Software Testing and Analysis: Process, Principles, and Techniques

Mauro Pezzè
University of Lugano
172 PUBLICATIONS **2,926** CITATIONS

Michal Young
University of Oregon
81 PUBLICATIONS **1,346** CITATIONS

Software Testing and Analysis: Process, Principles, and Techniques

Mauro Pezzè and Michal Young

Draft Version of 31st March 2000

Chapter 14

Structural Testing

The structure of the software itself is a valuable source of information for selecting test cases and determining whether a set of test cases has been sufficiently thorough. We can ask whether a test suite has "covered" a control flow graph or other model of the program.[1] It is simplest to consider structural coverage criteria as addressing the test adequacy question: "Have we tested enough." In practice we will be interested not so much in asking whether we are done, but in asking what unmet obligations with respect to the adequacy criteria suggest about additional test cases that may be needed, i.e., we will often treat the adequacy criterion as a heuristic for test case selection or generation. For example, if one statement remains unexecuted despite execution of all the test cases in a test suite, we may devise additional test cases that exercise that statement. Structural information should not be used as the primary answer to the question, "How shall I choose tests," but it is useful in combination with other test selection criteria (particularly specification-based testing) to help answer the question "What additional test cases are needed to reveal faults that may not become apparent through black-box testing alone."

Required Background

- Chapter 6

  The material on control flow graphs and related models of program structure is required to understand this chapter.

- Chapter 11

  The introduction to test case adequacy and test case selection in general sets the context for this chapter. It is not strictly required for un-

[1]In this chapter we use the term "program" generically for the artifact under test, whether that artifact is a complete application or an individual unit together with a test harness. This is consistent with usage in the testing research literature.

derstanding this chapter, but is helpful for understanding how the techniques described in this chapter should be applied.

14.1 Overview

Testing can reveal a fault only when the execution of the corresponding faulty element causes a failure. For example, if there were a fault in the statement at line **??** of the program in Figure 14.1, it could be revealed only with test cases in which the input string contains the character % followed by two hexadecimal digits, since only these cases would cause this statement to be executed. Based on this simple observation, a program has not been adequately tested if some of its elements have not been executed.[2] Control flow testing criteria are defined for particular classes of elements by requiring the execution of all such elements of the program. Control flow elements include statements, branches, conditions, and paths.

Unfortunately, a set of correct program executions in which all control flow elements are exercised does not guarantee the absence of faults. Execution of a faulty statement may not always result in a failure. The state may not be corrupted when the statement is executed with some data values, and a corrupt state may not propagate through execution to eventually lead to a failure. Let us assume for example to have erroneously typed 6 instead of 16 in the statement at line **??** of the program in Figure 14.1. Test cases that execute the faulty statement with value 0 for variable digit_high would not corrupt the state, thus leaving the fault unrevealed despite having executed the faulty statement.

The statement at line 31 of the program in Figure 14.1 contains a fault, since variable eptr used to index the input string is incremented twice without checking the size of the string. If the input string contains a character % in one of the last two positions, eptr* will point beyond the end of the string when it is later used to index the string. Execution of the program with a test case where string encoded terminates with character % followed by at most one character causes the faulty statement to be executed. However, due to the memory management of C programs, execution of this faulty statement may not cause a memory failure, since the program will read the next character available in memory ignoring the end of the string. Thus, this fault may remain hidden during testing despite having produced an incorrect intermediate state. Such a fault could be revealed using memory checking software tool that identifies memory violations[3].

Control flow testing complements specification-based testing by including cases that may not be identified from specifications alone. A typical case is implementation of a single item of the specifications with multiple parts of

[2]This is an over-simplification, since some of the elements may not be executed by any possible input. The issue of infeasible elements is discussed in Section 14.9

[3]At the time of writing commercial tools providing such capability include Purify and BoundsChecker

```c
 /* External file hex_values.h defines Hex_Values[128]
 * with value 0 to 15 for the legal hex digits (case-insensitive)
 * and value -1 for each illegal digit including special characters
 */

#include "hex_values.h"
/**
 *  @title cgi_decode
 *  @desc
 *      Translate a string from the CGI encoding to plain ascii text
 *       '+' becomes space, %xx becomes byte with hex value xx,
 *       other alphanumeric characters map to themselves
 *
 *      returns 0 for success, positive for erroneous input
 *              1 = bad hexadecimal digit
 */
int cgi_decode(char *encoded, char *decoded) {
  char *eptr = encoded;
  char *dptr = decoded;
  int ok=0;
  while (*eptr) {
    char c;
    c = *eptr;
    /* Case 1:  '+' maps to blank */
    if (c == '+') {
      *dptr = ' ';
    } else if (c == '%') {
      /* Case 2:  '%xx' is hex for character xx */

      int  digit_high = Hex_Values[*(++eptr)];
      int  digit_low  = Hex_Values[*(++eptr)];
      if (   digit_high == -1 || digit_low == -1 ) {
        /* *dptr='?'; */
        ok=1; /* Bad return code */
      } else {
        *dptr = 16* digit_high + digit_low;
      }

      /* Case 3:  All other characters map to themselves */
    } else {
      *dptr = *eptr;
    }
    ++dptr;
    ++eptr;
  }
  *dptr = '\0';            /* Null terminator for string */
  return ok;
}
```

Figure 14.1: The C function cgi_decode, which translates a cgi-encoded string to a plain ASCII string (reversing the encoding applied by the common gateway interface of most web servers).

