Chapter 3
Specification-Based Test-Case Selection Methods

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

Basic idea

Test cases can also be selected based on the information extracted from the program specification.

The two principles of test-case selection are applicable just like in selecting test cases based on the source code.

Specification-based vs. source-code based

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.

Specification-based vs. source-code based

That is true.

But the fact is that the programmer is not only capable of making mistakes by omission but also by commission as well.

Specification-based vs. source-code based

For example, the programmer often partition the input domain into more subdomains prescribed by the specification.

In that case, errors in some subdomain may not be detected by the test cases selected based on the specification alone.

Specification-based vs. source-code based

Therefore, it is more accurate to say the code-based methods complement the specification-based methods.

Certain types of fault cannot be detected by using the code-based methods alone, and certain types of faults cannot be detected by using the specification-based methods alone.

Specification-based vs. source-code based

It is harder and costlier to select test cases based on the program specification because the specification is often written in a natural language.

Basic idea

The specification-based methods can be similarly developed by applying the two principles.

It involves identification of the components to be exercised during the test, and selection of test cases that are computationally as loosely coupled among one another as possible.

An important difference

Because execution paths cannot be readily identified in a program specification, different schemes have to be devised to find test cases that are computationally loosely coupled.

Constructs of interest

What constructs in a program spec. can be used to select test cases?

Condition clauses and action clauses

A *condition clause* is an expression that can be true or false.

An *action clause* is an expression that prescribes an action to be taken by the program during execution.

Example

Specification 3.1:

Write a program that takes three positive integers as input and determine if they represent three sides of a triangle, and if they do, indicate what type of triangle it is. To be more specific, it should read three integers and set a flag as follows:

Example (continued)

if they represent a scalene triangle then set the flag to 1,

if they represent an isosceles triangle then set the flag to 2,

if they represent an equilateral triangle then set the flag to 3,

and if they do not represent a triangle then set the flag to 4.

Example (continued)

We can see that in this specification there are four condition clauses:

They represent a scalene triangle
They represent a isosceles triangle
They represent an equilateral triangle
They do not represent a triangle at all

Example (continued)

and four action clauses:

Set the flag to 1
Set the flag to 2
Set the flag to 3
Set the flag to 4

Example (continued)

Such clauses can be used to find out how the function implemented by the program can be decomposed into a set of subprograms. We can then choose the subprograms as the components to be exercised during the test.

Theoretical significance

A program can be viewed as an artifact that embodies a mathematical function $f: X \rightarrow Y$, where f is often implemented as a set of n subfunctions

$f = \{f_1, f_2, ..., f_n\}$,
$f_i: X_i \rightarrow Y$, and
$X = X_1 \cup X_2 \cup ... \cup X_n$.

Thus we can choose f_is as the components to be exercised.

The two principles

In applying the two principles to develop methods for test-case selection, we can choose f_is as the components to be exercised.

Note that if two test cases exercise two different subfunctions they will also be loosely coupled computationally as well because they will be processed by two different subprograms.

Methods to be discussed

- Subfunction testing
- Predicate testing
- Boundary-value analysis
- Error guessing

Subfunction testing

The components to be exercised are the subfunctions to be implemented by the program.

The test set should consist of at least one element from each subdomain.

To be more precise, if the specification says that $f: X \rightarrow Y$ is the function to be implemented, $f = \{f_1, f_2, \ldots f_n\}$, $f_i: X_i \rightarrow Y$ and $X = X_1 \cup X_2, \cup \ldots \cup X_n$ then the test should contain at least one element from each and every X_i.

Subfunction testing (continued)

We assume that subfunction f_i will compute its value for every element in X_i by performing the same sequence of operations. Otherwise f_i has to be further decomposed until this property is satisfied.

Subfunction testing (continued)

A subfunction test therefore will cause every component in the specification to be exercised at least once, and all test cases will be computationally loosely coupled.

Example 1 (continued)

Consider Specification 3.1.

The input domain is the set of all triples of the form $<x_1, x_2, x_3>$, where $x_i \geq$ MININT and $x_i \leq$ MAXINT for all i = 1, 2, and 3.

