Dependable Software Systems

Topics in Control-Flow Testing

Material drawn from [Beizer, Mancoridis]
Control-Flow Testing

- Control-flow testing is a structural testing strategy that uses the program’s control flow as a model.
- Control-flow testing techniques are based on judiciously selecting a set of test paths through the program.
- The set of paths chosen is used to achieve a certain measure of testing thoroughness.
- E.g., pick enough paths to assure that every source statement is executed at least once.
Motivation

• Control-flow testing is most applicable to new software for unit testing.

• Control-flow testing assumptions:
  – specifications are correct
  – data is defined and accessed properly
  – there are no bugs other than those that affect control flow

• Structured and OO languages reduce the number of control-flow bugs.
Control Flowgraphs

- The control flowgraph is a graphical representation of a program’s control structure.
Flowgraphs Consist of Three Primitives

- A **decision** is a program point at which the control can diverge.
  - (e.g., if and case statements).
- A **junction** is a program point where the control flow can merge.
  - (e.g., end if, end loop, goto label)
- A **process block** is a sequence of program statements uninterrupted by either decisions or junctions. (*i.e.*, straight-line code).
  - A process has one entry and one exit.
  - A program does not jump into or out of a process.
Example of a Flowgraph

1. INPUT X,Y
   Z:=X+Y
   V:=X-Y
2. IF Z>=0 GOTO SAM
3. JOE: Z:=Z+V
4. SAM: Z:=Z+V
5. U:=0
6. LOOP
   B(U),Q(V):=(Z+V)*U
7. IF B(U)=0 GOTO JOE
   Z:=Z-1
8. IF Z=0 GOTO ELL
   U:=U+1
9. UNTIL U=Z
   B(U-1):=B(U+1)+Q(V-1)
10. ELL: B(U+Q(V)):=U+V
11. IF U=V GOTO JOE
12. IF U>V THEN U := Z
13. YY: Z:=U
2. END

Dependable Software Systems (Control-Flow Testing) © SERG
Exponentiation Algorithm

1. scanf("%d %d", &x, &y);
2. if (y < 0)
   pow = -y;
else
   pow = y;
3. z = 1.0;
4. while (pow != 0) {
   z = z * x;
   pow = pow - 1;
}
5. if (y < 0)
   z = 1.0 / z;
7. printf("%f", z);
Bubble Sort Algorithm

1 for (j=1; j<N; j++) {
    last = N - j + 1;
2 for (k=1; k<last; k++) {
3     if (list[k] > list[k+1]) {
            temp = list[k];
            list[k] = list[k+1];
            list[k+1] = temp;
        }
5    }
6 }
7 print(“Done
”);
Paths

- A path through a program is a sequence of statements that starts at an entry, junction, or decision and ends at another (possible the same), junction, decision, or exit.
- A path may go through several junctions, processes, or decisions, one or more times.
- Paths consist of segments.
- The smallest segment is a link. A link is a single process that lies between 2 nodes.
Paths (Cont’d)

- The **length** of a path is the number of links in a path.
- An **entry/exit path** or a **complete path** is a path that starts at a routine’s entry and ends at the same routine’s exit.
Paths (Cont’d)

• Complete paths are useful for testing because:
  – It is difficult to set up and execute paths that start at an arbitrary statement.
  – It is difficult to stop at an arbitrary statement without changing the code being tested.
  – We think of routines as input/output paths.
Path Selection Criteria

- There are many paths between the entry and exit points of a typical routine.
- Even a small routine can have a large number of paths.
How do we define “complete” testing?

• 1) Exercise every path from entry to exit.
• 2) Exercise every statement at least once.
• 3) Exercise every branch (in each direction) at least once.

• Clearly, 1 implies 2 and 3
• However, 1 is impractical for most routines.
• Also, 2 is not equal to 3 in languages with goto statements.
Demonstration that 2 does not imply 3

- **2.Statement Coverage:** For $x < 0$ the program produces the correct result AND every statement has been executed.

- **3.Branch Coverage:** Would have found the bug! Therefore 2 does not imply 3.

**Correct Code**

```java
1   if (x >= 0 ) {
2       x = x + A;
}
2   x = x + A
```

**Buggy Code**

```java
1   if (x >= 0 ) {
     /* missing statement */
}
2   x = x + A
```
Demonstration that 3 Does not Imply 2

• **Branch Coverage:**
  Does not exercise dead code. Therefore 3 does not imply 2.

• However, 3 implies 2 for programs written in a structured programming language without `goto` statements.

```plaintext
1           if (x < 0) {
 2               goto 200;
      x = x + A
} else
      x = x + A
3  200:  x = x + A
```
Control-flow Testing Criteria

- We have explored 3 testing criteria from an infinite set of strategies:
  - 1) Path Testing ($P_\infty$):
    - 100% path coverage.
    - Execute all possible control flow paths through the program.
Control-flow Testing
Criteria (Cont’d)

– 2) Statement Testing ($P_1$):
  • 100% statement coverage.
  • Execute all statements in a program at least once under some test.