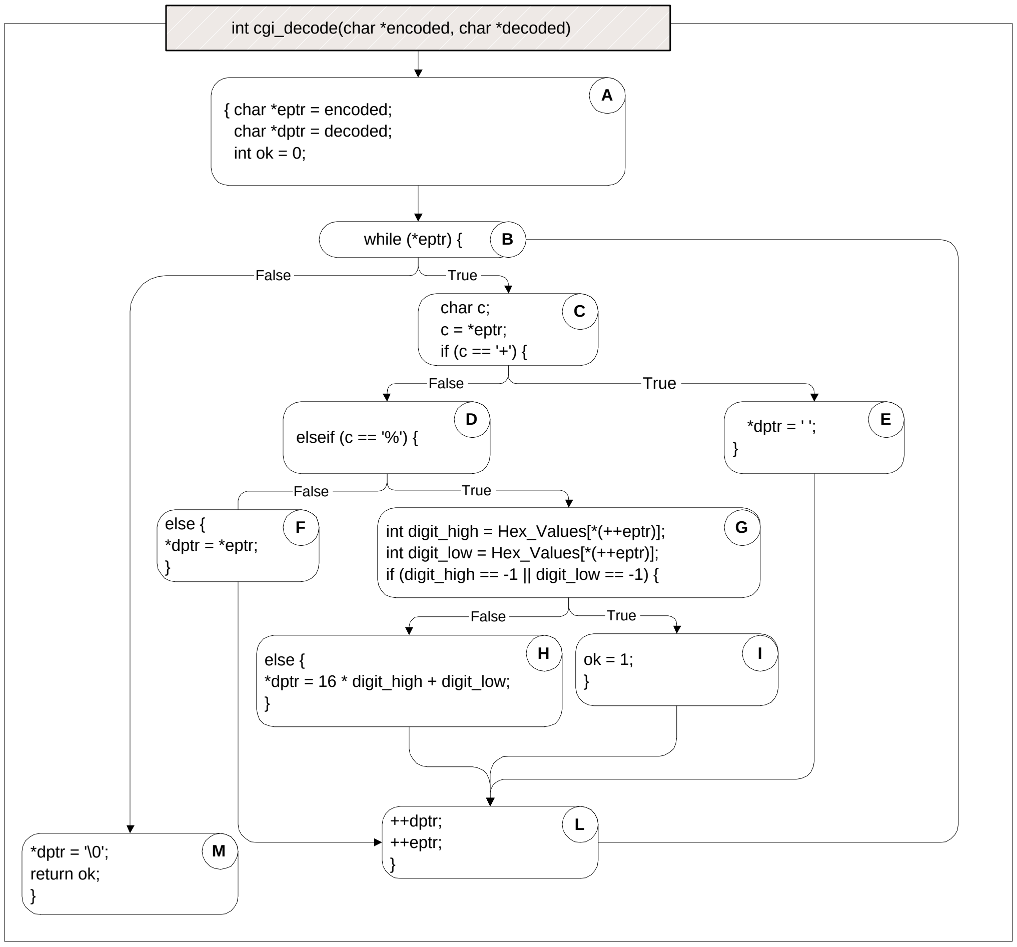

Figure 14.2: The control flow graph of function cgi_decode from Figure 14.1

T_0	=	{ “ ”, “test”, “test+case%1Dadequacy” }
T_1	=	{ “adequate+test%0Dexecution%7U” }
T_2	=	{ “%3D”, “%A”, “a+b”, “test” }
T_3	=	{ “ ”, “+%0D+%4J” }
T_4	=	{ “first+test%9Ktest%K9” }

Table 14.1: Sample test suites for C function cgi_decode from Figure 14.1

the program. For example, a good specification of a table would leave data structure implementation decisions to the programmer. If the programmer chooses a hash table implementation, then different portions of the insertion code will be executed depending on whether there is a hash collision. Selection of test cases from the specification would not ensure that both the collision case and the non-collision case are tested. Even the simplest control flow testing criterion would require that both of these cases are tested.

On the other hand, test suites satisfying control flow adequacy criteria could fail in revealing faults that can be caught with specification-based criteria. The most notable example is the class of so called *missing path* faults. Such faults result from the missing implementation of some items of the specifications. For example, the program in Figure 14.1 transforms all hexadecimal ASCII codes to the corresponding characters. Thus, it is not a correct implementation of a specification that requires control characters to be identified and skipped. A test suite designed only to adequately cover the control structure of the program will not explicitly include test cases to test for such faults, since no elements of the structure of the program correspond to this feature of the specifications.

In practice, control flow testing criteria are used to evaluate the thoroughness of test suites derived from specification-based testing criteria, by identifying elements of the programs not adequately exercised. Unexecuted elements may be due to natural differences between specification and implementation, or they may reveal flaws of the software or its development process: inadequacy of the specifications that do not include cases present in the implementation; coding practice that radically diverges from the specification; or inadequate specification-based test suites.

Control flow adequacy can be easily measured with automatic tools[4]. The degree of control flow coverage achieved during testing is often used as an indicator of progress and can be used as a criterion of completion of the testing activity[5].

[4]At the time of writing commercial tools providing such capability include Cantata, Testbed, PureCoverge,...

[5]Application of test adequacy criteria within the testing process is discussed in Chapter 11.

14.2 Statement Testing

The first set of control flow elements to be exercised are statements, i.e., nodes of the control flow graph. The statement coverage criterion requires each statement to be executed at least once, reflecting the idea that a fault in a statement cannot be revealed without executing the faulty statement.

Δ Statement Adequacy Criterion

Let T be a test suite for a program P. T satisfies the statement adequacy criterion for P, iff, for each statement S of P, there exists at least one test case in T that causes the execution of S.

This is equivalent to stating that every node in the control flow graph model of program P is visited by some execution path exercised by a test case in T.

Δ Statement Coverage

The statement coverage $C_{Statement}$ of T for P is the fraction of statements of program P executed by at least one test case in T.

$$C_{Statement} = \frac{\text{number of executed statements}}{\text{number of statements}}$$

T satisfies the statement adequacy criterion if $C_{Statement} = 1$. The ratio of visited control flow graph nodes to total nodes may be different from the ratio of executed statements to all statements, depending on the granularity of the control flow graph representation. Nodes in a control flow graph often represent *basic blocks* rather than individual statements, and so some standards (notably *DOD-178B*) refer to basic block coverage, thus indicating node coverage for a particular granularity of control flow graph. For the standard control flow graph models discussed in Chapter 6, the relation between coverage of statements and coverage of nodes is monotonic, i.e., if the statement coverage achieved by test suite T_1 is greater than the statement coverage achieved by test suite T_2, then the node coverage is also greater. In the limit, statement coverage is 1 exactly when node coverage is 1.

Δ Basic Block Coverage

Let us consider for example the program of Figure 14.1. The program contains 18 statements. A test suite T_0

$$T_0 = \{\text{`` ''}, \text{``test''}, \text{``test+case\%1Dadequacy''}\}$$

does not satisfy the statement adequacy criterion, because it does not cause the execution of statement ok = 1 at line 34. The test suite T_0 results in statement coverage of .94 (17/18), or node coverage of .91 (10/11) relative to the control flow graph of Figure 14.2. On the other hand, a test suite with only test case

$$T_1 = \{\text{``adequate+test\%0Dexecution\%7U''}\}$$

causes all statements to be executed, and thus satisfies the statement adequacy criterion, reaching a coverage of 1. A test suite

$$T_2 = \{\text{``\%3D''}, \text{``\%A''}, \text{``a+b''}, \text{``test''}\}$$

also causes all statements to be executed and thus satisfies the statement adequacy criterion. Coverage is not monotone with respect to the size of the test suites, i.e., test suites that contain fewer test cases may achieve a higher coverage than test suites that contain more test cases. In the former example, T_1 contains only one test case, while T_0 contains three test cases, but T_1 achieves a higher coverage than T_0. (Test suites used in this chapter are summarized in Table 14.1.)

Criteria can be satisfied by many test suites of different sizes. In the former example both T_1 and T_2 satisfy the statement adequacy criterion for program cgi_decode although one consists of a single test case and the other consists of 4 test cases. Notice that while we typically wish to limit the size of test suites, in some cases we may prefer a larger test suite over a smaller suite that achieves the same coverage. A test suite with fewer test cases may be more difficult to generate or may be less helpful in debugging. Let us consider, for example, to have missed the 1 in the statement at line **??** of the program in Figure 14.1. Both test suites would reveal the fault, resulting in a failure, but T_2 would provide better information for localizing the fault, since the program fails only for test case "%1D", the only test case of T_2 that exercises the statement at line **??**.

On the other hand, a test suite obtained by adding test cases to T_2 would satisfy the statement adequacy criterion, but would not have any particular advantage over T_2 with respect to the total effort required to reveal and localize faults. Designing complex test cases that exercise many different elements of a unit is seldom a good way to optimize a test suite, although it may occasionally be justifiable when there is large and unavoidable fixed cost (e.g., setting up equipment) for each test case regardless of complexity.

