<ch4 toc ch6>

Chapter 5
DATA-FLOW TESTING

1. SYNOPSIS

Data-flow testing uses the control flowgraph to explore the unreasonable things that can happen to data **(data-flow anomalies).** Consideration of data-flow anomalies leads to test path selection strategies that fill the gaps between complete path testing and branch and statement testing. Comparisons of strategies, generalization, tools, and effectiveness.

2. DATA-FLOW TESTING BASICS

2.1. Motivation and Assumptions

2.1.1. What Is It?

Data-flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects. For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something.

2.1.2. Motivation

I can't improve on Rapps and Weyuker's (RAPP82) eloquent motivation for data-flow testing:

> "It is our belief that, just as one would not feet confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation."

For more motivation, reread Section 3.4, Chapter 2. *At least half of contemporary source code consists of data declaration statements—that is, statements that define data structures, individual objects, initial or default values, and attributes.* To the extent that we achieve the widely sought goal of reusable code, we can expect the balance of source code statements to shift ever more toward data statement domination.

In all known hardware technologies, memory components have been, are, and are expected to be cheaper than processing components. A flipflop (basic storage element) requires two transistors, but I bit of an adder takes six transistors.* Another advantage to memory is geometrical regularity. Memory chips are regular—they look as if they were woven—conversely, logic elements are more haphazard.** The memory units' geometric regularity permits tighter packing and therefore more elements per chip. This fact will, as it has in the past, continue to push software into data-dominated designs.

---

*Memory elements can be built with one transistor (or none) and adders can do with less than six, but for all known technologies, whatever design wizardry is applied, memory is cheaper than processing.

**Programmed logic arrays (PLAs) are geometrically regular, but they pay for that by effectively discarding many of their components, and therefore they can't achieve as high an effective density as memory.

2.1.3. New Paradigms—Data-Flow Machines

Low-cost computational and memory elements have made possible massively parallel machines that can break the time logjam of current architectures. Most computers today are **Von Neumann machines.** This architecture features interchangeable storage of instructions and data in the same memory units. The Von Neumann architecture executes one instruction at a time in the following, typical, microinstruction sequence:

1. Fetch instruction from memory.
2. Interpret instruction.
3. Fetch operand(s).
4. Process (execute).
5. Store result (perhaps in registers).
6. Increment program counter (pointer to next instruction).
7. GOTO 1.

The pure Von Neumann machine has only one set of control circuitry to interpret the instruction, only one set of registers in which to process the data, and only one execution unit (e.g., arithmetic/logic unit). This design leads to a sequential, instruction-by-instruction execution, which in turn leads to control-flow dominance in our thinking. The Von Neumann machine forces sequence onto problems that may not inherently be sequential.

Massively parallel **(multi-instruction, multidata—MIMD)** machines, by contrast, have multiple mechanisms for executing steps 1–7 above and can therefore fetch several instructions and/or objects in parallel. They can also do arithmetic or logical operations simultaneously on different data objects. While such machines are still in the R&D phase, it is clear that they are dominated by data-flow thinking to which control flow takes a back seat (OXLE84). Even though you may not now be programming a real MIMD machine, current language and compiler development trends are toward abstract MIMD machines in which the decision of how to sequence parallel computation steps are left to the compiler (SHIY87). Just as we found that compilers do a better job (on average) than humans at assigning registers, we can expect such compilers to do a better job of sequencing. If the implementation is for an MIMD machine, then the compiler will produce parallel (data-flow) instructions while for a conventional machine it will produce sequential instructions. In other words, the sequential (Von Neumann) machine is the special case of a parallel machine with only one processor. It is reasonable to expect languages to evolve toward the support of the more general case. Whether compiled to one or many processing units, from the programmer's point of view, it will be data-flow software that has to be tested.

Figure 5.1. Von

Given L, t, and d, solve for Z and H_c.

Neumann Navigation Calculation PDL.

$$\cos C = \cos L \sin t$$
$$\tan M = \cot L \cos t$$
$$\tan (Z + F) = -\sin L \tan t$$
$$\tan F = \cos C \tan (M + d)$$
$$\sin H_c = \sin C \sin (M + d)$$
$$Z = (Z + F) - F$$

```
t1 := cot L
t2 := cos t
t3 := t1*t2          */ tan M /*
t1 := tan^-1 t3      */ M /*
t1 := t1 + d         */ M + d /*
t2 := -sin L
t3 := tan t
t2 := t2*t3          */ tan (Z + F) /*
t2 := tan^-1 t2      */ Z + F /*
t3 := cos L
t4 := sin t
t3 := t3*t4          */ cos C /*
t4 := tan t1         */ tan (M + d) /*
t4 := t3*t4          */ tan F /*
t4 := tan^-1 t4      */ F /*
Z := t2 - t4
t3 := cos^-1 t3      */ C /*
t3 := sin t3         */ sin C /*
t1 := sin t1         */ sin (M + d) /*
Hc := t1*t3          */ sin Hc /*
Hc := sin^-1 Hc
```

Figure 5.1 shows the PDL for solving some navigation equations (BOWD77) on a Von Neumann machine that has a coprocessor to calculate trigonometric functions. The control flowgraph corresponding to Figure 5.1 is trivial: it has exactly one link. The implementation shown requires twenty-one steps and four temporary memory locations. Figure 5.2 shows a data-flow PDL for solving the same set of equations. A corresponding data flowgraph is shown in Figure 5.3. The “PAR DO” notation means that the calculations within its scope (PAR MEND PAR) can be done in parallel. The nodes, which I’ve shown as expressions, denote operations (for example, tan(x), *, sin(x)) and the links denote that the result of the operation at the arrow’s tail is needed for the operation at the arrow’s head. This flowgraph is simplified because we’re really doing two things at each node: an assignment and an arithmetic or trigonometric operation. As always when we have a graph, we should think about ways to cover it.

Figure 5.2. Data-flow Machine Navigation Calculation PDL.

