SOFTWARE TESTING

Tester’s Job:

- Find as many faults of different kinds as he can.
- Certify some kind of quality measure for the software based on the test results, meaning he must
  - carefully select test cases, and
  - evaluate the test results.

Software testing does not show that there are no faults, even when every test-case gives the correct output.

Basic Assumption in Software Testing:

- Errors are not intentional by the programmer, i.e., they are not specially crafted.
- All Testing methods depend on this assumption.

Test Coverage Measures:

- They are based on program’s structure or more abstract forms like finite-state model or data-flow model.

Mapping Test Results to Error Discovery: ???
INPUT-OUTPUT SPECIFICATIONS, PROGRAM-BEHAVIOR, AND TEST-CASES

Program-Behavior:

- *Actual* observable input-output behavior of the program; it may be different from the *expected* (based on requirements) behavior.
  - We typically do not have a complete knowledge of the program-behavior even if we have the code

Test-Case Behaviors:

- The input-output behavior that we want to test/observe.

Example. *Requirement:* compute $n^2$ for $-20 \leq n \leq 20$.
- Program computes $n^2$ for $0 \leq n \leq 10$ and $n^3$ for $10 < n \leq 100$.

```latex
\begin{align*}
\text{(1) } \text{Areas } 1+2 &= \{(n, n^2): 0 \leq n \leq 10\} \text{ and } \\
\text{Areas } 4+5 &= \{(n, n^2): -20 \leq n < 0 \text{ or } 10 < n \leq 20\} \\
\text{(2) } (-3, 10) &\in \text{Area#7} \cup \text{Area#8.}
\end{align*}
```

† Fig. 1.4 in "Software Testing" (3rd ed.) by P.C. Jorgensen.
Question:

• Mark all parts of the input-space (horizontal axis) corresponding to the other areas in the top diagram. (It may not be meaningful to do the same for output-space – why?)

• Which points in the second diagram belong to the input-output behavior space in the top diagram?

• Show a modified version of the "alternative" view if the specification allows multiple different acceptable outputs for some inputs.
BLACK-BOX TESTING

- Based on requirements; uses an executable code only.

**Example Requirements** (for an WordCharCounts-function):

(1) Words in the input text file are at most 20 characters long. (This is not same as saying that longer words are to be ignored.)

(2) Blanks, tabs, and new-lines are considered word-separators.

(3) Comma, semicolon, and colon are not part of a word; hyphens as in "son-in-law" are part of a word.

**Example Test-case** (input file is shown as a string):

- The test-case below (\[ for a tab) can verify requirement (1) and parts of (2), making it of category 1 or 4 (see page 3).

  "This text   \t has five words  "

- This would not verify requirement (3), i.e., the required behavior; requirement (3) falls in category 2 or 5 w.r.t this test case.

**Question:** Which of the requirements in (1)-(3) are of category 2 w.r.t the above test case and the source-code below? Give a requirement of category 5 w.r.t this test case.

```c
#define WORDLEN 20
void WordCharCounts(FILE *inFile)
{ int i;
  char word[WORDLEN+1];
  wordCount = charCount = 0;
  while (fscanf(inFile, "%s", word) > 0) {
    wordCount++;
    for (i=0; i<=WORDLEN; i++)
      if (\'\0\' == word[i]) break;
    else charCount++;
  }
}
```
EXERCISE

1. Read the manual page for strcpy-function in C (Unix; type "man strcpy" to see the manual-page); why do you think the source-string pointer is not to be changed by strcpy-function? Make sure that you understand what would go wrong with strcpy-function for the situation below; here, the destination and source strings are next to each other but they do not overlap. Things would also go wrong if we let destination = source + 2, i.e., the destination-string starts at two places after the start of source-string.

   \[
   \begin{array}{cccccccc}
   a & b & c & \text{\null} & d & e & f & g & h & \text{\null} \\
   \text{destination} & \downarrow & \text{source} \\
   \end{array}
   \]

   Then, write a "good" set of requirements for a function, whose profile is given below, and explain what should the new safeStrcpy-function do in each of the above cases. Indicate what should be the return-values in each of the cases.

   \[
   \text{int safeStrcpy(char * destination, char *source)}
   \]

   A programmer should be able to find out if a successful copy action has been properly carried out while using safeStrcpy-function or the nature of the problem that would have happened, so that he can take appropriate alternate action (which could be to use the old strcpy-function). Finally, give an implementation (or a pseudocode) that meets the requirements you formulated.

