<ch2 toc ch4>

Chapter 3
FLOWGRAPHS AND PATH TESTING

1. SYNOPSIS

Path testing based on the use of the program's control flow as a structural model is the cornerstone of testing. Methods for generating tests from the program's control flow, criteria for selecting paths, and how to determine path–forcing input values are discussed.

2. PATH–TESTING BASICS

2.1. Motivation and Assumptions

2.1.1. What Is It?

Path testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.

2.1.2. Motivation

Path–testing techniques are the oldest of all structural test techniques. They are recorded as being in use at IBM for more than two decades (HIRS67, SCH169, WARN64). The commonsense idea of executing every statement and branch at least once under some test comes to almost every person who examines software testing in depth. Path–testing techniques were also the first techniques to come under theoretical scrutiny. There is considerable (anecdotal) evidence that path testing was independently discovered and used many many times in many different places.

Path testing is most applicable to new software for unit testing. It is a structural technique. It requires complete knowledge of the program's structure (i.e., source code). It is most often used by programmers to unit–test their own code. The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases. Path testing is rarely, if ever, used for system testing. For the programmer, it is *the* basic test technique.

I'm often asked by independent testers and system testers why they should bother learning path testing because they're not likely to ever use it directly: the reason to learn path testing is that it's fundamental to all testing techniques. That's also the reason this chapter is the longest in the book. It is almost impossible to discuss any testing strategy without relying on the conceptual vocabulary established in path testing. And when we discuss functional testing techniques such as transaction–flow testing (Chapter 4) or domain testing (Chapter 6), which are very useful to system testers even though we don't deal with paths directly, understanding the strategy depends on understanding that there are paths that can be distinguished from one another.

2.1.3. The Bug Assumption

The bug assumption for the path–testing strategies is that something has gone wrong with the software that makes it take a different path than intended. As an example, "GOTO X" where "GOTO Y" had been intended. As another example, “IF A is *true* THEN DO X ELSE DO Y”, instead of “IF A is *false* THEN . . .”. We also assume, in path testing, that specifications are correct and achievable, that there are no processing bugs other than those that affect the control flow, and that data are properly defined and accessed. Structured programming languages prevent many of the bugs targeted by path testing: as a consequence, the effectiveness of path testing for these languages is reduced. Conversely, old code, especially in assembly languages, COBOL, FORTRAN, and Basic, has a higher proportion of control–flow bugs than contemporary code, and for such software path testing is indispensable.

2.2. Control Flowgraphs

2.2.1. General

The **control flowgraph** (or **flowgraph** alone when the context is clear) is a graphical representation of a program’s control structure. It uses the elements shown in Figure 3.1: **process blocks, decisions,** and **junctions.** The control flowgraph is similar to the earlier **flowchart,** with which it is not to be confused. The differences between control flowgraphs and flowcharts are discussed in Section 2.2.5 below.

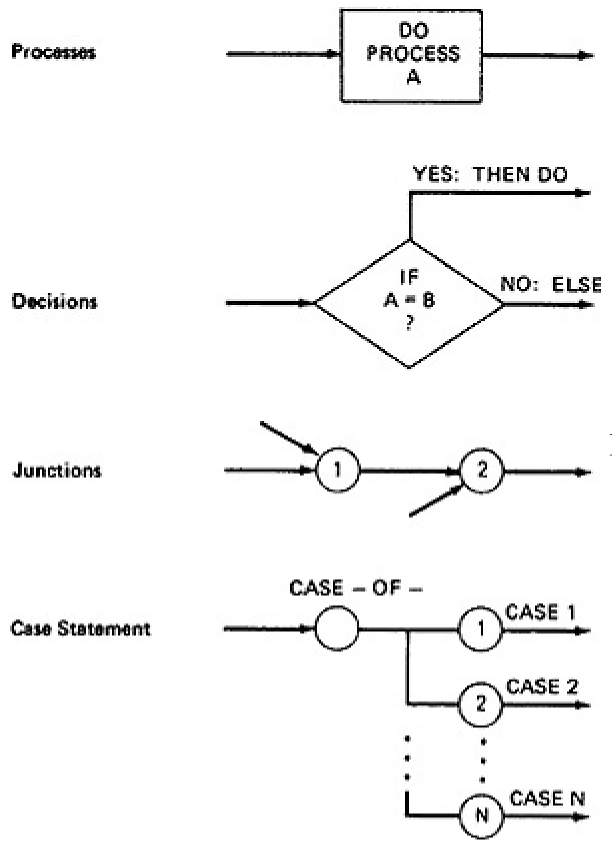


Figure 3.1. Flowgraph Elements.

2.2.2. Process Block

A **process block**[*] is a sequence of program statements uninterrupted by either decisions or junctions. Formally, it is a sequence of statements such that if any one statement of the block is executed, then all statements thereof are executed. Less formally, a process block is a piece of straight–line code. A process block can be one source statement or hundreds. The point is that, bugs aside, once a process block is initiated, every statement within it will be executed. Similarly, there is no point within the process block that is the target of a GOTO. The term “process” will be used interchangeably with “process block.”

*Also called "Process," "block," "basic block," and "program block."

A process has one entry and one exit. It can consist of a single statement or instruction, a sequence of statements or instructions, a single–entry/single–exit subroutine, a macro or function call, or a sequence of these. The program does not (rather, is not intended to) jump into or out of processes. From the point of view of test cases designed from control flowgraphs, the details of the operation within the process are unimportant if those details do not affect the control flow. If the processing does affect the flow of control, the effect will be manifested at a subsequent decision or case statement.

2.2.3. Decisions and Case Statements

A **decision** is a program point at which the control flow can diverge. Machine language conditional branch and conditional skip instructions are examples of decisions. The FORTRAN IF and the Pascal IF–THEN–ELSE constructs are decisions, although they also contain processing components. While most decisions are two–way or **binary,** some (such as the FORTRAN IF) are three–way branches in control flow. The design of test cases is generally easier with two–way branches than with three–way branches, and there are also more powerful test–design tools that can be used. A **case statement** is a multiway branch or decision. Examples of case statements include a jump table in assembly language, the FORTRAN–computed GOTO and assigned GOTO, and the Pascal CASE statement. From the point of view of test design, there are no fundamental differences between decisions and case statements.

2.2.4. Junctions

A **junction** is a point in the program where the control flow can merge. Examples of junctions are: the target of a jump or skip instruction in assembly language, a label that is the target of a GOTO, the END–IF and CONTINUE statements in FORTRAN, and the Pascal statement labels, END and UNTIL.

Unconditional branches such as FORTRAN's GOTO or unconditional jump instructions are not fundamental to programming. Although it can be proven (BOHM66) that unconditional branches are not essential to programming, their use is ubiquitous and often practically unavoidable. Testing and testing theory are easier if there are no GOTOs, but in the real world we must often test dirty old software. The methods discussed in this book apply to any kind of software—assembly languages, FORTRAN, COBOL, C, Pascal, Ada, etc. Some of the differences between the old and new languages is that the newer languages are generally nicer from the point of view of control flow and test design. Whatever language you program, if you can apply test techniques to assembly language spaghetti code, you'll have no trouble with the newer, gentler languages.

2.2.5. Control Flowgraphs Versus Flowcharts

A program's **flowchart** resembles a control flowgraph, but differs in one important way. In control flowgraphs, we don't show the details of what is in a process block; indeed, the entire block, no matter how many statements in it, is shown as a single process. In flowcharts, conversely, every part of the process block is drawn: if a process block consists of 100 steps, the flowchart may have 100 boxes. Although some flowcharting conventions permit grouping adjacent processing statements into "higher–level" steps, there are no fixed rules about how this should be done.* The flowchart focuses on process steps, whereas the control flowgraph all but ignores them. The flowchart forces an expansion of visual complexity by adding many off–page connectors, which confuse the control flow, but the flowgraph compacts the representation and makes it easier to follow.

*Flowcharts as usually drawn do combine program steps so that there may be only a few boxes for many statements but the rules by which program steps are combined into flowchart boxes are arbitrary and peculiar to the flowcharting convention used. This makes it difficult, if not impossible, to create theoretical constructs for flowcharts. Control flowgraphs, conversely, use mathematically precise formation rules.

Flowcharts have been falling out of favor for over a decade, and before another decade passes they'll be regarded as curious, archaic relics of a bygone programming era. Indeed, flowcharts are rarely used today; they're created mainly to satisfy obsolete documentation specifications. But before we throw flowcharts forever into the dustbin of programming practices past, we should ask why they had been so popular and why otherwise rational thinkers about the programming process (myself included) should have so fervently promoted their use. I can't speak for others, but I think that the flowchart was a first step toward the control flowgraph: when we used flowcharts for design we were actually using cluttered–up control flowgraphs. The trouble with them is that there's too much information—especially if the flowcharting standard requires one flowchart box per source statement.

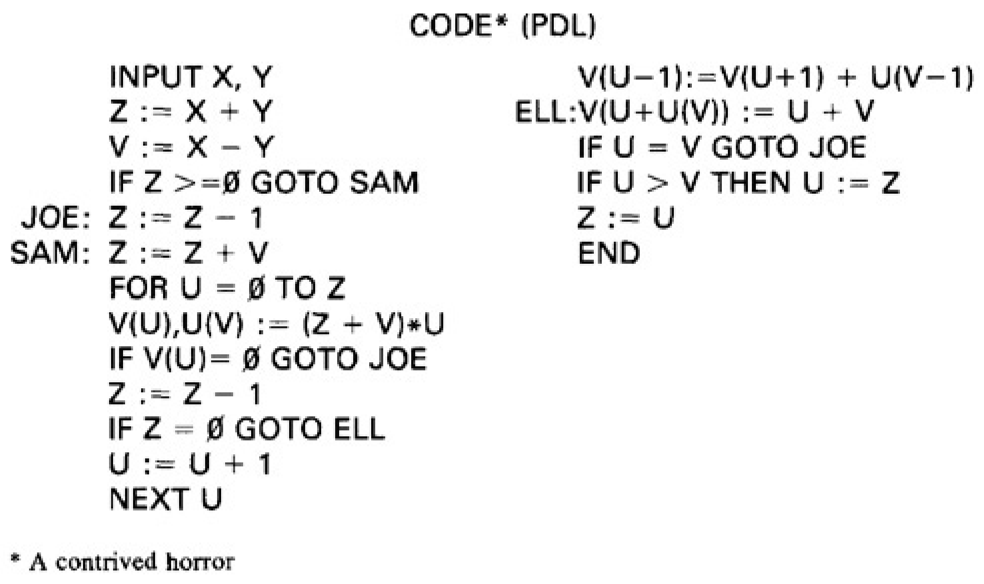
So while the exercise of drawing detailed flowcharts may not be an effective step in the design process, the act of drawing a control flowgraph (and also data flowgraph—Chapter 5) is a useful tool that can help us clarify the control flow and data flow issues. Accordingly, because we are all good programmers, I'll assume that there is a design control–flowgraph that has been created and checked prior to coding. There may also be specification control–flowgraphs prior to design.

2.2.6. Notational Evolution

The control flowgraph is a simplified (i.e., more abstract) representation of the program's structure. To understand its creation and use, we'll go through an example, starting with Figure 3.2—a little horror written in a FORTRAN–like **program design language** (PDL). The first step in translating this to a control flowgraph is shown in Figure 3.3, where we have the typical one–for–one classical flowchart. Note that complexity has increased, clarity has decreased, and that we had to add auxiliary labels (LOOP, XX, and YY), which have no actual program counterpart. In Figure 3.4 we merged the process steps and replaced them with the single process box. We now have a control flowgraph. But this representation is still too busy. We simplify the notation further to achieve Figure 3.5, where for the first time we can really see what the control flow looks like. To do that we had to make several more notational changes.

1. The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions, especially if we allow do–nothing or dummy processes.
2. We don't need to know (at this time) the specifics of the decisions, just the fact that there is a branch—so we can do away with things such as "U > V?", "yes", "no", and so on.

Figure 3.2. Program Example (PDL).



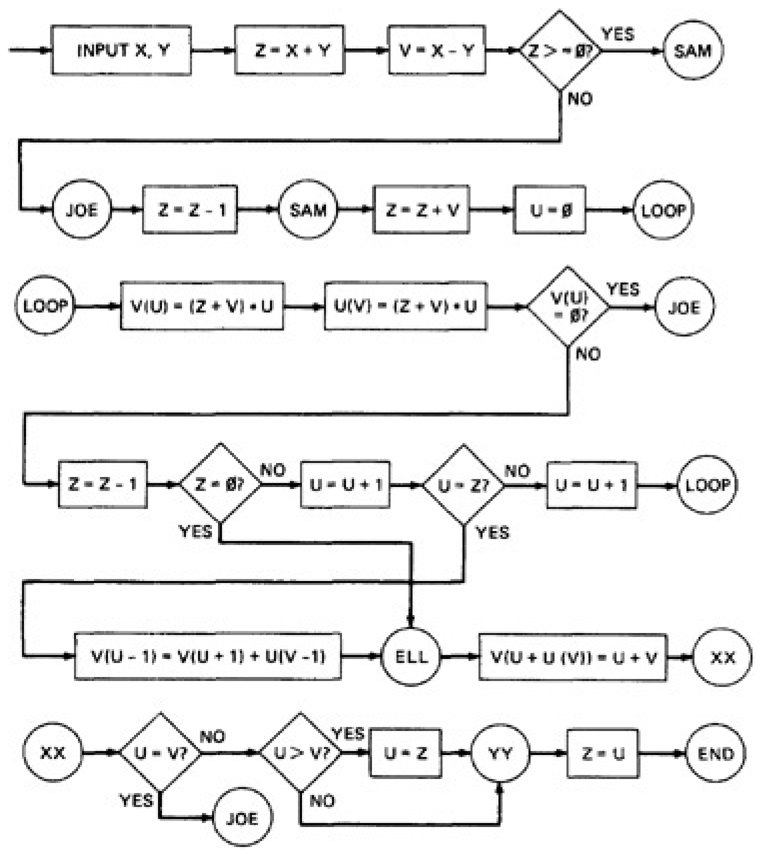
```
                        CODE* (PDL)

      INPUT X, Y                      V(U−1):=V(U+1) + U(V−1)
      Z := X + Y                  ELL:V(U+U(V)) := U + V
      V := X − Y                      IF U = V GOTO JOE
      IF Z >=Ø GOTO SAM               IF U > V THEN U := Z
JOE:  Z := Z − 1                      Z := U
SAM:  Z := Z + V                      END
      FOR U = Ø TO Z
      V(U),U(V) := (Z + V)*U
      IF V(U)= Ø GOTO JOE
      Z := Z − 1
      IF Z = Ø GOTO ELL
      U := U + 1
      NEXT U
```

* A contrived horror

3. The specific target label names aren't important—just the fact that they exist. So we can replace them by simple numbers.

Figure 3–5 is the way we usually represent the program's control flowgraph. There are two kinds of components: circles and arrows that join circles. A circle with more than one arrow leaving it is a **decision;** a circle with more than one arrow entering is a **junction.** We call the circles **nodes** and the arrows **links.** Note also that the entry and exit are also denoted by circles and are thereby also considered to be nodes. Nodes are usually numbered or labeled by using the original program labels. The link name can be formed from the names of the nodes it spans. Thus a link from node 7 to node 4 is called link (7,4), whereas one from node 4 to node 7 is called link (4,7). For parallel links between a pair of nodes, (nodes 12 and 13 in Figure 3.5) we can use subscripts to denote each one or some unambiguous notation such as "(12,13 upper)" and "(12,13 lower)". An alternate way to name links that avoids this problem is to use a unique lowercase letter for each link in the flowgraph.