Control flow coverage may be measured incrementally while executing a test suite. In this case, the contribution of a single test case to the overall coverage that has been achieved depends on the order of execution of test cases. For example, in test suite T_2 introduced above, execution of test case "%1D" exercises 16 of the 18 statements of the program cgi_decode, but it exercises only 1 new statement if executed after "%A". The increment of coverage due to the execution of a specific test case does not measure the absolute efficacy of the test case. Measures independent from the order of execution may be obtained by identifying *independent statements*. However, in practice we are only interested in the coverage of the test suite and in the statement not exercised by the test suite, not in the coverage of test cases.

14.3 Branch Testing

A test suite can achieve complete statement coverage without executing all the possible branches in a program. Consider, for example, a faulty program cgi_decode′ obtained from program cgi_decode by removing line 41. The control flow graph of program cgi_decode′ is shown in Figure 14.3. In the new program there are no statements following the false branch exiting node D.

Thus, a test suite that tests only translation of specially treated characters but not treatment of strings containing other characters that are copied without change satisfies the statement adequacy criterion, but would not reveal the missing code in program cgi_decode′. For example, a test suite T_3

$$T_3 = \{\text{“ ”}, \text{“+\%0D+\%4J”}\}$$

satisfies the statement adequacy criterion for program cgi_decode′ but does not exercise the false branch from node D in the control flow graph model of the program.

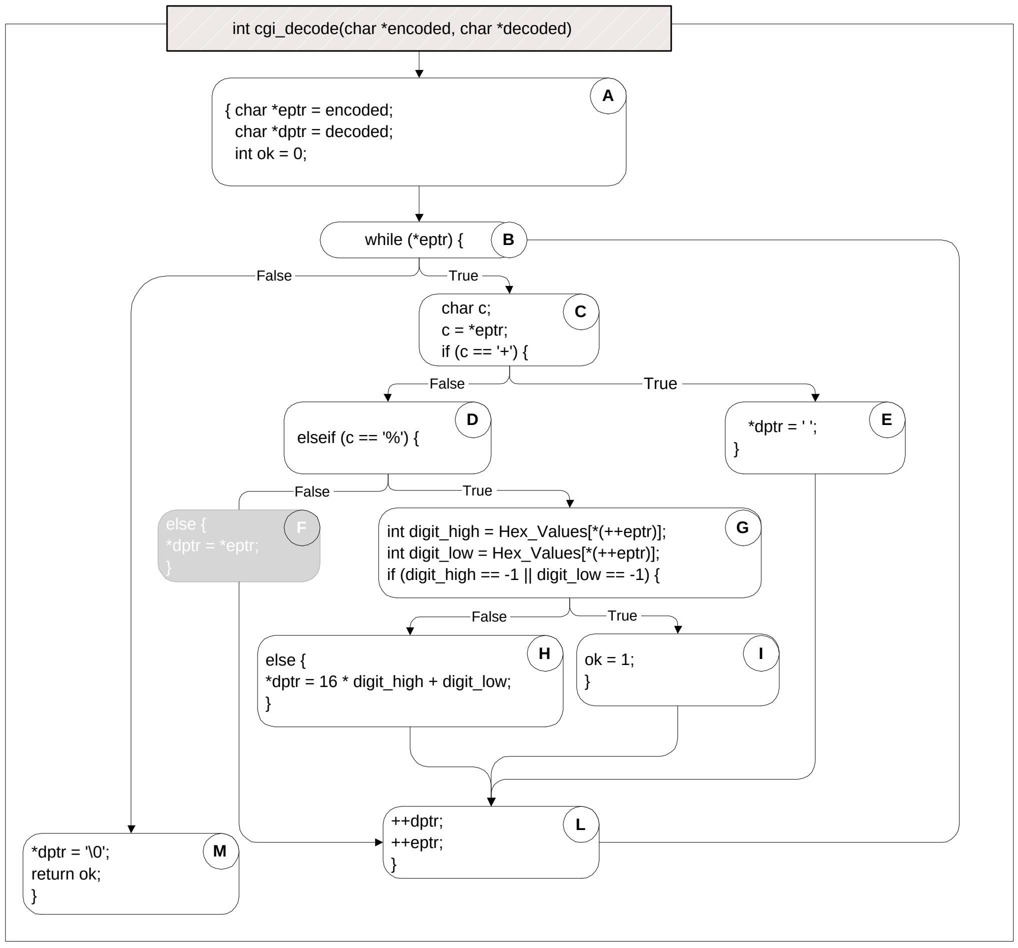

Figure 14.3: The control flow graph of function cgi_decode′ which is obtained from the program of Figure 14.1 after removing node F.

The branch adequacy criterion requires each branch of the program to be executed by at least one test case.

Δ Branch Adequacy Criterion

Let T be a test suite for a program P. T satisfies the branch adequacy criterion for P, iff, for each branch B of P, there exists at least one test case in T that causes execution of B.

This is equivalent to stating that every edge in the control flow graph model of program P belongs to some execution path exercised by a test case in T.

Δ Branch Coverage

The branch coverage C_{Branch} of T for P is the fraction of branches of program P executed by at least one test case in T.

$$C_{Branch} = \frac{\text{number of executed branches}}{\text{number of branches}}$$

T satisfies the branch adequacy criterion if $C_{Branch} = 1$.

Test suite T_3 achieves branch coverage of .88 since it executes 7 of the 8 branches of program cgi_decode$'$. Test suite T_2 satisfies the branch adequacy criterion, and would reveal the fault. Intuitively, since traversing all edges of a graph causes all nodes to be visited, test suites that satisfy the branch adequacy criterion for a program P satisfy also the statement adequacy criterion for the same program.[6] The contrary is not true, as illustrated by test suite T_3 for the program cgi_decode$'$ presented above.

14.4 Condition Testing

Branch coverage is useful for exercising faults in the way a computation has been decomposed into cases. Condition coverage considers this decomposition in more detail, forcing exploration not only of both possible results of a boolean expression controlling a branch, but also of different combinations of the individual conditions in a compound boolean expression.

Assume, for example, that we have forgotten the first operator '$-$' in the conditional statement at line 32 resulting in the faulty expression

```
(digit_high == 1 || digit_low == -1).
```

As trivial as this fault seems, it can easily be overlooked if only the outcomes of complete boolean expressions are explored. The branch adequacy criterion can be satisfied, and both branches exercised, with test suites in which the first comparison evaluates always to **False** and only the second is varied. Such tests do not systematically exercise the first comparison, and will not reveal the fault in that comparison. Condition adequacy criteria overcome this problem by requiring different elementary conditions of the decisions to be separately exercised.

[6]We can consider entry and exit from the control flow graph as branches, so that branch adequacy will imply statement adequacy even for units with no other control flow.

The simplest condition adequacy criterion, called basic (or elementary) condition coverage requires each elementary condition to be covered, i.e., each elementary condition shall have a **True** and a **False** outcome at least once during the execution of the test set. The basic conditions, sometimes also called elementary conditions, are comparisons, references to boolean variables, and other boolean-valued expressions whose component sub-expressions are not boolean values.

Δ Basic Condition Adequacy Criterion

Let T be a test suite for a program P. T covers all basic conditions of P, i.e., it satisfies the basic condition adequacy criterion for P, iff each basic condition has a *true* outcome for for at least one test case in T and a *false* outcome for at least one test case in T.