2. The next-page gives several incorrect versions of safeStrcpy-function; comments are added by me. Find out what is the problem in each case.
INCORRECT safeStrcpy-FUNCTIONS

Comments are added by me.

1. int safeStrcpy(char *destination, char *source)
   { int *ptr = malloc(strlen(source));
     strcpy(ptr, source);
     strcpy(destination, ptr);
     source = ptr;
     return destination;
   }

2. int safeStrcpy(char *destination, char *source)
   { int length = strlen(source);
     char array[length]; //does not work - use malloc
     strcpy(temp, source); //what is temp?
     strcpy(destination, temp);
     return 1;
   }

3. int safeStrcpy(char *destination, char *source)
   { int length = strlen(source);
     char *temp[length];
     strcpy(temp, source);
     strcpy(destination, temp);
     return 1;
   }

4. int safeStrcpy(char *destination, char *source)
   { int n = LENGTH; //what is the relevance of LENGTH?
     if (n != 0) {
       char *d = dst;
       const char *s = src;
       while (--n != 0) {
         if (*d++ = *s++) == 0) { //you meant \0'
           while (--n != 0) *d++ = 0; // \0'
           break;
         }
       }
     }
   return(&des);
   }

5. Pseudocode for safeStrcpy(char *destination, char *source)
   int length = getLength of string; //which string?
   int arrayLength = getLength of array; //what array?
   if (length < arrayLength)
       strcpy(destination, course)
   else strncpy(destination, source, arrayLength)
WHITE-BOX TESTING

- Uses the source-code, in addition to the requirements.
- Can focus on the way an output variable is affected by inputs (static code analysis) and relationship among output variables.
- Allows more detailed testing, taking advantage of *automated* code instrumentation. (Automated instrumentation prevents erroneous code modification.)
- Helps to identify sources of error.
- Better assess the quality of testing in terms of test-coverage measures.

**Example.** Static-analysis can show that we will always have "char-Count ≥ wordCount", which is obviously true (if we do not exclude single character words).

```c
#define WORDLEN 20
void WordCharCounts(FILE *inFile)
{
    int i;
    char word[WORDLEN+1];
    wordCount = charCount = 0;
    while (fscanf(inFile, "%s", word) > 0) {
        wordCount++;
        for (i=0; i<=WORDLEN; i++)
            if ('\0' == word[i]) break;
        else charCount++;
    }
}
```

**Question:** Give another property (relationship) among the outputs or between inputs and outputs that can be obtained by examining the code.
PATH-EQUIVALENCE OF INPUTS

Path-Equivalence of Inputs:

- Two inputs $I_1$ and $I_2$ are \textit{path-equivalent}, denoted by $I_1 \approx I_2$, if they have the same execution paths $\pi(I_1) = \pi(I_2)$.
  - $\pi(I_1)$ and $\pi(I_2)$ follow the same true/false branch at each decision-node for each execution of them, executing same sequence of actions.

Characteristics of An Equivalence Relation:

- Reflexive: $x \approx x$.
- Symmetric: If $x \approx y$, then $y \approx x$.
- Transitive: If $x \approx y$ and $y \approx z$, then $x \approx z$.

Equivalence Class: $[x]_\approx = \{y : y \approx x\}$.

Example: Considering the input file as a string of characters and $I_1 = "abc\ de\", I_2 = "abc\ ed\",$ and $I_3 = "ab\ cde\",$ we only have $I_1 \approx I_2$, i.e., $[I_1] = [I_2] \neq [I_3]$. Why?

```c
#define WORDLEN 20
void WordCharCounts(FILE *inFile)
{ int i;
  char word[WORDLEN+1];
  wordCount = charCount = 0;
  while (fscanf(inFile, "%s", word) > 0) {
    wordCount++;
    for (i=0; i<=WORDLEN; i++)
      if ("\0" == word[i]) break;
    else charCount++;
  }
}
```

Question: Which of $I_1$, $I_2$, and $I_3$ are path-equivalent if we replace the for-loop by "charCount += length(word)"?
REACHABILITY RELATION
IS NOT AN EQUIVALENCE RELATION

Reachability Relation in a Flowchart (any directed graph):
• $xRy$, meaning $y$ can be reached from $x$.

![Flowchart Diagram]

Here, $A_1 RA_2$ but not $A_2 RA_1$ although $D_1 RA_2$ and $A_2 RD_1$.

Question:
• ? Which of the three equivalence-relation properties are violated for the reachability relation?

