Structural Testing

John T. Bell

Department of Computer Science
University of Illinois, Chicago

based on material from chapter ? of "Software Testing and Analysis", by Pezze and Young.

Overview

- Structural testing considers the internal structure of code when determining tests.
- A flaw will not reveal itself unless the line containing the flaw is executed.
- Unfortunately, executing a flaw does not always produce (noticeable) error results.
- Best used as a metric for measuring progress and estimating coverage than as an end goal.

Preview of Structural Methods

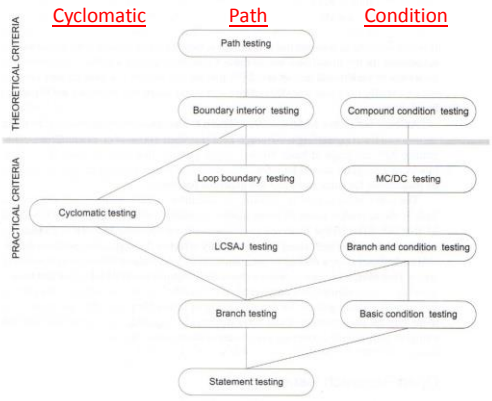


Figure 12.8: The subsumption relation among structural test adequacy criteria described in this chapter.

Running Example (See Handout)
Flaw on line 25 may not always show

```
1  #include "hex_values.h"
2  /**
3   * @file 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).

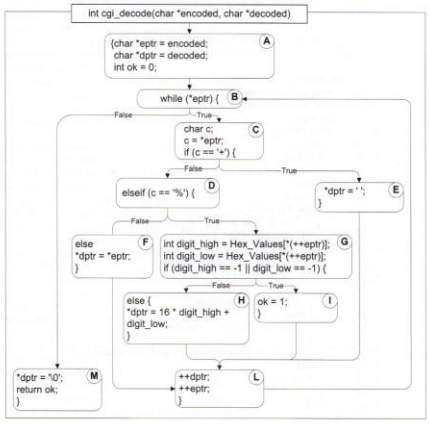


Figure 12.2: Control flow graph of function cgi_decode from Figure 12.1

Statement Testing

- $C_{Statement} = \frac{\#\ of\ executed\ statements}{\#\ of\ executable\ statements}$
- Statement coverage is the most basic, and is subsumed by all other coverages.
- Basic block coverage is similar, in that it attempts to reach every node on the flow graph.
- Consider the trade-offs of more simple tests versus fewer complex tests in the test suite.

Condition Testing

- Branch Coverage
- Basic Condition Coverage
- Branch and Condition Coverage
- Compound Condition Coverage
- MC/DC – Modified Condition/Decision Coverage

Branch Coverage

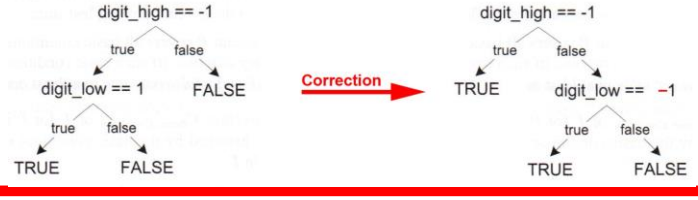
- $C_{Branch} = \frac{\#\ of\ executed\ branches}{\#\ of\ branches}$
- Branches correspond to arcs on the flow graph.
- A test (suite) may cover all statements without covering all branches, whenever a branch contains no statements. E.g. a missing "else".
- Consider if node F were removed from the sample control flow graph.

Basic Condition Coverage

- $C_{basic\_condition} = \frac{\#\ of\ truth\ values\ assumed}{2 * \#\ of\ basic\ conditions}$
- Goal is to evaluate every basic condition in both true and false cases.
- Basic condition criteria can be covered without covering all branches (or statements.)
  - For example a compound conditional in which the overall result stays the same for all test cases.
  - See (corrected) example on next slide.

Corrections to page 220

- Test case T_4 = { “first+test%9Ktest%K9” } always evaluates to TRUE on line 27, even though both true and false cases are evaluated for the basic conditions on that line.
- Tree shown is for an && test, not an || test:

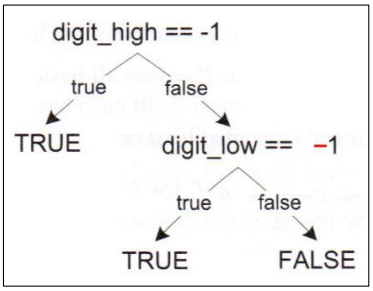


Branch and Condition Coverage

- A test suite satisfies branch and condition coverage criteria if it satisfies both branch coverage criteria and condition coverage criteria.
- A more complete extension is the compound condition adequacy criteria covered on the next slide.

