Chapter 12

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 the 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 functional 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 5

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

- Chapter 9

  The introduction to test case adequacy and test case selection in general sets the context for this chapter. It is not strictly required for understanding this chapter, but is helpful for understanding how the techniques described in this chapter should be applied.

[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.

12.1 Overview

Testing can reveal a fault only when execution of the faulty element causes a failure. For example, if there were a fault in the statement at line 31 of the program in Figure 12.1, 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, or a corrupt state may not propagate through execution to eventually lead to failure. Let us assume for example to have erroneously typed 6 instead of 16 in the statement at line 31 of the program in Figure 12.1. 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 26 of the program in Figure 12.1 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 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 a dynamic memory checking tool that identifies memory violations.

Control flow testing complements functional testing by including cases that may not be identified from specifications alone. A typical case is implementation of a single item of the specification by multiple parts of 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 functional criteria. The most notable example is the class of so called *missing path* faults. Such faults result from the missing im-

[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 12.8

```
 1  #include "hex_values.h"
 2  /**
 3   * @title cgi_decode
 4   * @desc
 5   *     Translate a string from the CGI encoding to plain ascii text
 6   *     '+' becomes space, %xx becomes byte with hex value xx,
 7   *     other alphanumeric characters map to themselves
 8   *
 9   *     returns 0 for success, positive for erroneous input
10   *           1 = bad hexadecimal digit
11   */
12  int cgi_decode(char *encoded, char *decoded) {
13    char *eptr = encoded;
14    char *dptr = decoded;
15    int ok=0;
16    while (*eptr) {
17      char c;
18      c = *eptr;
19      /* Case 1: '+' maps to blank */
20      if (c == '+') {
21        *dptr = ' ';
22      } else if (c == '%') {
23        /* Case 2: '%xx' is hex for character xx */
24        int   digit_high = Hex_Values[*(++eptr)];
25        int   digit_low  = Hex_Values[*(++eptr)];
26        /* Hex_Values maps illegal digits to -1 */
27        if (    digit_high == -1 || digit_low == -1 ) {
28          /* *dptr='?'; */
29          ok=1; /* Bad return code */
30        } else {
31          *dptr = 16* digit_high + digit_low;
32        }
33        /* Case 3: All other characters map to themselves */
34      } else {
35        *dptr = *eptr;
36      }
37      ++dptr;
38      ++eptr;
39    }
40    *dptr = '\0';              /* Null terminator for string */
41    return ok;
42  }
```

Figure 12.1: 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).

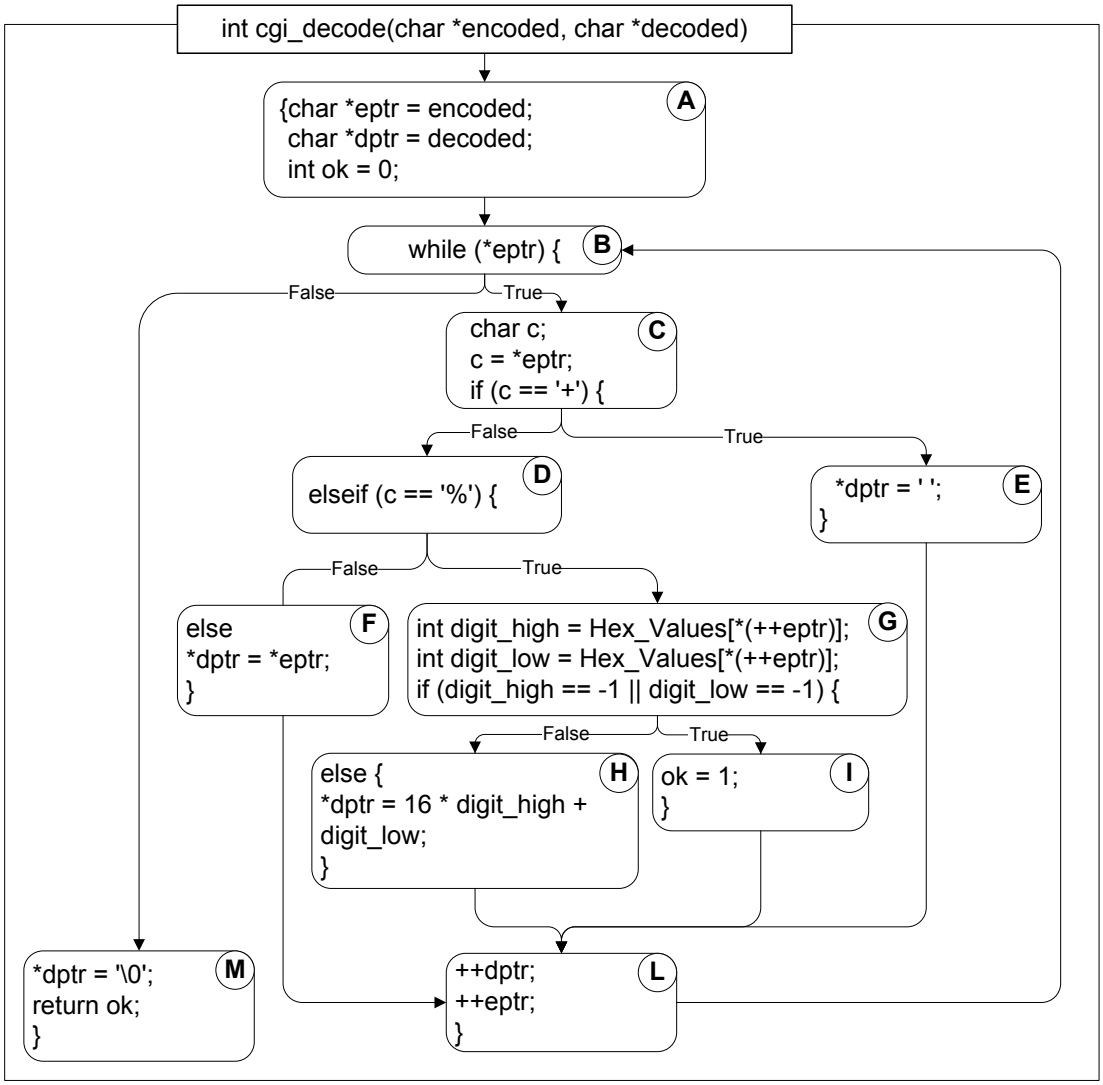


Figure 12.2: Control flow graph of function cgi_decode from Figure 12.1

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 12.1: Sample test suites for C function cgi_decode from Figure 12.1

plementation of some items in the specification. For example, the program in Figure 12.1 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 specification.