Figure 3.3. One–to–one Flowchart for Figure 3.2 Example.



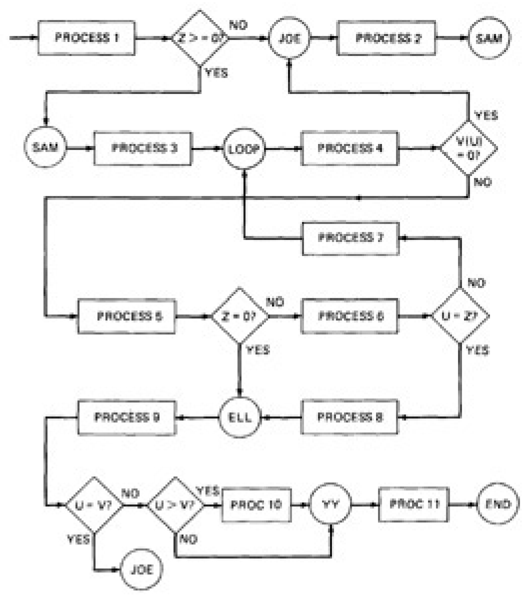


Figure 3.4. Control Flowgraph for Figure 3.2 Example.

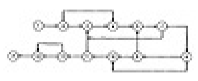


Figure 3.5. Simplified Flowgraph Notation.

The final transformation is shown in Figure 3.6, where we've dropped the node numbers to achieve an even simpler representation. The way to work with control flowgraphs is to use the simplest possible representation—that is, no more information than you need to correlate back to the source program or PDL.

Although graphical representations are revealing, they are often inconvenient—especially if we have to work with them. The alternative is to use a **linked–list** representation, shown in Figure 3.7. Each node has a name and there is an entry on the list for each link in the flowgraph. Only the information pertinent to the control flow is shown—that is, the labels and the decisions. The linked list is the representation of choice for programs that manipulate or create flowgraphs.

2.2.7. Flowgraph–Program Correspondence

A flowgraph is a pictorial representation of a program and not the program itself, just as a topographic map, no matter how detailed, is not the terrain it represents. This distinction is important: failure to make it can lead to bugs. You can't always associate the parts of a program in a unique way with flowgraph parts because many program structures, such as IF–THEN–ELSE constructs, consist of a combination of decisions, junctions, and processes. Furthermore, the translation from a flowgraph element to a statement and vice versa is not always unique. Myers (MYER77) cites an anomaly based on different representations of the FORTRAN statement "IF (A=0) AND. (B=1) THEN . . .". It has the alternate representations shown in Figure 3.8.

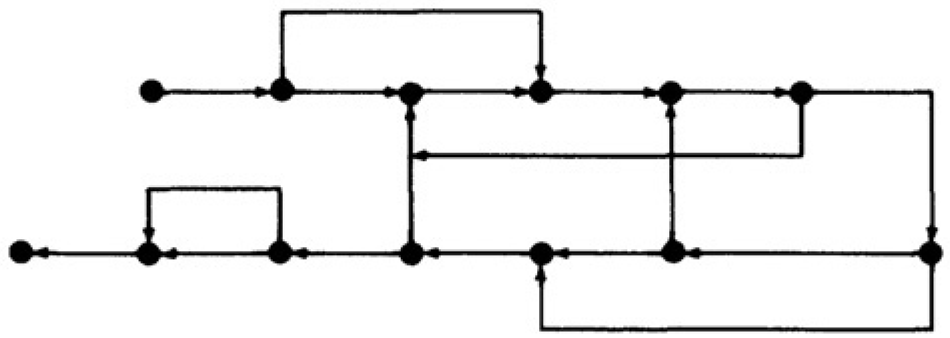


Figure 3.6. Even Simpler Flowgraph Notation.

```
 1 (BEGIN)    : 3
 2 (END)      :          Exit, no outlink
 3 (Z>Ø?)     : 4 (FALSE)
              : 5 (TRUE)
 4 (JOE)      : 5
 5 (SAM)      : 6
 6 (LOOP)     : 7
 7 (V(U)=Ø?)  : 4 (TRUE)
              : 8 (FALSE)
 8 (Z=Ø?)     : 9 (FALSE)
              :10 (TRUE)
 9 (U=Z?)     : 6 (FALSE) = LOOP
              :10 (TRUE) = ELL
10 (ELL)      :11
11 (U=V?)     : 4 (TRUE) = JOE
              :12 (FALSE)
12 (U>V?)     :13 (TRUE)
              :13 (FALSE)
13            : 2 (END)
```

Figure 3.7. Linked–List Control–Flowgraph Notation.

A FORTRAN DO has three parts: a decision, an end–point junction, and a process that iterates the DO variable. The FORTRAN IF–THEN–ELSE has a decision, a junction, and three processes (including the processing associated with the decision itself). Therefore, neither of these statements can be translated into a single flowgraph element. Some computers have looping, iterating, and EXECUTE instructions or other instruction options and modes that prevent the direct correspondence between instructions and flowgraph elements. Such differences are so familiar to us that we often code without conscious awareness of their existence. It is, however, important that the distinction between a program and its flowgraph representation be kept in mind during test design. An improper translation from flowgraph to code during coding can lead to bugs, and an improper translation (in either direction) during test design can lead to missing test cases and consequently, to undiscovered bugs. When faced with a possibly ambiguous translation from code to flowgraph or from flowgraph to code, as in the above example, it is better to pick the more complicated representation rather than the simpler one. At worst, you will design a few extra test cases.

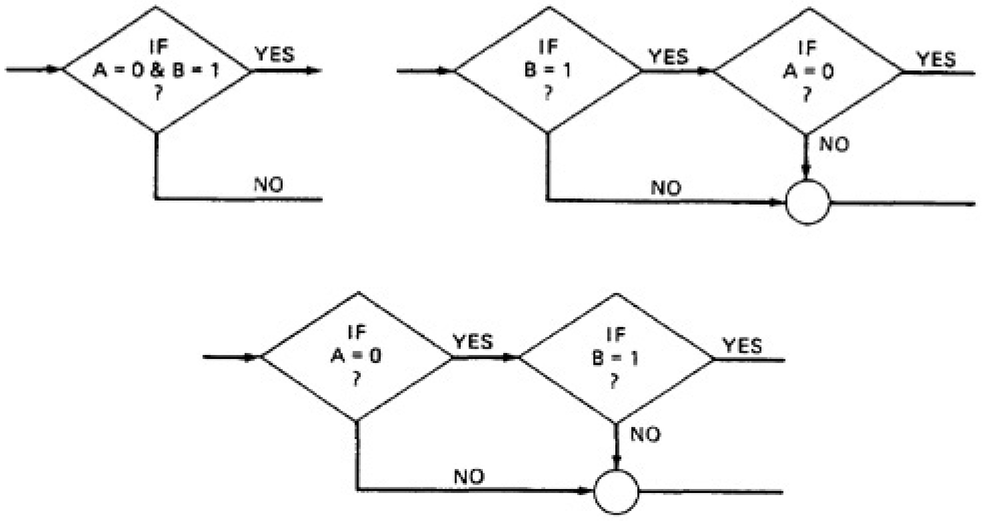


Figure 3.8. Alternative Flowgraphs for the Same Logic.

2.2.8. Flowgraph and Flowchart Generation

The control flowgraph is a simplified version of the earlier flowchart. Although there are many commercially available flowcharting packages, there are relatively few control flowgraph generators as of the time of writing. This situation is especially irksome because the information needed to generate the control flowgraph is a by–product of most compilers. Therefore, we will have to discuss the earlier flowchart generators and hope that, sooner or later, you'll be able to substitute "control flowgraph" for "flowchart" throughout the sequel.

Flowcharts can be (1) hand–drawn by the programmer, (2) automatically produced by a flowcharting program based on a mechanical analysis of the source code, or (3) semiautomatically produced by a flowcharting program based in part on structural analysis of the source code and in part on directions given by the programmer. The semiautomatic flowchart is most common with assembly language source code. A flowcharting package that provides controls over how statements are mapped into process boxes can be used to produce a flowchart that is reasonably close to the control flowgraph. You do this by starting process boxes just after any decision or GOTO target and ending them just before branches or GOTOs.

The designer's original flowchart and the automatically produced flowchart would be identical in a perfect world, but often they are not. The programmer's design control–flowgraph or flowchart is a statement of intentions and not a program. Those intentions become corrupted through the action of malevolent forces such as keyboards, compilers, and other programmers. A typographical error in a case statement can cause a different path to be implemented or can make a path unachievable. In assembly language, the possibility of manipulating addresses, registers, page boundaries, or even instructions (inadvertently, it is to be hoped) makes the potential differences between the specification control–flowgraph and the actual control flow wilder yet. If you have automatically produced flowcharts, then it is effective to check the correspondence between the specification control–flowgraph and the flowchart produced from code. Better yet, have someone else, someone who has not seen the specification flowgraph, translate the code back into a flowgraph, or compare the automatically produced flowchart with the specification flowgraph. Do it as part of an inspection or review. Many bugs can be caught this way. It may seem like a lot of extra work, but consider which you would rather do:

1. A calm, manual retranslation of the code back into a flowgraph with an unbiased comparison, or
2. A frenzied plea to a colleague in the heat of the test floor late at night: “Look this over for me, will you? I just can’t find the bug.”

2.3. Path Testing

2.3.1. Paths, Nodes, and Links

A **path** through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times. Paths consist of **segments.** The smallest segment is a link—that is, a single process that lies between two nodes (e.g., junction–process–junction, junction–process–decision, decision–process–junction, decision–process–decision). A direct connection between two nodes, as in an unconditional GOTO, is also called a “process” by convention, even though no actual processing takes place. A **path segment** is a succession of consecutive links that belongs to some path. The **length** of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. An alternative way to measure the length of a path is by the number of nodes traversed—this method has some analytical and theoretical benefits. If programs (by convention) are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed. Because links are named by the pair of nodes they join, the **name of a path** is the name of the nodes along the path. For example, the shortest path from entry to exit in Figure 3.5 is called “(1,3,5,6,7,8,10,11,12,13,2)”. Alternatively, if we choose to label the links, the name of the path is the succession of link names along the path. A path has a **loop** in it if any node (link) name is repeated. For example, path (1,3,4,5,6,7,4,5,6,7,8,9,10,11,12,13,2) in Figure 3.5 loops about nodes 4,5,6, and 7.

The word “path” is also used in the more restricted sense of a path that starts at the routine’s entrance and ends at its exit. In practice, test paths are usually entry–to–exit paths. Where we have to make a distinction between arbitrary paths and entry–to–exit paths, we’ll use the term “entry/exit path” as needed. The terms **entry/exit path** and **complete path** are also used in the literature to denote a path that starts at an entry and goes to an exit. Our interest in entry/exit paths in testing is pragmatic because: (1) it’s difficult to set up and execute paths that start at an arbitrary statement; (2) it’s hard to stop at an arbitrary statement without setting traps or using patches and (3) entry/exit paths are what we want to test because we use routines that way.

2.3.2. Multi–Entry/Multi–Exit Routines

Throughout this book I implicitly assume that all routines and programs have a single entry and a single exit. Although there are circumstances in which it is proper to jump out of a routine and bypass the normal control structure and exit method, I cannot conceive of any rational reason why one would want to jump into the middle of a routine or program.* You might want to jump out of a routine when an illogical condition has been detected for which it is clear that any further processing along that path could damage the system's operation or data. Under such circumstances, the normal return path must be bypassed. In such cases, though, there is only one place to go—to the system's recovery–processing software. Jumping into a routine is almost always done in a misguided attempt to save some code or coding labor (to be paid for by a manifold increase in test design and debugging labor). If the routine performs several variations on the same processing and it is effective to bypass part of the processing, the correct way to design the routine is to provide an entry parameter that within the routine (say, by a case statement), directs the control flow to the proper point. Similarly, if a routine can have several different kinds of outcomes, then an exit parameter should be used. Another alternative is to encapsulate the common parts into subroutines. Instead of using direct linkages between multiple exits and entrances, we handle the control flow by examining the values of the exit parameter that can serve as an entry parameter for the next routine or a return parameter for the calling routine. Note that the parameter does not have to be passed explicitly between the routines—it can be a value in a register or in a common memory location.

*Not to be confused with instances in which a collection of independent routines are accessed by a common name. For example, the set of routines is called "SET" and it is loaded as a set in order to have efficient overlays; but within "SET" there are independent, single–entry subroutines. It might superficially seem that "SET" is a multi–entry/multi–exit routine.

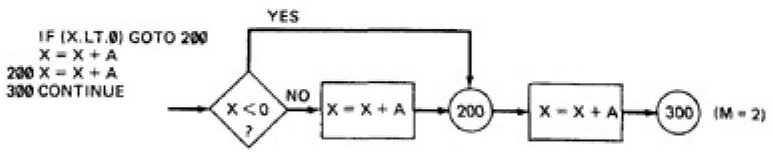
The trouble with multi–entry and multi–exit routines is that it can be very difficult to determine what the interprocess control flow is, and consequently it is easy to miss important test cases. Furthermore, the use of multi–entry and multi–exit routines increases the number of entries and exits and therefore the number of interfaces. This practice leads to more test cases than would otherwise be needed. Multi–entry/multi–exit routine testing is discussed in further detail in Section 2.5 below.

2.3.3. Fundamental Path Selection Criteria

There are many paths between the entry and exit of a typical routine. Every decision doubles the number of potential paths, and every loop multiplies the number of potential paths by the number of different iteration values possible for the loop. If a routine has one loop, each pass through that loop (once, twice, three times, and so on) constitutes a different path through the routine, even though the same code is traversed each time. Even small routines can have an incredible number of potential paths (see Chapter 8 for path–counting methods). A lavish test approach might consist of testing all paths, but that would not be a complete test, because a bug could create unwanted paths or make mandatory paths unexecutable. And just because all paths are right doesn't mean that the routine is doing the required processing along those paths. Such possibilities aside for the moment, how might we define "complete testing"?

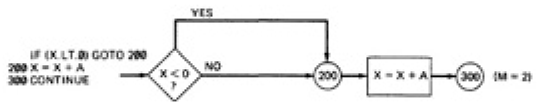
1. Exercise every path from entry to exit.
2. Exercise every statement or instruction at least once.
3. Exercise every branch and case statement, in each direction, at least once.