Output-Equivalence of Inputs:
• $I_1$ and $I_2$ are output-equivalent if $P(I_1) = P(I_2)$.

Question:
• ? Which of $I_1$, $I_2$, and $I_3$ in the previous page are output-equivalent? How about for the modified program with the for-loop replaced by "charCount += length(word)"?
• ? Does $P(I_1) = P(I_2)$ imply that $I_1 \approx I_2$? How about the converse?
• ? For a function $P$, what else should be considered as the output $P(I)$ other than the value returned by $P$?
IMPORTANCE OF
PATH-EQUIVALENCE RELATION

An Elementary Form of Error:

- An error in an action $A$ which does not affect any branch-test condition (hence the execution path $\pi(I)$ for any $I$).
- If $P'$ be an erroneous version of program $P$ due to an elementary error in the action $A$ in $P$, then for each input $I$
  - Path $\pi(I)$ in $P$ equals $\pi'(I)$ in $P'$, with each occurrence of $A$ in $\pi(I)$ replaced by $A'$ in $\pi'(I)$.

Question: Give examples of elementary and non-elementary errors in WordCharCounts-program below.

```c
#define WORDLEN 20
void WordCharCounts(FILE *inFile)
{
    int i;
    char word[WORDLEN+1];
    wordCount = charCount = 0;
    while (fscanf(inFile, "%s", word) > 0) {
        wordCount++;
        for (i=0; i<=WORDLEN; i++)
            if ('\0' == word[i]) break;
        else charCount++;
    }
}
```

Assumption in Program Testing for an Elementary Error:

- Each test-case $I$ for which $\pi(I)$ goes through the erroneous action will show an error in the output.
- If $I$ shows the error and $I' \approx I$, then $I'$ will also show the error.
TESTING STRATEGY FOR
ELEMENTARY ERRORS

Single Elementary Error:

- If we select one test-case $I_j$ from each path-equivalence class of inputs such that
  
  (a) the execution paths $\pi(I_j)$ together cover all actions, and
  
  (b) each $I_j$ produces correct output,

  then the program is error free.

Assumption for Multiple Elementary Errors:

- No two errors cancel each other’s effect.
- Thus, a test case $I_j$ whose execution path $\pi(I_j)$ goes through one or more errors will result in an error in the output.

  Same testing strategy applies for multiple elementary errors.

Simplest Test Coverage Measure:

- $C_0 = \text{The percentage of actions covered by the test-cases.}$
- We want $C_0 = 100\%$ for any acceptable level of testing.
- If $C_0 < 100\%$, then there is some action $A$ such that $A \notin \pi(I_j)$ for any of the test cases $I_j$ and we can replace $A$ by an arbitrary $A'$ without any impact on the test-outputs $P(I_j)$.

Question: How can we always introduce an elementary error in a program $P$ to create a new program $P'$ with the property that the error shows up in the output?
MEASURING ACTION-COVERAGE

Code Instrumentation:

- At the start of each non-trivial action-block introduce a suitable print-operation to indicate that this block is entered:

**Example.** An instrumentation of WordCharCounts-function.

```c
#define WORDLEN 20
void WordCharCounts(FILE *inFile)
{int i;
 char word[WORDLEN+1];
 printf(testCovFile, "entered block A1\n");
 wordCount = charCount = 0;
 while (fscanf(inFile, "%s", word) > 0) {
    printf(testCovFile, "entered block A2\n");
    wordCount++;
    for (i=0; i<=WORDLEN; i++)
       if ('\0' == word[i]) break;
    else {printf(testCovFile, "entered block A3\n");
          charCount++;
    }
}
}
```

**Output in testCovFile for I = "abc de"** (without indentations):

```
entered block A1
    entered block A2
    entered block A3
    entered block A3
    entered block A3
    entered block A2
    entered block A3
entered block A3
```

\[
\begin{align*}
C_0\text{-coverage: } & \frac{\sum_{A_i \text{ covered in all test cases}} (#actions in A_i)}{\sum_{A_j \text{ in the program}} (#actions in all A_j)} = 100\% \text{ for this } I. \\
\end{align*}
\]

**Question:** Give an I with the smallest \( C_0 \)-coverage and give that value of \( C_0 \).
IT CAN BE DIFFICULT TO ACHIEVE $C_0 = 100\%$

**Difficulties:**

- Unreachable code; no execution ever goes through some action-blocks. Cannot achieve $C_0 = 100\%$.
- Difficulty in finding test-input $I$ for which $\pi(I)$ contains a specific action-block.
- Both can happen when there are many interdependent if-statements.