In practice, control flow testing criteria are used to evaluate the thoroughness of test suites derived from functional 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 functional test suites.

Control flow adequacy can be easily measured with automatic tools. 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[3].

12.2 Statement Testing

The most intuitive 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.

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

[3] Application of test adequacy criteria within the testing process is discussed in Chapter 20.

T satisfies the statement adequacy criterion if $C_{Statement} = 1$. The ratio of visited control flow graph nodes to total nodes may differ from the ratio of executed statements to all statements, depending on the granularity of the control flow graph representation.

Δ basic block coverage

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 5, 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.

Let us consider for example the program of Figure 12.1. 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 execute statement $\mathsf{ok} = 1$ at line 29. 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 12.2. 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.

Coverage is not monotone with respect to the size of test suites, i.e., test suites that contain fewer test cases may achieve a higher coverage than test suites that contain more test cases. 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 12.1.)

Criteria can be satisfied by many test suites of different sizes. A test suite Both T_1 and Both T_1 and

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

cause all statements to be executed and thus 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 suppose, for example, we omitted the 1 in the statement at line 31 of the program in Figure 12.1. Both test suites T_1 and T_2 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 31.

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 whole test suite, and not in the contribution of individual test cases.

12.3 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 34. The control flow graph of program cgi_decode′ is shown in Figure 12.3. 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.

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.

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

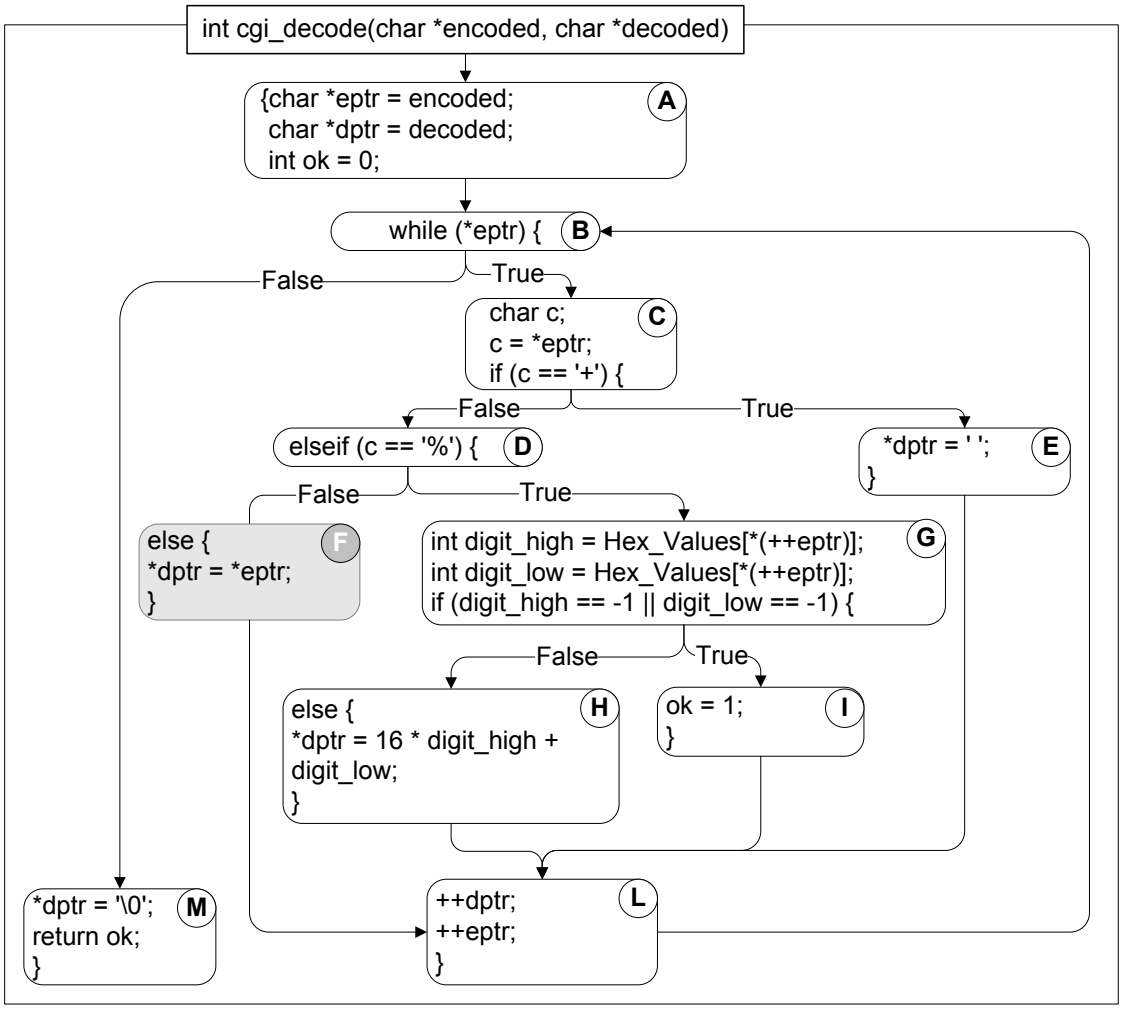


Figure 12.3: The control flow graph of function cgi_decode′ which is obtained from the program of Figure 12.1 after removing node F.

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 also satisfy the statement adequacy criterion for the same program.[4] The contrary is not true, as illustrated by test suite T_3 for the program cgi_decode′ presented above.

12.4 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 27 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 basic conditions of the decisions to be separately exercised. 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.

The simplest condition adequacy criterion, called basic condition coverage requires each basic condition to be covered. Each basic condition must have a *True* and a *False* outcome at least once during the execution of the test suite.

Δ basic condition adequacy criterion

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

Δ basic condition coverage

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.

$$C_{\text{Basic\_Condition}} = \frac{\text{total number of truth values assumed by all basic conditions}}{2 \times \text{number of basic conditions}}$$

[4] 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.