If prescription 1 is followed, then prescriptions 2 and 3 are automatically followed; but prescription 1 is impractical for most routines. It can be done only for routines that have no loops, in which case it includes all the cases included in prescriptions 2 and 3. Prescriptions 2 and 3 might appear to be equivalent, but they are not. Here is a correct version of a routine:



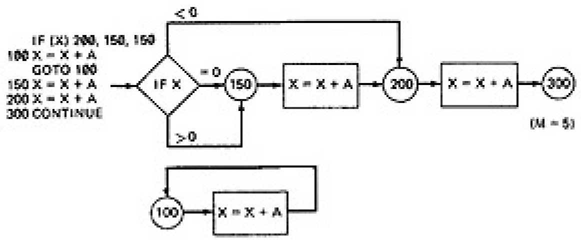
```
    IF (X.LT.0) GOTO 200
    X = X + A
200 X = X + A
300 CONTINUE
```

For X negative, the output is X + A, while for X greater than or equal to zero, the output is X + 2A. Following prescription 2 and executing every statement, but not every branch, would not reveal the bug in the following incorrect version:



```
    IF (X.LT.0) GOTO 200
200 X = X + A
300 CONTINUE
```

A negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden. The next example uses a test based on executing each branch but does not force the execution of all statements:



```
    IF (X) 200, 150, 150
100 X = X + A
    GOTO 100
150 X = X + A
200 X = X + A
300 CONTINUE
```

The hidden loop around label 100 is not revealed by tests based on prescription 3 alone because no test forces the execution of statement 100 and the following GOTO statement. Furthermore, label 100 is not flagged by the compiler as an unreferenced label and the subsequent GOTO does not refer to an undefined label.

A **static analysis** (that is, an analysis based on examining the source code or structure) cannot determine whether a piece of code is or is not reachable. There could be subroutine calls with parameters that are subroutine labels, or in the above example there could be a GOTO that targeted label 100 but could never achieve a value that would send the program to that label. Only a **dynamic analysis** (that is, an analysis based on the code's behavior while running—which is to say, to all intents and purposes, testing) can determine whether code is reachable or not and therefore distinguish between the ideal structure we think we have and the actual, buggy structure.

2.3.4. Path–Testing Criteria

Although real bugs are rarely as blatant as the above examples, the examples demonstrate that prescriptions 2 and 3 alone are insufficient: but note that for software written in a structured programming language, satisfying prescription 3 *does* imply that prescription 2 is also satisfied.

Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions. A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested. We have, therefore, explored three different **testing criteria** or **strategies** out of a potentially infinite family of strategies. They are:

1. *Path Testing (P∞)*—Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program. If we achieve this prescription, we

are said to have achieved **100% path coverage.**[*] This is the strongest criterion in the path–testing strategy family: it is generally impossible to achieve.

2. *Statement Testing (P_1)*—Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved **100% statement coverage.**[*] An alternate, equivalent characterization is to say that we have achieved **100% node coverage** (of the nodes in the program's control flowgraph). We denote this by C1. This is the weakest criterion in the family: testing less than this for new software is unconscionable and should be criminalized.

[*]We usually drop the "100%," so when we say that we have "achieved branch coverage" it means we have achieved 100% branch coverage. Similarly for statement coverage, path coverage, and other notions of coverage that will be introduced in the sequel. The use of the word "coverage" in the literature is bound to be confusing to the newcomer. The term "complete coverage" or "coverage" alone is often used in the literature to mean 100% statement and branch coverage (i.e., C1 + C2). Because C2 usually implies C1, the term "branch coverage" usually means "branch and statement coverage." The term "coverage" alone is also used to mean coverage with respect to the specific criterion being discussed, (e.g., predicate coverage), and also for C1 + C2. I will endeavor to make the specific coverage notion intended clear.

3. *Branch Testing (P_2)*—Execute enough tests to assure that every branch alternative has been exercised at least once under some test. If we do enough tests to achieve this prescription, then we have achieved **100% branch coverage** (see page 74 footnote). An alternative characterization is to say that we have achieved **100% link coverage** (of the links in the control flowgraph of the program). For structured software, branch testing and therefore branch coverage strictly includes statement coverage. We denote branch coverage by C2.

What about strategies that are stronger than branch testing but weaker than path testing? The notation P_1, P_2, . . . P_∞ should alert you to the fact that there is an infinite number of such strategies, but even that's insufficient to exhaust testing.

2.3.5. Common Sense and Strategies

Branch and statement coverage are accepted today as the minimum mandatory testing requirement. Statement coverage is established as a minimum testing requirement in the IEEE unit test standard (ANS187B). Statement and branch coverage have also been used for more than two decades as minimum mandatory unit test requirements for new code at IBM (HIRS67, SCH169, WARN64) and other major computer and software companies. The justification for insisting on statement and branch coverage isn't based on theory but on common sense. One can argue that 100% statement coverage (C1) is a lot of work, to which I reply that debugging is a lot more work. Why not use a judicious sampling of paths?[**] What's wrong with leaving some code, especially code that has a low probability of execution, untested? Common sense and experience show why such proposals are ineffectual:

[**]One reason that programmers and programming managers (especially) who are unfamiliar with testing may object to statement and branch coverage is that they confuse it with executing all possible paths. I recall several instances in which hours of argument ended when the recalcitrant programmer was finally made to realize that testing all paths was not at all the same as branch and statement coverage —whereupon she said, "Oh! Is *that* all you wanted?"

1. Not testing a piece of code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs.
2. The high–probability paths are always thoroughly tested if only to demonstrate that the system works properly. If you have to leave some code untested at the unit level, it is more rational to leave the normal, high–probability paths untested, because someone else is sure to exercise them during integration testing or system testing.
3. Logic errors and fuzzy thinking are inversely proportional to the probability of the path's execution.
4. The subjective probability of executing a path as seen by the routine's designer and its objective execution probability are far apart. Only analysis can reveal the probability of a path, and most programmers' intuition with regard to path probabilities is miserable (see BEIZ78).
5. The subjective evaluation of the importance of a code segment as judged by its programmer is biased by aesthetic sense, ego, and familiarity. Elegant code might be heavily tested to demonstrate its elegance or to defend the concept, whereas straightforward code might be given cursory testing because "How could anything go wrong with that?"

Unit testing of new code based on less than C1 forces us to decide what code should be left untested. Ask yourself, "What immunity from bugs has been granted to the untested code?" If such code has no special immunity from bugs, then what criterion will you use to decide which code shall and which shall not be tested? Such criteria are inevitably biased, rarely rational, and always grievous. Realistically, the practice of putting untested code into systems is common, and so are system failures. The excuse I&ve most often heard for putting in untested code is that there wasn&146;t enough time or money left to do the testing. If there wasn't enough time and money to test the routine, then there wasn't enough time and money to create it in the first place. What you think is code, before it has been properly tested, is not code, but the mere promise of code—not a program, but a perverse parody of a program. If you put such junk into a system, its bugs will show, and because there hasn&146;t been a rigorous unit test, you'll have a difficult time finding the bugs. As Hannah Cowley said, "Vanity, like murder, will out." For it's vanity to think that untested code has no bugs, and murder to put such code in. It is better to leave out untested code altogether than to put it in. Code that doesn't exist can't corrupt good code. A function that hasn't been implemented is known not to work. An untested function may or may not work itself (probably not), but it can make other things fail that would otherwise work. In case I haven't made myself clear, leaving untested code in a system is stupid, shortsighted, and irresponsible.

How about branch coverage (C2)? Why does common sense lead us to this requirement? Control flow is at the heart of programming. Typical software has a high density of conditional branches, loops, and other control–flow statements—approximately 25% in most languages. If we stop at C1, then one of the alternatives in the IF–THEN–ELSE statements might never be tested. It's obviously worse with case statements and loops. Let's apply common sense to our testing. We should expect that if we apply a testing criterion to source code, say C1, then applying our test cases to the object code produced should achieve comparable coverage. For example, if we achieved C1 for Pascal source, then is it not reasonable to expect that we achieved C1 for the object code into which that source was translated? Statement coverage (C1) does not guarantee this result, nor does it even come close. A good piece of logic–intensive modern code might actually be only 75% or less C1 covered at the object level when C1 is the criterion used for source. One can easily design a routine in an HOL such that its machine language translation is at best covered at 50%, or for that matter, at an arbitrarily low percentage under C1. Therefore, the commonsense argument we used for statement coverage also forces us into branch coverage. That is, if we are to achieve object level Cl, then we need at least source level C2 for most source languages.

Commonsense considerations will force us to even more thorough notions of testing, because for example, even C2 applied to source does not guarantee Cl for object. Every testing technique leads us to define more justifiable tests and leads us to a further recognition of just how weak statement and branch coverage are as criteria for "complete testing." Have we tested all reasonable data–usage patterns? Have we checked all the interesting extreme input combinations? Have we tested all interesting loop conditions? Have all features been verified? The more we learn about testing, the more we realize that statement and branch coverage are minimum *floors* below which we dare not fall, rather than ceilings to which we should aspire.

2.3.6. Which Paths

You must pick enough paths to achieve C1 + C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate path testing, but it is not crucial to the pragmatic design of tests. It's better to take many simple paths than a few complicated paths. Furthermore, there's no harm in taking paths that will exercise the same code more than once. As an example of how to go about selecting paths, consider the unstructured monstrosity of Figure 3.9.

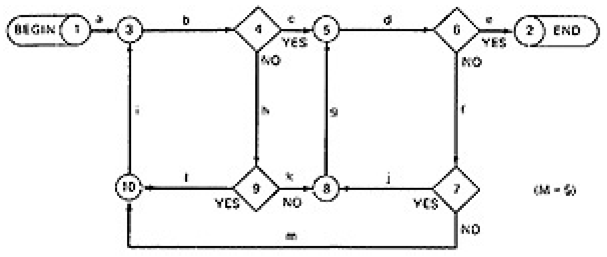


Figure 3.9. An Example of Path Selection.

Start at the beginning and take the most obvious path to the exit—it typically corresponds to the normal path. The most obvious path in Figure 3.9 is (1,3,4,5,6,2), if we name it by nodes, or *abcde* if we name it by links. Then take the next most obvious path, *abhkgde.* All other paths in this example lead to loops. Take a simple loop first—building, if possible, on a previous path, such as *abhlibcde.* Then take another loop, *abcdfjgde.* And finally, *abcdfmibcde.* Here are some practical suggestions:

1. Draw the control flowgraph on a single sheet of paper.
2. Make several copies—as many as you'll need for coverage (C1 + C2) and several more.
3. Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheet.
4. Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1 + C2.

I say "appear" because some of the paths you've selected might not be achievable. This is discussed in Section 3 below.

As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision. The above paths lead to the following table:

PATHS	**DECISIONS**				**PROCESS—LINK**												
	4	**6**	**7**	**9**	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**
abcde	YES	YES			✓	✓	✓	✓	✓								
abhkgde	NO	YES		NO	✓	✓		✓	✓		✓	✓			✓		
abhlibcde	NO,YES	YES		YES	✓	✓	✓	✓	✓			✓	✓			✓	
abcdfjgde	YES	NO,YES	YES		✓	✓	✓	✓	✓	✓	✓			✓			
abcdfmibcde	YES	NO,YES	NO		✓	✓	✓	✓	✓	✓			✓				✓

After you have traced a covering path set on the master sheet and filled in the table for every path, check the following:

1. Does every decision have a YES and a NO in its column? (C2)
2. Has every case of all case statements been marked? (C2)
3. Is every three–way branch (less, equal, greater) covered? (C2)
4. Is every link (process) covered at least once? (C1)*

Note: For structured languages and well–formed programs this check is redundant.

Select successive paths as small variations of previous paths. Try to change only one thing at a time—only one decision's outcome if possible. It's better to have several paths, each differing by only one thing, than one path that covers more but along which several things change.** The *abcd* segment in the above example is common to many paths. If this common segment has been debugged, and a bug appears in a new path that uses this segment, it's more likely that the bug is in the new part of the path (say *fjgde*) rather than in the part that's already been debugged. Using small changes from one test to the next may seem like more work; however,

**Not only does this strategy make sense, but it has been formally explored (PRAT87) to create the path prefix strategy, which is stronger than branch testing, but nevertheless efficient from the point of view of the number of test cases needed. It appears to be a satisfactory basis for building structural test generation tools.

1. Small changes from path to path mean small, easily documented, and gradual changes in the test setup. Setting up long, complicated paths that share nothing with other test cases is also a lot of extra work.
2. Testing is experimenting. Good experiments rely on changing only one thing at a time. The more you change from test to test, the likelier you are to get confused.
3. The costs of extra paths are a few more microseconds of computer time, the time to run another case, and the cost of additional documentation. Many more and different kinds of tests are required beyond path testing. A few extra paths represent only a small increment in the total test labor.

You could select paths with the idea of achieving coverage without knowing anything about what the routine is supposed to do. Path selection based on pure structure without regard to function has the advantage of being free of bias. Conversely, such paths are likely to be confusing, counterintuitive, and hard to understand. I favor paths that have some sensible functional meaning.* With this in mind, the path selection rules can be revised as follows:

*The term "function" is here understood to mean function in the context of the routine's specification and not necessarily in the sense of overall function as viewed by the user.

1. Pick the simplest, functionally sensible entry/exit path.
2. Pick additional paths as small variations from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths over long paths, simple paths over complicated paths, and paths that make sense over paths that don't.
3. Pick additional paths that have no obvious functional meaning only if it's necessary to provide

coverage. But ask yourself first why such paths exist at all. Why wasn't coverage achieved with functionally sensible paths?

4. Be comfortable with your chosen paths. Play your hunches and give your intuition free reign as long as you achieve C1 + C2.

5. Don't follow rules slavishly—except for coverage.

2.4. Loops

2.4.1. The Kinds of Loops

I had a physics professor who said that there were only two kinds of mensuration systems: the metric and the barbaric. I say that there are only three kinds of loops: **nested, concatenated,** and **horrible.** Figure 3.10 shows examples of each kind.

2.4.2. Cases for a Single Loop

A single loop can be covered with two cases: looping and not looping. But experience shows that many loop–related bugs are not discovered by C1 + C2. Bugs lurk in corners and congregate at boundaries—in the case of loops, at or around the minimum and maximum number of times the loop can be iterated. The minimum number of iterations is often zero, but it need not be.

Case 1—Single Loop, Zero Minimum, N Maximum, No Excluded Values

1. Try bypassing the loop (zero iterations). If you can't, you either have a bug, or zero is not the minimum and you have the wrong case.

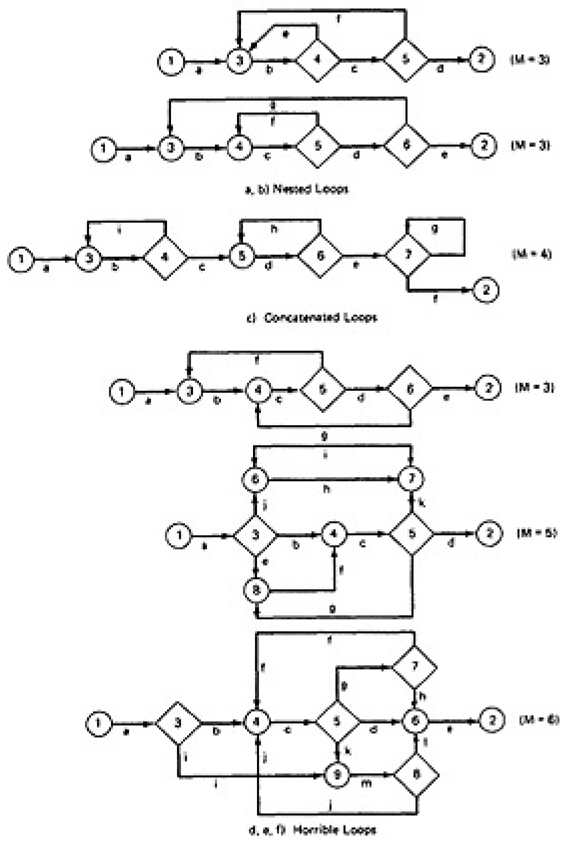


Figure 3.10. Examples of Loop Types.

