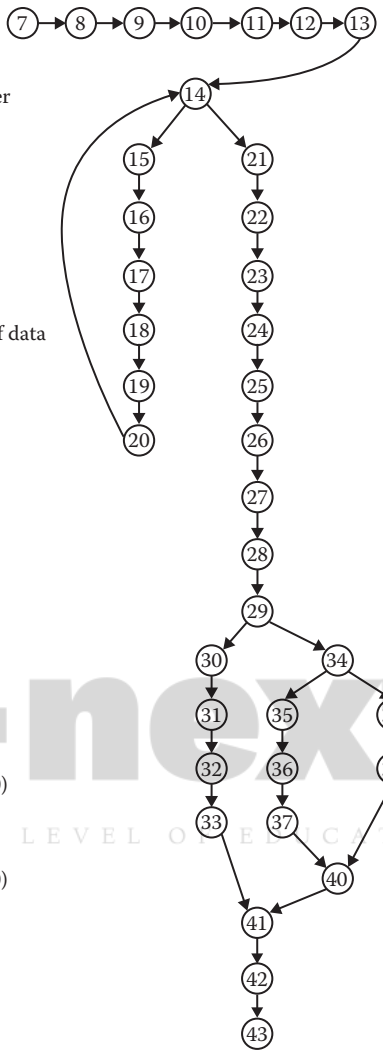

Figure 9.1 Commission problem and its program graph.

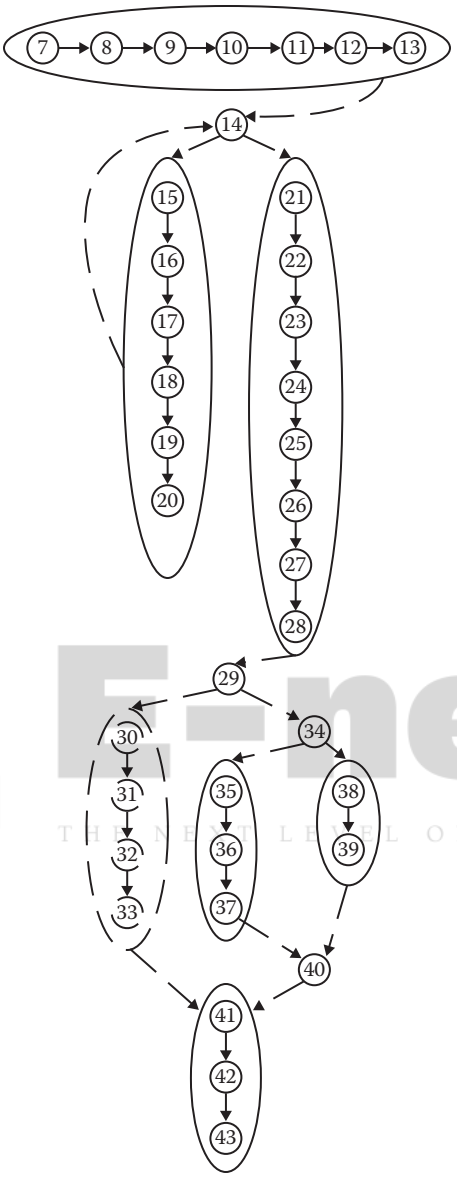

Figure 9.2 DD-path graph of commission problem pseudocode (in Figure 9.1).

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.

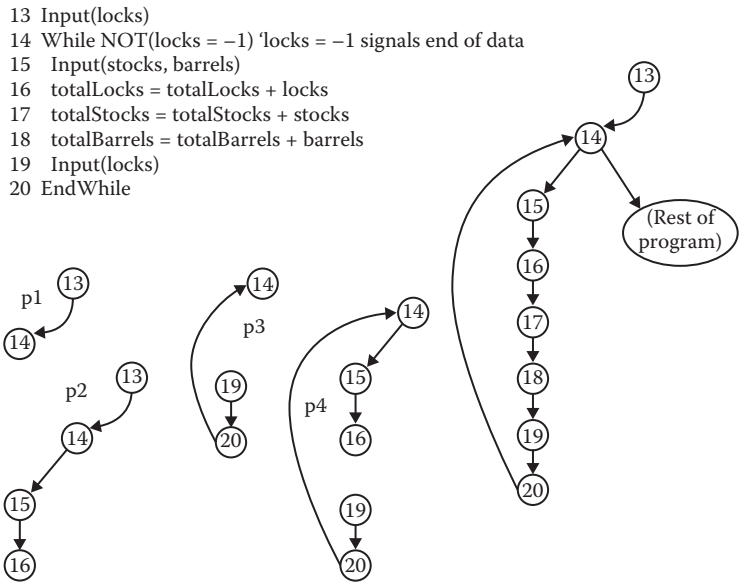

Figure 9.3 Du-paths for locks.