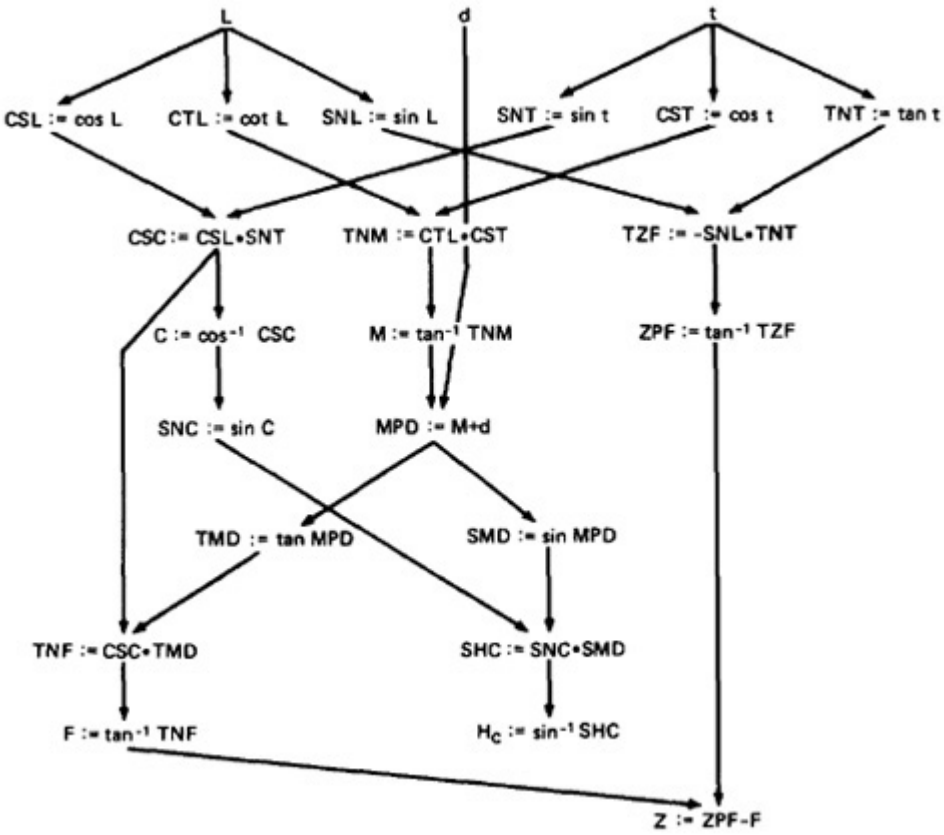
Given L, t, and d, solve for Z and H_c.

$$\cos C = \cos L \sin t$$
$$\tan M = \cot L \cos t$$
$$\tan (Z + F) = -\sin L \tan t$$
$$\tan F = \cos C \tan (M + d)$$
$$\sin H_c = \sin C \sin (M + d)$$
$$Z = (Z + F) - F$$

```
PAR DO: CSL  := cos L   SNT := sin t
        CTL  := cot L   CST := cos t
        SNL  := sin L   TNT := tan t
END PAR:
PAR DO: CSC  := CSL*SNT     */ cos C /*
        TNM  := CTL*CST     */ tan M /*
        TZF  := -SNL*TNT    */ tan (Z+F) /*
END PAR:
PAR DO:  C   := cos⁻¹  CSC   */ C /*
         M   := tan⁻¹  TNM   */ M /*
        ZPF  := tan⁻¹  TZF   */ Z + F /*
END PAR
PAR DO: MPD := M + d      */ M + d /*
        SNC := sin C      */ sin C /*
END PAR:
PAR DO: TMD := tan MPD    */ tan (M + d) /*
        SMD := sin MPD    */ sin (M + d) /*
END PAR:
PAR DO: TNF  := CSC*TMD   */ tan F /*
        SHC  := SNC*SMD   */ sin Hc /*
END PAR:
PAR DO:   Hc := sin⁻¹  SHC
          F  := tan⁻¹  TNF   */ F /*
END PAR:
          Z  := ZPF - F
```

One of the advantages of the data flowgraph (as we saw for transaction flows) is that there's no restriction to uniprocessing as there is for the control flowgraph. There is a control flow here—actually, there are several independent, parallel streams. When we go back to the Von Neumann machine for the rest of this book, we'll be forcing these control streams into one sequential control flow as usual. The Von Neumann machine implementation shown in Figure 5.1 is not unique. There are thousands of different orders, all correct, in which the operations could have been done. There might be a different control flowgraph for each such variant, but the underlying relation between the data objects shown in Figure 5.3 would not change. From our (Von Neumann) perspective, the kind of data flowgraph shown in Figure 5.3 is a specification of relations between objects. When we say that we want to cover this graph, we mean that we want to be sure that all such relations have been explored under test.

Figure 5.3. Data Flowgraph for Figure 5.2 Navigation Calculations.



2.1.4. The Bug Assumptions

The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects are not available when they should be, or silly things are being done to data objects. Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.

2.2. Date Flowgraphs

2.2.1. General

The **data flowgraph** is a graph consisting of nodes and **directed links** (i.e., links with arrows on them). Although we'll be doing data-flow testing, we won't be using data flowgraphs as such. Rather, we'll use an ordinary control flowgraph annotated to show what happens to the data objects of interest at the moment. If we were using a data-flow machine or language, the graph of Figure 5.3 would be the graph we would work with directly. With a Von Neumann machine, our objective is to expose deviations between the data flows we have (however they be implemented) and the data flows we want—such as that which might be specified by Figure 5.3.

2.2.2. Data Object State and Usage

Data objects can be created, killed and/or used. They can be used in two distinct ways: in a calculation or as part of a control flow predicate. The following symbols* denote these possibilities:

---

[*] I don't like these particular symbols because the literature is contradictory. Some authors use *u* to mean "undefined," *c* for "control use," *p* for "processing use," and *r* for "reference"; some use other conventions. Like it or not, the above notation is dominant.

---

d—defined, created, initialized, etc.
k—killed, undefined, released.
u—used for something.
c—used in a calculation.
p—used in a predicate.

1. *Defined*—An object is **defined** explicitly when it appears in a data declaration or implicitly (as in FORTRAN) when it appears on the left-hand side of an assignment statement. "Defined" can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, and so on. Be broad in your interpretation of "defined" or "create" and you'll find more applications to data-flow testing than you might think of at first.
2. *Killed or Undefined*—An object is **killed** or **undefined** when it is released or otherwise made unavailable, or when its contents are no longer known with certitude. For example, the loop control variable in FORTRAN is undefined when the loop is exited; release of dynamically allocated objects back to the availability pool is "killing" or "undefining"; return of records; the old top of the stack after it is popped; a file is closed. Note that an assignment statement can simultaneously kill and (re)define. For example, if A had been previously defined and we do a new assignment such as A : = 17, we have killed A's previous value and redefined A. A very precise model would denote a typical assignment statement by *kd*. In practice, however, because killing the old value is usually implicit in an assignment statement and because a use between the (implicit) *k* and the *d* is impossible, we can drop the *k* and use a simple *d* to model assignment statements.
Define and kill are complementary operations. That is, they generally come in pairs and one does the opposite of the other. When you see complementary operations on data objects it should be a signal to you that a data-flow model, and therefore data-flow testing methods, might be effective.
3. *Usage*—A variable is used for **computation** (*c*) when it appears on the right-hand side of an assignment statement, as a pointer, as part of a pointer calculation, a file record is read or written, and so on. It is **used** in a **predicate** (*p*) when it appears directly in a predicate (for example, IF A>B . . .), but also implicitly as the control variable of a loop, in an expression used to evaluate the control flow of a case statement, as a pointer to an object that will be used to direct control flow. Predicate usage does not preclude computational use, or vice versa; in some languages, usage can be for both predicate and computation simultaneously—for example, a test-and-clear machine language instruction.