2. Could the loop–control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?

3. One pass through the loop.
4. Two passes through the loop for reasons discussed below.
5. A typical number of iterations, unless covered by a previous test.
6. One less than the maximum number of iterations.
7. The maximum number of iterations.
8. Attempt one more than the maximum number of iterations. What prevents the loop–control variable from having this value? What will happen with this value if it is forced?

The reason for two passes through the loop is based on a theorem by Huang (HUAN79) that states that some data–flow anomalies, such as some initialization problems, can be detected only by two passes through the loop. The problem occurs when data are initialized within the loop and referenced after leaving the loop. If, because of bugs, a variable is defined within the loop but is not referenced or used in the loop, only two traversals of the loop would show the double initialization. Similar problems are discussed in further detail in Chapter 5.

Case 2—Single Loop, Nonzero Minimum, No Excluded Values

1. Try one less than the expected minimum. What happens if the loop control variable's value is less than the minimum? What prevents the value from being less than the minimum?
2. The minimum number of iterations.
3. One more than the minimum number of iterations.
4. Once, unless covered by a previous test.
5. Twice, unless covered by a previous test.
6. A typical value.
7. One less than the maximum value.
8. The maximum number of iterations.
9. Attempt one more than the maximum number of iterations.

Case 3—Single Loops with Excluded Values

Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as Cases 1 and 2 above. Say that the total range of the loop–control variable was 1 to 20, but that values 7, 8, 9, and 10 were excluded. The two sets of tests are 1–6 and 11–20. The test cases to attempt would be 0,1,2,4,6,7, for the first range and 10,11,15,19,20,21, for the second range. The underlined cases are not supposed to work, but they should be attempted. Similarly, you might want to try a value within the excluded range, such as 8. If you had two sets of excluded values, you would have three sets of tests, one for each allowed range. If the excluded values are very systematic and easily typified, this approach would entail too many tests. Say that all odd values were excluded. I would test the extreme points of the range as if there were no excluded values, for the extreme points, for the excluded values, and also for typical excluded values. For example, if the range is 0 to 20 and odd values are excluded, try – 1,0, 1,2,3,10,11,18,19,20,21,22.

2.4.3. Nested Loops

If you had five tests (assuming that one less than the minimum and one more than the maximum were not achievable) for one loop, a pair of nested loops would require 25 tests, and three nested loops would require 125. This is heavy even by my standards. You can't always afford to test all combinations of nested loops' iteration values. Here's a tactic to use to discard some of these values:

1. Start at the innermost loop. Set all the outer loops to their minimum values.
2. Test the minimum, minimum + 1, typical, maximum – 1, and maximum for the innermost loop,

while holding the outer loops at their minimum–iteration–parameter values. Expand the tests as required for out–of–range and excluded values.

3. If you've done the outermost loop, GOTO step 5, ELSE move out one loop and set it up as in step 2—with all other loops set to typical values.

4. Continue outward in this manner until all loops have been covered.

5. Do the five cases for all loops in the nest simultaneously.

This procedure works out to twelve tests for a pair of nested loops, sixteen for three nested loops, and nineteen for four nested loops. Practicality may prevent testing in which all loops achieve their maximum values simultaneously. You may have to compromise. Estimate the processing time for the loop and multiply by the product of loop–iteration variables to estimate the time spent in the loop (for details in the general case and precise methods, see Chapter 8 or BEIZ78). If the expected execution time is several years or centuries, ask yourself whether this is reasonable. Why isn't there a check on the combination of values? Unbounded processing time could indicate a bug.

These cases can be expanded by taking into account potential problems associated with initialization of variables and with excluded combinations and ranges. In general, Huang's twice–through theorem should also be applied to the combination of loops to assure catching data–initialization problems. Hold the outer loops at the minimum values and run the inner loop through its cases. Then hold the outer loop at one and run the inner loop through its cases. Finally, hold the outer loop at two and run through the inner–loop cases. Next, reverse the role of the inner and outer loop and do it over again, excluding cases that have been tried earlier. A similar strategy can be used with combinations of allowed values in one loop and excluded values in another.

2.4.4. Concatenated Loops

Concatenated loops fall between single and nested loops with respect to test cases. Two loops are **concatenated** if it's possible to reach one after exiting the other while still on a path from entrance to exit. If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops. Even if the loops are on the same path and you can be sure that they are independent of each other, you can still treat them as individual loops; but if the iteration values in one loop are directly or indirectly related to the iteration values of another loop, *and* they can occur on the same path, then treat them as you would nested loops. The problem of excessive processing time for combinations of loop–iteration values should not occur because the loop–iteration values are additive rather than multiplicative as they are for nested loops.

2.4.5. Horrible Loops

Although the methods of Chapter 8 may give you some insight into the design of test cases for horrible loops, the resulting cases are not definitive and are usually too many to execute. The thinking required to check the end points and looping values for intertwined loops appears to be unique for each program. It's also difficult at times to see how deeply nested the loops are, or indeed whether there are any nested loops. The use of code that jumps into and out of loops, intersecting loops, hidden loops, and cross–connected loops, makes iteration–value selection for test cases an awesome and ugly task, which is another reason such structures should be avoided.

2.4.6. Loop–Testing Time

Any kind of loop can lead to long testing time, especially if all the extreme value cases are to be attempted (MAX – 1, MAX, MAX + 1). This situation is obviously worse for nested and dependent concatenated loops. In the context of real testing, most tests take a fraction of a second to execute, and even deeply nested loops can be tested in seconds or minutes. I said earlier that unreasonably long test execution times (i.e., hours or centuries) could indicate bugs in the software or the specification. Consider nested loops in which testing the combination of extreme values leads to long test times. You have several options:

1. Show that the combined execution time results from an unreasonable or incorrect specification. Fix the specification.
2. Prove that although the combined extreme cases are hypothetically possible, they are not possible in the real world. That is, the combined extreme cases cannot occur. Don’t test the cases and accept the risk. The risk is high because hardware glitches or human input errors could drive you into the case.
3. Put in limits or checks that prevent the combined extreme cases. Then you have to test the software that implements such safety measures.
4. Test with the extreme–value combinations, but use different numbers.

What goes wrong at loop limits, especially at simultaneous limits on several nested or dependent loops? It’s rarely the specific numbers; it’s the fact that several limits are hit simultaneously. For example, three nested loops have limits at 10,000, 200, and 8,000 for a total of 16,000,000,000 iterations for one extreme–value check. The bug, if any, is not likely to be associated with the specific values 10,000, 200, and 8,000. Almost any other three limiting values would show the bug—for example, 100, 20, and 80. The new limiting values may not make operational sense, nor need they, because you’re not trying to simulate reality, you’re trying to break the software. The test time problem is solved by rescaling the test limit values to a set of numbers that lead to reasonable execution times. This can be done by a separate compile, by patching, by setting parameter values, etc. The only numbers that may be special are binary powers such as 2, 4, 8, and especially 256, and 65,536. These values should be avoided in rescaling a problem lest a peculiarity of the number mask a bug.

2.5. More on Testing Multi–Entry/Multi–Exit Routines

2.5.1. A Weak Approach

Suppose we have to test a program that has multi–entry and/or multi–exit routines. An approach to testing such junk, assuming that we cannot get the routines changed, is to create fictitious single–entry segments with fictitious case statements and fictitious processes that set fictitious exit parameters and go to a fictitious common junction. This fictitious code (provided you don’t get trapped into confusing it with real code) will help you organize the test case design and keep the control flow relatively clean from an analytical point of view (see Figure 3.11).

This technique involves a lot of extra work because you must examine the cross–reference listings to find all references to the labels that correspond to the multiple entries. But it can be rewarding. Present the fictitious input case statement and the list of entrances to the designer of the multi–entry routine, and you may be rewarded with a statement like, “That’s not right! So–and–so isn’t supposed to come in via that entry point.” Similarly, every exit destination corresponding to your pseudoexit parameter should be discussed with the designer of the multi–exit routine, and again you may find that some targets are not correct, although this is less likely to happen because multiple entries are more likely to be misused than multiple exits. After all, the designers of routines should know how they want to exit, but it’s difficult to control an entry that can be initiated by many other programmers.

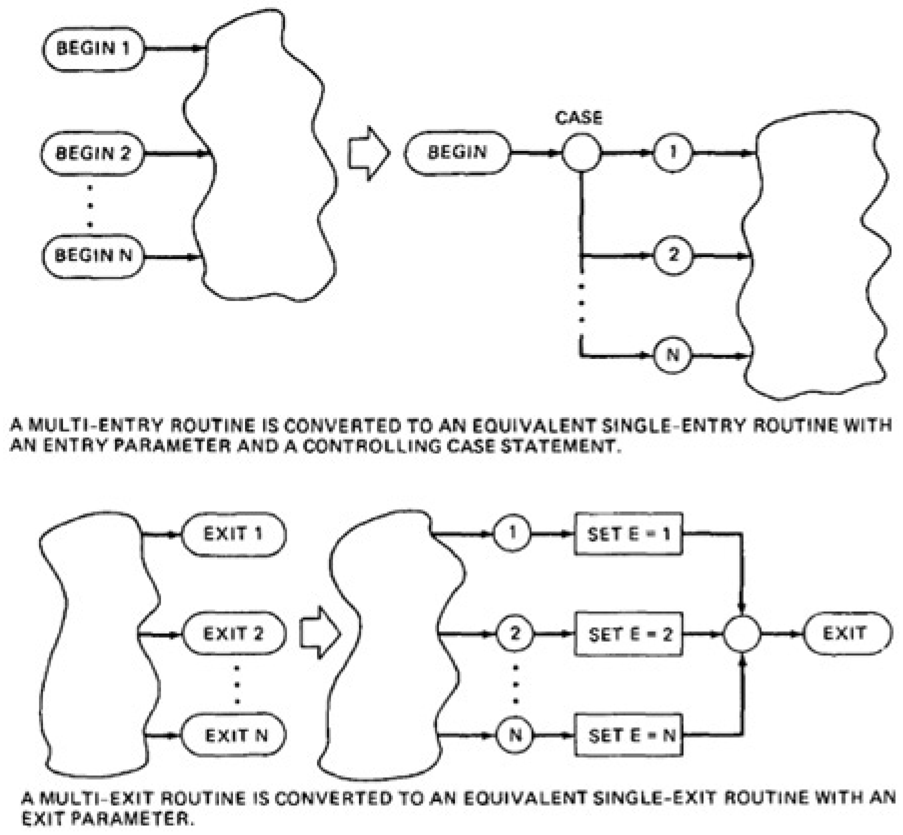

BEGIN 1
BEGIN 2
BEGIN N
BEGIN
CASE
1
2
N
A MULTI-ENTRY ROUTINE IS CONVERTED TO AN EQUIVALENT SINGLE-ENTRY ROUTINE WITH AN ENTRY PARAMETER AND A CONTROLLING CASE STATEMENT.
EXIT 1
EXIT 2
EXIT N
1
2
N
SET E = 1
SET E = 2
SET E = N
EXIT
A MULTI-EXIT ROUTINE IS CONVERTED TO AN EQUIVALENT SINGLE-EXIT ROUTINE WITH AN EXIT PARAMETER.

Figure 3.11. Conversion of Multi–Exit or Multi–Entry Routines.

In assembly language it's possible to calculate the address of the entry point either in absolute terms or relative to the routine's origin. Such multiple entries are truly abominable, because absolute address calculations make the routine's operation dependent upon location, and relative address calculations change the control flow when the routine is modified. Furthermore, there is no way to tell, without effectively simulating the machine's operation, just what is going on. Absolute addressing for control purposes should be forbidden in almost all instances, although there are mitigating circumstances in which it is not only desirable, but mandatory. In some computers, it is not possible to write an interrupt–handling routine or a device–control routine without resorting to absolute addresses. Where this is essential, all such code should be centralized, tightly controlled, and approved in writing, one such address at a time.

2.5.2. The Integration Testing Issue

Treating the multi–entry/multi–exit routine as if it were reasonably structured by using a fictional entry case statement and a fictional exit parameter is a weak approach because it does not solve the essential testing problem. The essential problem is an integration testing issue and has to do with paths within called components.

In Figure 3.12a we have a multi–entry routine with three entrances and three different callers. The first entrance is valid for callers A and B, the second is valid only for caller A, and the third is valid for callers B and C. Just testing the entrances (as we would in unit testing) doesn't do the job because in integration testing it's the interface, the validity of the call, that must be established. In integration testing, we would have to do at least two tests for the A and B callers—one for each of their entrances. Note also that, in general, during unit testing we have no idea who the callers are to be.

Figure 3.12b shows the situation for a multi–exit routine. It has three exits, labeled 1, 2, and 3. It can be called by components X or Y. Exits 1 and 2 are valid for the X calls and 2 and 3 are valid for the Y calls. Component testing must not only confirm that exits 1 and 2 are taken for the X calls, but that there are no paths for the X calls that lead to exit 3—and similarly for exit 1 and the Y calls. But when we are doing unit tests, we do not know who will call this routine with what restrictions. As for the multi–entry routine, we must establish the validity of the exit for every caller. Note that we must confirm that not only does the caller take the expected exit, but also that there is no way for the caller to return via the wrong exit.

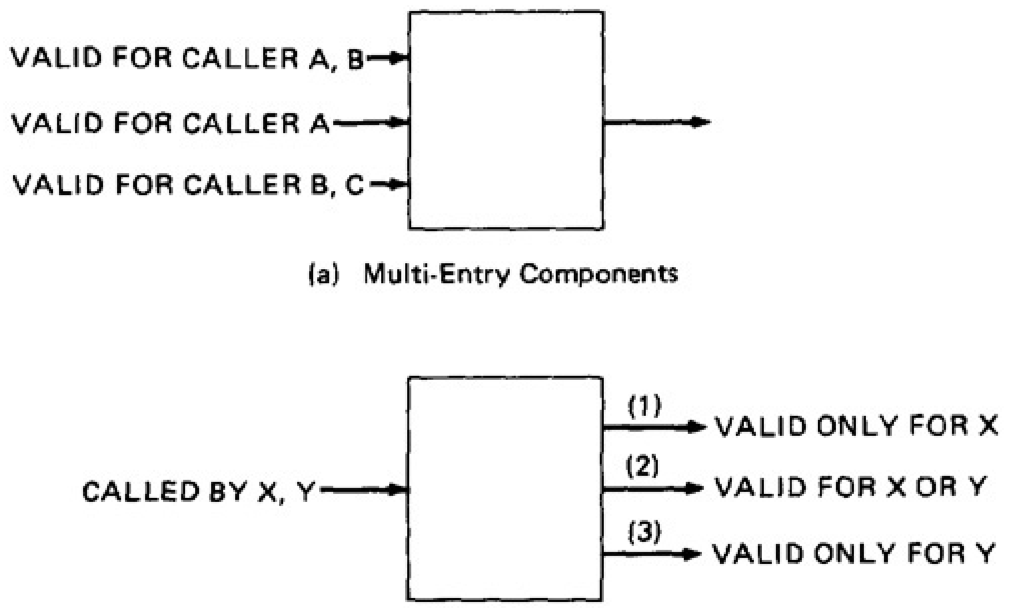


Figure 3.12. Invalid Paths in a Multi–Exit Routine.