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

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

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

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>

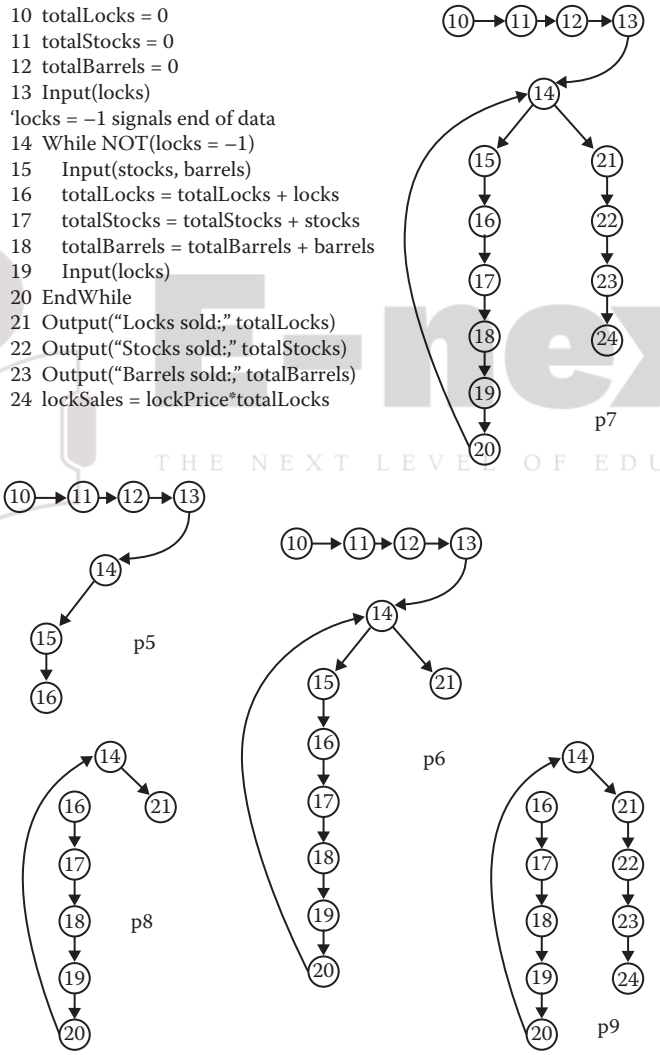

Figure 9.4 Du-paths for totalLocks.

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.

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.

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.

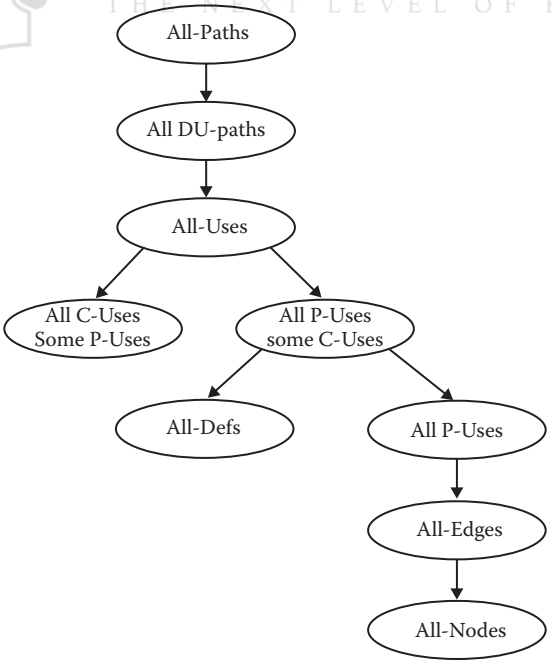

Figure 9.5 Rapps–Weyuker hierarchy of data flow coverage metrics.

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.