2.2.3. Data-Flow Anomalies

There are as many notions of data-flow anomalies as there are modelers. The notions presented here are by consensus those considered to be most useful. An anomaly is denoted by a two-character sequence of actions. For example, *ku* means that the object is killed and then used (possible in some languages), whereas *dd* means that the object is defined twice without an intervening usage. What is an anomaly may depend on the application. For example, the sequence

```
A:= C + D
IF A>O THEN X :=1 ELSE X :=-1
A:= B + C
```

seems reasonable because it corresponds to *dpd* for variable A, but in the context of some secure systems it might be objectionable because the system doctrine might require several assignments of A to zero prior to reuse. For example,

```
A:= C + D
IF A>O THEN X:=1 ELSE X:= -1
A:= 0
A:= 0
A:= 0
A:= B + C
```

There are nine possible two-letter combinations for *d, k* and *u*. Some are bugs, some are suspicious, and some are okay.

dd—probably harmless but suspicious. Why define the object twice without an intervening usage?
dk—probably a bug. Why define the object without using it?
du—the normal case. The object is defined, then used.
kd—normal situation. An object is killed, then redefined.
kk—harmless but probably buggy. Did you want to be sure it was really killed?
ku—a bug. The object doesn't exist in the sense that its value is undefined or indeterminate. For example, the loop-control value in a FORTRAN program after exit from the loop.
ud—usually not a bug because the language permits reassignment at almost any time.
uk—normal situation.
uu—normal situation.

In addition to the above two-letter situations there are six single-letter situations. We'll use a leading dash to mean that nothing of interest (*d,k,u*) occurs prior to the action noted along the entry-exit path of interest and a trailing dash to mean that nothing happens after the point of interest to the exit.

–*k*: possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We're killing a variable that does not exist; but note that the variable might have been created by a called routine or might be global.
–*d*: okay. This is just the first definition along this path.
–*u*: possibly anomalous. Not anomalous if the variable is global and has been previously defined.
k–: not anomalous. The last thing done on this path was to kill the variable.
d–: possibly anomalous. The variable was defined and not used on this path; but this could be a global definition or within a routine that defines the variables for other routines.
u–: not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If *d* and *k* mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use—not a bug if we expect some other routine to return it.

The single-letter situations do not lead to clear data-flow anomalies but only the possibility thereof. Also, whether or not a single-letter situation is anomalous is an integration testing issue rather than a component testing issue because the interaction of two or more components is involved. Although the easier data-flow anomalies can be exhibited by testing a single component, the more difficult (and more frequent) anomalies are those that involve several components—i.e., integration data-flow bugs.

2.2.4. Data-Flow Anomaly State Graph

Our data-flow anomaly model prescribes that an object can be in one of four distinct states:

K—undefined, previously killed, does not exist.
D—defined but not yet used for anything.
U—has been used for computation or in predicate.
A—anomalous.

Don't confuse these capital letters (K,D,U,A), which denote the state of the variable, with the program action, denoted by lowercase letters (*k,d,u*). Figure 5.4 (after HUAN79) shows what I call the "unforgiving model," because it holds that once a variable becomes anomalous it can never return to a state of grace.*

*This state graph doesn't support the single-letter anomalies. We'd have to add more states to do that.

My data-flow anomaly state graph differs from Huang's because I call the *kk* sequence anomalous, whereas he does not. Assume that the variable starts in the K state—that is, it has not been defined or does not exist. If an attempt is made to use it or to kill it (e.g., say that we're talking about opening, closing, and using files and that "killing" means closing), the object's state becomes anomalous (state A) and, once it is anomalous, no action can return the variable to a working state. If it is defined (*d*), it goes into the D, or defined but not yet used, state. If it has been defined (D) and redefined (*d*) or killed without use (*k*), it becomes anomalous, while usage (*u*) brings it to the U state. If in U, redefinition (*d*) brings it to D, *u* keeps it in U, and *k* kills it.

Huang (HUAN79) has a more forgiving alternate model (Figure 5.5). This graph has three normal and three anomalous states and he considers the *kk* sequence not to be anomalous. The difference between this state graph and Figure 5.4 is that redemption is possible. A proper action from any of the three anomalous states returns the variable to a useful working state. The point of showing you this alternative anomaly state graph is to demonstrate that the specifics of an anomaly depends on such things as language, application, context, or even your frame of mind. In principle, you must create a new definition of data flow anomaly (e.g., a new state graph) in each situation. You must at least verify that the anomaly definition behind the theory or imbedded in a data flow anomaly test tool is appropriate to your situation.

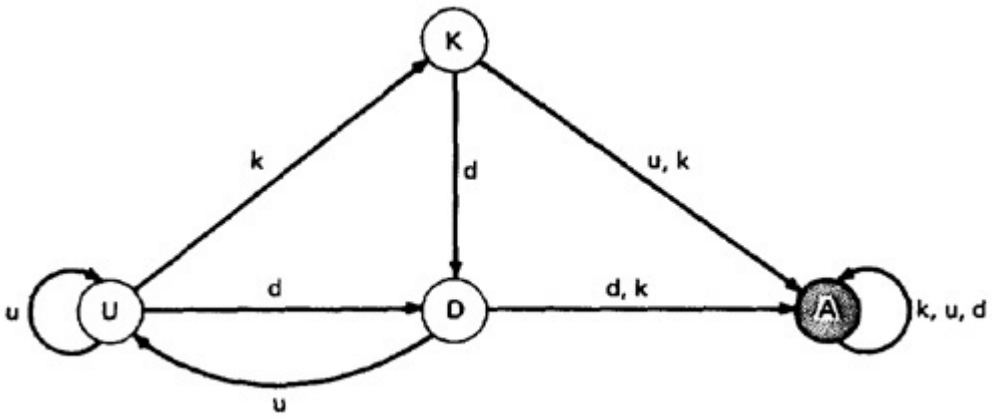


Figure 5.4. Unforgiving Data Flow Anomaly State Graph.

2.2.5. Static Versus Dynamic Anomaly Detection

Static analysis is analysis done on source code without actually executing it (GHEZ81). **Dynamic analysis** is done on the fly as the program is being executed and is based on intermediate values that

result from the program's execution. Source-code syntax error detection is the archetypal static analysis result, whereas a division by zero warning is the archetypal dynamic analysis result. If a problem, such as a data-flow anomaly, can be detected by static analysis methods, then it does not belong in testing—it belongs in the language processor.

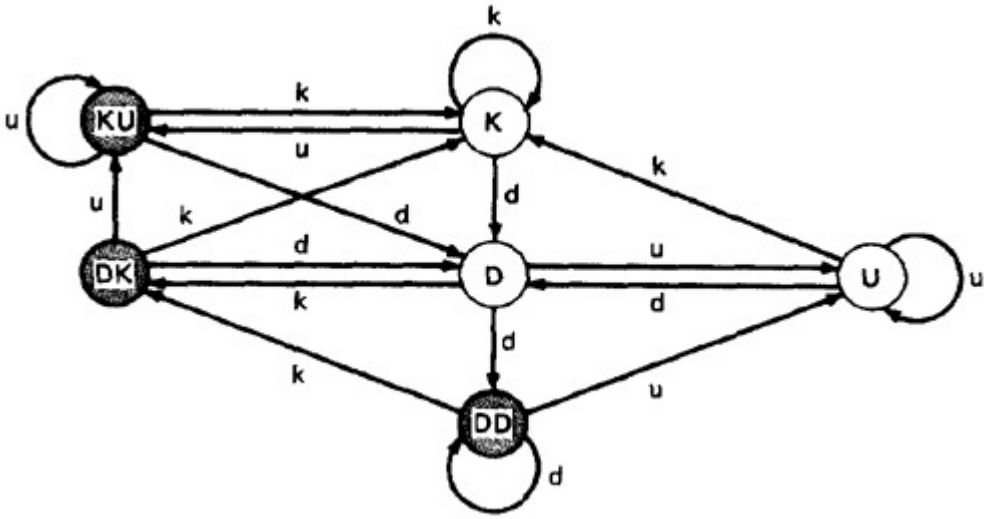


Figure 5.5. Forgiving Data Flow Anomaly State Graph.