When we combine the multi–entry routine with the multi–exit routine, we see that in integration testing we must examine every combination of entry and exit for every caller. Since we don't know, during unit design, which combinations will or will not be valid, unit testing must at least treat each such combination as if it were a separate routine. Thus, a routine with three entrances and four exits results in twelve routines' worth of unit testing. Integration testing is made more complicated in proportion to the number of exits, or fourfold.

In an attempt to save 2% of the coding labor, the programmer has increased unit testing cost twelve–fold and integration testing cost by a factor of 4. If only the routine's designers would actually go through all the testing required, it would not be so bad. Unfortunately, most designers of multi–entry/multi–exit routines do not understand the testing ramifications and, as a consequence, quit at simple C1 or C2 (and we're lucky to get that). It's not that such routines can't be tested, but that they're hardly ever tested adequately.

2.5.3. The Theory and Tools Issue

Another objection to multi–entry/multi–exit routines is that much of testing theory depends on the fact that such routines do not or cannot exist. Most of testing theory assumes that all routines have a single entry and exit. Software which has this property is called **well–formed.** Software which does not have this property is called **ill–formed.**

There are other characterizations of well–formed software: the most common variant is to insist on strict structuring in addition to single–entry/single–exit. In applying any result from theory, it is wise to check the notion of well–formed used and to see whether the software to which the result is to be applied does or does not conform—because the theoretical results may or may not be true in your case. The most common trap occurs in comparing testing strategies. Theory may show that strategy X includes strategy

Y, but only for certain notions of well–formed. In other cases, they may be incomparable and thus, running tests based on X would not guarantee tests based on Y and you would have to do both sets of tests. Because multi–entry/multi–exit routines are ill–formed for most theories, the proofs of how strategies relate to one another may be invalid.

The theory issue extends to tools. Test tools, especially test generators based on structure, embody combinations of formal strategies and heuristics. Some of the strategies may be proprietary and may be based on theoretical results—which may or may not hold for ill–formed software. Consequently, the tool may fail to generate important test cases.

2.5.4. Strategy Summary

The proper way to test multi–entry or multi–exit routines is:

1. Get rid of them.
2. Completely control those you can't get rid of.
3. Augment the flowgraph with fictitious, equivalent, input case statements and exit parameters to help you organize the tests for those that remain and accept the weakness of the approach.
4. Do stronger unit testing by treating each entry/exit combination as if it were a completely different routine.
5. Recognize that you are dealing with a much more complicated beast than you thought, and pay for it in much heavier integration testing.
6. Be sure you understand the strategies and assumptions built into your automatic test generators and confirm that they do (don't) necessarily work for multi–entry/multi–exit routines.

2.6. Effectiveness of Path Testing

2.6.1. Effectiveness and Limitations

Approximately 65% of all bugs can be caught in unit testing, which is dominated by path–testing methods, of which statement and branch testing dominates. Precise statistics based on controlled experiments on the effectiveness of path testing are sparse (but see BASI87). What statistics there are show that path testing catches approximately half of all bugs caught during unit testing or approximately 35% of all bugs (BASI87, BOEH75B, ENDR75, GANN79, GIRG86, HENN84, HOWD76, HOWD78D, KERN76, MILL77C, THAY76). When path testing is combined with other methods, such as limit checks on loops, the percentage of bugs caught rises to 50% to 60% in unit testing. Path testing is more effective for unstructured than for structured software. The statistics also indicate that path testing as a sole technique is limited. Here are some of the reasons:

1. Planning to cover does not mean you will cover. Path testing may not cover if you have bugs.
2. Path testing may not reveal totally wrong or missing functions.
3. Interface errors, particularly at the interface with other routines, may not be caught by unit–level path testing.
4. Database and data–flow errors may not be caught.
5. The routine can pass all of its tests at the unit level, but the possibility that it interferes with or perturbs other routines cannot be determined by unit–level path tests.
6. Not all initialization errors are caught by path testing.
7. Specification errors can't be caught.

2.6.2. A Lot of Work?

Creating the flowgraph, selecting a set of covering paths, finding input data values to force those paths, setting up the loop cases and combinations—it's a lot of work. Perhaps as much work as it took to design the routine and certainly more work than it took to code it. The statistics indicate that you will spend half of your time testing and debugging—presumably that time includes the time required to design and document test cases. I would rather spend a few quiet hours in my office doing test design than twice those hours on the test floor debugging, going half–deaf from the clatter of a high–speed printer that's producing massive dumps, the reading of which will make me half–blind. Furthermore, *the act of careful, complete, systematic, test design will catch as many bugs as the act of testing.* It's worth repeating here and several times more in this book: *The test design process, at all levels, is at least as effective at catching bugs as is running the test designed by that process.* Personally, I believe that it's far more effective, but I don't have statistics to back that claim. And when you consider the fact that bugs caught during test design cost less to fix than bugs caught during testing, it makes test design bug catching even more attractive.

2.6.3. More on How to Do It

Although fancy tools are nice, the only tools you need is the source code listing (or PDL), a yellow marking pen, and a copying machine. At first you may want to create the control flowgraph and use that as a basis for test design, but as you gain experience with practice (and I mean a few days, not months or years), you'll find that you can select the paths directly on the source code without bothering to draw the control flowgraph. If you can path trace through code for debugging purposes then you can just as easily trace through code for test design purposes. And if you can't trace a path through code, are you a programmer?

You do it with code almost the same way as you would with a pictorial control flowgraph. Make several copies of the source. Select your path, marking the statements on the path with the marking pen—but be sure to mark only the executed parts of IF . . . THEN . . . ELSE statements lest you achieve only C1 instead of C2. Transcribe your markings to a master sheet. When it's all yellow, you have a covering path set. Note the difference between C1 and C2. For C1, you only need to check the statement off; for C2, you must also mark out every part of the statement, especially for IF . . . THEN . . . ELSE and case statement parts.

2.7. Variations

Branch and statement coverage as basic testing criteria are well established as effective, reasonable, and easy to implement. How about the more complicated test criteria in the path–testing family? We know that there are an infinite number of such criteria and we seemed, for a while, destined to an infinite number of Ph.D. dissertations that explored them. There are two main classes of variations:

1. Strategies between P_2 and total path testing.
2. Strategies weaker than P_1 or P_2.

The stronger strategies, typically require more complicated path selection criteria, most of which are impractical for human test design. Typically, the strategy has been embedded in a tool that either selects a covering set of paths based on the strategy or helps the programmer to do so. While research can show that strategy A is stronger than B in the sense that all tests generated by B are included in those generated by A, it is much more difficult to ascertain cost–effectiveness. For example, if strategy A takes 100 times as many cases to satisfy as B, the effectiveness of A would depend on the probability that there are bugs of the type caught by A and not by B. We have almost no such statistics and therefore we know very little about the pragmatic effectiveness of this class of variations (but see WEYU88A). A survey of the main strategies can be found in NTAF88.

As an example of how we can build a family of path–testing strategies, consider a family in which we construct paths out of segments that traverse one, two, or three nodes or more. If we build all paths out single–node segments P_1 (hardly to be called a "path," then we have achieved C1. If we use two–node segments (e.g., links = P_2) to construct paths, we achieve C2.

The weaker strategies—that is, those based on doing less than C2 or C1—may seem to directly contradict our position that C2 is a minimum requirement. This observation is true for wholly new software, but not necessarily true for modified software in a maintenance situation. It does not appear to be reasonable to insist that complete C2 testing, say, be redone when only a small part of the software has been modified. Recent research (LEUN88, LINJ89) shows that the incremental testing (i.e., regression testing) situation is not the same as the new software situation and therefore, weaker path–testing strategies may be effective.

3. PREDICATES, PATH PREDICATES, AND ACHIEVABLE PATHS

3.1. General

Selecting a path does not mean that it is achievable. If all decisions are based on variables whose values are independent of the processing and of one another, then all combinations of decision outcomes are possible (2^n outcomes for n binary decisions) and all paths are achievable: in general, this is not so. Every selected path leads to an associated boolean expression, called the **path predicate expression,** which characterizes the input values (if any) that will cause that path to be traversed.

3.2. Predicates

3.2.1. Definition and Examples

The direction taken at a decision depends on the value of decision variables. For binary decisions, decision processing ultimately results in the evaluation of a logical (i.e., boolean) function whose outcome is either TRUE or FALSE. Although the function evaluated at the decision can be numeric or alphanumeric, when the decision is made it is based on a logical function's truth value. The logical function evaluated at a decision is called a **predicate** (GOOD75, HUAN75). Some examples: "A is greater than zero," "the fifth character has a numerical value of 31," "X is either negative or equal to 10," "$X + Y = 3Z^2 - 44$," "Flag 21 is set." Every path corresponds to a succession of TRUE/FALSE values for the predicates traversed on that path. As an example:

> " 'X is greater than zero' is TRUE."

AND

> " '$X + Y = 3Z^2 - 44$' is FALSE."

AND

> " 'W is either negative or equal to 10' is TRUE."

is a sequence of predicates whose truth values will cause the routine to take a specific path. A predicate associated with a path is called a **path predicate.**

3.2.2. Multiway Branches

The path taken through a multiway branch such as computed GOTO's (FORTRAN), case statements (Pascal), or jump tables (assembly language) cannot be directly expressed in TRUE/FALSE terms. Although it is possible to describe such alternatives by using multivalued logic, an easier expedient is to express multiway branches as an equivalent set of IF . . . THEN . . . ELSE statements. For example, a three–way case statement can be written as:

```
IF case=1 DO A1 ELSE
      (IF case=2 DO A2 ELSE DO A3 ENDIF) ENDIF
```

The translation is not unique because there are many ways to create a tree of IF . . . THEN . . . ELSE statements that simulates the multiway branch. We treat multiway branches this way as an analytical convenience in order to talk about testing—we don't replace multiway branches with nested IF's just to test them. Here is another possible variance between our model (the flowgraph) and the real program. Although multiway branches are effective programming construct, we'll talk about predicates as if they are all binary. That means that we'll have to take special care (such as the artificial construct above) to handle multiway predicates.

3.2.3. Inputs

In testing, the word **input** is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it—for example, inputs in a calling sequence, objects in a data structure, values left in a register. Although inputs may be numerical, set members, boolean, integers, strings, or virtually any combination of object types, we can talk about data as if they are numbers. No generality is lost by this practice. Because any array can be mapped onto a one–dimensional array, we can treat the set of inputs to the routine as if it is a one–dimensional array, which we call the **input vector.**

3.3. Predicate Expressions

3.3.1. Predicate Interpretation

The simplest predicate depends only on input variables. For example, if X_1 and X_2 are inputs, the predicate might be "$X_1 + X_2 >= 0$". Given the values of X_1 and X_2 the direction taken through the decision based on the predicate is determined at input time and doesn't depend on processing. Assume that the predicate is "$X_1 + Y >= 0$", that along a path prior to reaching this predicate we had the assignment statement "$Y := X_2 + 7$", and that nothing else on that path affected the value of Y. Although our predicate depends on the processing, we can substitute the symbolic expression for Y to obtain an equivalent predicate "$X_1 + X_2 + 7 >= 0$". The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation.** The interpretation may depend on the path; for example,

```
          INPUT X
          ON X GOTO A, B, C, ...
     A: Z := 7 @ GOTO HEM
     B: Z := –7 @ GOTO HEM
     C: Z := 0 @ GOTO HEM
          .........
HEM: DO SOMETHING
          .........
```

```
    HEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM
```

The predicate interpretation at HEN (the IF statement) depends on the path we took through the first multiway branch (the assigned GOTO). It yields for the three cases respectively, “IF Y + 7 > 0 GOTO. . .”,“IF Y – 7>0 GOTO . . .”, and “IF Y>0 GOTO . . .”. It is also possible that the predicate interpretation does not depend on the path and for that matter, appearances aside, when all is said and done (especially if there are bugs) that after interpretation the predicate does not depend on anything (for example, “7 >= 3”). Because every path can lead to a different interpretation of predicates along that path, a predicate that after interpretation does not depend on input values does not necessarily constitute a bug. Only if all possible interpretations of a predicate are independent of the input could we suspect a bug.

The **path predicates** are the specific form of the predicates of the decisions along the selected path after interpretation. In our discussion of path testing we assume, unless stated otherwise, that all predicates have been interpreted—ignoring for the moment the difficulty that such interpretation could entail.

3.3.2. Independence and Correlation of Variables and Predicates

The path predicates take on truth values (TRUE/FALSE) based on the values of input variables, either directly (interpretation is not required) or indirectly (interpretation is required). If a variable’s value does not change as a result of processing, that variable is **independent** of the processing. Conversely, if the variable’s value can change as a result of the processing the *variable* is **process dependent.** Similarly, a *predicate* whose truth value can change as a result of the processing is said to be **process dependent** and one whose truth value does not change as a result of the processing is **process independent.** Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based. For example, the input variables are X and Y and the predicate is “X + Y = 10”. The processing increments X and decrements Y. Although the numerical values of X and Y are process dependent, the predicate “X + Y = 10” is process independent. As another example, the predicate is “X is odd” and the process increments X by an even number. Again, X’s value is process dependent but the predicate is process independent. However, if all the variables on which a predicate is based are process independent, it follows that the predicate must be process independent and therefore its truth value is determined by the inputs directly. Also keep in mind that we are looking only at those variables whose values can affect the control flow of the routine and not at all variables whose values may change as a result of processing.

Variables, whether process dependent or independent, may be **correlated** to one another. Two variables are **correlated** if every combination of their values cannot be independently specified. For example, two 8–bit variables should lead to 2^{16} combinations. If there is a restriction on their sum, say the sum must be less than or equal to 2^8, then only 2^9 combinations are possible. Variables whose values can be specified independently without restriction are **uncorrelated.** By analogy, a pair of predicates whose outcomes depend on one or more variables in common (whether or not those variables are correlated) are said to be **correlated predicates.** As an example, let X and Y be two input variables that are independent of the processing and are not correlated with one another. Let decision 10 be based on the predicate “X = Y” and decision 12 on the predicate “X + Y = 8”. If we select values for X and Y to satisfy decision 10, we may have forced the predicate’s truth value for decision 12 and may not be able to make that decision branch the way we wish. Every path through a routine is achievable only if all predicates in that routine are uncorrelated. If a routine has a loop, then at least one decision’s predicate must depend on the processing or there is an input value that will cause the routine to loop indefinitely.

3.3.3. Path Predicate Expressions

The following is a conceptual exercise—an aid to understanding testing issues, but not the way you design test cases. Select an (entry/exit) path through a routine. Write down the uninterpreted predicates for the decisions you meet along the path, being sure if there are loops, to distinguish each passage through the predicates in the loop (or the loop–control predicate) by noting the value of the loop–control variable that applies to that pass. Interpret the predicates to convert them into predicates that contain only input variables. The result of this mental exercise is a set of boolean expressions, all of which must be satisfied to achieve the selected path. This set is called the **path predicate expression.** Assuming (for the sake of our example) that the input variables are numerical, the expression is equivalent to a set of inequalities such as

$$X_1 + 3X_2 + 17 >= 0$$
$$X_3 = 17$$
$$X_4 - X_1 >= 14X_2$$

Any set of input values that satisfy *all* of the conditions of the path predicate expression will force the routine to the path. If there is no such set of inputs, the path is not achievable. The situation can be more complicated because a predicate could have an .OR. in it. For example:

IF $X_5 > 0$ .OR. $X_6 > 0$ THEN . . .

