**Structural Testing**

Software Engineering
Andreas Zeller • Saarland University

---

**Testing Tactics**

- **Functional**
  - “black box”
  - Tests based on spec
  - Test covers as much specified behavior as possible

- **Structural**
  - “white box”
  - Tests based on code
  - Test covers as much implemented behavior as possible

---

**Why Structural?**

- If a part of the program is never executed, a defect may loom in that part.
  - A “part” can be a statement, function, transition, condition…
- Attractive because automated

---

From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 14
And Pezze + Young, “Software Testing and Analysis”, Chapters 12-13

In contrast to functional tests (discussed the last time), structural tests are based on the code structure.

Structural tests are automated – and can be much more fine-grained than functional tests.
Why Structural?

- Complements functional tests
  Run functional tests first, then measure what is missing
- Can cover low-level details missed in high-level specification

A Challenge

class Roots {
    // Solve ax² + bx + c = 0
    public roots(double a, double b, double c)
    { ... }

    // Result: values for x
    double root_one, root_two;
}

- Which values for a, b, c should we test?

  assuming a, b, c, were 32-bit integers, we’d have (2³²)³ ≈ 10²⁸ legal inputs
  with 1.000.000.000.000 tests/s, we would still require 2.5 billion years

The Code

// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{  
    double a = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        // code for handling two roots
    }
    else if (q == 0) {
        // code for handling one root
    }
    else {
        // code for handling no roots
    }
}

Typically, both techniques are used.

Recall this example from last lecture.

If we know the code (“white box”) and thus the structure, we can design test cases accordingly

Test this case

and this

and this!
The Test Cases

// Solve \( ax^2 + bx + c = 0 \)
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        // code for handling two roots
    }
    else if (q == 0) {
        // code for handling one root
    }
    else {
        // code for handling no roots
    }
}

(a, b, c) = (3, 4, 1)
(a, b, c) = (0, 0, 1)
(a, b, c) = (3, 2, 1)

A Defect

// Solve \( ax^2 + bx + c = 0 \)
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        // code for handling two roots
    }
    else if (q == 0) {
        // code must handle a = 0
        x = (-b) / (2 * a);
    }
    else {
        // code for handling no roots
    }
}

(a, b, c) = (0, 0, 1)

Expressing Structure

// Solve \( ax^2 + bx + c = 0 \)
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a != 0) {
        // code for handling two roots
    }
    else if (q == 0) {
        x = (-b) / (2 * a);
    }
    else {
        // code for handling no roots
    }

What is relevant in her is the program structure – the failure occurs only if a specific condition is true and a specific branch is taken.

Finding appropriate input values is a challenge in itself which may require external theory – but in this case, the external theory is just maths.
Control Flow Graph

- A control flow graph expresses paths of program execution.
- Nodes are basic blocks – sequences of statements with one entry and one exit point.
- Edges represent control flow – the possibility that the program execution proceeds from the end of one basic block to the beginning of another.

Structural Testing

- The CFG can serve as an adequacy criterion for test cases.
- The more parts are covered (executed), the higher the chance of a test to uncover a defect.
- “parts” can be: nodes, edges, paths, conditions…

Control Flow Patterns

Every part of the program induces its own patterns in the CFG.
The function `cgi_decode` translates a CGI-encoded string (i.e., from a Web form) to a plain ASCII string, reversing the encoding applied by the common gateway interface (CGI) on common Web servers.

(from Pezze + Young, “Software Testing and Analysis”, Chapter 12)
While the program is executed, one statement (or basic block) after the other is covered — i.e., executed at least once — but not all of them. Here, the input is “test”: checkmarks indicate executed blocks.

We’d like to test every statement, so we come up with more test cases.

We’d like to test every statement, so we come up with more test cases.
Test Adequacy Criteria

- How do we know a test suite is “good enough”?
- A test adequacy criterion is a predicate that is true or false for a pair \( \langle \text{program, test suite} \rangle \)
- Usually expressed in form of a rule – e.g., “all statements must be covered”

Statement Testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once
- Rationale: a defect in a statement can only be revealed by executing the defect
- Coverage: \( \frac{\# \text{executed statements}}{\# \text{statements}} \)
The initial coverage is 7/11 blocks = 63%. We could also count the statements instead (here: 14/20 = 70%), but conceptually, this makes no difference.

and the coverage increases with each additionally executed statement...
Computing Coverage

- Coverage is computed automatically while the program executes
- Requires instrumentation at compile time
  With GCC, for instance, use options `-ftest-coverage -fprofile-arcs`
- After execution, coverage tool assesses and summarizes results
  With GCC, use `gcov source-file` to obtain readable .gcov file

For Java, use jcoverage or like tools.