There's actually a lot more static analysis for data flow anomalies going on in current language processors than you might realize at first. Languages which force variable declarations (e.g., Pascal) can detect *–u* and *ku* anomalies and optimizing compilers can detect some (but not all) instances of dead variables. The run-time resident portion of the compiler and/or the operating system also does dynamic analysis for us and therefore helps in testing by detecting anomalous situations. Most anomalies are detected by such means; that is, we don't have to put in special software or instrumentation to detect an attempt, say, to read a closed file, but we do have to assure that we design tests that will traverse paths on which such things happen.

Why isn't static analysis enough? Why is testing required? Could not a vastly expanded language processor detect a anomalies? No. The problem is provably unsolvable. Barring unsolvability problems, though, there are many things for which current notions of static analysis are inadequate.

1. *Dead Variables*—Although it is often possible to prove that a variable is dead or alive at a given point in the program (in fact, optimizing compilers depend on being able to do just that), the general problem is unsolvable.
2. *Arrays*—Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array. Array pointers are usually dynamically calculated, so there's no way to do a static analysis to validate the pointer value. And even if the pointer is within bounds, how do we know that the specific array element accessed has been initialized? In many languages, dynamically allocated arrays contain garbage unless explicitly initialized and therefore, *–u* anomalies are possible. In Rocky Mountain Basic for example, arrays can be dynamically redimensioned without losing their contents. This feature is very handy, but think of what that does to static validation of array pointer values.
3. *Records and Pointers*—The array problem and the difficulty with pointers is a special case of multipart data structures. We have the same problem with records and the pointers to them. Also, in many applications we create files and their names dynamically and there's no way to determine, without execution, whether such objects are in the proper state on a given path or, for that matter, whether they exist at all.

4. *Dynamic Subroutine or Function Names in a Call*—A subroutine or function name is a dynamic variable in a call. What is passed, or a combination of subroutine names and data objects, is constructed on a specific path. There's no way, without executing the path, to determine whether the call is correct or not.
5. *False Anomalies*—Anomalies are specific to paths. Even a "clear bug" such as *ku* may not be a bug if the path along which the anomaly exist is unachievable. Such "anomalies" are **false anomalies.** Unfortunately, the problem of determining whether a path is or is not achievable is unsolvable.
6. *Recoverable Anomalies and Alternate State Graphs*—What constitutes an anomaly depends on context, application, and semantics. Huang provided two anomaly state graphs (Figures 5.4 and 5.5), but I didn't agree with his first one so I changed it. The second graph is also a good model, one based on the idea that anomaly recovery is possible. How does the compiler know which model I have in mind? It can't because the definition of "anomaly" is not fundamental. The language processor must have a built-in anomaly definition with which you may or may not (with good reason) agree.
7. *Concurrency, Interrupts, System Issues*—As soon as we get away from the simple single-task uniprocessor environment and start thinking in terms of systems, most anomaly issues become vastly more complicated. How often do we define or create data objects at an interrupt level so that they can be processed by a lower-priority routine? Interrupts can make the "correct" anomalous and the "anomalous" correct. True concurrency (as in an MIMD machine) and pseudoconcurrency (as in multiprocessing) systems can do the same to us. Much of integration and system testing is aimed at detecting data-flow anomalies that cannot be detected in the context of a single routine.

Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods, especially for data flow anomaly detection. That's good because it means there's less for us to do as testers and we have far too much to do as it is.

2.3. The Data-Flow Model

2.3.1. General

Our data-flow model is based on the program's control flowgraph—don't confuse that with the program's data flowgraph. We annotate each link with symbols (for example, *d, k, u, c, p*) or sequences of symbols (for example, *dd, du, ddd*) that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called **link weights.** There is a (possibly) different set of link weights for every variable and for every element of an array. The control flowgraph's structure (that is, the nodes and the links that connect them) is the same for every variable: it is the weights that change.*

---

*Readers familiar with the data-flow testing literature may note (and object) that I assign the weights to the links rather than to the nodes, contrary to the usual practice, which assigns weights to nodes, except predicate uses, which are assigned to outlinks. The model described in Section 2.3.2 results in equivalent graphs because it is always possible to convert a node-weighted model to a link-weighted model, or vice versa. My use of link weights rather than node weights is not capricious—it is to provide consistency with the general node reduction algorithm of Chapters 8 and 12. I'd rather, especially in Chapter 12, use only relation matrices. Node weights would have forced me to introduce incidence matrices in addition to relation matrices in a text already overloaded with abstractions.

2.3.2. Components of the Model

Here are the modeling rules:

1. To every statement there is a node, whose name (number) is unique. Every node has at least one outlink and at least one inlink except exit nodes, which do not have outlinks, and entry nodes, which do not have inlinks.
2. **Exit nodes** are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, **entry nodes** are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.
3. The outlink of **simple statements** (statements with only one outlink) are weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter. For example, the assignment statement A:= A + B in most languages is weighted by *cd* or possibly *ckd* for variable A. Languages that permit multiple simultaneous assignments and/or compound statements can have anomalies within the statement. The sequence must correspond to the order in which the object code will be executed for that variable.
4. **Predicate nodes** (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the *p*-use(s) on *every* outlink, appropriate to that outlink.
5. Every sequence of simple statements (e.g., a sequence of nodes with one infink and one outlink) can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.
6. If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.
7. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data-flow action for any variable. In all that follows, from the point of view of discussing the strategies, we'll assume that such a transformation has been done.*

---

*The strategies, as defined by Weyuker et al. (FRAN88, RAPP82, RAPP85) are defined over a model programming language that avoids complexities met in many real languages. For example, anomalies within a statement are effectively forbidden, as is isolated code. Although the strategies are developed, and theorems proved over a simplified language, it does not obviate the utility of the strategies when applied to real languages.

2.3.3. Putting It Together

Figure 5.6a shows the control flowgraph that we first saw as Figures 3.2 to 3.6 in Chapter 3. I've kept the node labels and marked the decision nodes with the variables in the control-flow predicate. We don't need the actual predicate here—it's enough to know the names of the predicate variables. Also, the nodes are numbered so that we can more easily refer to them and to the intervening links.

Figure 5.6b shows this control flowgraph annotated for variables X and Y data flows (they're identical). There's a *dcc* on the first link (1,3) and nothing else because nothing else is done with these variables. I've assumed that killing is implicit in assignment and that explicit killing is not necessary.