A single .OR., such as the above, gives us two sets of expressions, either of which, if solved, forces the path. If we added the above expression to the original three we would have the following two sets of inequalities:

A: $X_5 > 0$	E: $X_6 < 0$
B: $X_1 + 3X_2 + 17 >= 0$	B: $X_1 + 3X_2 + 17 >= 0$
C: $X_3 = 17$	C: $X_3 = 17$
D: $X_4 - X_1 >= 14X_2$	D: $X_4 - X_1 >= 14X_2$

We can simplify our notation by using an uppercase letter to denote each predicate's truth value and then use boolean algebra notation to denote the predicate expression: concatenation means "AND", a plus sign means "OR", and negation is denoted by an overscore. The above example, using the boolean variable names shown above then becomes

$$ABCD + EBCD = (A+E)BCD$$

If we had taken the opposite branch at the fourth predicate, the inequality would be $X_4 - X_1 < 14X_2$ and the resulting predicate expression for that path would be $(A+E)BC\overline{D}$

3.4. Predicate Coverage

3.4.1. Compound Predicates

Most programming languages permit **compound predicates** at decisions—that is, predicates of the form A .OR. B or A .AND. B. and more complicated boolean expressions. The branch taken at such decisions is determined by the truth value of the entire boolean expression. Even if a given decision's predicate is not compound, it may become compound after interpretation because interpretation may require us to carry forward a compound term. Also, a simple negation can introduce a compound predicate. For example, say the predicate at some decision is

X = 17 (that is, IF X=17 THEN . . .)

The opposite branch is X .NE. 17, which is equivalent to X > 17 .OR. X < 17. Therefore, whether or not the language permits compound predicates, we can at any decision have an arbitrarily complicated compound predicate after interpretation.

3.4.2. Predicate Coverage

Assume for the sake of discussion that a predicate on a selected path is a compound predicate which depends directly on the input vector—i.e., interpretation is not required. Consider a compound predicate such as A+B+C+D. It consists of four subsidiary predicates, any of which must be true for the branch to be taken. We don't know, offhand, the order in which the subsidiary predicates will be evaluated: that depends on how the compiler implements things. It could be A,B,C,D, or more likely, D,C,B,A. It's difficult to predict the sequence in which the predicates are evaluated because the compiler can rearrange that sequence to optimize the code. Let's say it evaluates the terms in the order A,B,C,D. Typically (but not necessarily) the compiler will create code which will stop predicate evaluation as soon as the truth value of the compound predicate is determined. For example, if the first term (A) is true, evaluation stops because the direction of the branch is determined. Therefore, the desired path could be taken, but there could still be a bug in predicate B that would not be discovered by this test. The same situation holds for a form such as ABCD for the FALSE branch, because the negation of ABCD is

Returning to the general case in which we allow arbitrarily complicated compound predicates at any decision and in which compound predicates arise as a result of interpretation, it is clear that achieving the desired direction at a given decision could still hide bugs in the associated predicates. The desired path is achieved for the test case you chose, but for some other case, in which the truth value of the controlling predicate is determined in some other order, the branch goes the wrong way. For example, the predicate is A + B and A is correct, but B is buggy. The first test makes A true, and consequently B is not exercised. The next test case (the one we didn't try) has A false and B should be true, which would make the path go the same way as the first case. The bug in B makes B false and consequently we have the wrong path. An even nastier case is A + B + C because the intended path will be taken despite the bug in B because B's failure is masked by the fact that C is true.

We'll have a lot more to say about predicates and boolean algebra in Chapter 10. For now, it's enough to realize that achieving C2 could still hide control flow bugs. A stronger notion of coverage is indicated: **predicate coverage.** We say that **predicate coverage** has been achieved if all possible combinations of truth values corresponding to the selected path have been explored under some test. Predicate coverage is clearly stronger than branch coverage. If all possible combinations of all predicates under all interpretations are covered, we have the equivalent of total path testing. Just as there are hierarchies of path testing based on path–segment link lengths, we can construct hierarchies based on different notions of predicate coverage.

3.5. Testing Blindness

3.5.1. The Problem

Let's leave compound predicates for now and return to simple predicates. Is it enough to run one path through a predicate to test its validity? Consider the following example:

```
          IF A GOTO BOB ELSE GOTO COB
BOB:DO SOMETHING
          . . .
          GOTO SAM
COB:DO SOMETHING ELSE
          . . .
          GOTO SAM
          . . .
SAM:IF X DO ALPHA ELSE DO BETA
```

Our question concerns the X predicate in SAM. We can reach it either via the BOB sequence or via the COB sequence. Whichever we do, there are two alternatives at SAM and presumably, two different cases would be sufficient to test the SAM statement. Is it possible that we might have to actually do four cases, corresponding to two cases for reaching SAM via BOB and two more cases for reaching SAM via COB? The answer, unfortunately, is yes, because of **testing blindness** (ZEIL81).

Testing blindness is a pathological situation in which the desired path is achieved for the wrong reason. It can occur because of the interaction of two or more statements that makes the buggy predicate "work" despite its bug and because of an unfortunate selection of input values that does not reveal the situation. Zeil (ZEIL81, WHIT87) discusses three kinds of predicate blindness: **assignment blindness, equality blindness,** and **self–blindness.** There are probably many more kinds of blindness yet to be discovered. We don't know whether instances of different kinds of blindness bugs are significant to the point where it pays to design special test methods to discover them. The point of discussing blindness is to expose further limitations of path testing and to justify the need for other strategies.

3.5.2. Assignment Blindness

Assignment blindness occurs when the buggy predicate appears to work correctly because the specific value chosen for an assignment statement works with both the correct and incorrect predicate. Here's an example:

Correct	*Buggy*
X := 7	X := 7
.....	.....
IF Y > 0 THEN	IF X + Y > 0 THEN

If the test case sets Y := 1 the desired path is taken in either case, but there is still a bug. Some other path that leads to the same predicate could have a different assignment value for X, so the wrong path would be taken because of the error in the predicate.

3.5.3. Equality Blindness

Equality blindness occurs when the path selected by a prior predicate results in a value that works both for the correct and buggy predicate. For example,

Correct	*Buggy*
IF Y = 2 THEN. . .	IF Y = 2 THEN. . .
.....	.....
IF X + Y > 3 THEN. . .	IF X > 1 THEN. . .

The first predicate (IF Y = 2) forces the rest of the path, so that for any positive value of X, the path taken at the second predicate will be the same for the correct and buggy versions.

3.5.4. Self–Blindness

Self–blindness occurs when the buggy predicate is a multiple of the correct predicate and as a result is indistinguishable along that path. For example,

```
        Correct                 Buggy
X := A                  X := A
.....                   .....
IF X – 1 > 0 THEN...    IF X + A – 2 > 0 THEN
```

The assignment (X := A) makes the predicates multiples of each other (for example, A – 1 > 0 and 2A – 2 > 0), so the direction taken is the same for the correct and buggy version. A path with another assignment could behave differently and would reveal the bug.

4. PATH SENSITIZING

4.1. Review; Achievable and Unachievable Paths.

Let’s review the progression of thinking to this point.

1. We want to select and test enough paths to achieve a satisfactory notion of test completeness such as C1 and/or C2.
2. Extract the program’s control flowgraph and select a set of tentative covering paths.
3. For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general, individual predicates are compound or may become compound as a result of interpretation.
4. Trace the path through, multiplying (boolean) the individual compound predicates to achieve a boolean expression such as

(A + BC)(D + E)(FGH)(IJ)(K)(L),

where the terms in the parentheses are the compound predicates met at each decision along the path and each letter (A, B, . . .) stands for simple predicates.
5. Multiply out the expression to achieve a **sum–of–products form:**
ADFGHIJKL + AEFGHIJKL + BCDFGHIJKL + BCEFGHIJKL
6. Each product term denotes a set of inequalities that, if solved, will yield an input vector that will drive the routine along the designated path. Remember that equalities are a special case of inequalities and that these need not be numerical; for example, A could mean “STRING ALPHA = ‘Help Me’ ” and B could mean “BIT 17 IS SET”.
7. Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.

If you can find a solution, then the path is **achievable.** If you can’t find a solution to any of the sets of inequalities, the path is **unachievable.** The act of finding a set of solutions to the path predicate expression is called **path sensitization.**

Is there a general algorithm that will solve the inequalities in order to sensitize the path and failing that, tell us that there is no solution? The answer is a resounding no! The question is known to lead to unsolvable problems from several different points of view. In the pragmatic world of testing we never let trivial things such as provable unsolvability get in our way—especially if there are effective heuristics around.

4.2. Pragmatic Observations

The purpose of the above discussion has been to explore the sensitization issues and to provide insight into tools that help us sensitize paths. If in practice you really had to do the above in the manner indicated then test design would be a difficult procedure suitable only to the mathematically inclined. It doesn't go that way in practice: it's much easier. You select a path and with little fuss or bother you determine the required input vector. Furthermore, if there is any difficulty, it's likelier that there are bugs rather than a truly difficult sensitization problem.

4.3. Heuristic Procedures for Sensitizing Paths

Here is a workable approach. Instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.

1. Identify all variables that affect the decisions. Classify them as to whether they are process dependent or independent. Identify correlated input variables. For dependent variables, express the nature of the process dependency as an equation, function, or whatever is convenient and clear. For correlated variables, express the logical, arithmetic, or functional relation that defines the correlation.
2. Classify the predicates as dependent or independent. A predicate based only on independent input variables must be independent. Identify correlated predicates and document the nature of the correlation as for variables. If the same predicate appears at more than one decision, the decisions are obviously correlated.
3. Start path selection with uncorrelated, independent predicates. Cover as much as you can. If you achieve coverage and you had identified supposedly dependent predicates, something is wrong. Here are some of the possibilities:
  - **a.** The predicates are correlated and/or dependent in such a way as to nullify the dependency. The routine's logic can probably be simplified. See the methods of Chapter 10.
  - **b.** The predicates are incorrectly classified. Check your work.
  - **c.** Your path tracing is faulty. Look for a missing path or incomplete coverage.
  - **d.** There is a bug.

4. If coverage hasn't been achieved using independent uncorrelated predicates, extend the path set by using correlated predicates; preferably those whose resolution is independent of the processing ––i.e., those that don't need interpretation.
5. If coverage hasn't been achieved, extend the cases to those that involve dependent predicates (typically required to cover loops), preferably those that are not correlated.
6. Last, use correlated, dependent predicates.
7. For each path selected above, list the input variables corresponding to the predicates required to force the path. If the variable is independent, list its value. If the variable is dependent, list the relation that will make the predicate go the right way (i.e., interpret the predicate). If the variable is correlated, state the nature of the correlation to other variables. Examine forbidden combinations (if any) in detail. Determine the mechanism by which forbidden combinations of values are prohibited. If nothing prevents such combinations, what will happen if they are supplied as inputs?
8. Each path will yield a set of inequalities, which must be simultaneously satisfied to force the path.

4.4. Examples

4.4.1. Simple, Independent, Uncorrelated Predicates

The uppercase letters in the decision boxes of Figure 3.13 represent the predicates. The capital letters on the links following the decisions indicate whether the predicate is true (unbarred) or false (barred) for that link. There are four decisions in this example and, consequently, four predicates. Because they are uncorrelated and independent by assumption, each can take on TRUE or FALSE independently, leading to $2^4 = 16$ possible values; but the number of possible paths is far less (8) and the number of covering paths is smaller still. A set of covering paths and their associated predicate truth values can be trivially obtained from the flowgraph:

Path	*Predicate Values*
abcdef	$A\ \bar{C}$
aghcimkf	$\bar{A}\ B\ C\ D$
aglmjef	$\bar{A}\ \bar{B}\ \bar{D}$

A glance at the path column shows that all links are represented at least once. The predicate value column shows all predicates appearing both barred and unbarred. Therefore, every link has been covered and every decision has been taken both ways. I violated my own rules here because two things changed on the second path. Using a few more but simpler paths with fewer changes to cover the same flowgraph, I get:

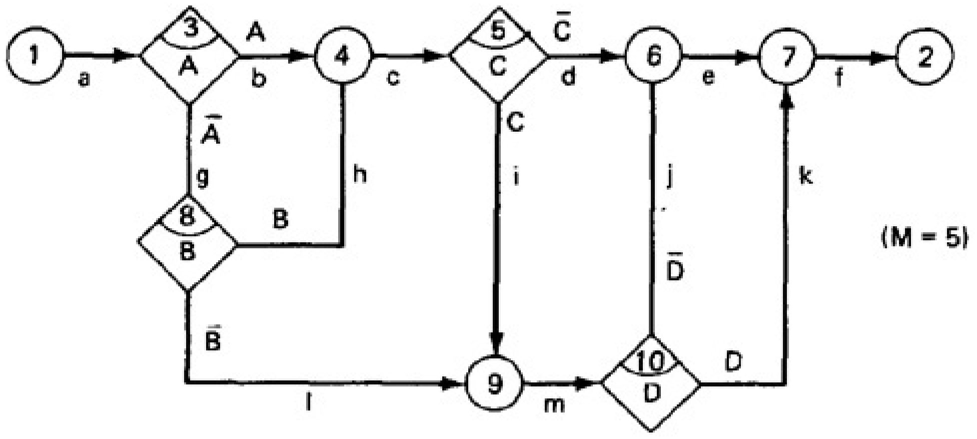


Figure 3.13. Predicate Notation.

Path	*Predicate Values*
abcdef	$A\ \bar{C}$
abcimjef	$A\ C\ \bar{D}$
abcimkf	$A\ C\ D$
aghcdef	$\bar{A}\ B\ \bar{C}$
aglmkf	$\bar{A}\ \bar{B}\ D$

Because you know what each predicate means (for example, A means “X = 0?”), you can now determine the set of input values corresponding to each path.

4.4.2. Correlated, Independent Predicates

The two decisions in Figure 3.14 are correlated because they used the identical predicate (A). If you picked paths *abdeg* and *acdfg,* which seem to provide coverage, you would find that neither of these paths is achievable. If the A branch (*c*) is taken at the first decision, then the A branch (*e*) must also be taken at the second decision. There are two decisions and therefore a potential for four paths, but only two of them, *abdfg* and *acdeg,* are achievable. The flowgraph can be replaced with Figure 3.15, in which we have reproduced the common code, or alternatively, we can embed the common link *d* code into a subroutine.

Predicates correlation will not often be so blatant as when both decisions have exactly the same tests on the same variables. Suppose you didn't know or didn't realize that some predicates were correlated. You would find that setting one decision's variables would force another's and that you did not have the freedom to choose the rest of the path. In general, correlated predicates mean that some paths are not achievable, although this does not mean that coverage is unachievable. If you select paths from the design flowgraph or the PDL specification without considering the details of path sensitization and subsequently you find that a path is not achievable, even though it was selected on the basis of seemingly meaningful cases, it is an occasion for joy rather than frustration. One of the following must be true:

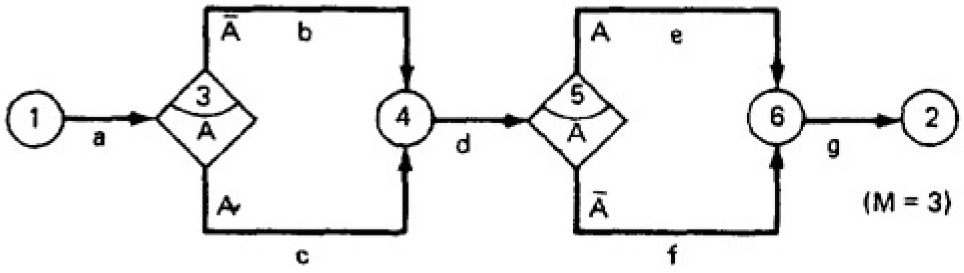


Figure 3.14. Correlated Decisions.