**Problems with Code Instrumentation:**

- Although code instrumentation can be done via automated tools, it can increase the program size significantly.
- It can slow down the execution significantly.
- The instrumentation output file can be too large.

**Approximate Methods:**

- The runtime machine code execution is sampled.
- Each executed machine code is mapped to the program source-code.
- Reduces program overhead in terms of program-memory, execution time, and measurement-output file.

**Question:**

- How can we instrument a code more intelligently to minimize the instrumentation-output and still have enough information to compute $C_0$? Show the instrumented form for WordCharCounts-function and the instrumentation output for $I = "abc\ de"$. 

A REFINEMENT OF $C_0$-MEASURE

- Not all action-blocks $A_i$ are of equal relevance in execution paths.

(i) A flowchart.
(ii) The domination tree.
(iii) The reverse domination tree.

- If a set of begin-end paths cover the terminal nodes \{A3, A4, A5\} in the domination-tree, then it covers all flowchart nodes.
- Similarly, for the terminal nodes of the reverse-domination tree.

For 100% $C_0$-coverage, we only need to cover the common terminal nodes in domination and reverse-domination trees.

Question:

- State the definition of the new $C_0$-coverage measure based on the common terminal nodes of the two domination trees.
- Give the new measure for $I = $ empty-text-file for the WordChar-Count function (with bounded word-length).
BRANCH-COVERAGE MEASURE

$$C_1\text{-coverage:} \quad \frac{\sum_{D_i \text{ covered in all test runs}} (#T/F \text{ branches covered at } D_i)}{2 \times (#\text{branch nodes in program})}$$

= 100% for I = "abc de".

Additional Instrumentation (for empty then/else blocks):

```c
#define WORDLEN 20
void WordCharCounts(FILE *inFile)
{
    int i;
    char word[WORDLEN+1];
    printf(testCovFile, "entered block A1\n");
    wordCount = charCount = 0;
    while (fscanf(inFile, "%s", word) > 0) {
        printf(testCovFile, "entered block A2\n");
        wordCount++;
        for (i=0; i<=WORDLEN; i++)
        if (\'0\' == word[i]) {
            printf(testCovFile, entered block A4\n");
            break;
        }
    else {printf(testCovFile, "entered block A3\n");
            charCount++;
        }
    }
}
```

Output in testCovFile for I = "ab cde" (without indentations):

```
entered block A1
    entered block A2
    entered block A3
    entered block A3
    entered block A4
entered block A2
    entered block A3
    entered block A3
    entered block A4
```

**Question:** Should we instrument the exits from loops (for, while-do, etc.)? What is the minimum $C_1$-coverage for an I here?
BOUNDARY TESTING

- This may involve both valid test-cases that are "within specification limits" and also invalid test-cases that are outside the limits (testing for graceful-failing vs. "abort").

Requirement based:

- Many requirements represent constraints on inputs and outputs, and they can give rise to the respective boundary values.
  - Boundary testing can apply to inputs and to the outputs (trying to push respectively the input and/or the output to the boundary limits).

- The boundary values can be related to both entities and relationships in the data-model.

Example.

- For WordCharCounts-program,
  - Input text files with words of size 1 and of max length WORDLEN = 20 represent a form of boundary case.
  - Empty input file itself is also a boundary case.
  - An input file with words longer than WORDLEN = 20 represent a test-case for testing graceful-degradation.

- For TriangleClassification-function
  - An input file containing triplets for each category of triangles (equilateral, isosceles, and scalar) and also non-triangular triplets is a regular test-case.
  - An input to test graceful-failing would be an \(abc\)-triplets where the input-condition \(a \leq b \leq c\) is violated, which can happen in more than one way.
NOTION OF PROGRAM SLICE

Program Slice:

- Given an output variable \( x \), it is part of the program involving only those (parts of) statements that may affect \( x \).
- Includes relevant branch-statements, variables \( y \) that affect those branches, and the statements that affect those \( y \) in turn.
- The slice may be a small fragment of the original program and hence easier to test or debug.