T satisfies the basic condition adequacy criterion if $C_{Basic\_Conditions} = 1$. Notice that the total number of truth values that the basic conditions can take is twice the number of basic conditions, since each basic condition can assume value *true* or *false*. For example, the program in Figure 12.1 contains five basic conditions, which in sum may take ten possible truth values. Three basic conditions correspond to the simple decisions at lines 18, 22, and 24, i.e., decisions that each contain only one basic condition. Thus they are covered by any test suite that covers all branches. The remaining two conditions occur in the compound decision at line 27. 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 case 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

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

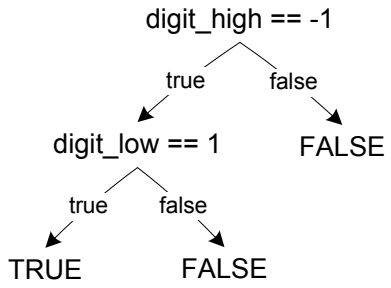
satisfies the basic condition adequacy criterion, but not the branch condition adequacy criterion, since the outcome of the decision at line 27 is always *False*. Thus branch and basic condition adequacy criteria are not directly comparable.

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

Δ branch and condition adequacy

A more complete extension that includes both the basic condition and the branch adequacy criteria is the *compound condition adequacy criterion*,[5] which requires a test for each possible evaluation of compound conditions. It is most natural to visualize compound condition adequacy as covering paths to leaves of the evaluation tree for the expression. For example, the compound condition at line 27 would require covering the three paths in the following tree:

Δ compound condition adequacy



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

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

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:

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/decision coverage* or MCDC, also known as the *modified condition adequacy criterion*. The modified condition/decision 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 evaluated 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 can be satisfied with $N+1$ test cases, making it an attrac-

Δ modified condition/decision coverage (MCDC)

tive compromise between number of required test cases and thoroughness of the test. It 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/decision 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*

The values underlined in the table independently affect the outcome of the decision. Note that the same test case can cover the values of several basic conditions. For example, test case (1) covers value *True* for the basic 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.

12.5 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

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

Δ path coverage

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.

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 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.

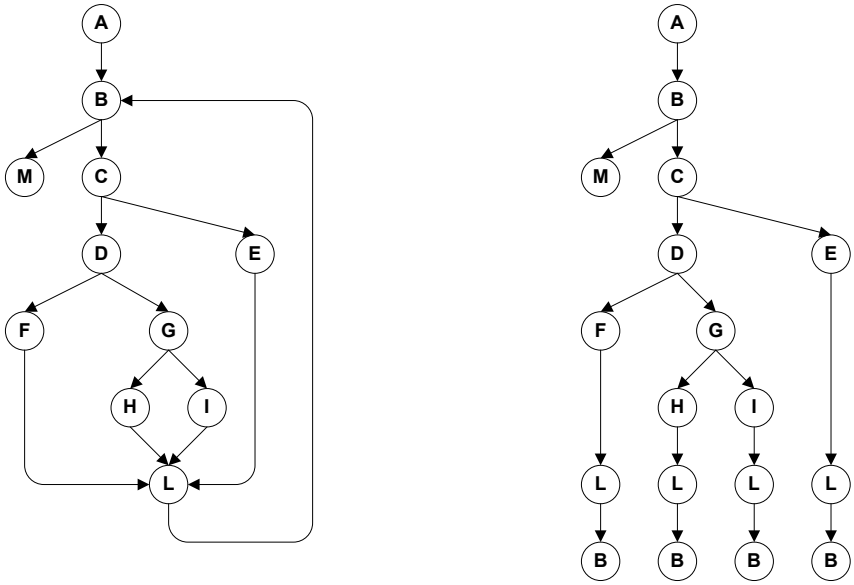


Figure 12.4: Deriving a tree from a control flow graph to derive sub-paths for boundary/interior testing. Part (i) is the control flow graph of the C function cgi_decode, identical to Figure 12.1 but showing only node identifiers without source code. Part (ii) is a tree derived from part (i) by following 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.

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 12.4 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 12.5 illustrates a fault that may not be uncovered using statement or decision testing, but will assuredly be detected if the boundary interior path 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

```
1  typedef struct cell {
2     itemtype itemval;
3     struct cell *link;
4  } *list;
5  #define NIL ((struct cell *) 0)
6
7  itemtype  search( list *l, keytype k)
8  {
9     struct cell *p = *l;
10    struct cell *back = NIL;
11
12    /* Case 1: List is empty */
13    if (p == NIL) {
14       return NULLVALUE;
15    }
16
17    /* Case 2: Key is at front of list */
18    if (k == p->itemval) {
19       return p->itemval;
20    }
21
22    /* Default: Simple (but buggy) sequential search */
23    p=p->link;
24    while (1) {
25       if (p == NIL) {
26          return NULLVALUE;
27       }
28       if (k==p->itemval) {    /* Move to front */
29          back->link = p->link;
30          p->link = *l;
31          *l = p;
32          return p->itemval;
33       }
34       back=p; p=p->link;
35    }
36 }
```

Figure 12.5: 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.