Figure 5.6c shows the same control flowgraph annotated for variable Z. Z is first defined by an assignment statement on the first link. Z is used in a predicate (Z >= 0?) at node 3, and therefore both outlinks of that node—(3,4) and (3,5)—are marked with a *p*. Link (4,5) has Z := Z–1, which is a computational use followed by a definition and hence the *cd* annotation. There are two more instances of predicate use—at nodes 8 and 9.

The data-flow annotation for variable V is shown in Figure 5.6d. Don't confuse array V() with variable V; similarly for U() versus U. The assignment statement V(U),U(V) := (Z+V)*U (see Figure 3.2) is compound. I've modeled it as two separate statements with the result that there are a total of three *c*-uses of this variable at link (6,7): two on the right-hand side of the assignment and once for the array pointer.

If there aren't too many variables of interest, you can annotate the control flowgraph by using the standard data-flow actions with subscripts to denote the variables. For example: d_x, k_x, u_x, c_x, and p_x denote the actions for variable x, whereas d_y, k_y, u_y, c_y, and p_y denote the actions for variable y.

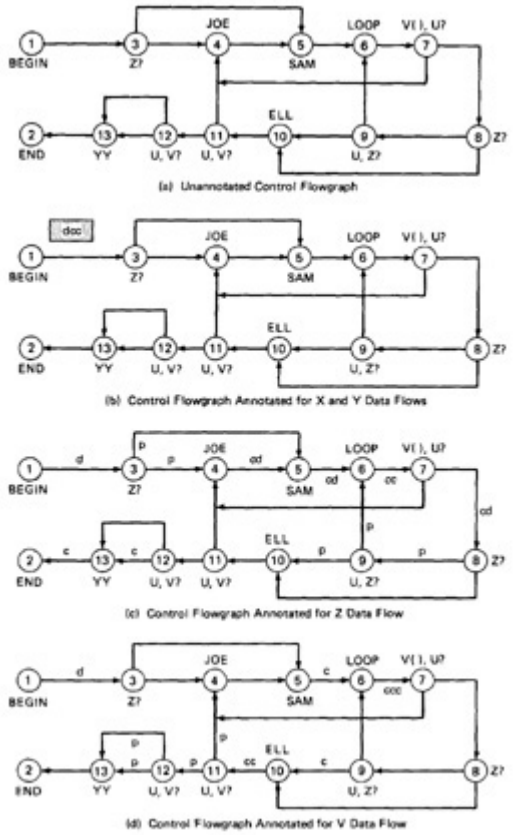


Figure 5.6. Control Flowgraphs.

3. DATA-FLOW TESTING STRATEGIES

3.1. General

Data-flow testing strategies are structural strategies. We saw in Chapter 3, Section 2.7, how to construct a family of structural testing strategies based on the length (in links or traversed nodes) of path segments to be used to construct test paths. This is only one family of an infinite number of families of possible test strategies. Ntafos (NTAF84B) generalizes the idea by defining **required element testing** as a way of generating a family of test strategies based on a structural characterization of the way test cases are to be defined (i.e., how we pick nodes, links, and/or sequences of nodes or links to be included in a test case) and a functional characterization that test cases must satisfy. Pure structural and functional testing as well as hybrid strategies can be defined within his conceptual framework.

Restricting ourselves to structural strategies, any algorithm for selecting links and/or nodes defines a corresponding (possibly useless) test strategy. In path-testing strategies, the only structural characteristic used was the raw program-control flowgraph without consideration of what happened on those links. In

other words, nodes and links are considered to have no property other than the fact that they exist. Higher-level path-testing strategies based, say, on adjacent link pairs or triplets take more of the control-flow structure into account, but still no other information than is implicit in the control flowgraph.

In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph. In other words, data flow strategies require data-flow link weights (*d,k,u,c,p*).* Data-flow testing strategies are based on selecting test path segments (also called **subpaths**) that satisfy some characteristic of data flows for all data objects. For example, all subpaths that contain a *d* (or *u, k, du, dk*). Given a rule for selecting test path segments, a major objective of testing research has been to determine the relative strength of the strategy corresponding to that rule—that is, to find out whether it is stronger or weaker than some other strategy, or incomparable. A strategy X is **stronger** than another strategy Y if all test cases produced under Y are included in those produced under X—conversely for **weaker**. All structural strategies are weaker than total path testing. Data-flow testing strategies provide one set of families that fill the gap.

*Other structural testing strategies could require some other property that can be expressed by link weights. For example, suppose our testing concern is the mean execution time of the routine and that passing or failing is determined by whether or not timing objectives are met. In that case, the link weights would be link execution time and branch probabilities (see Chapter 8, Section 5.5.3).

3.2. Terminology

We'll assume for the moment that all paths are achievable. Some terminology:

1. A **definition-clear path segment*** (with respect to variable X) is a connected sequence of links such that X is (possibly) defined on the first link and not redefined or killed on any subsequent link of that path segment. All paths in Figure 5.6b are definition clear because variables X and Y are defined only on the first link (1,3) and not thereafter. Similarly for variable V in Figure 5.6d. In Figure 5.6c, we have a more complicated situation. The following path segments are definition-clear: (1,3,4), (1,3,5), (5,6,7,4), (7,8,9,6,7), (7,8,9,10), (7,8,10), (7,8,10,11). Subpath (1,3,4,5) is not definition-clear because the variable is defined on (1,3) and again on (4,5). For practice, try finding all the definition-clear subpaths for this routine (i.e., for all variables).

*The definitions of this section are informal variants of those presented in the literature, which do not usually take into account the possibility that a sequence of data-flow actions can take place on a given link for a given variable. The theory, as developed (e.g., FRAN88, RAPP82, RAPP85) is over a language that does not permit such complications. The strategies are based on models for which one and only one data-flow action occurs on each link (actually, at each node) for any variable. Associating the data-flow action with the links rather than with the nodes (as we do) and allowing multiple data-flow actions per link complicates the model considerably but does not change the fundamental results of the theory. The more realistic model, with multiple data-flow actions on links, can be converted to the simpler model of the theory by just breaking up a link on which multiple actions take place into a sequence of equivalent links. For example, a *dcc* link becomes three successive links with a *d*, a *c* and a *c* action on them, respectively. Then the definitions, as given, apply. Predicate uses in the literature are always associated with links and, in this respect, the models coincide.

The fact that there is a definition-clear subpath between two nodes does not imply that all

subpaths between those nodes are definition-clear; in general, there are many subpaths between nodes, and some could have definitions on them and some not. Note that a definition-clear path segment does not preclude loops. For example, a loop consisting of links (*i,j*) and (*j,i*) could have a definition on (*i,j*) and a use on (*j,i*). Observe that if we have included such loop segments in a test, by the definition of definition-clear path segment, there is no need to go around again. Thus, although these strategies allow loops, the loops need be traversed at most once; therefore, the number of test paths is always finite. As a consequence, all of these strategies must be weaker than all paths because we can always "create" a bug that is manifested only after a loop has been iterated an arbitrarily high number of times.
We consider one variable (X) at a time and look at all the links on which X is defined. We want to examine interactions between a variable's definition and the subsequent uses of that definition. If there are several definitions without intervening uses along a given path, then there can be no interaction between the first definition and its uses on that path. The test criteria are based on the uses of a variable which can be reached from a given definition of that variable (if any).