This is the output of the GCOV coverage tool for `cgi_decode`. Each statement (each line) is annotated with the number of executions so far. Zero executions is suspicious and would be marked by “#####”; the tag “–” stands for lines without executable code.
Statement testing is a simple criterion for assessing the adequacy of a test suite – but there are many more such criteria.

As an example, consider branch testing, which is a criterion that subsumes statement testing. In other words, if the branch testing criterion is satisfied by a pair \(\langle\text{program, test suite}\rangle\), so is the statement testing criterion for the same pair.
Why is branch testing useful? Assume block F were missing (= a defect). Then, we could achieve 100% statement coverage without ever triggering the defect.

If we focus on whether branches have been taken, though, we get a different picture.

Here, we’d find that the test case executes only 7 out of 8 branches, or 87%.
Branch Testing

- Adequacy criterion: each branch in the CFG must be executed at least once
- Coverage: \( \frac{\text{# executed branches}}{\text{# branches}} \)
- Subsumes statement testing criterion because traversing all edges implies traversing all nodes
- Most widely used criterion in industry

Condition Testing

- Consider the defect
  \( \text{(digit_high == 1 || digit_low == -1)} \)
  // should be -1
- Branch adequacy criterion can be achieved by changing only `digit_low`
  i.e., the defective sub-expression may never determine the result
- Faulty sub-condition is never tested although we tested both outcomes of the branch

With another test case, we can cover this remaining branch – and find the defect.

(from Pezze + Young, "Software Testing and Analysis", Chapter 12)
Condition Testing

- Key idea: also cover *individual conditions* in compound boolean expression
e.g., both parts of `digit_high == 1 || digit_low == -1`

Condition Testing

- Adequacy criterion: each basic condition must be evaluated at least once
- Coverage:
  \[
  \text{# truth values taken by all basic conditions} = 2 \times \text{# basic conditions}
  \]
- Example: "%test+%9k%k9"
  100% basic condition coverage
  but only 87% branch coverage

The basic condition criterion is not comparable with branch or statement coverage criteria – neither implies (subsumes) the other.
Compound Conditions

- Assume \(((a \lor b) \land c) \lor d \land e\)
- We need 13 tests to cover all possible combinations
- In general case, we get a combinatorial explosion

<table>
<thead>
<tr>
<th>Test Case</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>True</td>
<td>–</td>
<td>True</td>
<td>–</td>
<td>True</td>
</tr>
<tr>
<td>(2)</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>(3)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>(4)</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>(5)</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>(6)</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>(7)</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>(8)</td>
<td>True</td>
<td>–</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>(9)</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>(10)</td>
<td>True</td>
<td>True</td>
<td>–</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>(11)</td>
<td>True</td>
<td>True</td>
<td>–</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>(12)</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>(13)</td>
<td>False</td>
<td>False</td>
<td>–</td>
<td>False</td>
<td>–</td>
</tr>
</tbody>
</table>
The combinatorial explosion is the reason why compound condition testing is a theoretical, rather than a practical criterion.

A possible compromise is MCDC or Modified Condition/Decision Coverage testing.

**MCDC Testing**
Modified Condition Decision Coverage

- Key idea: Test important combinations of conditions, avoiding exponential blowup

- A combination is “important” if each basic condition is shown to independently affect the outcome of each decision
MCDC Testing
Modified Condition Decision Coverage

- For each basic condition \( C \), we need two test cases \( T_1 \) and \( T_2 \)
- Values of all evaluated conditions except \( C \) are the same
- Compound condition as a whole evaluates to \( True \) for \( T_1 \) and \( false \) for \( T_2 \)
- A good balance of thoroughness and test size (and therefore widely used)

### MCDC Testing
Modified Condition Decision Coverage

- Assume \( (((a \lor b) \land c) \lor d) \land e) \)
- We need six tests to cover MCDC combinations

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>Decision</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>–</td>
<td>False</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 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.