Example 1 (continued)

It is obvious from the specification that the input domain X has to be partitioned into four subdomains, viz., $X = X_1 \cup X_2 \cup X_3 \cup X_4$, such that

$X_1 = \{<x_1, x_2, x_3> \mid <x_1, x_2, x_3> \in X \wedge \text{TRIANGLE}(x_1, x_2, x_3) \wedge \text{SCALENE}(x_1, x_2, x_3)\}$

$X_2 = \{<x_1, x_2, x_3> \mid <x_1, x_2, x_3> \in X \wedge \text{TRIANGLE}(x_1, x_2, x_3) \wedge \text{ISOSCELES}(x_1, x_2, x_3)\}$

$X_3 = \{<x_1, x_2, x_3> \mid <x_1, x_2, x_3> \in X \wedge \text{TRIANGLE}(x_1, x_2, x_3) \wedge \text{EQUILATERAL}(x_1, x_2, x_3)\}$

$X_4 = \{<x_1, x_2, x_3> \mid <x_1, x_2, x_3> \in X \wedge \text{TRIANGLE}(x_1, x_2, x_3) \wedge \neg\text{TRIANGLE}(x_1, x_2, x_3)\}$

Example 1 (continued)

where

$$\text{TRIANGLE}(x_1, x_2, x_3) \equiv x_1>0 \wedge x_2>0 \wedge x_3>0 \wedge x_1+x_2>x_3 \wedge x_2+x_3>x_1 \wedge x_3+x_1>x_2$$

$$\text{SCALENE}(x_1, x_2, x_3) \equiv x_1\neq x_2 \wedge x_2\neq x_3 \wedge x_3\neq x_1$$

$$\text{ISOSCELES}(x_1, x_2, x_3) \equiv x_1=x_2 \wedge x_2\neq x_3 \wedge x_3\neq x_1 \vee x_1\neq x_2 \wedge x_2=x_3 \wedge x_3\neq x_1 \vee x_1\neq x_2 \wedge x_2\neq x_3 \wedge x_3=x_1$$

$$\text{EQUILATERAL}(x_1, x_2, x_3) \equiv x_1=x_2 \wedge x_2=x_3 \wedge x_3=x_1$$

Example 1 (continued)

For this example, to do subfunction testing, therefore, is simply to test-execute the program with a test set consisting of four test cases, one each from subdomains X_1, X_2, X_3, and X_4.

A possible test set would be T = {<3, 4, 5>, <3, 3, 4>, <7, 7, 7>, <2, 3, 6>}.

The four subfunctions to be exercised, f_1, f_2, f_3, and f_4 are "set flag to 1", "set flag to 2", "set flag to 3", and "set flag to 4", respectively.

Example 2

Construct a test set based on the program specification given below.

Example 2 (continued)

Specification 3.2:

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. Set the alarm and terminate the program if the text contains an oversized word.

Example 2 (continued)

We shall identify the input domain first.

It is assumed that the input text is read by the program one character at a time. Therefore, the input is the set of all ASCII characters, assuming that the input is to be provided through a keyboard. We can start by partitioning the input domain into three subdomains, X_{EW}, X_{ET}, and X_{AN},

Example 2 (continued)

where

X_{EW}: the set of end-of-the-word markers in the input text consisting of BLANK (white space) and NEWLINE characters.

X_{ET}: the set consisting of end-of-the-text marker.

X_{AN}: the set consisting of all alphanumerical characters and punctuation marks that may appear in the input text.

Example 2 (continued)

So let us examine what the program needs to do if we partition the input domain into a set of subdomains as listed in Table 3.1.

Table 3.1
A trial partition of the input domain

subdomain	domain predicate	subfunction
1	$x \in X_{EW}$	f_1
2	$x \in X_{ET}$	f_2
3	$x \in X_{AN}$	f_3

f_1 (in Table 3.1)

```
[The input character is either a blank or new-line]
if wordlength > 0 then
    begin
        if linelength + wordlength ≥ MAXPOS then
            begin
                write(newline);
                linelength := wordlength
            end
        else
            begin
                write(blank);
                linelength := linelength + wordlength
            end;
        write(word);
        wordlength := 0;
    end;
read(char);
```

f_2 (in Table 3.1)

```
[The input character is an end-of-the-text mark]
if wordlength + linelength ≥ MAXPOS
then
    write(newline);
else
    write(blank);
write(word);
write(char);
exit;
```

f_3 (in Table 3.1)

```
[The input character is alphanumeric]
append(char, word);
wordlength := wordlength + 1;.
if wordlength > MAXPOS then
    begin
        write(alarm);
        exit
    end
else
    read(char);
```

Example 2 (continued)

Note that the description of every subfunction listed in Table 3.1 contains an "if" statement, indicating that the inputs in the same subdomain may be processed by different sequences of operations. This is due to the fact that what needs to be done for a particular input is not only dependent on its membership in a particular subdomain but also on the previous inputs as well.

