Subject – Software Testing
Structural Testing
Objective:

1. Understand Concept of structural testing
2. How structural (code-based or glass-box) testing complements functional (black-box) testing
3. Recognize and distinguish basic terms
   • Adequacy
   • Coverage
4. Recognize and distinguish characteristics of common structural criteria
5. Understand practical uses and limitations of structural testing

Definition:

Judging test suite thoroughness based on the structure of the program itself, it is also known as “white-box”, “glass-box”, or “code-based” testing. To distinguish from functional (requirements-based, “black-box” testing) “Structural” testing is still testing product functionality against its specification. Only the measure of thoroughness has changed. Structural testing, just like functional (spec-based) testing, still requires program specifications of some kind to judge whether a test execution was correct.

Why do we require structural testing?

One way of answering the question “What is missing in our test suite?”

If part of a program is not executed by any test case in the suite, faults in that part cannot be exposed but what’s a “part”?

Eg: Typically, a control flow element or combination: Statements (or CFG nodes), Branches (or CFG edges) Fragments and combinations: Conditions, paths

Complements functional testing: Another way to recognize cases that are treated differently

Recall fundamental rationale: Prefer test cases that are treated differently over cases treated the same

Eg: Sample test suites

**Test Suite**: Test Suite is a set of test cases it follows IEEE std.

\[ T_0 = \{ \text{“”,”test”,”test+case % 1DAdquacy”} \} \]

\[ T_1 = \{ \text{“adequate+test% 0DEExecution”} \} \]
Types of Structural (or Code Based) Testing

- Statement Testing
- Branch Testing
- Condition Testing
- Path Testing
- Cyclomatic complexity

**Statement Testing**

Adequacy criterion: each statement (or node in the CFG) must be executed at least once so that the faulty statement can be revealed.

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

\[
\text{Coverage} = \frac{\text{# executed statements}}{\text{# statements}}
\]

\( T \) satisfies statement adequacy criterion if \( c_{\text{stat}} = 1 \)

Rationale: a fault in a statement can only be revealed by executing the faulty statement

The statement coverage \( c_{\text{stat}} \) of \( T \) for program \( P \) is the fraction of statement of program \( P \) executed by at least one test case in \( T \)

Eg:

\[
\begin{align*}
T_0 & = \{"\", "test", "test+case\%1Dadequacy"\} & & 17/18 = 94\% \text{ Stmt Coverage.} \\
T_1 & = \{"adequate+test\%0Dexecution\%7U"\} & & 18/18 = 100\% \text{ Stmt Coverage.} \\
T_2 & = \{"%3D", "%A", "a+b", "test"\} & & 18/18 = 100\% \text{ Stmt Coverage.}
\end{align*}
\]

**Branch Testing**

Adequacy criterion: each branch (edge in the CFG) must be executed at least once. The branch coverage \( c_{\text{branch}} \) of \( T \) for \( P \) is the fraction of branches of program \( P \) executed by at least one test case in \( T \)
C_{\text{Branch}}: \frac{\text{# executed branches}}{\text{# branches}}

Eg:
T_3 = \{“”, “+%0D+%4\}”\} 100\% \text{ Stmt Cov. 88\% Branch Cov. (7/8 branches)}
T_2 = \{“%3D”, “%A”, “a+b”, “test”\} 100\% \text{ Stmt Cov. 100\% Branch Cov. (8/8 branches)}

\textbf{Conditional Testing}

Branch coverage exposes faults in how a computation has been decomposed into cases, intuitively attractive: check the programmer’s case analysis but only roughly groups cases with the same outcome.