– 3) Branch Testing ($P_2$):
  • 100% branch coverage.
  • Execute enough tests to assure that every branch alternative has been exercised at least once under some test.

\[
P_1 \leq P_2 \leq \ldots \leq P_\infty
\]

Dependable Software Systems (Control-Flow Testing) © SERG
Common Sense Strategies

- **Statement** and **branch** coverage have been used for over two decades as a **minimum mandatory** unit test requirement for new code developed at IBM and other companies.

- Insisting on statement and branch coverage is based on common sense rather than theory.
Common Sense Strategies (Cont’d)

- It makes sense to use **branch** coverage because software has a high density of conditional branches, loop, etc. (25% in most PLs)
- It is better to leave out untested code than to include it in a product release.
“The more we learn about testing, the more we realize that statement and branch coverage are minimum floors below which we dare not fall, rather that ceilings to which we should aspire.”

- B. Beizer.
Which Paths?

• You must pick enough paths to achieve statement and branch coverage.

• **Question:** What is the fewest number of paths to achieve statement and branch coverage?

• **Answer:** Un-ask the question.
  – It is better to take many simple paths than a few complicated ones.
  – There is no harm in taking paths that will exercise the same code more than once.
Example of P1 and P2 Coverage

![Graph Diagram]

<table>
<thead>
<tr>
<th>PATHS</th>
<th>DECISIONS</th>
<th>PROCESS LINKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcde</td>
<td>T T</td>
<td>* * * * *</td>
</tr>
<tr>
<td>abhkgede</td>
<td>F T F</td>
<td>** ** ** *</td>
</tr>
<tr>
<td>abhlibcde</td>
<td>TF T T</td>
<td>******** ** *</td>
</tr>
<tr>
<td>abcdjgde</td>
<td>T TF T</td>
<td>******** *</td>
</tr>
<tr>
<td>abcdmnibcde</td>
<td>T TF F</td>
<td>******** *</td>
</tr>
</tbody>
</table>
Branch and Statement Coverage

- **Question:** Does every decision have a T (true) and a F (false) in its column?
- **Answer:** Yes implies branch coverage.
- **Question:** Is every link covered at least once?
- **Answer:** Yes implies statement coverage.
Guidelines

• Select paths as small variations of previous paths.
• Try to change one thing in each path at a time.
Effectiveness of Control-flow Testing

- About 65% of all bugs can be caught in unit testing.
- Unit testing is dominated by control-flow testing methods.
- Statement and branch testing dominates control-flow testing.
Effectiveness of Control-flow Testing (Cont’d)

- Studies show that control-flow testing catches 50% of all bugs caught during unit testing.
  - About 33% of all bugs.
- Control-flow testing is more effective for unstructured code than for code that follows structured programming.
- Experienced programmers can bypass drawing flowgraphs by doing path selection on the source.
Limitations of Control-flow Testing

- Control-flow testing as a sole testing technique is limited:
  - Interface mismatches and mistakes are not caught.
  - Not all initialization mistakes are caught by control-flow testing.
  - Specification mistakes are not caught.
Path Predicates

- Every path corresponds to a succession of **true** or **false** values for the predicates traversed on that path.
- A **Path Predicate Expression** is a Boolean expression that characterizes the set of input values that will cause a path to be traversed.
- **Multiway branches** (e.g., **case/switch** statements) are treated as equivalent **if then else** statements.
Input Values to Path Predicate Expressions

• Any set of input values that satisfies ALL of the conditions of the path predicate expression will force the routine through that path.
• If there is no such set of inputs, the path is not achievable.
Example

X1,X2,X3,X4,X5,X6
if (X5 > 0 || X6 < 0) /* predicates A,B */
...
if(X1 + 3 * X2 + 17 >= 0) /* predicate C */
...
if(X3 == 17) /* predicate D */
...
if(X4 - X1 >= 14 * X2) /* predicate E */
...
Path Predicate Expression is: \((A+B)CDE\)
Input Vector

• The **input vector** of a routine is the set of input parameters to that routine along with any global variables used in that routine.
Process for Creating a Path Expression

• Write down the predicates for the decisions you meet along a path.
• The result is a set of path predicate expressions.
• All of these expressions must be satisfied to achieve a selected path.
Process (In)dependent Predicates

- A predicate whose truth value cannot/can change as a result of the processing is said to be **Process Independent/Dependent**, respectively.
- If all the variables on which a predicate is based are process independent, the predicate must be process independent.
- Process dependence of a predicate does not always follow from dependence of the input variables on which the predicate is based.
Correlated Predicates