Table 3.2 Further partitioning of the input domain.

subdomain	domain predicate	subfunction
1.1	$x \in X_{EW}$ *and* wordlength > 0	$f_{1.1}$
1.2	$x \in X_{EW}$ *and* wordlength ≤ 0	$f_{1.2}$
2.1	$x \in X_{ET}$ *and* wordlength+linelength $\geq$ MAXPOS	$f_{2.1}$
2.2	$x \in X_{ET}$ *and* wordlength+linelength < MAXPOS	$f_{2.2}$
3.1	$x \in X_{AN}$ and wordlength>MAXPOS	$f_{3.1}$
3.2	$x \in X_{AN}$ and wordlength$\leq$MAXPOS	$f_{3.2}$

$f_{1.1}$ (in Table 3.2)

```
[The input character is either a blank or new-line
 which marks the end of current word]
if linelength + wordlength ≥ MAXPOS then
      begin
            write(newline);
            linelength := wordlength
      end
else
      begin
            write(blank);
            linelength := linelength + wordlength
      end;
write(word);
wordlength := 0;
read(char);
```

$f_{1.2}$ (in Table 3.2)

[The input character is either a blank or new-line which is redundant]

read(char);

$f_{2.1}$ (in Table 3.2)

[The input character is an end of the text mark and the current word is too long to be written on the current line]

write(newline);

write(word);

write(char);

exit;

$f_{2.2}$ (in Table 3.2)

[The input character is an end of the text mark and the current word can be written on the current line]

write(blank);

write(word);

write(char);

exit;

$f_{3.1}$ (in Table 3.2)

[The input character is alphanumeric and the current word is too long]

append(char, word);

wordlength := wordlength + 1;

write(alarm);

$f_{3.2}$ (in Table 3.2)

[The input character is alphanumeric and the current word is not too long]

append(char, word);

wordlength := wordlength + 1;.

read(char);

Example 2 (continued)

Subdomain $X_{1.1}$ has to be partitioned further because the description of $f_{1.1}$ has to make use of a branch statement.

Table 3.3
A complete partitioning of the input domain.

subdomain	domain predicate	subfunction
1.1.1	$x \in XEW$ and wordlength > 0 and linelength+wordlength ≥ MAXPOS	$f_{1.1.1}$
1.1.2	$x \in X_{EW}$ *and* wordlength > 0 and linelength+wordlength < MAXPOS	$f_{1.1.2}$
1.2	$x \in X_{EW}$ *and* wordlength ≤ 0	$f_{1.2}$
2.1	$x \in X_{ET}$ *and* wordlength+linelength ≥ MAXPOS	$f_{2.1}$
2.2	$x \in X_{ET}$ *and* wordlength+linelength < MAXPOS	$f_{2.2}$
3.1	$x \in X_{AN}$ *and* wordlength>MAXPOS	$f_{3.1}$
3.2	$x \in X_{AN}$ and wordlength≤MAXPOS	$f_{3.2}$

$f_{1.1.1}$ (in Table 3.3)

[The input character is either a blank or new-line]

write(newline);

linelength := wordlength;

write(word);

wordlength := 0;

read(char);

$f_{1.1.2}$ (in Table 3.3)

[The input character is either a blank or new-line]

```
write(blank);
linelength := linelength + wordlength;
write(word);
wordlength := 0;
read(char);
```

$f_{1.2}$ (in Table 3.3)

[The input character is either a blank or new-line which is redundant]

read(char);

$f_{2.1}$ (in Table 3.3)

[The input character is an end of the text mark and the current word is too long to be written on the current line]

write(*newline*);

write(*word*);

write(*char*);

exit;

$f_{2.2}$ (in Table 3.3)

[The input character is an end of the text mark and the current word can be written on the current line]

write(blank);

write(word);

write(char);

exit

$f_{3.1}$ (in Table 3.3)

[The input character is alphanumeric and the current word is too long]

append(char, word);

wordlength := wordlength + 1;

write(alarm);

exit;

$f_{3.2}$ (in Table 3.3)

[The input character is alphanumeric and the current word is not too long]

append(char, word);

wordlength := wordlength + 1;.

read(char);

Example 2 (continued)

Alternatively, necessary context for the program to determine what to do for a particular input character can also be provided by defining the input domain as a set of sequences of two characters instead of single characters. For example, consider Specification 3.2 again. Since X, the set of all possible input characters, can be partitioned into three subdomains, the set of sequences of two characters, XX, can be partitioned into $3 \times 3 = 9$ subdomains as shown below.