Δ $C_{\text{Basic\_Condition}}$

The basic condition coverage $C_{Basic\_Condition}$ of T for P is the fraction of the total number of truth values assumed by the basic conditions of program P during the execution of all test cases in T.

$$C_{\text{Basic\_Condition}} = \frac{\text{total number of truth values assumed by all basic conditions}}{\text{total number of truth values of all basic conditions}}$$

T satisfies the basic condition adequacy criterion if $C_{Basic\_Conditions} = 1$. Notice that the total number of truth values of all basic conditions is twice the number of basic conditions, since each basic condition can assume value *true* or *false*. For example, the program in Figure 14.1 contains five basic conditions, which may take ten possible truth values. Three basic conditions correspond to the simple decisions at lines **??**, **??**, and 32, i.e., decisions that contain only one basic condition. Thus they are covered by any test suite that covers all branches. The last two conditions correspond to the compound decision at line 32. In this case, test suites T_1 and T_3 cover the decisions without covering the basic conditions. Test suite T_1 covers the decision since it has an outcome **True** for the substring $\%0D$ and an outcome **False** for the substring $\%7U$ of test case "adequate+test%0Dexecution%7U". However test suite T_1 does not cover the first condition, since it has only outcome **True**. To satisfy the basic condition adequacy criterion, we need to add an additional test that produces outcome *false* for the first condition, e.g., test case "basic%K7".

The basic condition adequacy criterion can be satisfied without satisfying branch coverage. For example, the test suite

$$T_4 = \{\text{“first+test\%9Ktest\%K9”}\}$$

check the example and decide if to ask why as a (solved) exercise or say it here.

satisfies the basic condition adequacy criterion, but not the branch condition adequacy criterion, since the outcome of the decision is always **False**. Thus branch and basic condition adequacy criteria are not directly comparable, i.e., they address different kinds of faults.

An obvious extension that includes both the basic condition and the branch adequacy criteria is called *branch and condition adequacy criterion*, with the obvious definition: A test suite satisfies the branch and condition adequacy

Δ Branch and Condition Adequacy

criterion if it satisfies both the branch adequacy criterion and the condition adequacy criterion.

A more complete extension that includes both the basic condition and the branch adequacy criteria is the *compound condition adequacy criterion,*[7] which requires a test for each possible combination of basic conditions. It is most natural to visualize compound condition adequacy as covering a truth table, with one column for each basic condition and one row for each combination of truth values that might be encountered in evaluating the compound condition. To satisfy the compound condition adequacy criterion, a test suite must encounter the following evaluations of the compound condition `(digit_high == -1 || digit_low == -1)` at line 32:

Δ Compound Condition Adequacy

Test Case	`digit_high == -1`	`digit_low == -1`
(1)	**True**	–
(2)	**False**	**True**
(3)	**False**	**False**

Notice that due to the left-to-right evaluation order and short-circuit evaluation of logical *OR* expressions in the C language, the value **True** for the first condition does not need to be combined with both values **False** and **True** for the second condition. The number of test cases required for compound condition adequacy can, in principle, grow exponentially with the number of basic conditions in a decision (all 2^N combinations of N basic conditions), which would make compound condition coverage impractical for programs with very complex conditions. Short circuit evaluation is often effective in reducing this to a more manageable number, but not in every case. The number of test cases required to achieve compound condition coverage even for expressions built from N basic conditions combined only with short-circuit boolean operators like the `&&` and `||` of C and Java can still be exponential in the worst case.

Consider the number of cases required for compound condition coverage of the following two boolean expressions, each with five basic conditions. For the expression `a && b && c && d && e`, compound condition coverage requires:

Test Case	a	b	c	d	e
(1)	**True**	**True**	**True**	**True**	**True**
(2)	**True**	**True**	**True**	**True**	**False**
(3)	**True**	**True**	**True**	**False**	–
(4)	**True**	**True**	**False**	–	–
(5)	**True**	**False**	–	–	–
(6)	**False**	–	–	–	–

For the expression `(((a || b) && c) || d) && e`, however, compound condition adequacy requires many more combinations:

[7]Compound condition adequacy is also know as multiple condition coverage

Test Case	a	b	c	d	e
(1)	**True**	–	**True**	–	**True**
(2)	**False**	**True**	**True**	–	**True**
(3)	**True**	–	**False**	**True**	**True**
(4)	**False**	**True**	**False**	**True**	**True**
(5)	**False**	**False**	–	**True**	**True**
(6)	**True**	–	**True**	–	**False**
(7)	**False**	**True**	**True**	–	**False**
(8)	**True**	–	**False**	**True**	**False**
(9)	**False**	**True**	**False**	**True**	**False**
(10)	**False**	**False**	–	**True**	**False**
(11)	**True**	–	**False**	**False**	–
(12)	**False**	**True**	**False**	**False**	–
(13)	**False**	**False**	–	**False**	–

An alternative approach that can be satisfied with the same number of test cases for boolean expressions of a given length regardless of short-circuit evaluation is the *modified condition adequacy criterion*, also known as *modified condition / decision coverage* or MCDC.

Δ Modified condition adequacy (MCDC)

The modified condition adequacy criterion requires that each basic condition be shown to independently affect the outcome of each decision. That is, for each basic condition C, there are two test cases in which the truth values of all conditions except C are the same, and the compound condition as a whole evaluates to **True** for one of those test cases and **False** for the other. The modified condition adequacy criterion represents a tradeoff between number of required test cases and thoroughness of the test, and is required by important quality standards in aviation, including RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," and its European equivalent EUROCAE ED-12B.

Recall the expression `(((a || b) && c) || d) && e`, which required 13 different combinations of condition values for compound condition adequacy. For modified condition adequacy, only 6 combinations are required. (Here they have been numbered for easy comparison with the previous table.)

	a	b	c	d	e	Decision
(1)	**True**	–	**True**	–	**True**	**True**
(2)	**False**	**True**	**True**	–	**True**	**True**
(3)	**True**	–	**False**	**True**	**True**	**True**
(6)	**True**	–	**True**	–	**False**	**False**
(11)	**True**	–	**False**	**False**	–	**False**
(13)	**False**	**False**	–	**False**	–	**False**

In the former table, each variable independently affects the true and false outcome of the expression at least once. The following table indicates which rows of the former table correspond to the independent effect for each variable:

	Outcome	
Variable	True	False
a	1	13
b	2	13
c	1	11
d	3	11
e	1	6

Note that the same test case can occur more than once in the table. Case 1 appears here three times in combination with other cases, showing the effect of conditions a, c, and e. Note also that this is not the only possible set of test cases to satisfy the criterion; a different selection of boolean combinations could be equally effective.

As an example, let us consider the following `if` statement, that appears in the Java source code of Grappa,[8] a graph layout engine distributed by AT&T Laboratories:

```
if(pos < parseArray.length
   &&  (parseArray[pos] == '{'
        || parseArray[pos] == '}'
        || parseArray[pos] == '|')) {
    continue;
  }
```

For brevity we will abbreviate each of the basic conditions as follows:

Room for `pos < parseArray.length`

Open for `parseArray[pos] == '{'`

Close for `parseArray[pos] == '}'`

Bar for `parseArray[pos] == '|')`

In this case again, the requirements for compound condition coverage and modified condition adequacy coverage are the same. We must have the following assignments of truth values:

Test Case	Room	Open	Close	Bar	Outcome
(1)	**False**	–	–	–	**False**
(2)	**True**	**True**	–	–	**True**
(3)	**True**	**False**	**True**	–	**True**
(4)	**True**	**False**	**False**	**True**	**True**
(5)	**True**	**False**	**False**	**False**	**False**

[8]The statement appears in file Table.java. This source code is copyright 1996, 1997, 1998 by AT&T Corporation. Grappa is distributed as "open source" software, available at the time of this writing from `http://www.research.att.com/sw/tools/graphviz/`. The formatting of the line has been altered for readability in this printed form.