Compound Condition Coverage

- Requires a test for all possible evaluations of compound conditions, e.g. reaching all leaves of logic trees such as the following, from line 27, if(DH == -1 || DL == -1)
- Note that short-circuiting may reduce the number of leaves that must be evaluated, but not always.



A && B && C && D && E requires only 6 test cases:

Test Case	A	B	C	D	E	Result
1	True	True	True	True	True	True
2	True	True	True	True	False	False
3	True	True	True	False		False
4	True	True	False			False
5	True	False				False
6	False					False

(((A||B) && C) || D) && E requires 13

Test Case	A	B	C	D	E	Result
1	True		True		True	True
2	False	True	True		True	True
3	True		False	True	True	True
4	False	True	False	True	True	True
5	False	False		True	True	True
6	True		True		False	False
7	False	True	True		False	False
8	True		False	True	False	False
9	False	True	False	True	False	False
10	False	False		True	False	False
11	True		False	False		False
12	False	True	False	False		False
13	False	False		False		False

MC/DC – Modified Condition/Decision Coverage

- Requires each basic condition to independently affect the result of each decision.
- I.e., for each basic condition, there must be 2 tests for which all other conditions are the same, and for which the result is true in one case and false in the other, as a direct result of the basic condition under evaluation.

Test Case	A	B	C	D	E	Result
1	True		True		True	True
6	True		True		False	False

- Can be shown to only require N + 1 tests

Now (((A||B) && C) || D) && E only requires 6

Test Case	A	B	C	D	E	Result
1	True	F	True	F	True	True
2	False	True	True	F	True	True
3	True	F	False	True	True	True
6	True	F	True	F	False	False
11	True	F	False	False	T	False
13	False	False	T	False	T	False

- Test case numbers match those of the previous slide.
- Boxes marked with T and F were blank (don't care) previously, but must now be given a specific value.

Path Testing

- Path Coverage
- Boundary Interior Coverage
- Loop Boundary Adequacy Criterion
- LCSAJ - Linear Code Sequence and Jump
- Cyclomatic Testing

Basic Path Coverage

- $C_{path} = \frac{\#\ of\ paths\ executed}{\#\ of\ paths}$
- For any program involving loops, the denominator is infinite, so C_{path} is always 0.

Boundary Interior Coverage

- Group paths that differ only by # of loop iterations.
- "Unfold" control flow graph at first repeating node to generate required sub-paths.

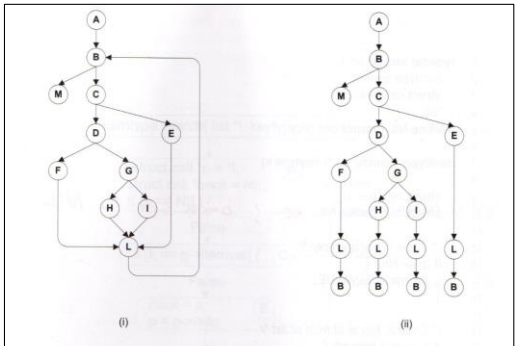


Figure 12.4: Deriving a tree from a control flow graph to derive subpaths for boundary/interior testing. Part (i) is the control flow graph of the C function cgi_decode, *identical to Figure 12.1 but showing only node identifiers without source code. Part (ii) is a tree derived from part (i) by following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

Sample Code with Problem When Line 29 is Executed On First Iteration