Example 2 (continued)

$$X = X_{AN} \cup X_{ET} \cup X_{EW}$$

$$\begin{aligned} XX &= (X_{AN} \cup X_{ET} \cup X_{EW})(X_{AN} \cup X_{ET} \cup X_{EW}) \\ &= X_{AN}X_{AN} \cup X_{AN}X_{ET} \cup X_{AN}X_{EW} \\ &\quad \cup X_{ET}X_{AN} \cup X_{ET}X_{ET} \cup X_{ET}X_{EW} \\ &\quad \cup X_{EW}X_{AN} \cup X_{EW}X_{ET} \cup X_{EW}X_{EW} \end{aligned}$$

Table 3.4 A trial partitioning of the input domain as a set of pairs.

subdomain	domain predicate	subfunction
1	$x_{i-1}x_i \in X_{AN}X_{AN}$	f_1
2	$x_{i-1}x_i \in X_{AN}X_{ET}$	f_2
3	$x_{i-1}x_i \in X_{AN}X_{EW}$	f_3
4	$x_{i-1}x_i \in X_{ET}X_{AN}$	f_4
5	$x_{i-1}x_i \in X_{ET}X_{ET}$	f_5
6	$x_{i-1}x_i \in X_{ET}X_{EW}$	f_6
7	$x_{i-1}x_i \in X_{EW}X_{AN}$	f_7
8	$x_{i-1}x_i \in X_{EW}X_{ET}$	f_8
9	$x_{i-1}x_i \in X_{EW}X_{EW}$	f_9

In the above, x_i is the current character and x_{i-1} is to the character read just before the current one.

f_1 (in Table 3.4)

```
[The current character is part of a new word]
append(char, word);
wordlength := wordlength + 1;
if wordlength > MAXPOS then
begin
      write(alarm);
      exit
end
else
      read(char);
```

f_2 (in Table 3.4)

[The current character marks the end of the text]

append(char, word);

if linelength + wordlength ≥ MAXPOS **then**

    write(newline);

write(word);

exit;

f_3 (in Table 3.4)

```
[The current character marks the end of a new
word]
if linelength + wordlength ≥ MAXPOS then
      begin
            write(newline);
            linelength := 0
      end
else
      write(blank);
write(word);
linelength := linelength + wordlength;
wordlength := 0;
read(char);
```

f_4, f_5, f_6 (in Table 3.4)

[The current character is redundant]

No reaction is expected from the program.

f_7 (in Table 3.4)

[The current character is the beginning of a new word]

append(char, word);

f_8 (in Table 3.4)

[The current character marks the end of the input text that has a space or new-line character at the end]

write(char);

exit;

f_9 (in Table 3.4)

[The current character is redundant]
read(char);

Example 2 (continued)

Subdomains 1, 2, and 3 need to be further partitioned to eliminate the need of using conditional statements in describing the subfunctions to be performed. By repeating the partitioning procedure illustrated above, we obtain Table 3.5.

Table 3.5 A complete partitioning of the input pairs.

Subdom.	domain predicate	Subfunc.
1.1	$x_{i-1}x_i \in X_{AN}X_{AN}$ *and* *wordlength* > *MAXPOS*	$f_{1.1}$
1.2	$x_{i-1}x_i \in X_{AN}X_{AN}$ *and* *wordlength* ≤ *MAXPOS*	$f_{1.2}$
2.1	$x_{i-1}x_i \in X_{AN}X_{ET}$ *and* *linelength*+*wordlength*≥*MAXPOS*	$f_{2.1}$
2.2	$x_{i-1}x_i \in X_{AN}X_{ET}$ *and* *linelength*+*wordlength*<*MAXPOS*	$f_{2.2}$
3.1	$x_{i-1}x_i \in X_{AN}X_{EW}$ *and* *linelength*+*wordlength*≥*MAXPOS*	$f_{3.1}$
3.2	$x_{i-1}x_i \in X_{AN}X_{EW}$ *and* *linelength*+*wordlength*<*MAXPOS*	$f_{3.2}$
4	$x_{i-1}x_i \in X_{ET}X_{AN}$	f_4
5	$x_{i-1}x_i \in X_{ET}X_{ET}$	f_5
6	$x_{i-1}x_i \in X_{ET}X_{EW}$	f_6
7	$x_{i-1}x_i \in X_{EW}X_{AN}$	f_7
8	$x_{i-1}x_i \in X_{EW}X_{ET}$	f_8
9	$x_{i-1}x_i \in X_{EW}X_{EW}$	f_9