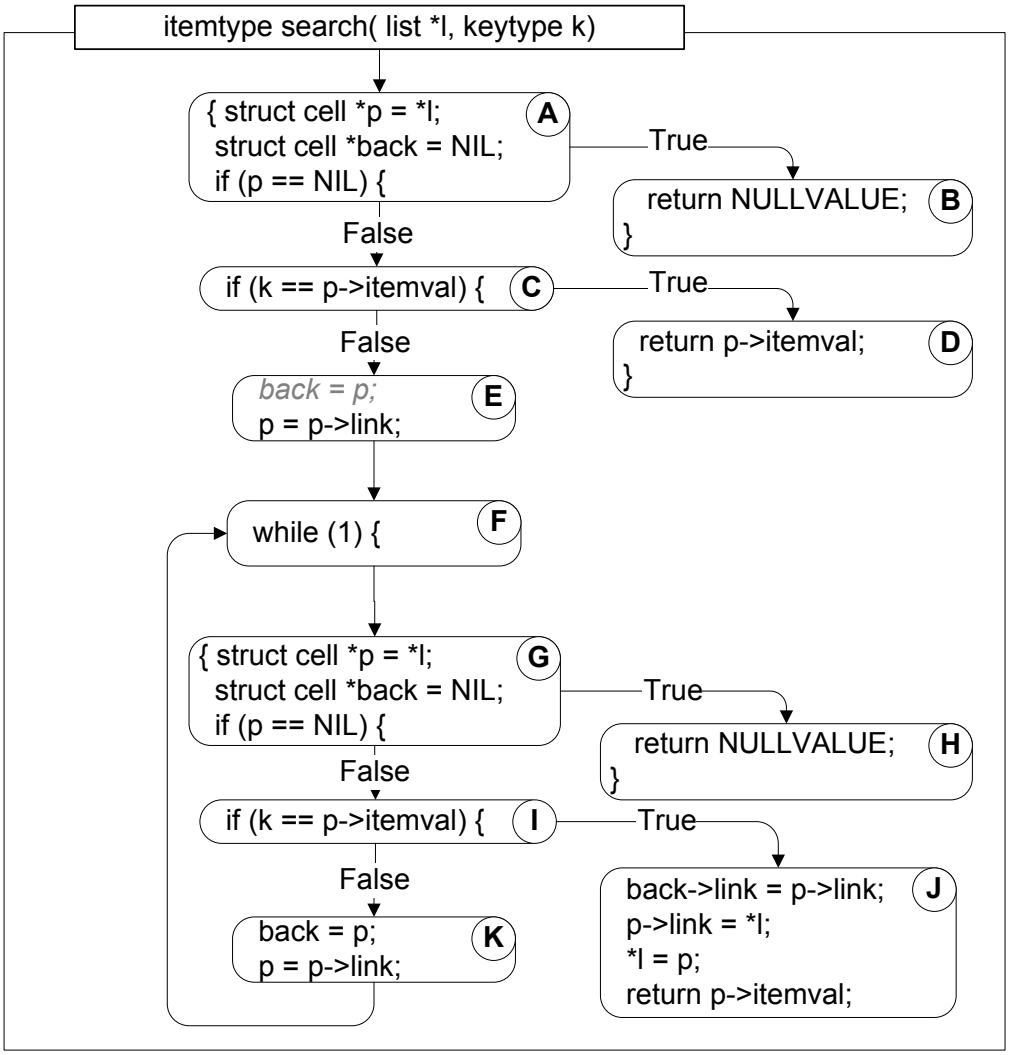


Figure 12.6: The control flow graph of C function search with move-to-front feature.

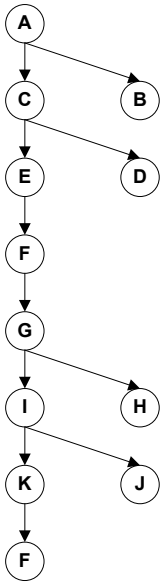


Figure 12.7: The boundary/interior sub-paths for C function search.

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 input data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent.[6]

Since coverage of non-looping paths is expensive, we can consider a variant of

[6] Section 12.8 below discusses infeasible paths.

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

A test suite T for a program P satisfies the loop boundary adequacy criterion 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.[7]
- 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 we would like the last part of the criterion to 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, 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.

Δ linear code sequence and jump (LCSAJ)

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.

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

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 $\langle True, False, True, False\rangle$ 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.

12.6 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 12.5 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 calls at all three points. 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 case, even statement coverage of *A* and *B* would ensure coverage of the calls relation (although not in the context of the entire component).

Δ call coverage

The search function in Figure 12.5 was originally part of a symbol table package in a small compiler. It was called at only one point, from one other C function in the same unit.[8] 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

[8] 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.

beginning to emerge mainly in the field of object-oriented testing, and we therefore cover them in Chapter 15.

12.7 Comparing Structural Testing Criteria

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 9. The relations among these criteria are illustrated in Figure 12.8. They are divided into practical criteria that 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 impractically 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.

Procedure call coverage criteria are not included in the figure, since they do not concern internal control flow of procedures and are thus incomparable with the control flow coverage criteria.

12.8 The Infeasibility Problem

Sometimes *no* set of test cases is capable of satisfying some test coverage criterion for a particular program, because the criterion requires execution of a 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 accommodated in testing processes that otherwise require satisfaction of coverage criteria.

Stronger coverage criteria tend to call for coverage of more elements that may be infeasible. 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