2. A **loop-free path segment** is a path segment for which every node is visited at most once. Path (4,5,6,7,8,10) in Figure 5.6c is loop free, but path (10,11,4,5,6,7,8,10,11,12) is not because nodes 10 and 11 are each visited twice.

3. A **simple path segment** is a path segment in which at most one node is visited twice. For example, in Figure 5.6c, (7,4,5,6,7) is a simple path segment. A simple path segment is either loop-free or if there is a loop, only one node is involved.

4. A ***du* path** from node *i* to *k* is a path segment such that if the last link has a computational use of X, then the path is simple and definition-clear; if the penultimate node is *j*—that is, the path is (*i,p,q,...,r,s,t,j,k*) and link (*j,k*) has a predicate use—then the path from *i* to *j* is both loop-free and definition-clear.

3.3. The Strategies

3.3.1. Overview

The structural test strategies discussed below (FRAN88, RAPP82, RAPP85) are based on the program's control flowgraph. They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set. The strategies also differ as to whether or not all paths of a given type are required or only one path of that type—that is, all predicate uses versus at least one predicate use; all computational uses versus at least one computational use; both computational and predicate uses versus either one or the other. Not all variations are interesting, nor have all been investigated.

3.3.2. All-du Paths

The **all-*du*-paths (ADUP) strategy** is the strongest data-flow testing strategy discussed here. It requires that *every du* path from *every* definition of *every* variable to *every* use of that definition be exercised under some test. In Figure 5.6b, because variables X and Y are used only on link (1,3), any test that starts at the entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy). The situation for variable Z (Figure 5.6c) is more complicated because the variable is redefined in many places. For the definition on link (1,3) we must exercise paths that include subpaths (1,3,4) and (1,3,5). The definition on link (4,5) is covered by any path that includes (5,6), such as subpath (1,3,4,5,6, ...). The (5,6) definition requires paths that include subpaths (5,6,7,4) and (5,6,7,8). Variable V (Figure 5.6d) is defined only once on link (1,3). Because V has a predicate use at node 12 and the subsequent path to the end must be forced for both directions at node 12, the all-*du*-paths strategy for this variable requires that we exercise all loop-free entry/exit paths and at least one path that includes the loop caused by (11,4). Note that we must test paths that include both subpaths (3,4,5) and (3,5) even though neither of these has V definitions. They must be included because they provide

alternate *du* paths to the V use on link (5,6). Although (7,4) is not used in the test set for variable V, it will be included in the test set that covers the predicate uses of array variable V() and U.

The all-*du*-paths strategy *is* a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

3.3.3. All-Uses Strategy

Just as we reduced our ambitions by stepping down from all paths (P_∞) to branch coverage (P_2), say, we can reduce the number of test cases by asking that the test set include *at least one* path segment from every definition to every use that can be reached by that definition—this is called the **all-uses (AU) strategy**. The strategy is that *at least one* definition-clear path from *every* definition of *every* variable to *every* use of that definition be exercised under some test. In Figure 5.6d, ADUP requires that we include subpaths (3,4,5) and (3,5) in some test because subsequent uses of V, such as on link (5,6), can be reached by either alternative. In AU either (3,4,5) or (3,5) can be used to start paths, but we don't have to use both. Similarly, we can skip the (8,10) link if we've included the (8,9,10) subpath. Note the hole. We must include (8,9,10) in some test cases because that's the only way to reach the *c* use at link (9,10)—but suppose our bug for variable V is on link (8,10) after all? Find a covering set of paths under AU for Figure 5.6d.

3.3.4. All-p-Uses/Some-c-Uses and All-c-Uses/Some-p-Uses Strategies

Weaker criteria require fewer test cases to satisfy. We would like a criterion that is stronger than P_2 but weaker than AU. Therefore, select cases as for All (Section 3.3.3) except that if we have a predicate use, then (presumably) there's no need to select an additional computational use (if any). More formally, the **all-*p*-uses/some-*c*-uses (APU+C)** strategy is defined as follows: for every variable and every definition of that variable, include at least one definition-free path from the definition to every predicate use; if there are definitions of the variable that are not covered by the above prescription, then add computational-use test cases as required to cover every definition. The **all-*c*-uses/some-*p*-uses (ACU+P)** strategy reverses the bias: first ensure coverage by computational-use cases and if any definition is not covered by the previously selected paths, add such predicate-use cases as are needed to assure that every definition is included in some test.

In Figure 5.6b, for variables X and Y, any test case satisfies both criteria because definition and uses occur on link (1,3). In Figure 5.6c, for APU+C we can select paths that all take the upper link (12,13) and therefore we do not cover the *c*-use of Z: but that's okay according to the strategy's definition because every definition is covered. Links (1,3), (4,5), (5,6), and (7,8) must be included because they contain definitions for variable Z. Links (3,4), (3,5), (8,9), (8,10), (9,6), and (9,10) must be included because they contain predicate uses of Z. Find a covering set of test cases under APU+C for all variables in this example—it only takes two tests. In Figure 5.6d, APU+C is achieved for V by (1,3,5,6,7,8,10,11,4,5,6,7,8,10,11,12[upper], 13,2) and (1,3,5,6,7,8,10,11,12[lower], 13,2). Note that the *c*-use at (9,10) need not be included under the APU+C criterion.

Figure 5.6d shows a single definition for variable V. C-use coverage is achieved by (1,3,4,5,6,7,8,9,10,11,12,13,2). In Figure 5.6c, ACU+P coverage is achieved for Z by path (1,3,4,5,6,7,8,10, 11,12,13[lower], 2), but the predicate uses of several definitions are not covered. Specifically, the (1,3) definition is not covered for the (3,5) *p*-use, the (7,8) definition is not covered for the (8,9), (9,6) and (9, 10) *p*-uses.

The above examples imply that APU+C is stronger than branch coverage but ACU+P may be weaker than, or incomparable to, branch coverage.

3.3.5. All-Definitions Strategy

The **all-definitions (AD) strategy** asks only that every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use. Path (1,3,4,5,6,7,8, . . .) satisfies this criterion for variable Z, whereas any entry/exit path satisfies it for variable V. From the definition of this strategy we would expect it to be weaker than both ACU+P and APU+C.