$f_{1.1}$ (in Table 3.5)

[The current character is part of a new word which is too long]

append(char, word);

wordlength := wordlength + 1;

write(alarm);

exit;

$f_{1.2}$ (in Table 3.5)

[The current character is part of a new word of proper length]

append(char, word);

wordlength := wordlength + 1;

read(char);

$f_{2.1}$ (in Table 3.5)

[The current character marks the end of the text and the last word has to be written on the next line]

append(char, word);

write(newline);

write(word);

exit;

$f_{2.2}$ (in Table 3.5)

[The current character marks the end of the text and there is enough room on the current line to write the last word]

append(char, word);

write(word);

exit;

$f_{3.1}$ (in Table 3.5)

[The current character marks the end of a new word which has to be written on the next line]

```
write(newline);
write(word);
linelength := wordlength;
wordlength := 0;
read(char);
```

$f_{3.2}$ (in Table 3.5)

[The current character marks the end of a new word which can be written on the current line]

```
write(blank);
write(word);
linelength := linelength + wordlength;
wordlength := 0;
read(char);
```

f_4, f_5, f_6 (in Table 3.5)

[The current character is redundant]

No reaction is expected from the program.

f_7 (in Table 3.5)

[The current character is the beginning of a new word]

append(char, word);

wordlength := wordlength + 1;

read(char);

f_8 (in Table 3.5)

[The current character marks the end of the input text that has a space or new-line character at the end]

write(char);

exit;

f_9 (in Table 3.5)

[The current character is redundant]
read(char);

Example 2 (continued)

Based on the above analysis, to do a subfunction test on the program specified by Specification 3.2, therefore, is to test the program with a set of texts that satisfies all 12 domain predicates listed in Table 3.5.

Predicate testing

In subfunction testing, we find domain predicates that partition the input domain into a set of subdomains in which those subfunctions are defined, and then select one element from each of these subdomains. For example, if C_1 and C_2 are the domain predicates, there will be four possible combinations of these two domain predicates, viz., $C_1 \wedge C_2$, $C_1 \wedge \neg C_2$, $\neg C_1 \wedge C_2$, and $\neg C_1 \wedge \neg C_2$. To do subfunction testing is to test the program with one test case from each subdomain defined by $C_1 \wedge C_2$, $C_1 \wedge \neg C_2$, $\neg C_1 \wedge C_2$, and $\neg C_1 \wedge \neg C_2$.

Predicate testing (continued)

In predicate testing, the components to be exercised are domain predicates. We exercise each domain predicate with two test cases, one that makes the predicate true, and another that makes the predicate false. For instance, if we find two domain predicates C_1 and C_2 in the specification, to do predicate testing is to test the program with four test cases, each individually satisfying C_1, $\neg C_1$, C_2, and $\neg C_2$, respectively.

Predicate testing (continued)

Since the selection of test cases is done piecemeal, some combination of C_1 and C_2 may not have a representative element in the resulting set of test cases. For example, we might select x_1 and x_2 to satisfy C_1 and $\neg C_1$, and select x_3 and x_4 to satisfy C_2 and $\neg C_2$, respectively. If it turns out that x_1, x_2, x_3, and x_4 individually satisfy $C_1 \wedge C_2$, $\neg C_1 \wedge \neg C_2$, $\neg C_1 \wedge C_2$, $C_1 \wedge \neg C_2$, respectively, then it would be perfect. It would be just the same as doing a subfunction test.

Predicate testing (continued)

But since the selection of x_1 and x_2 are done independent of the selection of x_3 and x_4, we might end up selecting x1 that satisfies $C_1 \wedge C_2$, x_2 that satisfies $\neg C_1 \wedge \neg C_2$, x_3 that satisfies $C_1 \wedge C_2$, and x_4 that satisfies $\neg C_1 \wedge \neg C_2$. The net effect is that the four test cases would come from two of the four subdomains only!

Predicate testing (continued)

What is wrong with testing the program with four test cases from two of the four subdomains?

There is nothing to guarantee that any of the four pairs, (x_1, x_3), (x_1, x_4), (x_2, x_3), and (x_2, x_4) are loosely coupled computationally.