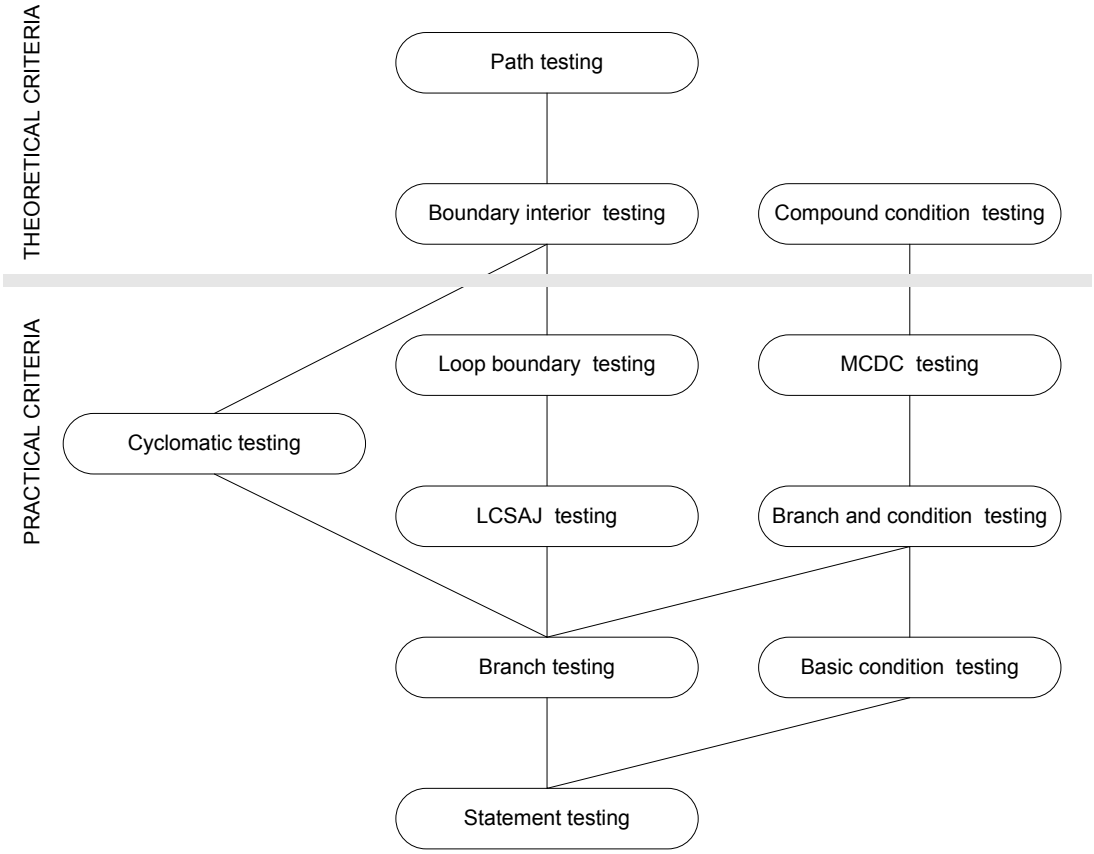


Figure 12.8: The subsumption relation among structural test adequacy criteria described in this chapter.

⟨*False*, *False*⟩ is not required for multiple condition coverage of this particular expression in a C or Java program.

The infeasibility problem is most acute for path-based structural coverage criteria, 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 salutary 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 cost-effective.

Open Research Issues

Devising and comparing structural criteria was a hot topic in the 1980s. It is no longer an active research area for imperative programming, but new programming paradigms or design techniques present new challenges. 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.

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]. Residual test coverage measurement is described by Pavlopoulou and Young [PY99].

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 Chapter 15, which addresses testing object-oriented programs. Readers wishing a more comprehensive view of unit testing may continue with Chapters 17 on test scaffolding and test data generation. Tool support for structural testing is discussed in Chapter 23.

Exercises

12.1. Let us consider the following loop, which appears in C lexical analyzers generated by the tool flex:[9]

```
1    for ( n = 0;
2            n < max_size && (c = getc( yyin )) != EOF && c != '\n';
3          ++n )
4                buf[n] = (char) c;
```

Devise a set of test cases that satisfy the compound condition adequacy criterion and a set of test cases that satisfy the modified condition adequacy criterion with respect to this loop.

[9] Flex is a widely used generator of lexical analyzers. Flex was written by Vern Paxson, and is compatible with the original AT&T lex written by M.E. Lesk. This excerpt is from version 2.5.4 of flex, distributed with the Linux operating system.

12.2. The following if statement appears in the Java source code of Grappa,[10] a graph layout engine distributed by AT&T Laboratories:

```
1  if(pos < parseArray.length
2      &&  (parseArray[pos] == '{'
3           || parseArray[pos] == '}'
4           || parseArray[pos] == '|')) {
5        continue;
6    }
```

(a) Derive a set of test case specifications and show that it satisfies the MCDC criterion for this statement. For brevity, 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] == '|')

(b) Do the requirements for compound condition coverage and modified condition/decision coverage differ in this case? Aside from increasing the number of test cases, what difference would it make if we attempted to exhaustively cover all combinations of truth values for the basic conditions?

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

12.4. The number of basis paths (cyclomatic complexity) does not depend on whether nodes of the control flow graph are individual statements or basic blocks which may contain several statements. Why?

12.5. Derive the subsumption hierarchy for the call graph coverage criteria described in this chapter, and justify each of the relationships.

12.6. If the modified condition/decision adequacy criterion requires a test case that is not feasible because of interdependent basic conditions, should this always be taken as an indication of a defect in design or coding? Why or why not?

[10] 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/`. Formatting of the line has been altered for readability in this printed form.

Chapter 13

Data Flow Testing

Exercising every statement or branch with test cases is a practical goal, but exercising every path is impossible, and even the number of simple (that is, loop-free) paths can be exponential in the size of the program. Path-oriented selection and adequacy criteria must therefore select a tiny fraction of control flow paths. Some control flow adequacy criteria, notably the loop boundary interior condition, do so heuristically. Data flow test adequacy criteria improve over pure control flow criteria by selecting paths based on how one syntactic element can affect the computation of another.

Required Background

- Chapter 6

  At least the basic data flow models presented in Chapter 6 section 6.1 are required to understand this chapter, although algorithmic details of data flow analysis can be deferred. Section 6.5 of that chapter is important background for section 13.4 of the current chapter. The remainder of Chapter 6 is useful background but not strictly necessary to understand and apply data flow testing.

- Chapter 9

  The introduction to test case adequacy and test case selection in general sets the context for this chapter. It is not strictly required for understanding this chapter, but is helpful for understanding how the techniques described in this chapter should be applied.

- Chapter 12

  The data flow adequacy criteria presented in this chapter complement control flow adequacy criteria. Knowledge about control flow adequacy criteria is desirable but not strictly required for understanding this chapter.

13.1 Overview

We have seen in Chapter 12 that structural testing criteria are practical for single elements of the program, from simple statements to complex combinations of conditions, but become impractical when extended to paths. Even the simplest path testing criteria require covering large numbers of paths that tend to quickly grow far beyond test suites of acceptable size for non-trivial programs.

Close examination of paths that need to be traversed to satisfy a path selection criterion often reveals that, among a large set of paths, only a few are likely to uncover faults that could have escaped discovery using condition testing coverage. Criteria that select paths based on control structure alone (e.g., boundary interior testing) may not be effective in identifying these few significant paths, because their significance depends not only on control flow but on data interactions.

Data flow testing is based on the observation that computing the wrong value leads to a failure only when that value is subsequently used. Focus is therefore moved from control flow to data flow. Data flow testing criteria pair variable definitions with uses, ensuring that each computed value is actually used, and thus selecting from among many execution paths a set that is more likely to propagate the result of erroneous computation to the point of an observable failure.

Consider for example the C function cgi_decode of Figure 13.1, which decodes a string that has been transmitted through the web's Common Gateway Interface. Data flow testing criteria would require one to execute paths that first define (change the value of) variable eptr, e.g., by incrementing it at line 37 and then use the new value of variable eptr, e.g., using variable eptr to update the array indexed by dptr at line 34. Since a value defined in one iteration of the loop is used on a subsequent iteration, we are obliged to execute more than one iteration of the loop to observe the propagation of information from one iteration to the next.