Condition coverage considers case analysis in more detail also \textit{individual conditions} in a compound Boolean expression e.g., both parts of digit_high == 1 || digit_low == -1

Condition testing aims to exercise all logical conditions in a program module. It define:
- Relational expression: (E1 op E2), where E1 and E2 are arithmetic expressions.
- Simple condition: Boolean variable or relational expression, Possibly preceded by a NOT operator.
- Compound condition: composed of two or more simple Conditions boolean operators and parentheses.
- Boolean expression: Condition without relational expressions

Adequacy criterion: Each basic condition must be executed at least once each basic condition must have true and a false outcome at least once during the execution of the test suite

The basic condition coverage \(c_{\text{basic}}\) of \(T\) for \(P\) is the fraction of the total no of truth values assumed by the basic condition of program \(P\) during execution of all test cases in \(T\)

\[
\text{Coverage}_{\text{Basic-condn}}: \frac{\text{# truth values taken by all basic condn}}{2 \times \text{# basic conditions}}
\]

\(T\) satisfies the basic condition adequacy criterion if \(c_{\text{basic}} = 1\)

Basic condition adequacy criterion can be satisfied without satisfying branch coverage
Eg: \( T_4 = \{ \text{“first+test%9Ktest%K9”} \} \) satisfies basic condition adequacy does not satisfy branch condition adequacy since if statement \((\text{digit\_high}=1 \lor \text{digit\_low}=-1)\) statement always false. Branch and basic condition are not comparable (neither implies the other).

Unlike all branches & statement covering all basic condition does not replace covering all branches.

Covering branches and condition: Branch and condition adequacy: cover all conditions and all decisions. Compound condition adequacy: Is also known as multiple condition coverage. Cover all possible evaluations of compound conditions, Cover all branches of a decision tree.

\[
\begin{align*}
digit\_high &= -1 \\
\text{true} & \quad \text{false} \\
\text{false} & \quad \text{true}
\end{align*}
\]

\[
\begin{align*}
digit\_low &= 1 \\
\text{false} & \quad \text{true}
\end{align*}
\]

FALSE

Compound conditions: Exponential complexity

**Eg., for Boolean expression** \( (((a \lor b) \&\& c) \lor d) \&\& e \) compound condition coverage requires

<table>
<thead>
<tr>
<th>Test</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>T</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>(2)</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>(3)</td>
<td>T</td>
<td>-</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>(4)</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>(5)</td>
<td>F</td>
<td>F</td>
<td>-</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>(6)</td>
<td>T</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>F</td>
</tr>
<tr>
<td>(7)</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>-</td>
<td>F</td>
</tr>
<tr>
<td>(8)</td>
<td>T</td>
<td>-</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>(9)</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>(10)</td>
<td>F</td>
<td>F</td>
<td>-</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>(11)</td>
<td>T</td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>(12)</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>(13)</td>
<td>F</td>
<td>F</td>
<td>-</td>
<td>F</td>
<td>-</td>
</tr>
</tbody>
</table>
Modified condition/decision (MC/DC)

Motivation: Effectively test important combinations of conditions, without exponential blowup in test suite size. “Important” combinations mean: Each basic condition shown to independently affect the outcome of each decision.

For each basic condition C, two test cases, values of all evaluated conditions except C are the same compound condition as a whole evaluates to true for one and false for the other.

MC/DC can be satisfied with N+1 test cases for N basic conditions.

\begin{align*}
(((a \lor b) \&\& c) \lor d) \&\& e
\end{align*}

<table>
<thead>
<tr>
<th>Test Case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>true</td>
<td>--</td>
<td>true</td>
<td>--</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>(2)</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>--</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>(3)</td>
<td>true</td>
<td>--</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>(6)</td>
<td>true</td>
<td>--</td>
<td>true</td>
<td>--</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>(11)</td>
<td>true</td>
<td>--</td>
<td>false</td>
<td>false</td>
<td>--</td>
<td>false</td>
</tr>
<tr>
<td>(13)</td>
<td>false</td>
<td>false</td>
<td>--</td>
<td>false</td>
<td>--</td>
<td>false</td>
</tr>
</tbody>
</table>

Underlined values independently affect the output of the decision. Required by the RTCA/DO-178B standard.