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

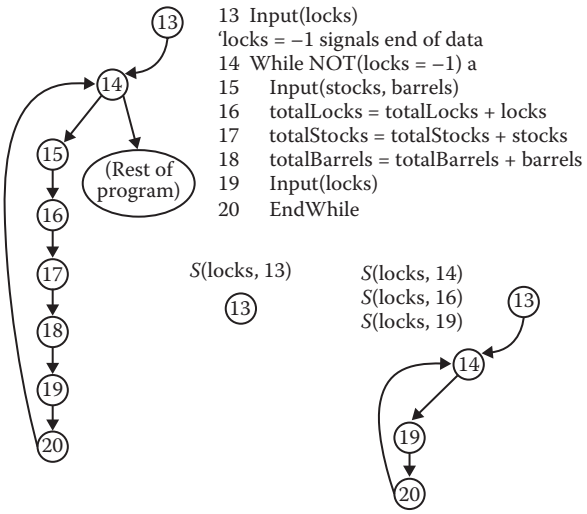

Figure 9.6 Selected slices on locks.

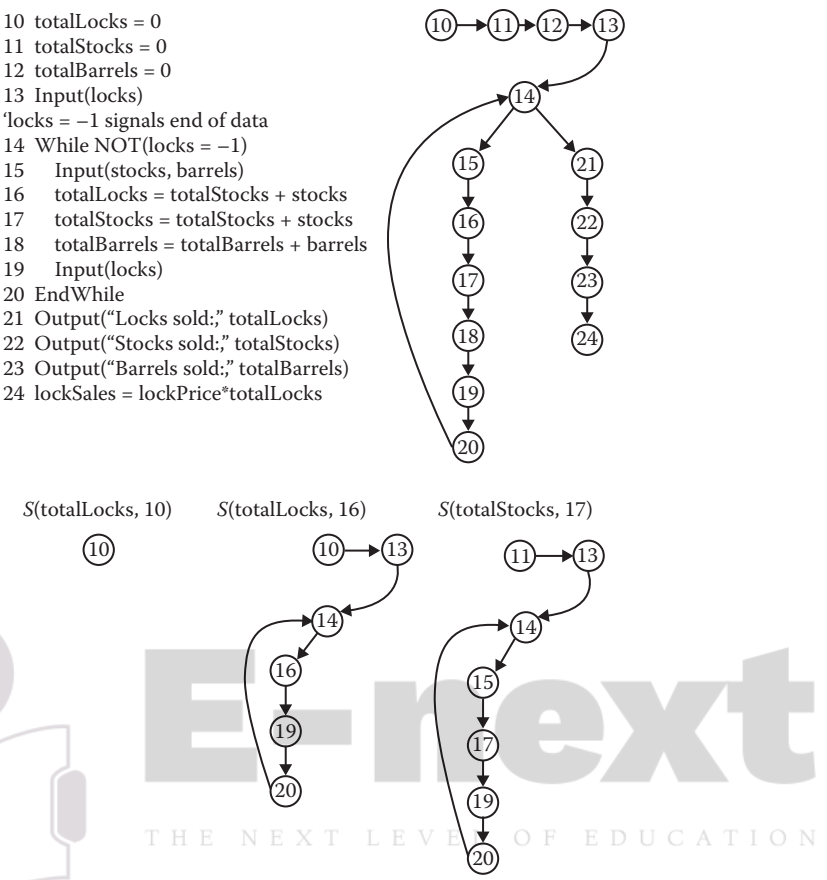

Figure 9.7 Selected slices in a loop.

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

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)

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}

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}

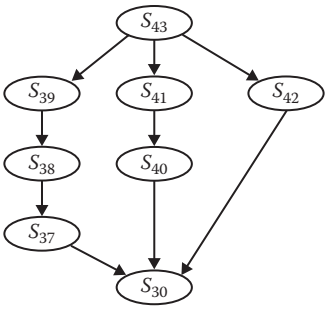

Figure 9.8 Partial lattice of slices on commission.