Path Testing
beyond individual branches

- Key idea: explore sequences of branches in control flow
- Many more paths than branches calls for compromises
For one thing, there is general path testing, i.e. covering all paths in the program. Since loops are unbounded, this is generally not feasible and therefore just a theoretical criterion. Its advantage, though, is that it subsumes almost all criteria.

Boundary interior testing groups together paths that differ only in the subpath they follow when repeating the body of a loop. In other words, we follow each path in the CFG up to the first repeated node.
Boundary Interior Adequacy
for cgi_decode

Original CFG

Paths to be covered

Issues

- The number of paths may still grow exponentially
  In this example, there are $2^4 = 16$ paths
- Forcing paths may be infeasible
  or even impossible if conditions are not independent

Test Criteria

Therefore, boundary interior testing belongs more to the “theoretical” criteria.
Test Criteria

Practical Criteria

Loop boundary testing

Boundary interior testing

Compound condition testing

MCDC testing

Branch and condition testing

LCSAJ testing

Basic condition testing

Statement testing

Branch testing

Basic condition testing

Boundary interior testing

Branch testing

Compound condition testing

LCSAJ testing

Path testing

Statement testing

Boundary interior testing

Basic condition testing

Theory Criteria

Subsumes

Path testing

Another alternative is loop boundary testing which forces constraints on how loops are to be executed. This is a practical criterion.

Typically combined with other adequacy criteria such as MCDC

Loop Boundary Adequacy

A test suite satisfies the loop boundary adequacy criterion if for every loop $L$:

- There is a test case which iterates $L$ zero times
- There is a test case which iterates $L$ once
- There is a test case which iterates $L$ more than once

This is a variant of the boundary/interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths

With these three test cases, we obtain loop boundary adequacy for the cgi_decode main loop.
LCSAJ Adequacy

Testing all paths up to a fixed length

- **LCSAJ = Linear Code Sequence And Jump**
- **A LCSAJ is a sequential subpath in the CFG starting and ending in a branch**

<table>
<thead>
<tr>
<th>LCSAJ length</th>
<th>corresponds to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>statement coverage</td>
</tr>
<tr>
<td>2</td>
<td>branch coverage</td>
</tr>
<tr>
<td>(n)</td>
<td>coverage of (n) consecutive LCSAJs</td>
</tr>
<tr>
<td>(\infty)</td>
<td>path coverage</td>
</tr>
</tbody>
</table>

Another alternative is loop boundary testing which forces constraints on how loops are to be executed.

LCSAJ is a generalization over branch and statement coverage.

Considering the exponential blowup in sequences of conditional statements (even when not in loops), we might choose to consider only sub-sequences of a given length. This is what LCSAJ gives us --- essentially considering full path coverage of (short) sequences of decisions.
Test Criteria

- Path testing
- Boundary interior testing
- Compound condition testing
- MCDC testing
- Branch and condition testing
- LCSAJ testing
- Branch testing
- Statement testing
- Basic condition testing
- Loop boundary testing

The adequacy of a coverage criterion can only be intuitively defined.

Established by a number of studies done by E. Weyuker at AT&T. “Any explicit relationship between coverage and error detection would mean that we have a fixed distribution of errors over all statements and paths, which is clearly not the case”.

Satisfying Criteria

Sometimes criteria may not be satisfiable:

- **Statements** may not be executed because of defensive programming or code reuse
- **Conditions** may not be satisfiable because of interdependent conditions
- **Paths** may not be executable because of interdependent decisions
Satisfying Criteria

- Reaching specific code can be very hard!
- Even the best-designed, best-maintained systems may contain unreachable code
- A large amount of unreachable code/paths/conditions is a serious maintainability problem
- Solutions: allow coverage less than 100%, or require justification for exceptions

More Testing Criteria

- Object-oriented testing
e.g., “Every transition in the object’s FSM must be covered”
e.g., “Every method pair in the object’s FSM must be covered”
- Interclass testing
e.g., “Every interaction between two objects must be covered”
- Data flow testing
e.g., “Every definition-use pair of a variable must be covered”

Data flow testing is based on the observation that computing the wrong value leads to a failure only when that value is subsequently used.
A typical data flow testing criterion is therefore that the tests must exercise every pair (definition, uses) of a variable (such as “ok” in this example).
Summary