Path Testing

Path adequacy: A test suite T for 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.

Adequacy criterion: each path must be executed at least once, so it helps in revealing failures that occurs when loop executed several times.

The path coverage $c_{\text{path}}$ of T for P is the fraction of paths of program P executed by at least one test case in T.

Coverage_{path}: \# executed paths
To ensure finite number of paths we must limit the no of times of execution of each loop eg: 10 iterations, 50 iterations etc..

Path testing Loop Eg:

Boundary interior path testing: A program may contain both looping and non-looping statement, but coverage of non-looping paths is expensive therefore we consider variant of the boundary / interior criterion.

Group together paths that differ only in the sub path they follow when repeating the body of a loop. Follow 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.
Deriving a tree from a CFG to derive sub paths for boundary/interior testing

i. Only node representation

ii. Derived from i., the path from root of tree to leaf is repeated.

Limitations of boundary interior adequacy

```c
if (a) {
    S1;
} else if (b) {
    S2;
} else 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
- Choosing input data to force execution of one particular path may be very difficult, or even impossible if the conditions are not independent

LCSAJ adequacy

There are additional path oriented coverage criteria that do not explicitly consider loops. Among these are criteria that consider path up to fixed length. The most common criteria is LCSAJ

LCSAJ defined as a body of code through which the flow of control may proceed sequentially and terminated by a jump in the control flow.

Linear Code Sequence And Jumps: sequential sub path in the CFG starting and ending in a branch. The criteria $\text{TER}_1 = \text{statement coverage}$

$\text{TER}_2 = \text{branch coverage}$

$\text{TER}_{n+2} = \text{coverage of } n \text{ consecutive LCSAJ}s$
Eg: 1 begin

NOTE: Each LCSAJ begins at a statement and end

2 int x,y,p;
3 input (x,y);
4 if(x<0) LCSAJ
5 p=g(y);
6 else
7 p=g(y*y);
8 end

LCSAJ StartLn EndLn Jump to
1 1 6 exit
2 1 4 7
3 7 8 exit

Cyclomatic adequacy

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 for an arbitrary graph

e - n + 2 which is called the cyclomatic complexity of CFG

Cyclomatic testing consists of attempting to exercise any set of execution paths that is basis set of CFG. Cyclomatic number: number of independent paths in the CFG

A path is representable as a bit vector, where each component of the vector represents an edge
Dependence” is ordinary linear dependence between (bit) vectors

Cyclomatic coverage counts the number of independent paths that have been exercised, relative to cyclomatic complexity. Cyclomatic Complexity calculations helps the developer / tester to decide whether the module under test is overly complex or well written.

Recommended limit value of Cyclomatic Complexity is 10.

>10
• Structure of module is overly complex
>5 and <10

• Structure of module is complex indicating logic is difficult to test
<5

• structure of the module is simple and logic is easy to test.

**Towards procedure call testing**

The criteria considered to this point measure coverage of control flow within individual procedures. not well suited to integration or system testing

Choose a coverage granularity commensurate with the granularity of testing

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

**Procedure entry and exit testing**

procedure may have multiple entry points (e.g., Fortran) and multiple exit points

Call coverage

The same entry point may be called from many points

**Subsumes Relation**

Test coverage criterion A subsumes test coverage criterion B iff, for every program p, every test satisfying A with respect to P also satisfies B with respect P.

The power and the cost of the structural test adequacy criteria can be formally compared using the subsumes relation

They are divided in to 2 broad categories

1. practical criteria can be always satisfied by test set whose size is linear function of program size.

2. theoretical criteria may require large no of test cases or infinite no of test cases.
Infeasibility Problem

No set of test cases available that are capable of satisfying test coverage criteria

Unreachable statements and code reuse

Infeasibility because of stronger coverage criteria that demands coverage of more software components

Boundary /interior coverage criteria in path based structural testing cannot be fulfilled

No execution of sub path even when branch returns true or false