3.0 Introduction

Test cases for a program can also be selected based on its specification. In the following we describe five methods for this purpose.

Critics of the code-based test-case selection methods often claim that specification-based methods are better because, if the source code did not implement some part of the specification, a code-based method will never yield a test case to reveal what is missing. This criticism is not entirely true. If there is a component missing in the source code, it should cause the program to malfunction in some way. Otherwise, it is an unnecessary rather than a missing part. Furthermore, programmers not only are capable of neglecting to incorporate needed parts into the program, they are equally capable of writing code segments that are uncalled-for as well. Such errors, by the same token, would be hard to detect if the test cases are selected based on program specification alone.

Obviously, it is more accurate to say that the code-based and specification-based methods complement one another.

Note that computerization is the most effective way to reduce the cost and time required to test a program. Compare to a code-based method, it is much harder to mechanize a specification-based method because in practice most software specifications are written in a natural language.

Just like in the development of a code-based method, the central idea used in the development of a specification-based method is still the same, i.e., identification of the components that need to be exercised during the test. The only difference is that now the components exist in the specification instead of the source code.

3.1 Error Guessing

Error guessing refers to the test-case selection technique used by some people who seem naturally adept at program testing. Given the description of a program, these people guess, both by intuition and experience, certain probable types of error and then select test cases to expose them. In other words, the components need to be exercised during the test are the subfunctions that are most likely to be implemented incorrectly.

This technique is largely an intuitive and ad hoc process. The basic idea is to list possible errors or error-prone situations and then select test cases based on that list.

For example, suppose a program was written to meet the following specification:

Given a text terminated by an ENDOFTEXT character and consisting of words separated by BLANK or NEWLINE characters, reformat it to a line-
by-line form in accordance with the following rules: (1) line breaks are made only where the given text has BLANK or NEWLINE; (2) each line is filled as far as possible as long as (3) no line will contain more than MAXPOS characters.

This specification will make an experienced programmer to wonder if the program will work correctly (and hence to select test data to exercise the program) for the following cases:

- an input text of length zero
- a text containing a very long word (of length greater than MAXPOS)
- a text containing nothing but BLANK's and NEWLINE's
- a text with an empty line
- words separated by two or more consecutive BLANK's or NEWLINE's
- a line with BLANK as the first or last character
- a text containing digits or special characters
- a text containing nonprintable characters
- MAXPOS set to a number greater than the system default line length

In addition to experience in the problem domain and software construction, testing, and debugging, study of work on programming style (e.g., [KEPL78]) may help one to cultivate skill in error guessing.

### 3.2 Equivalence Partitioning

**Basic idea:**

Partition the input domain of a program (as described in the specification) into a finite number of equivalence classes such that one can reasonably assume that a test of a representative value of each class is equivalent to a test of any other value.

The reason is that all elements in an equivalence class are expected to be treated in the same way by the program. If one test case in an equivalence class detects an error, all other test cases in the equivalence class would be expected to find the same error. Conversely, if a test case did not detect an error, we would expect that no other test cases in the equivalence class would find an error.

**Identification of equivalence classes:**

Study the program specification to find input conditions and

1. for each condition $P_i$, identify one valid equivalence class defined by $P_i$ and one invalid equivalence class defined by $\neg P_i$.

   e.g., if $P_i$ is "the first symbol of the identifier must be a letter" then the valid equivalence class is defined by "it is a letter" and the invalid one is defined by "it is not a letter".
(2) for each input condition of the form $P_i$ and $P_j$, identify one valid equivalence class defined by $P_i$ and $P_j$, and two invalid ones defined by $\neg P_i$ and $\neg P_j$, respectively.

E.g., if the input condition is $1 \leq n \leq 5$, the valid equivalence class is defined by $1 \leq n \leq 5$, and the two invalid equivalence classes are defined by $n < 1$ and $n > 5$, respectively.