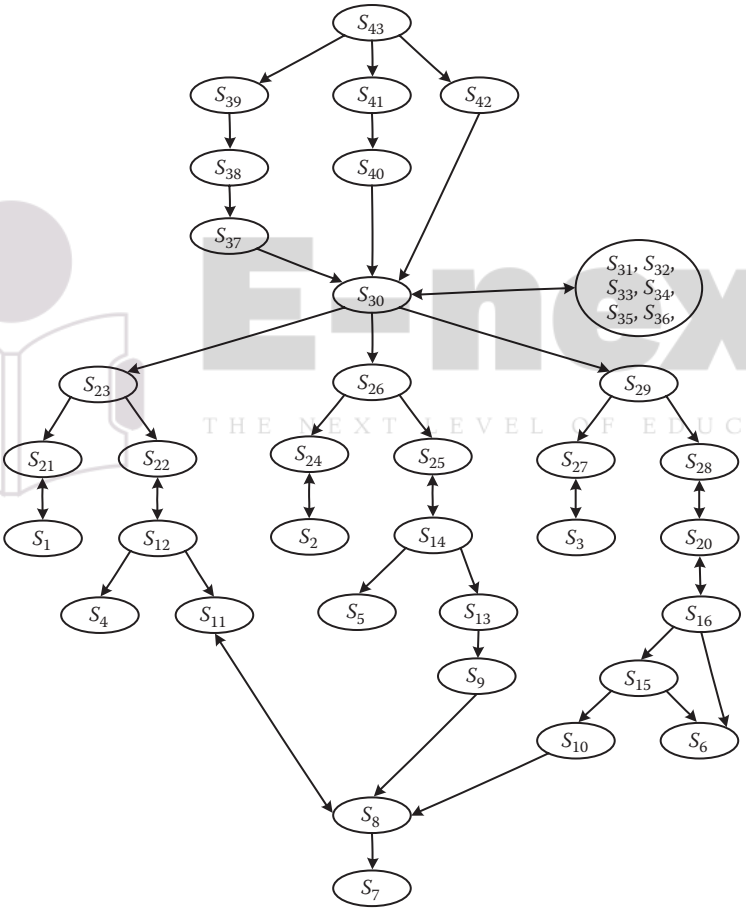

Figure 9.9 Full lattice on commission.

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}

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.

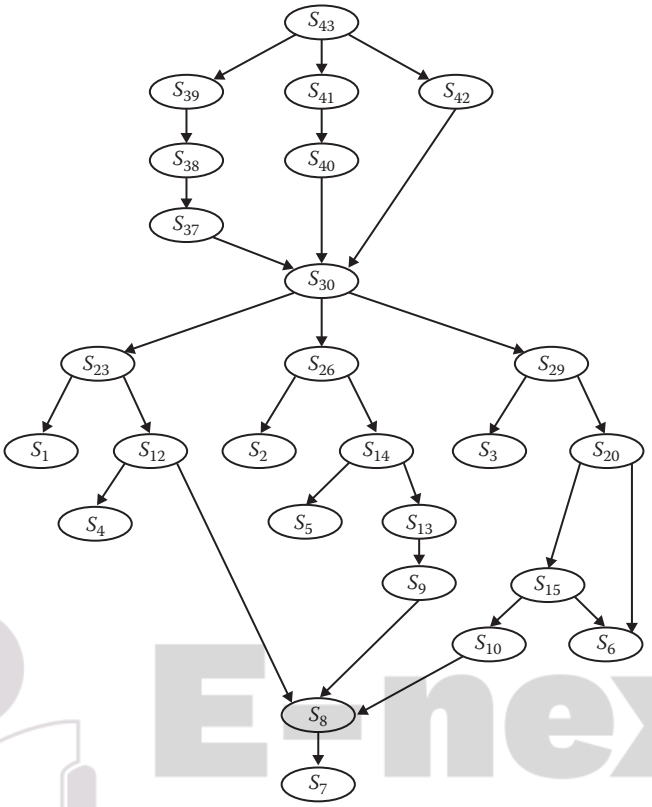

Figure 9.10 Simplified lattice on commission.

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.

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)
```

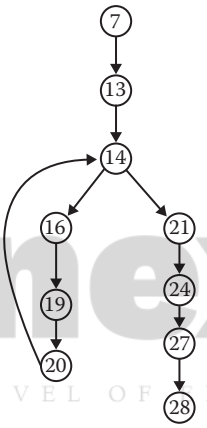

Figure 9.11 Slice 1.

```
 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)
```

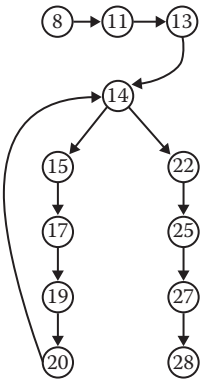

Figure 9.12 Slice 2.

```
 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)
```

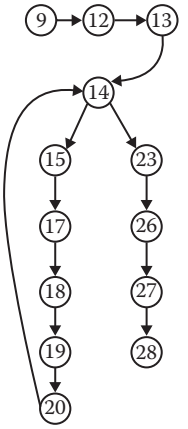

Figure 9.13 Slice 3.

```
 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
```

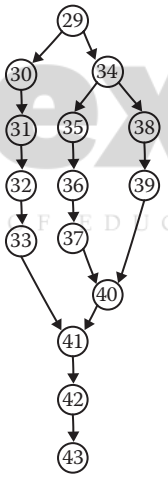

Figure 9.14 Slice 4.

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.

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.