Structural Testing

Software Engineering
Andreas Zeller • Saarland University
Testing Tactics

- Tests based on spec
- Test covers as much specified behavior as possible

- 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
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 \( ax^2 + bx + c = 0 \)
    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 \( (2^{32})^3 \approx 10^{28} \) legal inputs
  with \( 1.000.000.000.000.000 \) tests/s, we would still require 2.5 billion years
The Code

// 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
    }
}

$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

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 \neq 0) {
        // code for handling two roots
    }

    else if (q == 0) {
        // code for handling one root
    }

    else {
        // code for handling no roots
    }
}

\[
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

(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) {
        x = (-b) / (2 * a);
    }
    else {
        // code for handling no roots
    }
}

\[
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

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

code must handle $a = 0$
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
    }
}
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.
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

- while (COND)
  - BODY
- do
  - while (COND)
  - BODY
- for
  - INIT
  - COND
  - BODY
  - INCR

if (COND)
- THEN-BLOCK
- ELSE-BLOCK

if (COND)
- THEN-BLOCK;
else
- ELSE-BLOCK;

do {
  BODY
} while (COND);
cgi_decode

/**
 * @title cgi_decode
 * @desc
 * Translate a string from the CGI encoding to plain ascii text
 * '+' becomes space, %xx becomes byte with hex value xx,
 * other alphanumeric characters map to themselves
 *
 * returns 0 for success, positive for erroneous input
 * 1 = bad hexadecimal digit
 */

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
while (*eptr) /* loop to end of string (‘\0’ character) */
{
    char c;
    c = *eptr;
    if (c == '+') { /* ‘+’ maps to blank */
        *dptr = ' ';
    } else if (c == '%') { /* '%xx' is hex for char xx */
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1)
            ok = 1; /* Bad return code */
        else
            *dptr = 16 * digit_high + digit_low;
    } else { /* All other characters map to themselves */
        *dptr = *eptr;
    }
    ++dptr; ++eptr;
}
*dptr = '\0'; /* Null terminator for string */
return ok;
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        } else if (c == '%') {
            *dptr = *eptr;
        } else {
            *dptr = *eptr;
        }
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1) {
            ok = 1;
        } else {
            *dptr = 16 * digit_high + digit_low;
        }
        ++dptr;
        ++eptr;
    }
    *dptr = "0";
    return ok;
}

Figure 12.2: The control flow graph of function cgi_decode from Figure 12.1
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    char c;
    c = *eptr;
    if (c == '+') {
        *dptr = ' ';
    }
    while (*eptr) {
        True
        *dptr = '\0';
        return ok;
    }
    False
    else if (c == '%') {
        *dptr = *eptr;
    } else {
        *dptr = *eptr;
        int digit_high = Hex_Values[(++eptr)];
        int digit_low = Hex_Values[(++eptr)];
        if (digit_high == -1 || digit_low == -1) {
            True
            ok = 1;
            False
            else{
                *dptr = 16 * digit_high + digit_low;
            }
        }
    }
    ++dptr;
    ++eptr;
    return ok;
}
```
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        } else if (c == '%') {
            *dptr = *eptr;
        } else {
            *dptr = *eptr;
        }

        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1) {
            ok = 1;
        } else {
            *dptr = 16 * digit_high + digit_low;
        }

        ++dptr;
        ++eptr;
    }

    *dptr = "0";
    return ok;
}
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    char c;
    c = *eptr;
    if (c == '+') {
        *dptr = ' ';
    }
    while (*eptr) {
        True
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        }
        elseif (c == '%') {
            *dptr = *eptr;
        }
        else *
            *dptr = *eptr;
    }
    int digit_high = Hex_Values[*(++eptr)];
    int digit_low = Hex_Values[*(++eptr)];
    if (digit_high == -1 || digit_low == -1) {
        ok = 1;
    }
    else {
        *dptr = 16 * digit_high + digit_low;
        ok = 1;
    }
    return ok;
}
```
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        } else if (c == '%') {
            *dptr = *eptr;
        } else {
            *dptr = *eptr;
        }
        ++dptr;
        ++eptr;
    }