13.2 Definition Use Associations

Data flow testing criteria are based on data flow information, i.e., variable *definitions* and *uses*. Table 13.1 shows definitions and uses for the program cgi_decode of Figure 13.1. Recall that when a variable occurs on both sides of an assignment, it is first used and then defined, since the value of the variable before the assignment is used for computing the value of the variable after the assignment. For example, the ++eptr increment operation in C is equivalent to the assignment eptr = eptr + 1, and thus first uses and then defines variable eptr.

We will initially consider treatment of arrays and pointers in the current example in a somewhat ad hoc fashion and defer discussion of the general problem to Section 13.4. Variables eptr and dptr are used for indexing the input and output strings. In program cgi_decode, we consider the variables as both indexes (eptr and dptr) and strings (*eptr and *dptr). The assignment *dptr = *eptr is treated as a definition of the string *dptr as well as a use of the index dptr, the index eptr, and the string *eptr, since the result depends on both indexes as well as the contents of the source string. A change to an index is treated as a definition of both the index and the string, since a

```
1
2   /* External file hex_values.h defines Hex_Values[128]
3    * with value 0 to 15 for the legal hex digits (case-insensitive)
4    * and value -1 for each illegal digit including special characters
5    */
6
7   #include "hex_values.h"
8   /**     Translate a string from the CGI encoding to plain ascii text.
9    *      '+' becomes space, %xx becomes byte with hex value xx,
10  *      other alphanumeric characters map to themselves.
11  *     Returns 0 for success, positive for erroneous input
12  *           1 = bad hexadecimal digit
13  */
14  int  cgi_decode(char *encoded, char *decoded) {
15     char *eptr = encoded;
16     char *dptr = decoded;
17     int ok=0;
18     while (*eptr) {
19        char c;
20        c = *eptr;
21
22        if (c == '+') {    /* Case 1:  '+' maps to blank */
23           *dptr = ' ';
24        } else if (c == '%') { /* Case 2:  '%xx' is hex for character xx */
25           int   digit_high = Hex_Values[*(++eptr)];
26           int   digit_low  = Hex_Values[*(++eptr)];
27           if (     digit_high == -1 || digit_low == -1 ) {
28              /* *dptr='?'; */
29              ok=1; /* Bad return code */
30           } else {
31              *dptr = 16* digit_high + digit_low;
32           }
33        } else { /* Case 3:  All other characters map to themselves */
34           *dptr = *eptr;
35        }
36        ++dptr;
37        ++eptr;
38     }
39     *dptr = '\0';                    /* Null terminator for string */
40     return ok;
41  }
```

Figure 13.1: 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). This program is used also in Chapter 12 and is presented also in Figure 12.1 of Chapter 12.

Variable	Definitions	Uses
encoded	14	15
decoded	14	16
*eptr	15, 25, 26, 37	18, 20, 25, 26, 34
eptr	15, 25, 26, 37	15, 18, 20, 25, 26, 34, 37
*dptr	16, 23, 31, 34, 36, 39	40
dptr	16 36	16, 23, 31, 34, 36, 39
ok	17, 29	40
c	20	22, 24
digit_high	25	27, 31
digit_low	26	27, 31
Hex_Values	–	25, 26

Table 13.1: Definitions and uses for function cgi_decode. *eptr and *dptr indicate the strings, while eptr and dptr indicate the indexes.

change of the index changes the value accessed by it. For example in the statement at line 36 (++dptr), we have a use of variable dptr followed by a definition of variables dptr and *dptr.

It is somewhat counter-intuitive that we have definitions of the string *eptr, since it is easy to see that the program is scanning the encoded string without changing it. For the purposes of data flow testing, though, we are interested in interactions between computation at different points in the program. Incrementing the index eptr is a "definition" of *eptr in the sense that it can affect the value that is next observed by a use of *eptr.

Pairing definitions and uses of the same variable v identifies data interactions through v, i.e., definition-use pairs (DU pairs). Table 13.2 shows the DU pairs in program *cgi_decode* of Figure 13.1. Some pairs of definitions and uses from Table 13.1 do not occur in Table 13.2, since there is no definition-clear path between the two statements. For example, the use of variable eptr at line 15 cannot be reached from the increment at line 37, so there is no DU pair $\langle 37, 15 \rangle$. The definitions of variables *eptr and eptr at line 25, are paired only with the respective uses at line 26, since successive definitions of the two variables at line 26 kill the definition at line 25 and eliminate definition-clear paths to any other use.

Δ DU pair

A DU pair requires the existence of at least one *definition-clear path* from definition to use, but there may be several. Additional uses on a path do not interfere with the pair. We sometimes use the term *DU path* to indicate a particular definition-clear path between a definition and a use. For example, let us consider the definition of *eptr at line 37 and the use at line 34. There are infinitely many paths that go from line 37 to the use at line 34. There is one DU path that does not traverse the loop while going from 37 to 34. There are infinitely many paths from 37 back to 37, but only two DU paths, because the definition at 37 kills the previous definition at the same point.

Δ DU path

Data flow testing, like other structural criteria, is based on information obtained through static analysis of the program. We discard definitions and uses that cannot be statically paired, but we may select pairs even if none of the statically identifiable

Variable	DU Pairs
*eptr	⟨ 15, 18 ⟩, ⟨ 15, 20 ⟩, ⟨ 15, 25 ⟩, ⟨ 15, 34 ⟩ ⟨ 25, 26 ⟩, ⟨ 26, 37 ⟩ ⟨ 37, 18 ⟩, ⟨ 37, 20 ⟩, ⟨ 37, 25 ⟩, ⟨ 37, 34 ⟩
eptr	⟨ 15, 15 ⟩, ⟨ 15, 18 ⟩, ⟨ 15, 20 ⟩, ⟨ 15, 25 ⟩, ⟨ 15, 34 ⟩, ⟨ 15, 37 ⟩, ⟨ 25, 26 ⟩, ⟨ 26, 37 ⟩ ⟨ 37, 18 ⟩, ⟨ 37, 20 ⟩, ⟨ 37, 25 ⟩, ⟨ 37, 34 ⟩, ⟨ 37, 37 ⟩
*dptr	⟨ 39, 40 ⟩
dptr	⟨ 16, 16 ⟩, ⟨ 16, 23 ⟩, ⟨ 16, 31 ⟩, ⟨ 16, 34 ⟩, ⟨ 16, 36 ⟩, ⟨ 16, 39 ⟩, ⟨ 36, 23 ⟩, ⟨ 36, 31 ⟩, ⟨ 36, 34 ⟩, ⟨ 36, 36 ⟩, ⟨ 36, 39 ⟩
ok	⟨ 17, 40 ⟩, ⟨ 29, 40 ⟩
c	⟨ 20, 22 ⟩, ⟨ 20 24 ⟩
digit_high	⟨ 25, 27 ⟩, ⟨ 25, 31 ⟩
digit_low	⟨ 26, 27 ⟩, ⟨ 26, 31 ⟩
encoded	⟨ 14, 15⟩
decoded	⟨ 14, 16⟩