3.3.6. All-Predicate-Uses, All-Computational Uses Strategies

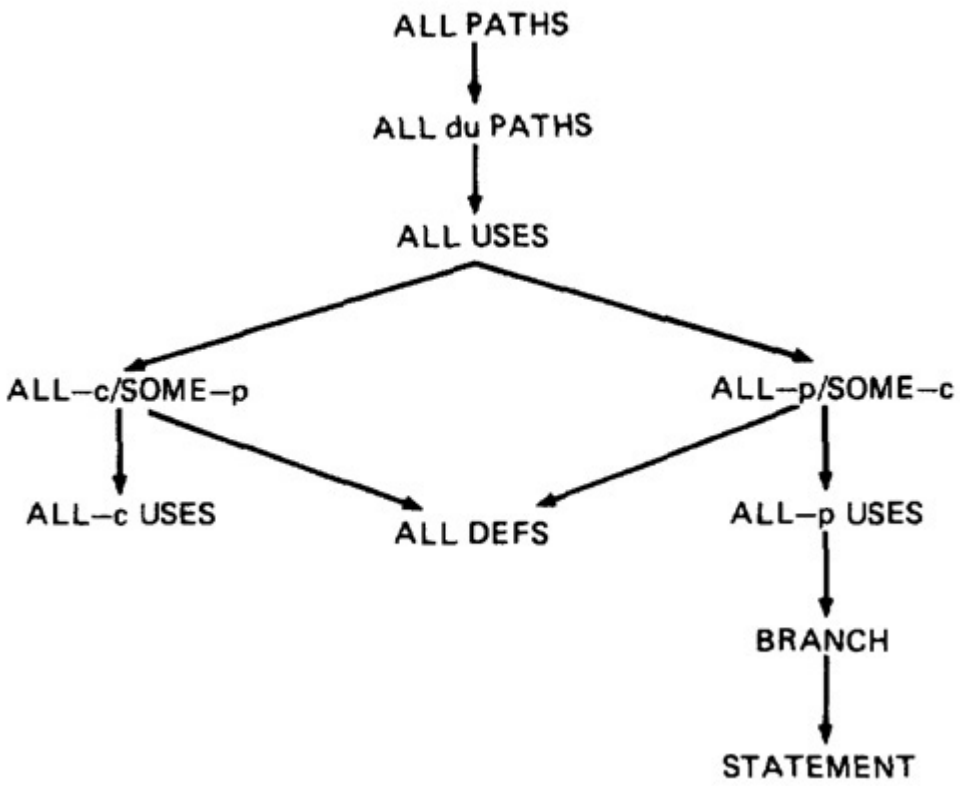
The **all-predicate-uses (APU) strategy** is derived from the APU + C strategy by dropping the requirement that we include a *c*-use for the variable if there are no *p*-uses for the variable following each definition. Similarly, the **all-computational-uses (ACU) strategy** is derived from ACU+P by dropping the requirement that we include a *p*-use if there are no *c*-use instances following a definition. It is intuitively obvious that ACU should be weaker than ACU+P and that APU should be weaker than APU+C.

3.3.7. Ordering the Strategies

Figure 5.7 compares path-flow and data-flow testing strategies. The arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow's head. Formal proofs of these relations are found in RAPP85 and FRAN88. The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practical applications. Note that although ACU+P is stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that "all definitions" is not comparable to ACU or APU.

The discussion of data-flow testing strategies in this chapter is introductory and only minimally covers the research done in this area. Variations of data-flow strategies exist, including different ways of characterizing the paths to be included and whether or not the selected paths are achievable. The strength relation graph of Figure 5.7 can be substantially expanded to fit almost all such strategies into it. Indeed, one objective of testing research has been to place newly proposed strategies into the hierarchy. For additional information see CLAR86, FRAN86, FRAN88, KORE85, LASK83, NTAF8413, NTAF88, RAPP82, RAPP85, and ZEIL88.

Figure 5.7. Relative Strength of Structural Test Strategies



3.4. Slicing, Dicing, Data Flow, and Debugging

3.4.1. General

Although an impressive theory of testing has emerged over the past 2 decades, debugging theory has lagged. Even those of us who eschew (unstructured) debugging because it is (in a sense) a failure of testing, recognize that debugging will always be with us. Testing in a maintenance context is not the same as testing new code—for which most testing theory and testing strategies have been developed. Maintenance testing is in many ways similar to debugging. I view debugging and testing as two ends of a spectrum of techniques in which maintenance testing falls someplace in the middle. It is interesting to note that the three concerns (testing, maintenance and debugging) come together in the context of data-flow testing techniques (KORE85, LASK83, LYLE88, OSTR88B).

3.4.2. Slices and Dices

A (static) program **slice** (WEIS82) is a part of a program (e.g., a selected set of statements) defined with respect to a given variable X (where X is a simple variable or a data vector) and a statement *i*: it is the set of all statements that could (potentially, under static analysis) affect the value of X at statement *i*—where the influence of a faulty statement could result from an improper computational use or predicate use of some other variables at prior statements. If X is incorrect at statement *i*, it follows that the bug must be in the program slice for X with respect to *i*. A program **dice** (LYLE87) is a part of a slice in which all statements which are known to be correct have been removed. In other words, a dice is obtained from a slice by incorporating information obtained through testing or experiment (e.g., debugging). The idea behind slicing and dicing is based on Weiser's observation (WEIS82) that these constructs are at the heart of the procedure followed by good debuggers. This position is intuitively sensible. The debugger first limits her scope to those prior statements that could have caused the faulty value at statement *i* (the slice) and then eliminates from further consideration those statements that

testing has shown to be correct. Debugging can be modeled as an iterative procedure in which slices are further refined by dicing, where the dicing information is obtained from ad hoc tests aimed primarily at eliminating possibilities. Debugging ends when the dice has been reduced to the one faulty statement.

Dynamic slicing (KORE88C) is a refinement of static slicing in which only statements on achievable paths to the statement in question are included. Korel and Laski further extend the notion of slices to arrays and data vectors and also compare and contrast data flow relations such as *dc* and *dp* within dynamic slices to analogous relations in static data flows (for example, *dc* and *dp*).

Slicing methods have been supported by tools and tried experimentally on a small scale. Current models of slicing and dicing incorporate assumptions about bugs and programs that weaken their applicability to real programs and bugs. It is too early to tell whether some form of slicing will lead to commercial testing and/or debugging tools. It is encouraging, though, to find that data-flow concepts appear to be central to closing the gap between debugging and testing.

4. APPLICATION, TOOLS, EFFECTIVENESS

Ntafos (NTAF84B) compared random testing, P_2, and AU strategies on fourteen of the Kernighan and Plauger (KERN76) programs (a set of mathematical programs with known bugs, often used to evaluate test strategies). The experiment had the following outcome:

Strategy	*Mean No. Test Cases*	*Bugs Found (%)*
Random testing	35	93.7
Branch testing	3.8	91.6
All uses	11.3	96.3

A second experiment (NTAF84A) on seven similar programs showed the following:

Strategy	*Mean No. Test Cases*	*Bugs Found (%)*
Random testing	100	79.5
Branch testing	34	85.5
All uses	84	90.0

Sneed (SNEE86) reports experience with real programs and compares branch coverage effectiveness at catching bugs to data-flow testing criteria. Sneed's data-flow coverage definition is pragmatic rather than mathematically precise: it corresponds most closely to AD. He reports that the number of bugs detected by requiring 90% "data coverage" was twice as high as those detected by requiring 90% branch coverage.