- A pair of predicates whose outcomes depend on one or more variables in common are said to be Correlated Predicates.
- Every path through a routine is achievable only if all predicates in that routine are uncorrelated.
Example of Correlated Predicates

E.g.,
X, Y
... /* no changes to X and Y here */
if(X == Y)
if(X + Y == 8)

To satisfy the first predicate we may have to pick values for X,Y that will force the truth value for the second predicate.
Path Sensitization

- The act of finding a set of solutions to the path predicate expression is called path sensitization.
Example:
Uncorrelated & Independent

Because the predicates are uncorrelated and independent … 4 binary decisions means
$2^4 = 16$
possible paths.
Example: Correlated & Independent

- Paths \textbf{abdeg} and \textbf{acdfg} seem to provide coverage, but neither of these paths is achievable.

- Only 2 paths are achievable: \textbf{abdfg} and \textbf{acdeg}.
Test Outcomes

- The **outcome** of test is what we expect to happen as a result of the test.
- Test outcomes include anything we can observe in the computer’s memory that should have (not) changed as a result of the test.
- Since we are not “kiddie testing” we must predict the outcome of the test as part of the test design process.
Testing Process

- run the test
- observe the actual outcome
- compare the actual outcome to the expected outcome.
Questions About Test Outcomes

• **Question:** If the predicted and actual outcomes match, can we say that the test has been passed?

• **Answer:** No! The desired outcome could have been achieved for the wrong reason. (coincidental correctness)
Questions About Test Outcomes

• **Question:** Assume that we ran a covering set of tests and achieved the desired outcomes for each case. Can we say that we’ve covered all branches?

• **Answer:** No! The desired outcome could have been reached by the wrong path!
  – Path instrumentation is necessary to confirm that the outcome was achieved by the intended path.
Path Instrumentation

- All instrumentation methods are a variation on a theme of an interpretive trace.
- An **interpretive trace program** executes every statement in order and records:
  - the intermediate values of all calculations
  - the statement labels traversed
  - ...
Path Instrumentation (Cont’d)

• If we run the tested routine under a trace, then we have all the information we need to confirm:
  – the outcome of the test
  – whether the outcome was achieved by the intended path.
Link Markers

- Name every link by a lowercase letter.
- Instrument the links so that the link’s name is recorded when the link is executed.
- The succession of letters produced in going from the routine’s entry to its exit should, if there are no bugs, exactly correspond to the path name.
Link Counters

- **Link Counters** is an instrumentation method based on counters.
- A link counter is incremented when a link is traversed.
- A path is “confirmed” if the length of the path is equal to the value of the counter.
Link Counters (Cont’d)

• Testing code should include code for tracing.
• Testing code should be executed only during testing.
  – Conditional compilation flags should be used to eliminate testing code for the release of the code.
• There exist tools that perform automatic instrumentation for a variety of PLs.
Integration Testing

- During control-flow testing, a new component is first tested as an independent unit.
- All called components are replaced by stubs.
- A stub is a simulator of a component that is presumably more reliable than the actual component.
Bottom-up Integration Testing

- Components are integrated one at a time with the stubs replaced by the real subroutines.
- This bottom-up integration process continues until the entire system has been integrated.
Problems with Bottom-up Integration Testing

• **Problems:**
  – Stubs may be buggy.
  – Selected paths may become unachievable because of the called component’s processing.

• During software maintenance, legacy code is used instead of stubs.
Two Detailed Examples Of Control-flow Testing
Using Control-flow Testing to Test Function ABS

- Consider the following function:

```c
/* ABS
   This program function returns the absolute value of the integer passed to the function as a parameter.
   INPUT: An integer.
   OUTPUT: The absolute value if the input integer.
*/
int ABS(int x)
{
    if (x < 0)
        x = -x;
    return x;
}
```
The Flowgraph for ABS

/* ABS
   This program function returns the absolute value of the integer
   passed to the function as a parameter.
   INPUT: An integer.
   OUTPUT: The absolute value if the input integer.
*/

int ABS(int x)
{
    if (x < 0)
        x = -x;
    return x;
}
Test Cases to Satisfy Path Coverage for $ABS$

- Complete path testing of $ABS$ is theoretically possible but not practical.
- $ABS$ takes as its input any integer. There are many integers (depending on the maximum size of an integer for the language) that could be input to $ABS$ making it impractical to test all possible inputs to $ABS$. 
# Test Cases to Satisfy Statement Testing Coverage for ABS