Table 13.2: DU pairs for function cgi_decode. Variable Hex_Values does not appear because it is not defined (modified) within the procedure.

definition-clear paths is actually executable. In the current example, we have made use of information that would require a quite sophisticated static data flow analyzer, as discussed below in Section 13.4.

13.3 Data Flow Testing Criteria

Various data flow testing criteria can be defined by requiring coverage of DU pairs in various ways.

The *All DU pairs adequacy criterion* requires each DU pair to be exercised in at least one program execution, following the idea that an erroneous value produced by one statement (the definition) might be revealed only by its use in another statement.

Δ all DU pairs adequacy criterion

A test suite T for a program P satisfies the all DU pairs adequacy criterion iff, for each DU pair du of P, at least one test case in T exercises du.

The corresponding coverage measure is the proportion of covered DU pairs:

Δ all DU pairs coverage

The all DU pair coverage $C_{DU\,pairs}$ of T for P is the fraction of DU pairs of program P exercised by at least one test case in T.

$$C_{DU\,pairs} = \frac{\text{number of exercised DU pairs}}{\text{number of DU pairs}}$$

The All DU pairs adequacy criterion assures a finer grain coverage than statement and branch adequacy criteria. If we consider for example function cgi_decode, we can easily see that statement and branch coverage can be obtained by traversing the while loop no more than once, e.g., with the test suite $T_{branch} = \{$"+", "%3D", "%FG", "t"$\}$ while several DU pairs cannot be covered without executing the while loop at least twice. The

pairs that may remain uncovered after statement and branch coverage correspond to occurrences of different characters within the source string, and not only at the beginning of the string. For example, the DU pair $\langle 37, 25 \rangle$, for variable *eptr can be covered with a test case $TC_{DU\,pairs}$"test%3D" where the hexadecimal escape sequence occurs inside the input string, but not with "%3D". The test suite $T_{DU\,pairs}$ obtained by adding the test case $TC_{DU\,pairs}$ to the test suite T_{branch} satisfies the all DU pairs adequacy criterion, since it adds both the cases of a hexadecimal escape sequence and an ASCII character occurring inside the input string.

One DU pair might belong to many different execution paths. The *All DU paths adequacy criterion* extends the all DU pairs criterion by requiring each simple (non-looping) DU path to be traversed at least once, thus including the different ways of pairing definitions and uses. This can reveal a fault by exercising a path on which a definition of a variable should have appeared but was omitted.

Δ all DU paths adequacy criterion

A test suite T for a program P satisfies the all DU paths adequacy criterion iff, for each simple DU path dp of P, there exists at least one test case in T that exercises a path that includes dp.

Δ all DU paths coverage

The corresponding coverage measure is the fraction of covered simple DU paths:

The all DU pair coverage $C_{DU\,paths}$ of T for P is the fraction of simple DU paths of program P executed by at least one test case in T.

$$C_{DU\,paths} = \frac{\text{number of exercised simple DU paths}}{\text{number of simple DU paths}}$$

The test suite $T_{DU\,pairs}$ does not satisfy the all DU paths adequacy criterion, since both DU pairs $\langle 37, 37 \rangle$ for variable eptr and $\langle 36, 23 \rangle$ for variable dptr correspond each to two simple DU paths, and in both cases one of the two paths is not covered by test cases in $T_{DU\,pairs}$. The uncovered paths correspond to a test case that includes character '+' occurring within the input string, e.g., test case $TC_{DU\,paths}$ = "test+case".

While the number of simple DU paths is often quite reasonable, in the worst case it can be exponential in the size of the program unit. This can occur when the code between the definition and use of a particular variable is essentially irrelevant to that variable, but contains many control paths. The procedure in Figure 13.2 illustrates. The code between the definition of ch in line 2 and its use in line 12 does not modify ch, but the all DU paths coverage criterion would require that each of the 256 paths be exercised.

We normally consider both *All DU paths* and *All DU pairs adequacy criteria* as relatively powerful and yet practical test adequacy criteria, as depicted in Figure 12.8 of page 233. However, in some cases, even the all DU pairs criterion may be too costly. In these cases, we can refer to a coarser grain data flow criterion, the *All definitions adequacy criterion*, which requires pairing each definition with at least one use.

Δ all definitions adequacy criterion

A test suite T for a program P satisfies the all definitions adequacy criterion for P iff, for each definition def of P, there exists at least one test case in T that exercises a DU pair that includes def.

```
1
2  void countBits(char ch) {
3    int count = 0;
4    if (ch & 1) ++count;
5    if (ch & 2) ++count;
6    if (ch & 4) ++count;
7    if (ch & 8) ++count;
8    if (ch & 16) ++count;
9    if (ch & 32) ++count;
10   if (ch & 64) ++count;
11   if (ch & 128) ++count;
12   printf("'%c' (0X%02x) has %d bits set to 1\n",
13          ch, ch, count);
14 }
```

Figure 13.2: A C procedure with a large number of DU paths. The number of DU paths for variable ch is exponential in the number of if statements, because the use in each increment and in the final print statement can be paired with any of the preceding definitions. The number of DU paths for variable count is the same as the number of DU pairs. For variable ch, there is only one DU pair, matching the procedure header with the final print statement, but there are 256 definition-clear paths between those statements — exponential in the number of intervening if statements.