Example. The bold lines below show the parts deleted to obtain the slices of WordCharCounts-function for the output variable wordCount and for charCount.

```c
#define WORDLEN 20 //slice for wordCount
void WordCharCounts(FILE *inFile)
{
    int i;
    char word[WORDLEN+1];
    wordCount = charCount = 0;
    while (fscanf(inFile, "%s", word) > 0) {
        wordCount++;
        for (i=0; i<=WORDLEN; i++)
            if ('\0' == word[i]) break;
        else charCount++;
    }
}
```

```c
#define WORDLEN 20 //slice for charCount
void WordCharCounts(FILE *inFile)
{
    int i;
    char word[WORDLEN+1];
    wordCount = charCount = 0;
    while (fscanf(inFile, "%s", word) > 0) {
        wordCount++;
        for (i=0; i<=WORDLEN; i++)
            if ('\0' == word[i]) break;
        else charCount++;
    }
}
```
**DEFINITION-USE RELATIONSHIP**

**Definition of a Variable:** $\text{def}(x, s)$

- A statement $s$ is a *definition* of $x$ if an execution of $s$ assigns a value to $x$.
- $s$ can be an input-statement from a file, an assignment statement, or a function-call statement.

**Use of a Variable:** $\text{use}(x, s)$

- A statement $s$ is an *use* of a variable $x$ if an execution of $s$ requires a value of $x$.

**Example:**
- The statement
  
  ```c
  fscanf(fp, "%s", word);
  ```

  is a definition of $\text{word}$, assuming that $\text{fp} \neq \text{NULL}$. It assigns a value to $\text{word}$ only if reading a non-empty string succeeds and otherwise the old value (if any) is retained.

  It is also a definition $\text{fp}$, as it may update $\text{fp}$.
- The above statement is not an use of $\text{word}$; it uses the address of $\text{word}$. It also uses the file-pointer $\text{fp}$.
- The statement
  
  ```c
  fscanf(fp, "%s %d", word, &i);
  ```

  may define $i$ and in that case it is also an use of $\text{word}$!
- The statement "$\text{i++;}" is both a definition and an use of $\text{i}$. 
1. void WordCharCounts(FILE *inFile)
2. {
3.     int i;
4.     char word[WORDLEN+1];
5.     wordCount = charCount = 0;
6.     while (fscanf(inFile, "%s", word) > 0) {
7.         wordCount++;
8.         for (i=0; i<=WORDLEN; i++)
9.             if ('\0' == word[i]) break;
10.        else charCount++;
11.    }
12. }

### Variables Definitions Uses

<table>
<thead>
<tr>
<th>Variables</th>
<th>Definitions</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>inFile</td>
<td>1, 5</td>
<td>5</td>
</tr>
<tr>
<td>charCount</td>
<td>4, 9</td>
<td>4, 9</td>
</tr>
<tr>
<td>i</td>
<td>7</td>
<td>7, 8</td>
</tr>
<tr>
<td>word (addr of word)</td>
<td>5 (3)</td>
<td>8 (5, 8)</td>
</tr>
<tr>
<td>wordCount</td>
<td>4, 6</td>
<td>6</td>
</tr>
</tbody>
</table>
DEF-USE RELATIONSHIP

Def-Use relationship:

- We say \( \text{def}(x, s) \) is related to \( \text{use}(x, s') \), where \( s \) may equal \( s' \), if there is an \( ss' \)-path of length \( \geq 0 \) such that there is no other definition of \( x \) on that path in between \( s \) and \( s' \).

Example of Def-Use Relationship.

```c
1. void WordCharCounts(FILE *inFile)
2. { int i;
3.    char word[WORDLEN+1];
4.    wordCount = charCount = 0;
5.    while (fscanf(inFile, "%s", word) > 0) {
6.        wordCount++;
7.        for (i=0; i<=WORDLEN; i++)
8.            if ('\0' == word[i]) break;
9.            else charCount++;
10.    }
11. }
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definitions</th>
<th>Uses of each definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>inFile</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>charCount</td>
<td>4</td>
<td>4, 9</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i</td>
<td>7 (twice)</td>
<td>7 (twice), 8</td>
</tr>
<tr>
<td>word</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>wordCount</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

- A definition with no uses is a potential flaw (e.g., a missing use).
- Even if a definition has an use, this may not be a "true" use because the def-use path is not executable.

Question: Is there any non-realizable (non-executable) def-use relationship above?
EXERCISE

1. Show the def-use relationships for the code below. Show a test-data that covers all the def-use relationships, but does not give 100% $C_1$-coverage; give the $C_1$-coverage measure for this test data and indicate the branch(es) not covered. Give another test-data to cover some of those uncovered branch(es). If we insert a suitable print-statement in the beginning of the body of the outer while-loop, then which def-use relationship-pairs will it track, and what happens if we put the print-statement just before line 4? (Draw the flowchart to see things more clearly.)