```
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 }
```

Missing in code

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.*

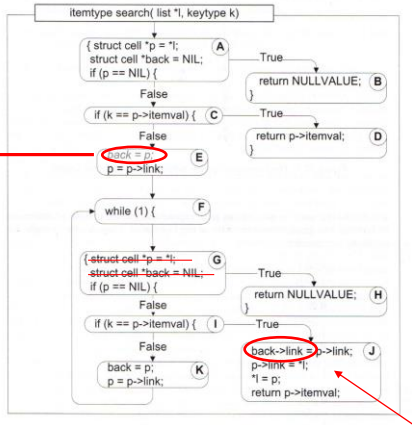


Figure 12.6: The control flow graph of C function search *with move-to-front feature.*

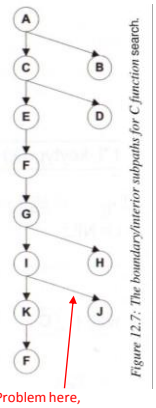


Figure 12.7: The boundary/interior subpaths for C function search.

Problem here, if loop exited on first pass.

Loop Boundary Adequacy

A test suite satisfies the loop boundary adequacy criteria iff for each loop present:

- 0: In at least one test, control reaches the loop and then the loop condition evaluates to false.
- 1: In at least one test, the loop is executed exactly one time.
- Many: In at least one test the loop is executed more than one time. (This can vary depending on circumstances. How many is “many”?)

LCSAJ – Linear Code Sequence and Jump

- An LCSAJ is a section of code through which execution can proceed sequentially, terminated by a jump.
- Specifications vary with the required length of sequential LCSAJs.
  - TER_{N+2} for N consecutive LCSAJs
  - TER_0 for statement testing
  - TER_1 for branch testing.

Cyclomatic Testing

- From graph theory, any connected graph with e edges and n nodes can be spanned by e-n+2 independent subpaths, where e-n+2 is the cyclomatic complexity of the graph.
- Any real path can be formed by the combination of independent subpaths.
- It is not required to test any particular set of independent subpaths, only that there be e-n+2 of them (as a goal.)

Procedure Call Testing

- For each procedure, there should be tests that exercise every entry point and every exit point.
  - This is normally covered by statement coverage, but should also be covered in context, i.e. when called by other procedures other than drivers.
- Every call to a procedure should be exercised, e.g. when a procedure is called from many places.
- The sequence of procedure calls, i.e. the "path" through the calling tree, may also be important.

Comparing Structural Testing Criteria

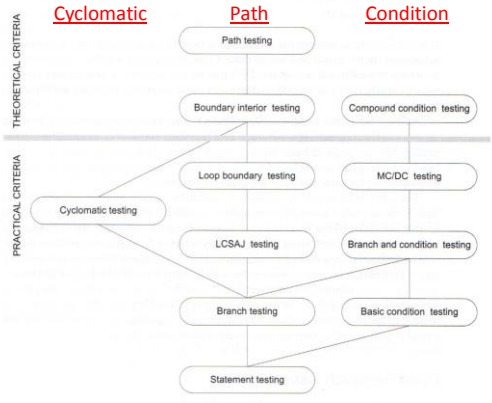


Figure 12.8: The subsumption relation among structural test adequacy criteria described in this chapter.

The Infeasibility Problem

- 100% Coverage is not always possible, or even desirable, in the face of redundant error checking and diminishing returns.
- One approach is to set a target less than 100%, but knowing exactly what is tricky.
- Another is to find and justify exceptions / exclusions, but that can be extremely difficult.