Predicate testing (continued)

In order to enhance the fault-discovery capability of the method, take all domain predicates into consideration when the test cases are being selected. Every time a test case is selected, make note of the subdomain to which it belongs, and avoid selecting a test case from any subdomain that is already represented, if all possible.

Predicate testing (continued)

The present method can be viewed as a counter-part of branch testing presented in the preceding chapter. If the counter part of any of the domain predicate identified in this method cannot be found in the source code, it should be investigated because it is an indication that the programmer might have neglected to implement some part of the program specification.

Boundary-value analysis

In this method the test cases are selected to exercise the limits or constraints imposed on the program input/output that can be found in, or derived from, the program specification.

Boundary-value analysis (continued)

In abstract, the input (output) domain is an infinite and multiple-dimensional space. In practice, it is partitioned into two major subdomains containing valid and invalid inputs (outputs).

Boundary-value analysis (continued)

In this method, a multiple of test cases are to be selected near or on the boundaries of each subdomain.

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

Boundary-value analysis (continued)

In particular

(1) If an input variable is defined in a range from the lower bound LB to the upper bound UB, use LB, UB, LB - δ, and UB + δ, where δ stands for the smallest value assumable by the variable, as the test cases.

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

Boundary-value analysis (continued)

(3) If the input or output of a program is a sequence, focus attention on the first and last element of the sequence, the sequence that is of zero/maximum length.

(4) Use one's ingenuity to search for additional boundary values.

Boundary-value analysis (continued)

The rationale: If the program is implemented properly and the operating system does what it is supposed to do while the program is being tested, the computational steps taken at LB, LB-δ, UB, and UB+δ of a datum used in the program should be significantly different than other points. That means the test cases selected at these points will be loosely coupled computationally. Hence the test cases so chosen will have an enhanced capacity for fault discovery.

Error guessing

We have been using the strategy of building an effective test set by adding to the set being constructed a new element that is loosely coupled to any other element already in the set. A new element is loosely coupled to other elements if it is to be processed by a different sequence of operations, or if it is located on, or near, the border of a domain as discussed in the preceding sections.

Error guessing (continued)

We avoid choosing two elements from the same subdomain. In general such inputs are computationally tightly coupled because they will be processed by the same sequence of operations. But if we know a subfunction well, we may be able to find two inputs that are not tightly coupled even if they belong to the same subdomain.

Error guessing (continued)

For example, if we know that the program input is a file and that file could be empty, that empty-file input will be loosely coupled to other non-empty inputs.

Error guessing (continued)

If we know that the arithmetic operation of division is included in computing a subfunction, we know that input that causes the divisor to become zero will be loosely coupled to other elements in the same subdomain.

Error guessing (continued)

If we know that the author of the program has the tendency to commit missed-by-one error in composing a loop construct, the input that causes a loop to iterate zero or one time will be loosely coupled to the others.

Error guessing (continued)

If we know that one of the subfunctions specified in the program specification has a mathematical pole at a certain point in the input domain, that input will be loosely coupled to others.

Error guessing (continued)

There are faults that can be found based on the knowledge about the skill or habit of the programmer, the programming language used, the programming environment in which the program will be developed and deployed, the intricacies in the application domain, and common sense about computer programming. We call that error-guessing if new test cases are found in the ways just mentioned.

Error guessing (continued)

For example, consider Specification 3.2 again. The tester may choose test cases to determine if the program will work correctly for

- an input text of length zero,
- a text containing no ENDOFTEXT character,
- a text containing a word of length MAXPOS,
- a text containing a very long word (of length greater than MAXPOS),
- a text containing nothing but BLANK's and NEWLINE's,

Error guessing (continued)

- 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.

Tools

Tools for the specification-based methods must have the capability to process the language in which the design/specification is written. Since such documents are usually written in a natural language, the required capability (of natural language processing) would be costly to acquire.

Tools (continued)

Furthermore, the specification of a program usually does not contain all information needed to complete the test-case selection process. Some of that has to be derived from the documents by inference or from the facts available from other sources.

The cost of building such tools can hardly be justified in most situations.

Tools (continued)

The analysis required to do test-case selection is similar to that required in doing detailed design.

The cost can be reduced by doing the test-case selection while doing the program design.

Automation

Generally speaking, automation of any of the specification-based test-case selection methods is difficult, and is not the most effective way to reduce the cost. It is the (software engineering) practice of building test cases whenever possible, and especially at the detail program design stage, that will lead to a significant reduction in the cost of testing.