The basic condition Room independently affects the outcome of the decision for test cases (1) and (2), i.e., in these two cases the outcome of the full decision changes if only the value of Room is changed.

Note that, for the purpose of determining modified condition adequacy coverage, we are permitted to assume arbitrary assignments of truth values to the "don't care" or "not evaluated" conditions, *except* for the basic condition whose effect we are showing. For example, we may consider case (1) as including the assignment $\langle \textbf{False}, \textbf{True}, \textbf{True}, \textbf{True} \rangle$ and (2) as including $\langle \textbf{True}, \textbf{True}, \textbf{True}, \textbf{True} \rangle$, thereby differing only in the assignment to the Room condition and in the outcome of the evaluation. Consider what would happen if we did not treat the "don't care" conditions in this manner: It is nonsense to require an actual test case with assignments $\langle \textbf{False}, \textbf{True}, \textbf{True}, \textbf{True} \rangle$, because the value of `parseArray[pos]` is undefined when

```
pos >= parseArray.length.
```

Test cases (2) and (3) establish that condition Open has had an independent affect on the decision, and likewise cases (3) and (4) show the effect of Close and cases (4) and (5) show the effect of Bar.

The modified condition adequacy criterion can be satisfied with $N+1$ test cases.

Q14.1. Prove that the number of test cases required to satisfy the modified condition adequacy criterion for a predicate with N basic conditions is $N+1$.

14.5 Path Testing

Decision and condition adequacy criteria force consideration of individual program decisions. Sometimes, though, a fault is revealed only through exercise of some sequence of decisions, i.e., a particular path through the program. It is simple (but impractical, as we will see) to define a coverage criterion based on complete paths rather than individual program decisions

Δ Path Adequacy Criterion

Let T be a test suite for a program P. T satisfies the path adequacy criterion for P, iff, for each path p of P, there exists at least one test case in T that causes the execution of p.

This is equivalent to stating that every path in the control flow graph model of program P is exercised by a test case in T.

Δ Path Coverage

The path coverage C_{Path} of T for P is the fraction of paths of program P executed by at least one test case in T.

$$C_{Path} = \frac{\text{number of executed paths}}{\text{number of paths}}$$

Unfortunately, the number of paths in a program with loops is unbounded, so this criterion cannot be satisfied for any but the most trivial programs. For

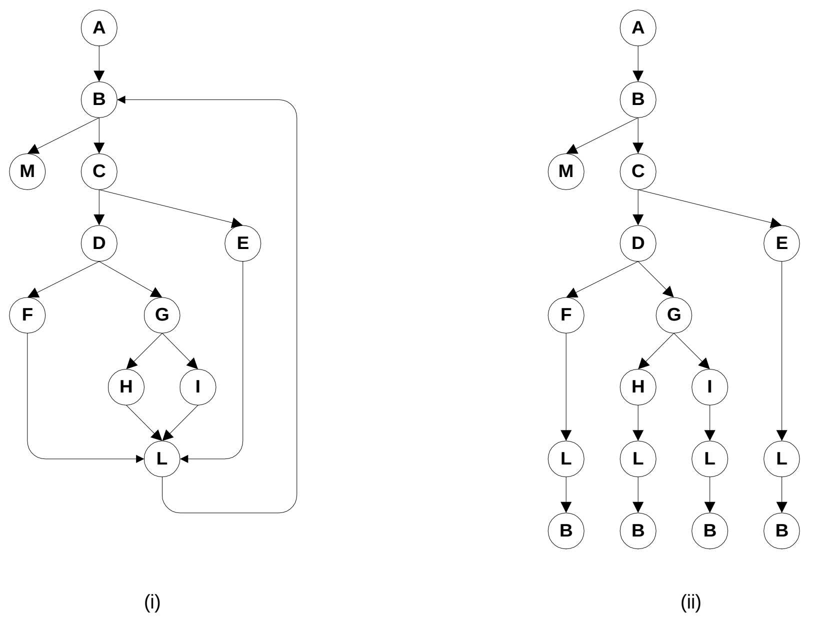

Figure 14.4: Deriving a tree from a control flow graph to derive sub-paths for boundary/interior testing. Part (i) is the control flow graph of the C function cgi_decode, identical to Figure 14.1 but showing only node identifiers without source code. Part (ii) is a tree derived from part (i) by following each path in the control flow graph up to the first repeated node. The set of paths from the root of the tree to each leaf is the required set of sub-paths for boundary/interior coverage.

program with loops, the denominator in the computation of the path coverage becomes infinite, and thus path coverage is zero no matter how many test cases are executed.

To obtain a practical criterion, it is necessary to partition the infinite set of paths into a finite number of classes, and require only that representatives from each class be explored. Useful criteria can be obtained by limiting the number of paths to be covered. Relevant subsets of paths to be covered can be identified by limiting the number of traversals of loops, the length of the paths to be traversed, or the dependencies among selected paths.

The boundary interior criterion groups together paths that differ only in the sub-path they follow when repeating the body of a loop.

Δ Boundary Interior Criterion

Figure 14.4 illustrates how the classes of sub-paths distinguished by the boundary interior coverage criterion can be represented as paths in a tree derived by "unfolding" the control flow graph of function *cgi_decode*.

Figure 14.5 illustrates a fault that may not be uncovered using statement or decision testing, but will assuredly be detected if the boundary interior path

```
 typedef struct cell {
  itemtype itemval;
  struct cell *link;
} *list;


#define NIL ((struct cell *) 0)

itemtype search( list *l, keytype k)
{
  struct cell *p = *l;
  struct cell *back = NIL;

  /* Case 1: List is empty */
  if (p == NIL) {
    return NULLVALUE;
  }

  /* Case 2: Key is at front of list,
   * No rearrangement is necessary
   */
  if (k == p->itemval) {
    return p->itemval;
  }

  /* Remaining cases are simpler because we can omit
   * special cases for beginning of list.
   * Simple sequential search, keeping a pointer one back
   * to enable relinking.
   */
  /* OOPS, we should have initialized  back=p; */
  p=p->link;
  while (1) {
    if (p == NIL) {
      return NULLVALUE;
    }
    if (k==p->itemval) {
      /* Move to front */
      back->link = p->link;
      p->link = *l;
      *l = p;
      return p->itemval;
    }
    back=p; p=p->link;
  }
}

```

Figure 14.5: A C function for searching and dynamically rearranging a linked list, excerpted from a symbol table package. Initialization of the `back` pointer is missing, causing a failure only if the search key is found in the second position in the list.

criterion is satisfied (the program fails if the loop body is executed exactly once, i.e., if the search key occurs in the second position in the list).