## Table

<table>
<thead>
<tr>
<th>PATHS</th>
<th>PROCESS LINKS</th>
<th>TEST CASES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>INPUT</td>
</tr>
<tr>
<td></td>
<td>a  b  c  d</td>
<td>A Negative</td>
</tr>
<tr>
<td>abc</td>
<td>√  √  √</td>
<td>Integer, x</td>
</tr>
<tr>
<td>adc</td>
<td>√  √  √</td>
<td>A Positive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integer, x</td>
</tr>
</tbody>
</table>

## Graph

```
1 -> a
    v    F
   /     |
  b  c  d
     /
    /  
   3   5
    v  v
  5  c  6
```

Dependable Software Systems (Control-Flow Testing)
## Test Cases to Satisfy Branch Testing Coverage for ABS

<table>
<thead>
<tr>
<th>PATHS</th>
<th>DECISIONS</th>
<th>TEST CASES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INPUT</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>abc</td>
<td>T</td>
<td>A Negative Integer, x</td>
</tr>
<tr>
<td>adc</td>
<td>F</td>
<td>A Positive Integer, x</td>
</tr>
</tbody>
</table>

Dependable Software Systems (Control-Flow Testing)
Example: Using Control-flow Testing to Test Program COUNT

- Consider the following program:

```c
/* COUNT
   This program counts the number of characters and lines in a text file.
   INPUT: Text File
   OUTPUT: Number of characters and number of lines.
*/

1     main(int argc, char *argv[])
2     {
3     int numChars = 0;
4     int numLines = 0;
5     char chr;
6     FILE *fp = NULL;
7```
8 if (argc < 2)
9 {
10  printf("\nUsage: %s <filename>", argv[0]);
11  return (-1);
12 }
13 fp = fopen(argv[1], "r");
14 if (fp == NULL)
15  {
16    perror(argv[1]); /* display error message */
17    return (-2);
18  }

Program COUNT (Cont’d)
while (!feof(fp))
{
    chr = getc(fp); /* read character */
    if (chr == '\n') /* if carriage return */
        ++numLines;
    else
        ++numChars;
}
printf("\nNumber of characters = %d", numChars);
printf("\nNumber of lines = %d", numLines);
• The junction at line 12 and line 18 are not needed because if you are at these lines then you must also be at line 14 and 19 respectively.
Test Cases to Satisfy Path Coverage for COUNT

- Complete path testing of *COUNT* is impossible because there are an infinite number of distinct text files that may be used as inputs to *COUNT*. 
Test Cases to Satisfy Statement Testing Coverage for COUNT

1. a → 8 (F) → b → 11 (T)
2. 14 (c) → 17 (F) → 19 (T)
3. 19 (d) → 22 (F) → 23 (T)
4. 24 (k) → 26 (f) → 29

Graph showing test cases and conditions.
# Test Cases to Satisfy Statement Testing Coverage for COUNT

<table>
<thead>
<tr>
<th>PATHS</th>
<th>PROCESS LINKS</th>
<th>TEST CASES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab</td>
<td>√  √</td>
<td>None</td>
</tr>
<tr>
<td>agc</td>
<td>√  √</td>
<td>Invalid Input Filename</td>
</tr>
<tr>
<td>aghdj</td>
<td>√  √</td>
<td>Input File with one character and no Carriage Return at the end of the line</td>
</tr>
<tr>
<td>aghdf</td>
<td>√  √  √  √  √  √  √  √</td>
<td>Input file with no characters and one carriage return</td>
</tr>
</tbody>
</table>

**INPUT**

**OUTPUT**

- “Usage: COUNT <filename>”
- Error Message
- Number of characters = 1
  Number of lines = 0
- Number of characters = 0
  Number of lines = 1
# Test Cases to Satisfy Branch Testing Coverage for COUNT

<table>
<thead>
<tr>
<th>PATHS</th>
<th>DECISIONS</th>
<th>TEST CASES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>INPUT</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>19</td>
</tr>
<tr>
<td>22</td>
<td>ab</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Usage: COUNT &lt;filename&gt;”</td>
</tr>
<tr>
<td></td>
<td>agc</td>
<td>F, T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invalid Input Filename</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Error Message</td>
</tr>
<tr>
<td>aghdjkli</td>
<td>F, F, T, F</td>
<td>Input File with one character and no Carriage Return at the end of the line</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of characters = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of lines = 0</td>
</tr>
<tr>
<td>aghdefli</td>
<td>F, F, T, T</td>
<td>Input file with no characters and one carriage return</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of characters = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of lines = 1</td>
</tr>
</tbody>
</table>
Summary

• The object of control-flow testing is to execute enough tests to assure that statement and branch coverage has been achieved.

• Select paths as deviation from the normal paths. Add paths as needed to achieve coverage.
Summary (Cont’d)

- Find path-sensitizing input data sets for each selected path.
- Use instrumentation (manual or using tools) to verify paths.
- Document all tests and expected test results.
- A test that reveals a bug has succeeded, not failed.