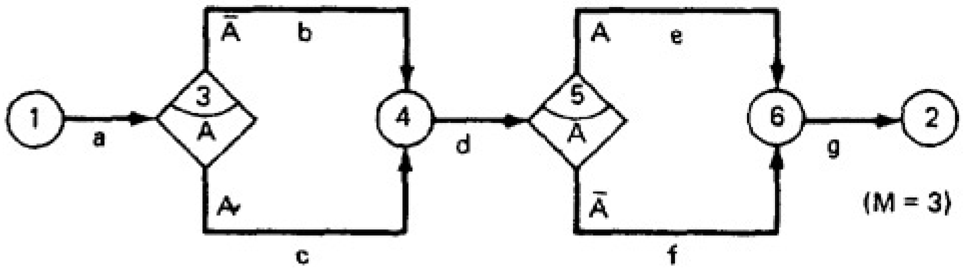


Figure 3.15. Correlated Decision of Figure 3.14 Removed.

1. You found a bug.
2. The design can be simplified by removing some decisions, possibly at the cost of a new subroutine or repeated code.
3. You have a better understanding of how the decisions are interrelated.

The question to ask is: "How did the decisions come to be correlated?" Correlated decisions are redundant. If you have n decisions in a routine and less than 2^n paths, there is redundancy in the design, even though you might not be able to take advantage of it. If n decisions give you only 2^{n-1} paths, then one decision is targeted for removal. If n decisions give 2^{n-2} paths, then two decisions should be targeted. Generally, $\log_2$(number of paths) tells you how many effective decisions you have. Comparing this with the actual number of decisions tells you whether any can be targeted for removal. Because getting rid of code is the best possible kind of test,[*] removing potentially redundant code is good testing at its best.

[*]The highest goal of testing is bug prevention. If bugs occur at 1% per source statement, then credit yourself with a bug "found" and "corrected" for every hundred lines of code you eliminate. Adjust this to the local bug rate. Bad routines tend to have a lot of superfluous code and a high bug rate—say, 3% to 4%; you can earn your pay on such routines.

One common and troublesome source of correlated decisions is the reprehensible practice of "saving code." Link *d* in the above example is typical. The designer had thought to save common code by doing the initial processing that followed the first decision, merging to execute the common code, and then splitting again to do the different code based on the second decision. It's relatively harmless in this example. Most often, the second decision will be based on a flag that was set on the link appropriate to the predicate's value at the first decision. The second decision is based on the flag's value, but it is obviously correlated to the first. Think of these two pieces as being widely separated and embedded within a larger, more complicated routine. Now the nature of the correlation is subtle and obscure. The way it's usually done, though, is not even this sensible. The programmer sees a sequence of code that seems to correspond to part of a case that's yet to be programmed. He jumps into the middle of the old code to take advantage of the supposedly common code and puts in a subsequent test to avoid doing the wrong processing and to get him back onto the correct path. Maintaining such code is a nightmare because such "code–saving" tricks are rarely documented. This can lead to future bugs even though there are no bugs at first. The potential for future bugs exists because the correlation is essential. If maintenance removes the correlation, then the previously unachievable (and undesirable) paths become possible.

Correlated decisions on a path, especially those that prevent functionally sensible paths, are suspect. At best they provide an opportunity for simplifying the routine's control structure and therefore for saving testing labor, or they may reveal a bug. It's rare that correlated decisions that prevent functionally sensible paths are due to subtleties in the requirements.

4.4.3. Dependent Predicates

Finding sensitizing values for dependent predicates may force you to "play computer." Usually, and thankfully, most of the routine's processing does not affect the control flow and consequently can be ignored. Simulate the computer only to the extent necessary to force paths. Loops are the most common kind of dependent predicates; the number of times a typical routine will iterate in the loop is usually determinable in a straightforward manner from the input variables' values. Consequently it is usually easy to work backward to determine the input value that will force the loop a specified number of times.

4.4.4. The General Case

There is no simple procedure for the general case. It is easy to state the steps involved but much harder to accomplish them.

1. Select cases to provide coverage on the basis of functionally sensible paths. If the routine is well structured, you should be able to force most of the paths without deep analysis. Intractable paths should be examined for potential bugs before investing time solving equations or whatever you might have to do to find path–forcing input values.
2. Tackle the path with the fewest decisions first. Give preference to nonlooping paths over looping paths.
3. Start at the end of the path and not the beginning. Trace the path in reverse and list the predicates in the order in which they appear. The first predicate (the last on the path in the normal direction) imposes restrictions on subsequent predicates (previous when reckoned in the normal path direction). Determine the broadest possible range of values for the predicate that will satisfy the desired path direction.
4. Continue working backward along the path to the next decision. The next decision may be restricted by the range of values you determined for the previous decision (in the backward direction). Pick a range of values for the affected variables as broad as possible for the desired direction and consistent with the set of values thus far determined.
5. Continue until you reach the entrance and therefore have established a set of input conditions

for the entire path.

Whatever manipulations you do can always be reduced to equivalent numerical operations. What you're doing in tracing the path backward is building a set of numerical inequalities or a combination of numerical inequalities, equations, and logical statements. You're trying to find input values that satisfy all of them—the values that force the path. If you can't find such a solution, the path is unachievable. Alternatively, it means that you couldn't solve the set of inequalities. Most likely it means that you've found a bug.

An alternate approach to use when you are truly faced with functionally sensible paths that require equation solving to sensitize is a little easier. Instead of working backward along the path, work forward from the first decision to the last. To do this, though, you may have to drop your preconceived paths. Let's say that you have already sensitized some paths and all the rest seem to be dependent and/or correlated.

1. List the decisions that have yet to be covered in the order in which you expect to traverse them. For each decision, write down the broadest possible range of input values that affect that decision.
2. Pick a direction at the first decision on the path that appears to go in the direction you want to go. Adjust all input values—that is, the range of input values—that are affected by your choice. For example, X was restricted to positive integers at input and Y was any letter between "D" and "G". The first decision restricted X to less than 10 and Y to "E" or "F". This restricted set of input values is now used for the next decision.
3. Continue, decision by decision, always picking a direction that gets you closer to the exit. Because the predicates are dependent and/or correlated, your earlier choices may force subsequent directions.
4. Assuming that the procedure does not lead to impossible or contradictory input values (which means that the attempted path is not achievable), start a new path using the last decision at which you had a choice, assuming that such a path will provide additional coverage.

The advantage of the forward method over the backward method is that it's usually less work to get the input values because you are solving the simultaneous inequalities as you go. The disadvantage is that you don't quite know where you're going. The routine is the master and not you.

5. PATH INSTRUMENTATION

5.1. The Problem

We haven't said much about the outcome of the test—for example, the outputs produced as a result of the processing along the selected path. Note that we use the word **"outcome"** rather than "output." The **outcome** of a test is what we expect to happen as a result of the test. That includes outputs, of course, but what if the desired outcome is that there be no output? As with inputs, test outcomes include anything we can observe in the computer's memory, mass storage, I/O, registers, that should have changed as a result of the test, or *not* changed as a result of the test—that bears repetition: *the expected outcome includes any expected changes or the* lack *of change* (*if that's what's expected*). We're not doing kiddie testing, so we predict the outcome of the test as part of the test design process. We then run the test, observe the actual outcome, and compare that outcome to the expected outcome. If the predicted and actual outcomes match, can we say that the test has been passed? We cannot. We can only say that some of the necessary conditions for passing are satisfied, but the conditions are not sufficient because the desired outcome could have been achieved for the wrong reason. This situation is called **coincidental correctness.** Continuing in this (paranoid) vein, assume that we ran a covering set of tests and achieved the desired outcomes for each case. Can we say that we've covered? Again no because the desired outcome could have been achieved by the wrong path.

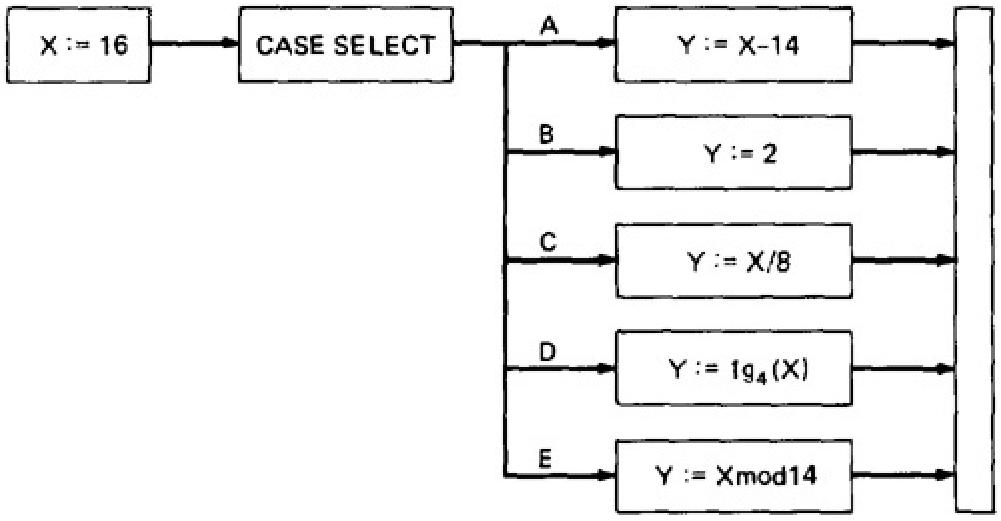


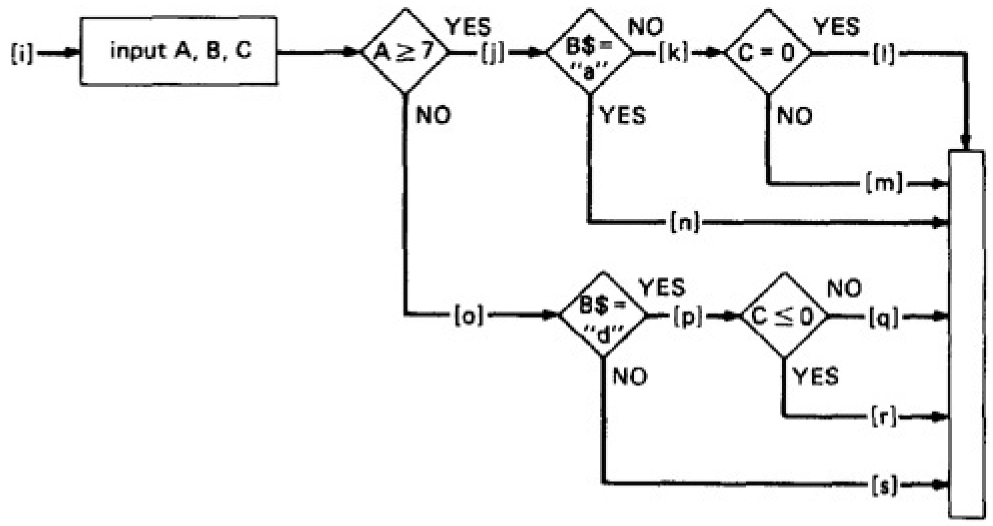
Figure 3.16. Coincidental Correctness.

Figure 3.16 is an example of a routine that, for the (unfortunately) chosen input value (X = 16), yields the same outcome (Y = 2) no matter which case we select. Therefore, the tests chosen this way will not tell us whether we have achieved coverage. For example, the five cases could be totally jumbled and still the outcome would be the same. **Path instrumentation** is what we have to do to confirm that the outcome was achieved by the intended path.

5.2. General Strategy

All instrumentation methods are a variation on a theme of an interpretive trace. An **interpretive trace program,** you'll recall, is one that executes every statement in order and records the intermediate values of all calculations, the statement labels traversed, etc. It's often part of a symbolic debugging package. If we run the tested routine under a trace, then we have all the information we need to confirm the outcome and, furthermore, to confirm that it was achieved by the intended path. The trouble with traces is that they give us far more information than we need. In fact, the typical trace program provides so much information that confirming the path from its massive output dump is more work than simulating the computer by hand to confirm the path. Even trace programs that provide some control over what is to be dumped usually do not give us the specific information we need for testing. Recognizing the limitations of the classical trace packages or symbolic debuggers, a variety of instrumentation methods more suitable to the testing process have emerged.

Figure 3.17. Single Link Marker Instrumentation.



5.3. Link Markers

A simple and effective form of instrumentation (RAMA75B) is called a **traversal marker** or **link marker.** Name every link by a lowercase letter. Instrument the links so that the link's name is recorded when the link is executed. The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name (i.e., the "name" you get when you concatenate the link names traversed on the path—e.g., *abcde*); see Figure 3.17.

Unfortunately, a single link marker may not do the trick because links can be chewed open by bugs. The situation is illustrated in Figure 3.18. We intended to traverse the *ikm* path, but because of a rampaging GOTO in the middle of the *m* link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug. The solution is to implement two markers per link: one at the beginning of each link and one at the end; see Figure 3.19. The two link markers now specify the path name and confirm both the beginning and end of the link.

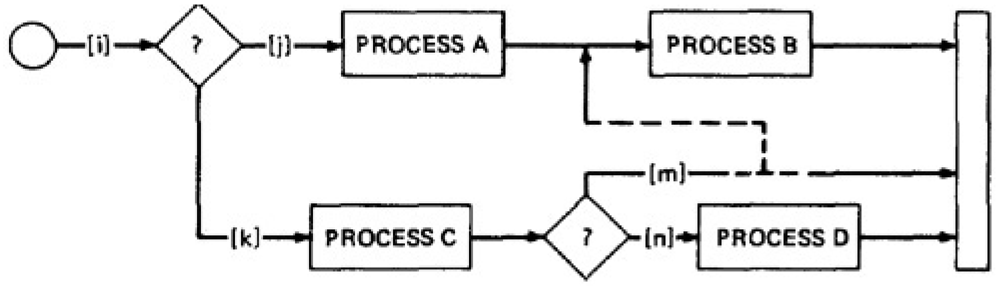


Figure 3.18. Why Single Link Markers Aren't Enough.

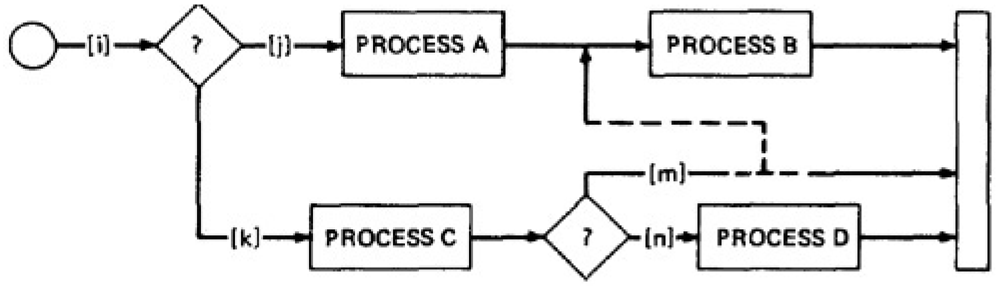


Figure 3.19. Double Link Markers.