    *dptr = "0";
    return ok;
}
```
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 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:  \[
\frac{\# \text{executed statements}}{\# \text{statements}}
\]
int cgi_decode(char *encoded, char *decoded) {
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    char c;
    c = *eptr;
    if (c == '+') {
        *dptr = ' ';
    }
    while (*eptr) {
        True
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        }
        elseif (c == '%') {
            *dptr = *eptr;
        }
        else {
            *dptr = *eptr;
        }
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1) {
            True
            ok = 1;
        }
        else {
            *dptr = 16 * digit_high + digit_low;
        }
        ++dptr;
        ++eptr;
    }
    True
    *dptr = "0";
    return ok;
}
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        char c = *eptr;
        if (c == '+') {
            *dptr = ' ';  // True
            ++dptr;
            ++eptr;
        } elseif (c == '%') {
            else
                *dptr = *eptr;
                ++dptr;
            }
        } else {
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1) {
                ok = 1;
            } else {
                *dptr = 16 * digit_high + digit_low;
            }
        }
    }

    *dptr = "0";
    return ok;
}
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    char c;
    c = *eptr;
    if (c == '+') {
        *dptr = ' ';
    }
    while (*eptr) {
        if (c == ' ') {
            *dptr = ' ';
            ++dptr;
            ++eptr;
            continue;
        } else if (c == '%') {
            *dptr = *eptr;
        } else {
            *dptr = *eptr;
        }
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1) {
            ok = 1;
        } else {
            *dptr = 16 * digit_high + digit_low;
            ok = 1;
        }
    }
    *dptr = '0';
    return ok;
}
```
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';  // False
        } else if (c == '%') {
            *dptr = *eptr;  // False
        } else {
            *dptr = *eptr;  // False
            int digit_high = Hex_Values[*++eptr];
            int digit_low = Hex_Values[*++eptr];
            if (digit_high == -1 || digit_low == -1) {
                ok = 1;  // True
            } else {
                *dptr = 16 * digit_high + digit_low;  // True
            }
        }
        ++dptr;
        ++eptr;
    }

    *dptr = "0";
    return ok;
}
```
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
int ok = 0;

while (*eptr) /* loop to end of string (',0' character) */
{
    char c;
    c = *eptr;
    if (c == '+') { /* '+' maps to blank */
        *dptr = ' ';
    } else if (c == '%') { /* '%xx' is hex for char xx */
        int digit_high = Hex_Values[*+(++eptr)];
        int digit_low  = Hex_Values[*+(++eptr)];
        if (digit_high == -1 || digit_low == -1)
            ok = 1; /* Bad return code */
    } else {
        *dptr = 16 * digit_high + digit_low;
    } else { /* All other characters map to themselves */
        *dptr = *eptr;
    }
    ++dptr; ++eptr;
}

*dptr = '\0'; /* Null terminator for string */
return ok;
Demo
Test Criteria

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

Theoretical Criteria

- Practical Criteria
Test Criteria

1. Statement testing
2. Branch testing
   - Boundary interior testing
   - LCSAJ testing
   - Path testing
3. Compound condition testing
4. MCDC testing
5. Basic condition testing
6. Branch and condition testing
7. Loop boundary testing
8. Boundary interior testing

Practical Criteria

Theoretical Criteria

subsumes
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' '; 
            ++dptr; 
            ++eptr;
        }
        else if (c == '%') {
            *dptr = *eptr;

        } else {
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1) {
                ok = 1;
            } else {
                *dptr = 16 * digit_high + digit_low;
            }
            ++dptr;
            ++eptr;
        }
    }
    *dptr = "0";
    return ok;
}
```

```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
    char c;
    while (*eptr) {
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        } else if (c == '%') {
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1) {
                ok = 1;
            } else {
                *dptr = 16 * digit_high + digit_low;
            }
        } else {
            *dptr = *eptr;
        }
        ++dptr;
        ++eptr;
    }
    *dptr = "0";
    return ok;
}
```
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    char c;
    c = *eptr;
    if (c == '+') {
        *dptr = ' ';
        ++dptr;
        ++eptr;
    }
    while (*eptr) {
        True
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
            ++dptr;
            ++eptr;
        } else if (c == '%') {
            *dptr = *eptr;
            ++eptr;
        } else {
            *dptr = 16 * digit_high + digit_low;
            ++dptr;
        }
        ok = 1;
    }
    return ok;
}
```

Figure 12.2: The control flow graph of function cgi_decode from Figure 12.1
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
            eptr++;
        } else if (c == '%') {
            *dptr = *eptr;
            ++eptr;
            ++dptr;
        } else {
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1) {
                ok = 1;
            } else {
                *dptr = 16 * digit_high + digit_low;
            }
        }
    }
    *dptr = '0';
    return ok;
}
```
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
  
  \[
  (\text{digit\_high} == 1 \ || \ \text{digit\_low} == -1)
  \]
  
  // should be -1

• Branch adequacy criterion can be achieved by changing only \text{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
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
Test Criteria

- Path testing
- Boundary interior testing
- LCSAJ testing
- Branch testing
- Statement testing
- Loop boundary testing

- Compound condition testing
- MCDC testing
- Branch and condition testing
- Basic condition testing