(3) If there is any reason to suspect that elements in an equivalence class are not treated in an identical manner by the program, split the equivalence class into smaller ones.

**Test-case selection:**

(1) Until all valid equivalence classes have been covered by test cases, find a new test case that covers as many of the uncovered valid equivalence classes as possible.

(2) For every invalid equivalence class, find a test case that covers that invalid equivalence class only.

### 3.3 Boundary-Value Analysis

**Basic idea:**

Select test cases that lie directly on, above, and beneath the boundaries of input equivalence classes and output equivalence classes to explore the program behavior along the border.

Note that this method differs from the equivalence partitioning in two respects: (a) rather than checking to see if the program will execute correctly for a representative element in an equivalence class, it attempts to determine if the program defines the equivalence class correctly, and (b) rather than selecting test cases based on input conditions only, it also requires derivation of test cases based on output conditions.

**Test-case selection:**

(1) If an input variable is defined in a range from LB to UB, use LB, UB, LB - $\delta$, and UB + $\delta$ as the test cases. (Here delta represents the smallest possible change in value.)

(2) Use rule (1) for each output variable.

(3) If the input or output of a program is a sequence (e.g., a sequential file or a linear list), focus attention on the first and last element of the sequence.

(4) Use your ingenuity to search for additional boundary values.
3.4 Cause-Effect Graphing

Basic idea:

A "cause" is an input condition, and an "effect" is a specific sequence of computations to be performed. A cause-effect graph is basically a directed graph that describes the logical combinations of causes and their relationship to the effects to be produced [MYER79].

Cause-effect graphing is a technique that aids in selecting test cases to check if the program will produce right effect for every possible combination of causes.

Test-case selection:

1. Divide the program specification into pieces of workable size.
2. Identify causes and effects in the specification.
3. Analyze the specification to determine the logical relationship among causes and effects, and express it as a cause effect graph.
4. Identify syntactic or environmental constraints that make certain combinations impossible.
5. Translate the graph into a limited-entry decision table.
6. Select a test case for every column in the decision table.

The format of a limited-entry decision table:

<table>
<thead>
<tr>
<th>condition stubs</th>
<th>condition entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>action stubs</td>
<td>action entries</td>
</tr>
</tbody>
</table>

♦ The condition stub contains a list of conditions, one to each row.
♦ The condition entry lists combinations of condition values in a column.
♦ The condition constraints express interactions between conditions.
♦ The action stub contains a list of actions.
♦ The action entry contains an "X" for each action to be performed.
♦ The application constraints express the conditions under which an action is to be performed.
Shown below is an example of a limited-entry decision table obtained by analyzing the following program specification:

Write a program to reformat text as follows.

Given a text terminated by an ENDOFTEXT character and consisting of words separated by BLANK or NEWLINE characters, reformat it to a line-by-line form in accordance with the following rules: (1) line breaks are made only where the given text has BLANK or NEWLINE; (2) each line is filled as far as possible as long as (3) no line will contain more than MAXPOS characters.

The resulting text should contain no blank lines. Sound an alarm if the text contains an oversized word.

<table>
<thead>
<tr>
<th>C1: The input character is BLANK or NEWLINE</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C2: The input character is ENDOFTEXT</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>(N)</td>
<td>(N)</td>
<td>(N)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C3: The current word is not empty</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>(Y)</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C4: The current output line still has enough space for the current word</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>(N)</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C5: The current output line is not empty</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C6: The current word is of the maximum size</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>(N)</td>
<td>-</td>
<td>(N)</td>
<td>(N)</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A1: Write BLANK and increment the current-word size</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A2: Write NEWLINE and reset the current-word size to 0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A3: Write the current word</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A4: Sound the alarm</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A5: Append this character to the current word and increment its size</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A6: Read a character and repeat this table</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A7: Exit from this table</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: In the above table, ".-" means either Y or N, and (Y) or (N) means it is implied.