The corresponding coverage measure is the proportion of covered definitions, where we say a definition is covered only if the value is used before being killed:

Δ all definitions coverage

The all definitions coverage C_{Def} of T for P is the fraction of definitions of program P covered by at least one test case in T.

$$C_{defs} = \frac{\text{number of covered definitions}}{\text{number of definitions}}$$

13.4 Data Flow Coverage with Complex Structures

Like all static analysis techniques, data flow analysis approximates the effects of program executions. It suffers imprecision in modeling dynamic constructs, in particular dynamic access to storage, such as indexed access to array elements or pointer access to dynamically allocated storage. We have seen in Chapter 6, (page 94) that the proper treatment of potential aliases involving indexes and pointers depends on the use to which analysis results will be put. For the purpose of choosing test cases, some risk of under-estimating alias sets may be preferable to gross over-estimation or very expensive analysis.

The precision of data flow analysis depends on the precision of alias information used in the analysis. Alias analysis requires a trade-off between precision and compu-

```
1  void pointer_abuse() {
2     int i=5, j=10, k=20;
3     int *p, *q;
4     p = &j + 1;
5     q = &k;
6     *p = 30;
7     *q = *q + 55;
8     printf("p=%d, q=%d\n", *p, *q);
9  }
```

Figure 13.3: Pointers to objects in the program stack can create essentially arbitrary definition-use associations, particularly when combined with pointer arithmetic as in this example.

tational expense, with significant over-estimation of alias sets for approaches that can be practically applied to real programs. In the case of data flow testing, imprecision can be mitigated by specializing the alias analysis to identification of definition-clear paths between a potentially matched definition and use. We do not need to compute aliases for all possible behaviors, but only along particular control flow paths. The risk of under-estimating alias sets in a local analysis is acceptable considering the application in choosing good test cases rather than offering hard guarantees of a property.

In the cgi_decode example we have made use of information that would require either extra guidance from the test designer or a sophisticated tool for data flow and alias analysis. We may know, from a global analysis, that the parameters encoded and decoded never refer to the same or overlapping memory regions, and we may infer that initially eptr and dptr likewise refer to disjoint regions of memory, over the whole range of values that the two pointers take. Lacking this information, a simple static data flow analysis might consider *dptr a potential alias of *eptr, and might therefore consider the assignment *dptr = *eptr a potential definition of both *dptr and *eptr. These spurious definitions would give rise to infeasible DU pairs, which produce test obligations that can never be satisfied. A local analysis that instead assumes (without verification) that *eptr and *dptr are distinct could fail to require an important test case if they can be aliases. Such under-estimation may be preferable to creating too many infeasible test obligations.

A good alias analysis can greatly improve the applicability of data flow testing, but cannot eliminate all problems. Undisciplined use of dynamic access to storage can make precise alias analysis extremely hard or impossible. For example, the use of pointer arithmetic in the program fragment of Figure 13.3 results in aliases that depend on the way the compiler arranges variables in memory.

13.5 The Infeasibility Problem

Not all elements of a program are executable, as discussed in Section 12.8 of Chapter 12. The path oriented nature of data flow testing criteria aggravates the problem

since infeasibility creates test obligations not only for isolated unexecutable elements, but also for infeasible combinations of feasible elements.

Complex data structures may amplify the infeasibility problem by adding infeasible paths as a result of alias computation. For example, while we can determine that x[i] is an alias of x[j] exactly when i = j, we may not be able to determine whether i can be equal to j in any possible program execution.

Fortunately the problem of infeasible paths is usually modest for the all definitions and all DU pairs adequacy criteria, and one can typically achieve levels of coverage comparable to those achievable with simpler criteria like statement and branch adequacy during unit testing. The all DU paths adequacy criterion, on the other hand, often requires much larger numbers of control flow paths. It presents a greater problem in distinguishing feasible from infeasible paths, and should therefore be used with discretion.

Open research issues

Data flow test adequacy criteria are close to the border between techniques that can be applied at low cost with simple tools, and techniques that offer more power but at much higher cost. While in principle data flow test coverage can be applied at modest cost (at least up to the all DU adequacy criterion), it demands more sophisticated tool support than test coverage monitoring tools based on control flow alone.

Fortunately data flow analysis and alias analysis have other important applications. Improved support for data flow testing may come at least partly as a side benefit of research in the programming languages and compilers community. In particular, finding a good balance of cost and precision in data flow and alias analysis across procedure boundaries (inter-procedural or "whole program" analysis) is an active area of research.

The problems presented by pointers and complex data structures cannot be ignored. In particular, modern object-oriented languages like Java use reference semantics — an object reference is essentially a pointer — and so alias analysis (preferably inter-procedural) is a prerequisite for applying data flow analysis to object oriented programs.

Further Reading

The concept of test case selection using data flow information was apparently first suggested in 1976 by Herman [Her76], but that original paper is not widely accessible. The version of data flow test adequacy criteria more widely known was developed independently by Rapps and Weyuker [RW85] and by Laski and Korel [LK83]. The variety of data flow testing criteria is much broader than the handful of criteria described in this chapter; Clarke et al. present a formal comparison of several criteria [CPRZ89]. Frankl and Weyuker consider the problem of infeasible paths and how they affect the relative power of data flow and other structural test adequacy criteria [FW93].

Marx and Frankl consider the problem of aliases and application of alias analysis on individual program paths [MF96]. A good example of modern empirical research on

costs and effectiveness of structural test adequacy criteria, and data flow test coverage in particular, is Frankl and Iakounenko [FI98].

Related Topics

The next chapter discusses model-based testing. Section 14.4 shows how control and data flow models can be used to derive test cases from specifications. Chapter 15 illustrates the use of data flow analysis for structural testing of object oriented programs.

Readers interested in the use of data flow for program analysis can proceed with Chapter 19.

Exercises

13.1. Sometimes a distinction is made between uses of values in predicates (*p-uses*) and other "computational" uses in statements (*c-uses*). New criteria can be defined using that distinction, for example:

all p-use some c-use: for all definitions and uses, exercise all (def, p-use) pairs and at least one (def, c-use) pair

all c-use some p-use: for all definitions and uses, exercise all (def, c-use) pairs and at least one (def, p-use) pair

(a) provide a precise definition of these criteria.

(b) describe the differences in the test suites derived applying the different criteria to function cgi-decode in Figure 13.1.

13.2. Demonstrate the subsume relation between all p-use some c-use, all c-use some p-use, all DU pairs, all DU paths and all definitions.

13.3. How would you treat the buf array in the transduce procedure shown in Figure 16.1?