Although the boundary/interior coverage criterion bounds the number of paths that must be explored, that number can grow quickly enough to be impractical. The number of sub-paths that must be covered can grow exponentially in the number of statements and control flow graph nodes, even without any loops at all. Consider for example the following pseudocode:

```
if (a) {
   S1;
}
if (b) {
   S2;
}
if (c) {
   S3;
}
   ...
if (x) {
   Sn;
}
```

The sub-paths through this control flow can include or exclude each of the statements S_i, so that in total N branches result in 2^N paths that must be traversed. Moreover, choosing test data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent.[9]

Since coverage of non-looping paths is expensive, we can consider a variant of the boundary/interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths.

Δ Loop Boundary Adequacy Criterion

Let T be a test suite for a program P. T satisfies the loop boundary adequacy criterion for P iff, for each loop l in P,

- In at least one execution, control reaches the loop and then the loop control condition evaluates to **False** the first time it is evaluated.[10]
- In at least one execution, control reaches the loop and then the body of the loop is executed exactly once before control leaves the loop.
- In at least one execution, the body of the loop is repeated more than once.

One can define several small variations on the loop boundary criterion. For example, we might excuse from consideration loops that are always executed a definite number of times (e.g., multiplication of fixed-size transformation matrices in a graphics application). In practice the last part of the

[9]Section 14.9 below discusses infeasible paths.

[10]For a *while* or *for* loop, this is equivalent to saying that the loop body is executed zero times.

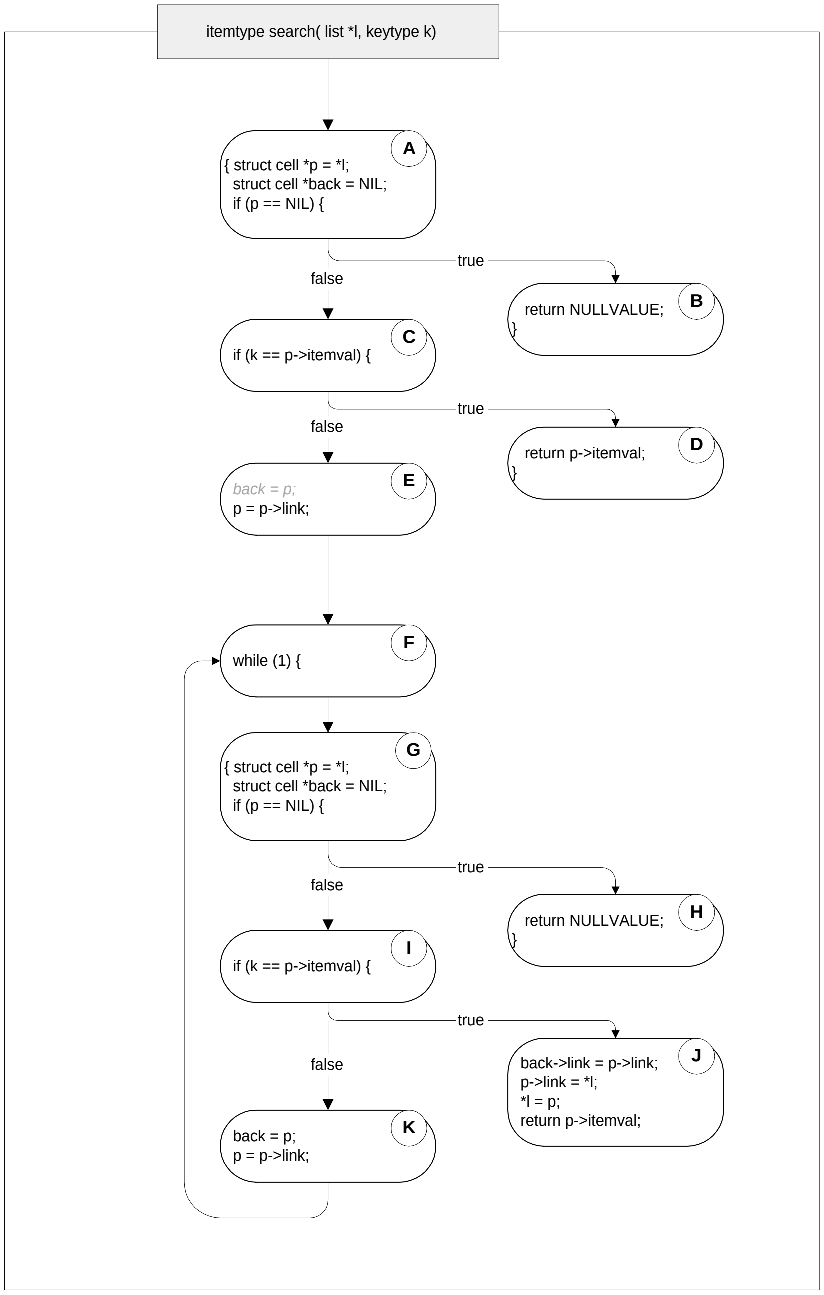

Figure 14.6: The control flow graph of C function search with move-to-front feature.

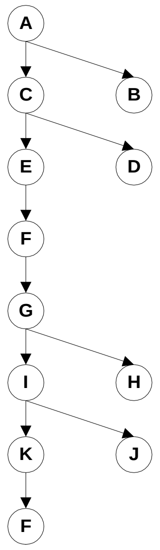

Figure 14.7: The boundary/interior sub-paths for C function search.

criterion should be "many times through the loop" or "as many times as possible," but it is hard to make that precise (how many is "many?").

It is easy enough to define such a coverage criterion for loops, but how can we justify it? Why should we believe that these three cases — zero times through, once through, and several times through — will be more effective in revealing faults than, say, requiring an even and an odd number of iterations? The intuition is that the loop boundary coverage criteria reflect a deeper structure in the design of a program. This can be seen by their relation to the reasoning we would apply if we were trying to formally verify the correctness of the loop. The basis case of the proof would show that the loop is executed zero times only when its postcondition (what should be true immediately following the loop) is already true. We would also show that an invariant condition is established on entry to the loop, that each iteration of the loop maintains this invariant condition, and that the invariant together with the negation of the loop test (i.e., the condition on exit) implies the postcondition. The loop boundary criterion does not require us to explicitly state the precondition, invariant, and postcondition, but it forces us to exercise essentially the same cases that we would analyze in a proof.

There are additional path-oriented coverage criteria that do not explicitly consider loops. Among these are criteria that consider paths up to a fixed length. The most common such criteria are based on *Linear Code Sequence and Jump (LCSAJ)*. An LCSAJ is defined as a body of code through which the flow of control may proceed sequentially, terminated by a jump in the control flow. Coverage of LCSAJ sequences of length 1 is almost, but not quite,

Δ Linear Code Sequence and Jump (LCSAJ) equivalent to branch coverage. Stronger criteria can be defined by requiring N consecutive LCSAJs to be covered. The resulting criteria are also referred to as TER_{N+2}, where N is the number of consecutive LCSAJs to be covered. Conventionally, TER_1 and TER_2 refer to statement and branch coverage, respectively.

Q14.2. We have stated that coverage of individual LCSAJs is almost, but not quite, equivalent to branch coverage. How can they differ? Devise a small example of code which requires more test cases to satisfy TER_3 (individual LCSAJs) than TER_2 (branch coverage).