5.4. Link Counters

A less disruptive (and less informative) instrumentation method is based on counters. Instead of a unique link name to be pushed into a string when the link is traversed, we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to double link counters. With these in place, we expect an even count that is exactly double the expected path length. This is probably paring instrumentation down to the point where it’s almost useless, so we’ll get a little more complicated and put a counter on every link. Then we can easily predict what the link counts should be for any test. If there are no loops, they will all equal 1. Similarly, we can accumulate counts over a series of tests, say a covering set, and confirm that the total link counts equals the sums we would expect from the series. The same reasoning as before leads us to double link counters—one at the start of the link and one at the end. The checkout procedure then consists of answering the following questions:

1. Do the begin–link counter values equal the end–link counter values for all links?
2. Do the input–link count of every decision equal the sum of the link counts of the output links from that decision?
3. Do the sum of the input–link counts for a junction equal the output–link count for that junction?
4. Do the counts match the values you predicted when you designed the test?

5.5. Other Instrumentation Methods.

The methods you can use to instrument paths are limited only by your imagination. Here’s a sample:

1. Mark each link by a unique prime number and multiply the link name into a central register. The path name is a unique number and you can recapture the links traversed by factoring. The order in which the links were traversed is lost, but it’s unlikely that a bug would cause a reversal of link execution order—but reversals of statements within a link are possible.
2. Use a bit map with a single bit per link and set that bit when the link is traversed (or two bits per link if you want to be more elaborate and protect yourself against link busters).
3. Use a hash coding scheme over the link names, or calculate an error–detecting code over the link names, such as a check sum.
4. Use your symbolic debugger or trace to give you a trace only of subroutine calls and return. Instrument the links that don’t have a real subroutine call on them by putting a call to a dummy subroutine, whose only interesting property is its name.
5. Set a variable value at the beginning of the link to a unique number for that link and use an assertion statement at the end of the link to confirm that you’re still on it. The sequence of satisfied assertions is the path name.

Every instrumentation probe (marker, counter) you insert gives you more information, but with each probe the information is further removed from reality. The ultimate instrumentation, a full interpretive trace, gives you the most information possible but so distorts timing relations that timing and race–condition bugs will hide under your probes. Location–dependent bugs also hide under probes. If, for example, someone made an absolute assumption about the location of something in the data base, the presence or absence of probes could modify things so that the bug could be seen only when the probe was inactive and not with the probe in place. In other words, the very probe designed to reveal bugs may hide them. Such **peek–a–boo** bugs are really tough.

5.6. Implementation

For unit testing, path instrumentation and verification can be provided by a comprehensive test tool that supports your source language. Unfortunately for you, you may be working with an unsupported language, or may be doing higher–level testing, in which case, you’ll have to consider how to install instrumentation the hard way.

The introduction of probes, especially if you have to put them in by hand, provides new opportunities for bugs. Automatically inserted probes are less bug–prone, but can only be inserted in terms of the real rather than the intended structure. This discrepancy can be great, especially if control is affected by what goes on in lower–level routines that are called by the routine under test. Instrumentation is relatively more important when path testing is used at the higher levels of program structure, such as with transaction flows (see Chapter 4), than when it is used at the unit level. Furthermore, at the higher levels the possibility of discrepancies between actual and intended structure is greater; but instrumentation overhead is relatively smaller.

It is easiest to install probes when programming in languages that support **conditional assembly** or **conditional compilation.** The probes are written in the source code and tagged into categories. Both counters and traversal markers can be implemented, and one need not be parsimonious with the number and placement of probes because only those that are activated for that test will be compiled or assembled. For any test or small set of tests, only some of the probes will be active. Rarely would you compile with all probes activated and then only when all else failed.

Conditional assembly and compilation must be used with caution, especially at higher program levels. A unit may take just a few seconds to compile or assemble. But (and this is language and language processor dependent), the same routine, if compiled or assembled in the context of a full system, could take many hours—thereby canceling many of the advantages of conditional assembly and compilation.

If conditional assembly or compilation are not available, use macros or function calls for each category of probe to be implemented. The probe can be turned on or off by modifying the macro or function definition or by setting ON/OFF parameters within the functions or macros. Use of macros or functions will also reduce bugs in the probes themselves. A general–purpose routine can be written to store the outputs of traversal markers. Because efficiency is not really an issue, you can afford the overhead and can use a piece of standard code to record things.

Plan your instrumentation in levels of ever–increasing detail so that when all probes are active at the most detailed level, they will serve as a diagnostic tool. Remember that path testing based on structure should comprise at most half of all the tests that are to be done, and although the instrumentation may be installed to verify path testing, it will also be useful in other tests.

6. IMPLEMENTATION AND APPLICATION OF PATH TESTING

6.1. Integration, Coverage, and Paths in Called Components

Path–testing methods are mainly used in unit testing, especially for new software. Let's consider an idealistic process by which components are integrated. The new component is first tested as an independent unit with all called components and corequisite components replaced by **stubs**—a simulator of lower–level components that is presumably more reliable than the actual component. This is classical unit testing. Path–testing methods at this stage are used to explore potential control–flow problems without the distraction of possible bugs in called or corequisite components. We then integrate the component with its called subroutines and corequisite components, one at a time, carefully probing the interface issues. Once the interfaces have been tested, we retest the integrated component, this time with the stubs replaced by the real subroutines and corequisite component. The component is now ready for the next level of integration. This bottom–up integration process continues until the entire system has been integrated. The idea behind integrating in this manner is that it avoids the confusion between problems in lower–level components and the component under test.

I said that the above procedure was idealistic—it's good pedagogy because it clarifies the integration issues, but that's hardly ever the way integration is done. Reality is more complicated, less rigid, far more dynamic, and more efficient. While a bottom–up integration strategy may be used in part, integration proceeds in associated blocks of components, some of which have been fully tested and are fully trusted and others of which have had little prior testing. Stubs may be correctly avoided because it is recognized that the bug potential for some stubs may be higher than that of the real routine. And there are always old, well–tested routines to be integrated.

As soon as we deal with partially tested lower–level subroutines to be integrated with the component on which we are momentarily focusing our attention, we have to think about paths within the subroutine—similarly for coroutines. The same situation, albeit less obviously, is true for corequisite component whose interface with our component is via a data object. What does it mean to achieve C1 or C2 coverage at this level when lower–level or corequisite components could affect our control flow? There are several consequences. First, predicate interpretation could require us to treat the subroutine as if it were in–line code, adding its complexity and processing to the problem. Sensitization obviously becomes more difficult because there's a lot more code to think about. Third, a selected path may be unachievable because the called component's processing blocks that path. In assembly language programming, all of these interactions and more must be considered, which is one of the reasons why, function–for–function, assembly language is much harder to program. One of the main services provided by higher–order languages is that they allow us to hide lower–level complexity. Although it is always possible to circumvent complexity hiding, the extent that the designer takes advantage of it is also the extent to which higher–level testing can ignore lower–level actions and paths within called components.

Path testing, and for that matter, most structural testing methods, rely on the assumption that we can do effective testing one level at a time without being overly concerned with what happens at the lower levels. It is a fundamental weakness, but no worse than the other weaknesses of path testing we've exposed, such as predicate coverage problems and blindness.

Although there are few hard statistics, we typically lose about 15% coverage (for most coverage metrics) with each level. Thus, while we may achieve C2 at the current level, path tests will achieve 85% one level down, 70% two levels down, 60% three levels down, and so on. When all testing, by all methods, is considered, C1 coverage at the system level ranges from a low of 50% to a high of 85%. We have no statistics for C2 coverage in system testing because it is impossible to monitor C2 coverage (in current hardware) without disrupting the system's operation to the point where testing is impossible. System–level coverage is generally restricted to Cl, which can be done by tools that minimally disturb the system. There are unsupported claims of achieving 95% C1 coverage in system testing, but this result was for software that was designed with coverage and integration issues in mind.

6.2. New Code

Wholly new or substantially modified code should always be subjected to enough path testing to achieve C2 (with additional C1 monitoring if it's an unstructured language). Stubs are used where it is clear that the bug potential for the stub is significantly lower than that of the called component. That means that old, trusted components will not be replaced by stubs. Some consideration is given to paths within called components, but only to the extent that we have to do so to assure that the paths we select at the higher level is achievable. Typically, we'll try to use the shortest entry/exit path that will do the job; avoid loops; avoid lower–level subroutine calls; avoid as much lower–level complexity as possible. The unit test suite should be mechanized so that it can be repeated as integration progresses. As a mechanized suite, it will be possible to redo most of the tests with very little effort as we achieve larger and larger aggregates of integrated components. The fact that a previously selected path is no longer achievable often means that we've found a bug arising from an unsuspected interaction. The path may also be blocked because of error conditions or other exceptional situations that can't easily be tested in context. We should expect to drop some percentage of the tests because they can't be run in an integrated component: typical values are 10%–20%.

6.3. Maintenance

The maintenance situation is distinctly different. Path testing is first used on the modified component, as for new software, but called and corequisite components will invariably be real rather than simulated. If we have a configuration–controlled, automated, unit test suite, then path testing will be repeated entirely with such modifications as required to accommodate the changes. Otherwise, selected paths will be chosen in an attempt to achieve C2 over the changed code. As we learn more about the differences between maintenance testing and new code testing, new, more effective strategies will emerge that will change the current, intuition–driven, maintenance test methods into efficient methodologies that provide the kind of coverage we should achieve in maintenance.

6.4. Rehosting

Path testing with C1 + C2 coverage is a powerful tool for rehosting old software. When used in conjunction with automatic or semiautomatic structural test generators, we get a very powerful, effective, rehosting process. Software is rehosted because it is no longer cost–effective to support the environment in which it runs (hardware, language, operating system). Because it's old software, we may have no specification to speak of, much of its operation may be poorly understood, and it is probably as unstructured as software can be. Such software, though, has one great virtue—it works. Whatever it is the software does, it does so correctly, and it is as bug–free as any software can be. The objective of rehosting is to change the operating environment and not the rehosted software. If you attempt to do both simultaneously, all is lost.

Here's how it's done. First, a translator from the old to the new environment is created and tested as any piece of software would be. The bugs in the rehosting process, if any, will be in the translation algorithm and the translator, and the rehosting process is intended to catch those bugs. Second, a complete (C1 + C2) path test suite is created for the old software in the old environment. Components may be grouped to reduce total testing labor and to avoid a total buildup and reintegration, but C1 + C2 is not compromised. The suite is run on the old software in the old environment and all outcomes are recorded. *That test suite and the associated outcomes become the specification for the rehosted software.* Another translator may be needed to convert or adapt the tests and outcomes to the new environment: such translation, if needed, is kept to a minimum, even if it may result in an inefficient rehosted program. The translator is run on all units. The translated units and higher–level component aggregates are then retested using the specification test suite. Coverage is monitored. Test failures or failure to achieve coverage leads to changes in the translator(s) that could necessitate translation reruns and retesting. When the entire system has passed all of the above tests, it is subjected to system–level functional verification testing.

The cost of the process is comparable to the cost of rewriting the software from scratch; it may even be more expensive, but that's not the point. The point is that this method avoids the risks associated with rewriting and achieves a stable, correctly working, though possibly inefficient, software base in the new environment without operational or security compromises. Once confidence in the rehosted software has been established, it can then be modified to improve efficiency and/or to implement new functionality, which had been difficult in the old environment.

This process, designed by the author, was successfully used to rehost the DOD security–accredited Overseas AUTODIN software, operated by the Defense Communications Agency. Assembly language software was translated from an obsolete Philco–Ford 102 computer into VAX assembly language: this entailed substantially different hardware architectures and instruction repertoires. Rehosting from one COBOL environment to another is easy by comparison.

7. TESTABILITY TIPS

Testable software has fewer bugs, is easier to test, and is easier to debug. What has path testing taught us about testability and—more important—which design goals should we adopt to create more testable software? Here are some tips.

1. Keep in mind three numbers: the total number of paths, the total number of achievable paths, and the number of paths required to achieve C2 coverage. The closer these numbers are to each other, the more testable the routine is because:

- **a.** Few unachievable paths means less sensitizing problems and fewer sensitizing dead ends (unsolvable equations).
- **b.** Fewer total paths means reduced opportunities for blindness and coincidental correctness.
- **c.** Testing is based on samples. The smaller the achievable path set is compared to the covering path set, the greater the relative size of the sample, and therefore the greater the confidence warranted in the test set we did execute.

2. Make your decisions once, only once, and stick to them—no correlated decisions. Design goal: n decisions means 2^n paths—all achievable, all needed for C2.
3. Don’t squeeze the code.
4. If you can’t test it, don’t build it.
5. If you don’t test it, rip it out.
6. Introduce no “extras,” “freebies,” unwanted generalizations, additional functionality, hooks, or anything else that will require more cases to cover if you won’t or can’t test them.
7. If you can’t sensitize a path you need for coverage, you probably don’t know what you’re doing.
8. Easy cover beats elegance every time.
9. Covering paths make functional sense.
10. Deeply nested and/or horrible loops aren’t a mark of genius but of a murky mind.
11. Flags, switches, and instruction modification, playing around with the program status word, and other violations of sane programming are evil. “The Devil made me do it!” is no excuse.
12. Don’t squeeze the code. *Don’t squeeze the code!* **DON’T SQUEEZE THE CODE!**

8. SUMMARY

1. Path testing based on structure is a powerful unit–testing tool. With suitable interpretation, it can be used for system functional tests (see Chapter 4).
2. The objective of path testing is to execute enough tests to assure that, as a minimum, C1 + C2 have been achieved.
3. Select paths as deviations from the normal paths, starting with the simplest, most familiar, most direct paths from the entry to the exit. Add paths as needed to achieve coverage.
4. Add paths to cover extreme cases for loops and combinations of loops: no looping, once, twice, one less than the maximum, the maximum. Attempt forbidden cases.
5. Find path–sensitizing input–data sets for each selected path. If a path is unachievable, choose another path that will also achieve coverage. But first ask yourself why seemingly sensible cases lead to unachievable paths.
6. Use instrumentation and tools to verify the path and to monitor coverage.
7. Incorporate the notion of coverage (especially C2) into all reviews and inspections. Make the ability to achieve C2 a major review agenda item.
8. Design test cases and path from the design flowgraph or PDL specification but sensitize paths from the code as part of desk checking. Do covering test case designs either prior to coding or

concurrently with coding.

9. Document all tests and expected test results as copiously as you would document code. Put test suites under the same degree of configuration control used for the software it tests. Treat each path like a subroutine. Predict and document the outcome for the stated inputs and the path trace (or name by links). Also document any significant environmental factors and preconditions. Your tests must be reproducible so that they can serve a diagnostic purpose if they reveal a bug. An undocumented test cannot be reproduced. Automate test execution.

10. Be creatively stupid when conducting tests. Every deviation from the predicted outcome or path must be explained. Every deviation must lead to either a test change, a code change, or a conceptual change.

11. A test that reveals a bug has succeeded, not failed (MYER79).

<ch2 toc ch4>