Theoretical Criteria

- Practical Criteria

subsumes
Test Criteria

Path testing

Boundary interior testing

LCSAJ testing

Branch testing

Statement testing

Basic condition testing

Compound condition testing

MCDC testing

Branch and condition testing

Loop boundary testing

Practical Criteria

Theoretical Criteria

subsumes
• 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>False</td>
<td>–</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>–</td>
<td>True</td>
<td>–</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>–</td>
<td>True</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>False</td>
<td>–</td>
<td>–</td>
<td>True</td>
</tr>
<tr>
<td>6</td>
<td>True</td>
<td>–</td>
<td>True</td>
<td>–</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>8</td>
<td>True</td>
<td>–</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>9</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>10</td>
<td>False</td>
<td>False</td>
<td>–</td>
<td>–</td>
<td>True</td>
</tr>
<tr>
<td>11</td>
<td>True</td>
<td>–</td>
<td>True</td>
<td>False</td>
<td>False</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>
Test Criteria

Practical Criteria

- Loop boundary testing
- Branch testing
- Statement testing

Theoretical Criteria

- Boundary interior testing
- LCSAJ testing
- Branch and condition testing
- Compound condition testing
- MCDC testing
- Basic condition testing

subsumes
Test Criteria

Path testing

Boundary interior testing

LCSAJ testing

Branch testing

Statement testing

Loop boundary testing

Compound condition testing

MCDC testing

Branch and condition testing

Basic condition testing

subsumes

Theoretical Criteria

Practical Criteria
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>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>
Path Testing
beyond individual branches

• Key idea: explore sequences of branches in control flow

• Many more paths than branches calls for compromises
Test Criteria

Practical Criteria

- Loop boundary testing
- Branch testing
- Statement testing
- LCSAJ testing
- Boundary interior testing
- Path testing

Theoretical Criteria

- Basic condition testing
- Branch and condition testing
- MCDC testing
- Compound condition testing

subsumes
Test Criteria

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

Practical Criteria

- Loop boundary testing

Theoretical Criteria

- subsumes
Test Criteria

- Path testing
  - Boundary interior testing
  - Branch testing
  - Statement testing
  - Loop boundary testing

- Compound condition testing
  - MCDC testing
  - Branch and condition testing
  - Basic condition testing

Theoretical Criteria

- LCSAJ testing

Practical Criteria

- subsumes
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

- Path testing
  - Boundary interior testing
  - Branch testing
  - Statement testing

- Compound condition testing
  - MCDC testing
  - Branch and condition testing
  - Basic condition testing

Practical Criteria

Theoretical Criteria

Subsumes
Test Criteria

Path testing

Boundary interior testing

Compound condition testing

MCDC testing

Branch and condition testing

Basic condition testing

Loop boundary testing

Statement testing

LCSAJ testing

Theoretical Criteria

Practical Criteria

subsumes
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*

Typically combined with other adequacy criteria such as MCDC
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        } else if (c == '%') {
            *dptr = *eptr;
        } else {
            *dptr = *eptr;
        }
        ++dptr;
        ++eptr;
    }

    while (*eptr) {
        char c;
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
        } else if (c == '%') {
            *dptr = *eptr;
        } else {
            *dptr = *eptr;
        }
        ++dptr;
        ++eptr;
    }

    *dptr = '0';
    return ok;
}
Test Criteria

Practical Criteria
- Path testing
  - Boundary interior testing
    - Loop boundary testing
  - Branch testing
    - Statement testing

Theoretical Criteria
- Compound condition testing
  - MCDC testing
  - Branch and condition testing
  - Basic condition testing
  - LCSAJ testing
Test Criteria

- Path testing
  - Boundary interior testing
    - LCSAJ testing
      - Branch testing
        - Loop boundary testing
      - Statement testing
    - MCDC testing
      - Compound condition testing
        - Basic condition testing
      - Branch and condition testing
        - Practical Criteria
          - Theoretical Criteria
            - subsumes
**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>
Test Criteria

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

Theoretical Criteria

- Practical Criteria

subsumes
Weyuker’s Hypothesis

The adequacy of a coverage criterion can only be intuitively defined.
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/pths/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” or “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”
```c
int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;

    while (*eptr) {
        c = *eptr;
        if (c == '+') {
            *dptr = ' ';
            ++dptr;
            ++eptr;
        } else if (c == '%') {
            else *dptr = *eptr;
        } else {
            *dptr = *eptr;
            ++dptr;
            ++eptr;
        }
        if (c == '+') {
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1) {
                ok = 1;
            } else {
                *dptr = 16 * digit_high + digit_low;
            }
        } else {
            *dptr = 16 * digit_high + digit_low;
        }
    }

    *dptr = *epr;
    return ok;
}
```
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

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

Summary

Test Criteria

- Path testing
- Boundary and condition testing
- MC/DC testing
- LCSAJ testing
- Branch testing
- Statement testing
- Loop boundary testing
- Subsumes