The number of paths to be exercised can also be limited by identifying a subset that can be combined (in a manner to be described shortly) to form all the others. Such a set of paths is called a "basis set," and from graph theory we know that every connected graph with n nodes, e edges, and c connected components has a basis set of only $e - n + c$ independent sub-paths. Producing a single connected component from a program flow graph by adding a "virtual edge" from the exit to the entry, the formula becomes $e - n + 2$ which is called the cyclomatic complexity of the control flow graph. Cyclomatic testing consists of attempting to exercise any set of execution paths that is a basis set for the control flow graph.

Δ Cyclomatic testing

To be more precise, the sense in which a basis set of paths can be combined to form other paths is to consider each path as a vector of counts indicating how many times each edge in the control flow graph was traversed, e.g., the third element of the vector might be the number of times a particular branch is taken. The basis set is combined by adding or subtracting these vectors (and not, as one might intuitively expect, by concatenating paths). Consider again the pseudocode

```
if (a) {
   S1;
}
if (b) {
   S2;
}
if (c) {
   S3;
}
   ...
if (x) {
   Sn;
}
```

While the number of distinct paths through this code is exponential in the number of `if` statements, the number of basis paths is small: Only $n + 1$ if there are n `if` statements. We can represent one basis set (of many possible) for a sequence of four such `if` statements by indicating whether each predicate evaluates to **True** or **False**:

1	**False**	**False**	**False**	**False**
2	**True**	**False**	**False**	**False**
3	**False**	**True**	**False**	**False**
4	**False**	**False**	**True**	**False**
5	**False**	**False**	**False**	**True**

The path represented as $\langle \textbf{True}, \textbf{False}, \textbf{True}, \textbf{False} \rangle$ is formed from these by adding paths 2 and 4 and then subtracting path 1.

Cyclomatic testing does not require that any particular basis set is covered. Rather, it counts the number of independent paths that have actually been covered (i.e., counting a new execution path as progress toward the coverage goal only if it is independent of all the paths previously exercised), and the coverage criterion is satisfied when this count reaches the cyclomatic complexity of the code under test.

14.6 Procedure Call Testing

The criteria considered to this point measure coverage of control flow within individual procedures. They are not well suited to integration testing or system testing. It is difficult to steer fine-grained control flow decisions of a unit when it is one small part of a larger system, and the cost of achieving fine-grained coverage for a system or major component is seldom justifiable. Usually it is more appropriate to choose a coverage granularity commensurate with the granularity of testing. Moreover, if unit testing has been effective, then faults that remain to be found in integration testing will be primarily interface faults, and testing effort should focus on interfaces between units rather than their internal details.

In some programming languages (FORTRAN, for example), a single procedure may have multiple entry points, and one would want to test invocation through each of the entry points. More common are procedures with multiple exit points. For example, the code of Figure 14.5 has four different return statements. One might want to check that each of the four returns is exercised in the actual context in which the procedure is used. Each of these would have been exercised already if even the simplest statement coverage criterion were satisfied during unit testing, but perhaps only in the context of a simple test driver; testing in the real context could reveal interface faults that were previously undetected.

Δ Procedure Entry and Exit Testing

Exercising all the entry points of a procedure is not the same as exercising all the calls. For example, procedure A may call procedure C from two distinct points, and procedure B may also call procedure C. In this case, coverage of calls of C means exercising all three of the points of calls. If the component under test has been constructed in a bottom-up manner, as is common, then unit testing of A and B may already have exercised calls of C. In that

Δ Procedure Call Testing

case, even statement coverage of A and B would ensure coverage of the calls relation (although not in the context of the entire component). Commonly available testing tools can measure coverage of entry and exit points.

The search function in Figure 14.5 was originally part of a symbol table package in a small compiler. It was called at only point, from one other C function in the same unit.[11] That C function, in turn, was called from tens of different points in a scanner and a parser. Coverage of calls requires exercising each statement in which the parser and scanner access the symbol table, but this would almost certainly be satisfied by a set of test cases exercising each production in the grammar accepted by the parser.

When procedures maintain internal state (local variables that persist from call to call), or when they modify global state, then properties of interfaces may only be revealed by sequences of several calls. In object-oriented programming, local state is manipulated by procedures called *methods*, and systematic testing necessarily concerns sequences of method calls on the same object. Even simple coverage of the "calls" relation becomes more challenging in this environment, since a single call point may be dynamically bound to more than one possible procedure (method). While these complications may arise even in conventional procedural programs (e.g., using function pointers in C), they are most prevalent in object-oriented programming. Not surprisingly, then, approaches to systematically exercising sequences of procedure calls are beginning to emerge mainly in the field of object-oriented testing, and we therefore cover them in Chapter 17.

14.7 Implicit Control Flow

Note for the readers of this draft version: This section will deal with execution paths that don't show up directly in the control flow graph model. This is primarily exception handling.

TO BE WRITTEN

14.8 Comparing Structural Testing Criteria

Advanced

The power and cost of the structural test adequacy criteria described in this chapter can be formally compared using the *subsumes* relation introduced in Chapter 11. The relations among these criteria are illustrated in Figure 14.8. They are divided into practical criteria which can always be satisfied by test sets whose size is at most a linear function of program size, and techniques which are of mainly theoretical interest because they may require impracti-

[11]The "unit" in this case is the C source file, which provided a single data abstraction through several related C functions, much as a C++ or Java class would provide a single abstraction through several methods. The search function was analogous in this case to a private (internal) method of a class.

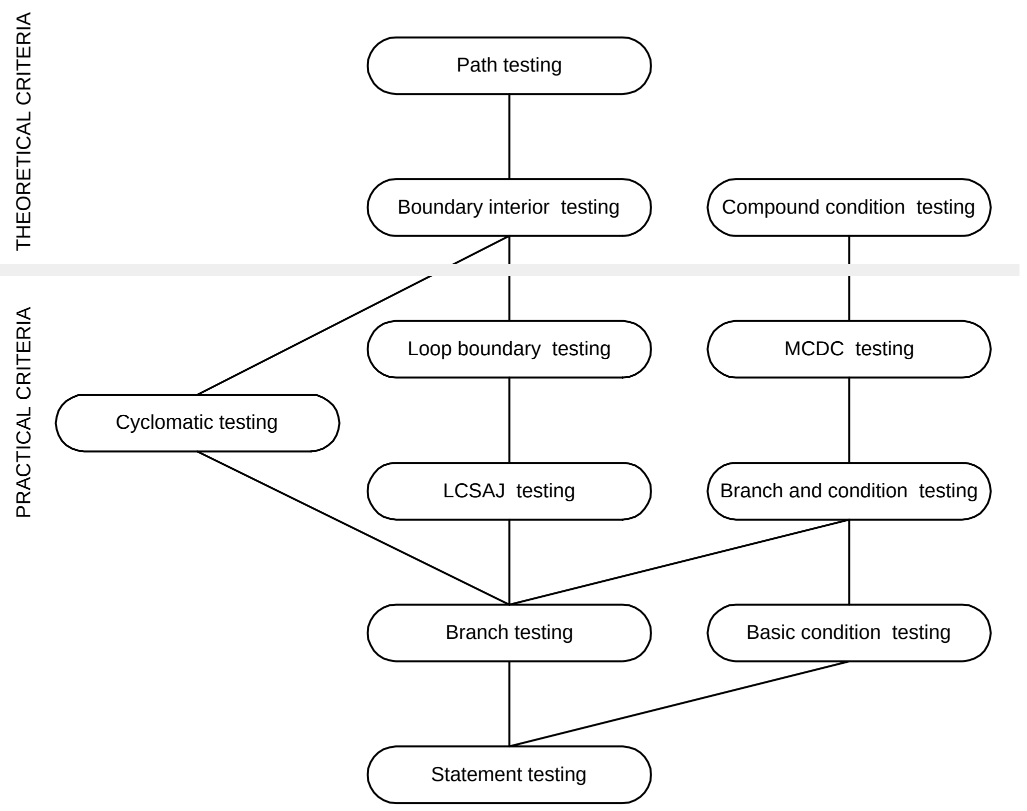

Figure 14.8: The subsumption relation among structural test adequacy criteria described in this chapter.

cally large numbers of test cases or even (in the case of path coverage) an infinite number of test cases.