Weyuker (WEYU88A, WEYU90) has published the most thorough comparison of data-flow testing strategies to date. Her study is based on twenty-nine programs from the Kernighan and Plauger set. Tests were designed using the ASSET testing system (FRAN88). Although the study is based on a small sample, the results are encouraging. The study examined the number of test cases needed to satisfy ACU, APU, AU, and ADUP. The number of test cases are normalized to the number of binary decisions in the program. A linear regression line of the form $t = a + bd$ was done for the four strategies; where a and b are constants obtained from the regression and d is the number of binary decisions in the program. The number of binary decisions in a program is an established program complexity metric (see Chapter

7). A loop-free program with *d* binary decisions requires at most *d* + 1 test cases to achieve branch coverage. Therefore, *d* is a measure of the number of test cases needed for branch coverage (actually, the number needed in practice is closer to *d*/4 than to *d*). The primary result is expressed in terms of the mean number of test cases required per decision, as shown in the table below (after WEYU88A):

Strategy	***t/d***	***t = a + bd***
ACU all-*c*	.43	1.87 + 0.52d
APU all-*p*	.70	1.01 + 0.76d
APU all-*p**	.39	104.28 + 0.02d
AU all-uses	.72	1.42 + 0.81d
ADUP all-*du*-paths	.81	1.40 + 0.92d

The all-*p** entry refers to a study by Shimeall and Levenson reported by Weyuker in a revised version of WEYU88A. The study covered eight numerical Pascal programs used in combat simulation. The programs ranged in size from 1186 to 2489 statements and had from 173 to 434 decisions, with a mean of 277 decisions. The regression line is not as meaningful a predictor of the number of test cases needed to achieve APU because of the relatively small range of number of decisions in the sample: *t/d* is probably a more reliable indicator of testing complexity for these programs. Note that this value is almost half of the value obtained for APU for the small Kernighan and Plauger programs. There are no data to reliably determine how *t/d* for APU changes with program size.

The surprising result is that ADUP doesn't even take twice as many test cases as ACU or APU; furthermore, the modest number of test cases is not much greater than required for P_2. The experiments further showed that although theoretically AC and AP are incomparable, achieving AP generally satisfied AC also. The results are comparable to those of Ntafos (NTAF84A, NTAF84B).

Data-flow testing concepts have been around a long time. Data flow testing practice predates the formal analysis of these strategies. Although data flow testing came under theoretical scrutiny after statement and branch testing the use of data flow testing strategies coincides with the use of branch testing in the late 1960s (BEND70C, SCHL70). Just as statement and branch coverage were found to be cost-effective testing strategies, even when unsupported by automation, data-flow testing has been found effective. Finding data-flow-covering test sets, especially for the all-uses strategy, is not more nor less difficult than finding branch-covering test sets—merely more tedious if you want to avoid redundant tests. There's more bookkeeping because you have to keep track of which variables are covered and where, in addition to branch coverage. Bender and Associates (BEND85) markets a proprietary design and test methodology in which AU (in addition to P_1 and P_2) testing plays a prominent role.* Experience with AU on critical software and a long history of experience at IBM (BEND70C) is convincing evidence of the practicality and effectiveness of data flow testing strategies.

---

*Because, as for statement versus branch coverage, for unstructured code such as assembly language, FORTRAN, or COBOL, AU does not necessarily guarantee APU, which in turn need not guarantee P_2, etc.

Data-flow testing does entail additional record keeping, for which a computer is most effective. Even relatively simple tools that just keep track of which variables are defined and where and in which (if any) subsequent statement the definition is used can significantly reduce the effort of data-flow testing. Because one entry/exit path test case typically passes through a bunch of definitions and uses for many

different variables, test design is not very different than for P_1 or P_2. In Weyuker's experiments (WEYU88A), subjects did not specifically try to use the selected strategy—they tested in accordance to a strategy of their choice, and the tool, ASSET, told them the extent to which the test set had satisfied the criteria. The following sources also deal with data-flow testing tools: BEND70C, FOSD76A, FRAN88, HARR89, HERM76, KORE85, KORE88A, KORE88B, KORE89, LASK90A, LYLE87, OSTR88B, POLL87B, SNEE86, WILS82, WEYU88A, and WEYU90.

While most users of data-flow testing strategies have found it expedient and cost-effective to create supporting tools, as of the time of writing, commercial data-flow testing tools have yet to come on the market. The ease with which data-flow testing tools can be incorporated into compilers, and the general increase in awareness of the efficacy of data-flow testing combine to foreshadow intense commercial tools development in the future.

5. TESTABILITY TIPS

1. No data-flow anomalies, even if "harmless."
2. Try to do all data-flow operations on one variable at the same program level. That is, avoid defining an object in a subroutine and using it in the calling program, or vice versa. When all data-flow operations on a variable are all done at the same program level, you avoid an integration testing problem for that variable. The closer your design meets this objective, the likelier you are to find a data-flow anomaly in unit tests rather than in integration or higher-level component tests.
3. If you can't achieve item 2 for a variable, then at least try to have complementary operations such as open/close or define/kill done at the same program level.
4. Try especially hard to keep the *p*-uses of a variable at the same level as they are defined. This helps you to avoid data-flow anomalies and control-flow problems that can only be detected in the integrated components; that is, more of the control flow can be verified in unit testing.
5. No data object aliases, "bit sharing," and the utilization of unassigned bits within bytes. "Waste" a little space if that makes data structures cleaner and references more consistent.
6. Use strong typing and user-defined types if and as supported by the source language. Use a preprocessor to enforce strong typing if not supported by the language.
7. Use explicit, rather than implicit (as in FORTRAN), declaration of all data objects even if not required by the language. Use explicit initialization of all data objects if it is not done automatically by the language processor.
8. Bias the design to a regular pattern of object creation, use, and release, in that order: i.e., create/assign/fetch, use, return/clear/ kill. Put data declarations at the top of the routine even if the language permits declarations anyplace. Return and clear objects at a central point, as close as possible to the exit. In other words, waste a little space and hold it a little longer in the interest of data-flow regularity.

6. SUMMARY

1. Data are as important as code and will become more important.
2. Data integrity is as important as code integrity. Just as common sense dictates that all statements and branches be exercised on under test, all data definitions and subsequent uses must similarly be tested.
3. What constitutes a data flow anomaly is peculiar to the application. Be sure to have a clear concept of data flow anomalies in your situation.
4. Use all available tools to detect those anomalies that can be detected statically. Let the extent and excellence of static data-flow anomaly detection be as important a criterion in selecting a language processor as produced object code efficiency and compilation speed. Use the slower

compiler that gives you slower object code if it can detect more anomalies. You can always recompile the unit after it has been debugged.

5. The data-flow testing strategies span the gap between all paths and branch testing. Of the various available strategies, AU probably has the best payoff for the money. It seems to be no worse than twice the number of test cases required for branch testing, but the resulting code is much more reliable. AU is not too difficult to do without supporting tools, but use the tools as they become available.

6. Don't restrict your notion of data-flow anomaly to the obvious. The symbols *d, k, u*, and the associated anomalies, can be interpreted (with profit) in terms of file opening and closing, resource management, and other applications.

<ch4 toc ch6>