```c
01. void WordCharCounts(FILE *inFile)
02. { char ch;
03. wordCount = charCount = 0;
04. while (fscanf(inFile, "%c", &ch) > 0)
05.    if ((ch != ' ') && (ch != '\n')) {
06.       charCount++; wordCount++;
07.      while (fscanf(inFile, "%c", &ch) > 0)
08.         if ((ch != ' ') && (ch != '\n'))
09.             charCount++;
10.           else break;
11.    }
12. }
```

2. How do you define a coverage measure based on the def-use relationship? Explain with an example.
A GLOBAL VIEW OF TESTING

Test Strategy: White-box or Black-box testing.

Test Goals/Objectives: Functional or performance testing.
- Select (user or design) requirements to be tested.
  - Identify functions and their input and output variables.
- Select test-coverage measures and the percentage coverage to be achieved for each measure.

Three Test Conclusions:
- More test needed, selected requirements satisfied, or not satisfied.

A Dataflow Diagram for Testing:
COMPARISON OF TEST-CASES

Basis of Comparison:

- Output point of view: how different are the outputs.
- Execution point of view: how different are the execution-paths (or the number of statements executed, etc)?
- Performance point of view: how different are the performance parameters like execution time and memory use?

Notes:

- The first and third above falls in black-box view, and the second one in white-box view.

Question:

- What are some other points of view for comparing test-cases, and which ones fall in black-box view and which ones fall in white-box view?
- How would you differentiate test-cases from input point of view?
COMPARISON OF TEST-CASES
VIA PROGRAM STRUCTURE

A Program Execution Path is More Than A Path:

- The nesting structure of program blocks gives a program execution-path more structure than just the linear (sequential) structure of a path in a general digraph.

Nesting Tree of Program Blocks:

- It is a rooted ordered tree, with each node represents an one-entry-one-exit block (disregarding breaks, continues, and returns).
  - Children of a node are ordered left-to-right representing sequential order of the associated subblocks.

- If-then-else decision nodes have two children: then-part forms the left-child and else-part forms the right-child.

- The decision-nodes for for-loop and whileDo-loop are shown as filled, and those for doWhile-loops are shown as double circles.
  - The subtrees of the children of these decision nodes form the body of the loop.

- Unlike the T/F labels of the links to children of an if-then-else decision node, there are no labels of the links to the children of decision-nodes for the loops.
NESTING TREE OF PROGRAM-BLOCKS

Flowchart and Nesting Tree for wordCharCounts-function:

- We are using below the version that uses WORDLEN and does not use strlen-function.
- Since the then-part of D3 has no action other than transfer of abnormal (semi-structured) control via "break" (to D1) it is shown as a dashed circle.

Question: Suppose we replace the for-loop "for (i=0; i≤WORDLEN; i++) ..." by the following; note that the for-loop now starts with i = 1. Show the new action blocks, the new flowchart, and the new nesting-tree.

```c
charCount++;
for (i=1; i<=WORDLEN; i++)
    if ('\0' == word[i]) break;
else charCount++;
```
APPROX. REPRESENTATION OF AN EXECUTION-PATH USING NESTING-TREE

• Shows the count of each node in the nesting-tree for an execution-path \( \pi(I) \) for some input \( I \), giving an abstraction of \( \pi(I) \).
  – Allows giving different weights for action-blocks at different levels and define a more refined form of \( C_0 \)-measure.
  – Allows giving different weights for branches of decision-nodes at different levels and define a more refined form of \( C_1 \)-measure.

Example.
• An action-block node shows its #(executions).
• A loop-decision node shows #(loop-body executions).
• An if-then-else decision node shows #(true-branch executions) and #(false-branch executions).
• The mark "?" shows an unknown value (based on the limited action-block instrumentation output); they can be derived if we know the source-code (or have the branch-instrumentation output) and they are indicated in parentheses next to '?'.

Instrumentation output for \( I = "abc\ de" \):

entered block A1
  entered block A2
    entered block A3
    entered block A3
    entered block A3
entered block A2
  entered block A3
  entered block A3

Diagram:

```
start 1
   \( \text{A1} \) 1
   \( \text{D1} \) 2
      \( \text{A2} \) 2
      \( \text{D2} \) ?(2)+5
         \( \text{D3} \) ?(2), 5
            ?(2)
            \( \text{A3} \) 5
```
EXERCISE

1. How can use the representation of test-paths to analyze a set of test-cases?