The hierarchy can be roughly divided into a part that relates requirements for covering program paths, and another part that relates requirements for covering combinations of conditions in branch decisions. The two parts come together at branch coverage. Above branch coverage, path-oriented criteria and condition-oriented criteria are generally separate, because there is considerable cost and little apparent benefit in combining them. Statement coverage is at the bottom of the subsumes hierarchy for systematic coverage of control flow. Applying any of the structural coverage criteria, therefore, implies at least executing all the program statements.

14.9 The Infeasibility Problem

Advanced

Sometimes *no* set of test cases is capable of satisfying some test coverage criterion A for a particular program P, because the criterion requires execution of some program element that can never be executed. This is true even for the statement coverage criterion, weak as it is. Unreachable statements can occur as a result of defensive programming (e.g., checking for error conditions that never occur) and code reuse (reusing code that is more general than strictly required for the application). Large amounts of "fossil" code may accumulate when a legacy application becomes unmanageable, and may in that case indicate serious maintainability problems, but some unreachable code is common even in well-designed, well-maintained systems, and must be accomodated in testing processes that otherwise require satisfaction of coverage criteria.

Stronger coverage criteria tend to require coverage of more infeasible elements. For example, in discussing multiple condition coverage, we implicitly assumed that basic conditions were independent and could therefore occur in any combination. In reality, basic conditions may be comparisons or other relational expressions and may be interdependent in ways that make certain combinations infeasible. For example, in the expression `(a > 0 && a < 10)`, it is not possible for both basic conditions to be **False**. Fortunately, short-circuit evaluation rules ensure that the combination $\langle\mathbf{False},\mathbf{False}\rangle$ is not required for multiple condition coverage of this particular expression in a C or Java program. For the modified condition/decision adequacy criterion in particular, one can make a strong argument that a test case that is required but not feasible indicates a defect in design or coding.

The infeasibility problem is most acute for structural coverage criteria for paths in the control flow graph, such as the boundary/interior coverage criterion. Consider, for example, the following simple code sequence:

```
if (a < 0) {
   a = 0;
}
if (a > 10) {
   a = 10;
}
```

It is not possible to traverse the sub-path on which the **True** branch is taken for both `if` statements. In the trivial case where these `if` statements occur together, the problem is both easy to understand and to avoid (by placing the second `if` within an `else` clause), but essentially the same interdependence can occur when the decisions are separated by other code.

An easy but rather unsatisfactory solution to the infeasibility problem is to make allowances for it by setting a coverage goal less than 100%. For example, we could require 90% coverage of basic blocks, on the grounds that no more than 10% of the blocks in a program should be infeasible. A 10% allowance for infeasible blocks may be insufficient for some units and too generous for others.

The other main option is requiring justification of each element left uncovered. This is the approach taken in some quality standards, notably RTCA/-DO-178B and EUROCAE ED-12B for modified condition/decision coverage (MCDC). Explaining why each element is uncovered has the salutory effect of distinguishing between defensive coding and sloppy coding or maintenance, and may also motivate simpler coding styles. However, it is more expensive (because it requires manual inspection and understanding of each element left uncovered) and is unlikely to be cost-effective for criteria that impose test obligations for large numbers of infeasible paths. This problem, even more than the large number of test cases that may be required, leads us to conclude that stringent path-oriented coverage criteria are seldom useful.

14.10 The Incrementality Problem

Advanced

One would like the cost of re-testing a program after a change to be proportional to the size of the change, and not proportional to the size of the program. Structural coverage criteria are not incremental in this sense. Even a small change has an unpredictable effect on coverage. If a set of test cases has achieved a certain level of coverage before the change, it is impossible to determine what level of coverage the same test set will produce after the program change. This implies that measurement of structural coverage in quickly evolving software cannot be used in the same way it might for a stable unit. For example, during development of a unit, structural coverage might be used to identify untested elements, but measures of satisfaction of coverage would be of little value. When the unit is delivered to an independent test group, structural coverage could be more profitably used as an indicator of the thoroughness of testing by the developers, or as a termination condition.

Open Research Issues

Devising and comparing structural criteria was a hot topic in the 80s. It is no longer an active research area for imperative programming, but new programming paradigms or design techniques present new challanges. Polymorphism, dynamic binding, object oriented and distributed code open new problems and require new techniques, as discussed in other chapters. Applicability of structural criteria to architectural design descriptions is still under investigation. Usefulness of structural criteria for implicit control flow has been addressed only recently.

Early testing research, including research on structural coverage criteria, was concerned largely with improving the fault-detection effectiveness of testing. Today, the most pressing issues are cost and schedule. Better automated techniques for identifying infeasible paths will be necessary before more stringent structural coverage criteria can be seriously considered in any but the most critical of domains. Alternatively, for many applications it may be more appropriate to gather evidence of feasibility from actual product use; this is called *residual* test coverage monitoring and is a topic of current research. The incrementality problem described above is particularly important in the context of rapid cycles of product development and change, and will surely be a topic of further testing research for the next several years. In particular we expect further research in inferring coverage during regression testing.

Further Reading

The main structural adequacy criteria are presented in Myers' *The Art of Software Testing* [Mye79], which has been a preeminent source of information for more than two decades. It is a classic despite its age, which is evident from the limited set of techniques addressed and the programming language used in the examples. The excellent survey by Adrion et al. [ABC82] remains the best overall survey of testing techniques, despite similar age. Frankl and Weyuker [FW93] provide a modern treatment of the subsumption hierarchy among structural coverage criteria.

Boundary/interior testing is presented by Howden [How75]. Woodward et al. [WHH80] present LCSAJ testing. Cyclomatic testing is described by McCabe [McC83].

Related Topics

Readers with a strong interest in coverage criteria should continue with the next chapter, which presents data flow testing criteria. Others may wish to proceed to chapters that describe application of structural testing in the particular problem domains. Chapter 16 describes testing programs with complex data structures, and Chapter 17 discusses testing object-oriented programs. Readers wishing a more comprehensive view of unit testing should continue with Chapters 20 on test data generation and 21 on design of test scaffolding. Tool support for structural testing discussed in Chapter 29. The process context of structural testing is described in Chapter 31.

Bibliography

[ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, June 1982.

[FW93] Phyllis. G. Frankl and Elain G. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, October 1993.

[How75] W. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computer*, 1975.

[McC83] T. McCabe. *Structured Testing*. IEEE Computer Society Press, 1983.

[Mye79] G. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[WHH80] M.R. Woodward, D. Hedley, and M.A. Hennel. Experience with path analysis and testing of programs. *TSE*, 1980.