Chapter 3
Specification-Based Test-Case Selection Methods

J. C. Huang
Department of Computer Science
University of Houston
Background

Recall that the essence of a test-case selection method is to identify certain programming components that need to be exercised during the test.

The components can be identified based on the source code or specification.
Code-Based vs. Specification-Based

Some critics argued that test-case selection based on the source code would not be effective because if a portion of the specification is not implemented, no test case will be picked to reveal that error. A specification based method will not have this shortcoming.
Code-Based vs. Specification-Based (continued)

The same can be said about spec-based methods. If a faulty component in a program is extraneous in nature, which is not uncommon, a spec-based method will suffer the same problem.
Comment

A programmer makes mistakes by *omission* as well as *commission*.
Specification-Based Test-Case Selection Methods

- Error Guessing
- Equivalence Partitioning
- Boundary-Value Analysis
- Cause-Effect Graphing
Error Guessing

Basic Idea:

Given the description (specification) of a program, the tester guesses, both by intuition and experience, certain probable types of error and then select test cases to expose them.
Error Guessing (continued)

The method:

Construct a list by enumerating all possible errors or error-prone situations, and then use it to select test cases.
Example

Given the following specification:

Given a text consisting of words separated by BL (blank space) and CR (carriage return) characters, reformat it so that (1) line breaks are made only where the given text has BL or CR; (2) each line is filled as far as possible; and (3) no line contains more than MAXCHAR characters.
Example (continued)

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

→ an input text of length zero
→ a text containing a very long word (of length greater than MAXCHAR)
→ a text containing nothing but BL's and CR's
→ a text with an empty line
Example (continued)

- words separated by two or more consecutive BL's or CR's
- a line with BL as the first or last character
- a text containing digits or special characters
- a text containing nonprintable characters
- MAXCHAR set to a number greater than the system default line length
Equivalence partitioning method

Basic idea:

Partition the input domain of a program 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 counter part of branch testing)
Equivalence partitioning method (continued)

The method:

Step 1: Study the program specification to find input conditions and identify the equivalence classes.
Equivalence partitioning method (continued)

(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".
Equivalence partitioning method (continued)

(2) for each input condition of the form \( P_i \text{ and } P_j \), identify one valid equivalence class defined by \( P_i \text{ and } P_j \), 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.
Equivalence partitioning method (continued)

(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.
Equivalence partitioning method  (continued)

The method:

Step 2: Select the test cases.

(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.
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.

(The counterpart of domain-strategy testing)
Boundary-value analysis

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 base on output conditions.
Boundary-value analysis (continued)

The method:

(1) If an input variable is defined in a range from LB to UB, use LB, UB, LB - δ, and UB + δ 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.
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.

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.
Cause-effect graphing (continued)

The method:

(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.
Cause-effect graphing (continued)

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>
Cause-effect graphing (continued)

• 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.
Cause-effect graphing (continued)

Example:

Shown below is an example of a limited-entry decision table obtained by analyzing and completing the following program specification:

*Given a text consisting of words separated by BLANK or NEWLINE characters, convert it to a line-by-line form in accordance with the following rules: (1) line breaks must be 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.*
### Cause-effect graphing (continued)

<table>
<thead>
<tr>
<th></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>C1: this word is BLANK or NEWLINE</td>
<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>
</tr>
<tr>
<td>C2: this word is ENDOFTEXT</td>
<td>(N)</td>
<td>(N)</td>
<td>(N)</td>
<td>(N)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>C3: the buffer is not empty</td>
<td>Y</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>
</tr>
<tr>
<td>C4: there is enough space in the buffer</td>
<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>
</tr>
<tr>
<td>C5: fill ° 0</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C6: the buffer is full</td>
<td>(N)</td>
<td>(N)</td>
<td>-</td>
<td>(N)</td>
<td>(N)</td>
<td>(N)</td>
<td>-</td>
<td>(N)</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>A1: write BLANK and increment fill</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2: write NEWLINE and set fill to 0</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3: write the contents of the buffer, reset fill and the buffer pointer</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4: set the alarm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>A5: increment the buffer pointer and put this word in the buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>A6: read a word and repeat this table</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>A7: exit from this table</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>