Chapter 5

Boundary Value Testing

In Chapter 3, we saw that a function maps values from one set (its domain) to values in another set (its range) and that the domain and range can be cross-products of other sets. Any program can be considered to be a function in the sense that program inputs form its domain and program outputs form its range. Input domain testing is the best-known functional testing technique. In this and the next two chapters, we examine how to use knowledge of the functional nature of a program to identify test cases for the program. Historically, this form of testing has focused on the input domain, but it is often a good supplement to apply many of these techniques to develop range-based test cases.

5.1 Boundary Value Analysis

For the sake of comprehensible drawings, the discussion relates to a function, F, of two variables, x_1 and x_2. When the function F is implemented as a program, the input variables x_1 and x_2 will have some (possibly unstated) boundaries:

$$a \leq x_1 \leq b$$

$$c \leq x_2 \leq d$$

Unfortunately, the intervals [a, b] and [c, d] are referred to as the ranges of x_1 and x_2, so right away we have an overloaded term. The intended meaning will always be clear from its context. Strongly typed languages (such as Ada® and Pascal) permit explicit definition of such variable ranges. In fact, part of the historical reason for strong typing was to prevent programmers from making the kinds of errors that result in faults that are easily revealed by boundary value testing. Other languages (such as COBOL, Fortran, and C) are not strongly typed, so boundary value testing is more appropriate for programs coded in such languages. The input space (domain) of our function F is shown in Figure 5.1. Any point within the shaded rectangle is a legitimate input to the function F.

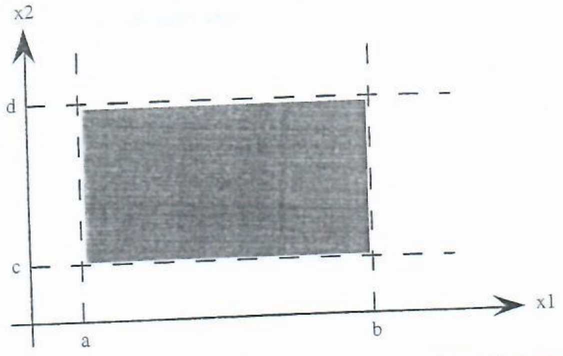

Figure 5.1 Input domain of a function of two variables.

Boundary value analysis focuses on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extreme values of an input variable. Loop conditions, for example, may test for $<$ when they should test for $\leq$, and counters are often "off by 1." The desktop publishing program with which this manuscript is written has an interesting boundary value problem. Two modes of textual display are used: one indicates new pages by a dotted line, and the other displays a page image showing where the text is placed on the page. If the cursor is at the last line of a page and new text is added, an anomaly occurs: in the first mode, the new line(s) simply appears, and the dotted line (page break) is adjusted. In the page display mode, however, the new text is lost—it does not appear on either page.

The basic idea of boundary value analysis is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and their maximum. A commercially available testing tool (originally named T) generates such test cases for a properly specified program. This tool has been successfully integrated with two popular front-end CASE tools (Teamwork from Cadre Systems, and Software through Pictures from Aonix, http://www.aonix.com/pdf/2140-AON.pdf). The T tool refers to these values as min, min+, nom, max–, and max. We will use these conventions here.

The next part of boundary value analysis is based on a critical assumption; it is known as the single fault assumption in reliability theory. This says that failures are only rarely the result of the simultaneous occurrence of two (or more) faults. Thus, the boundary value analysis test cases are obtained by holding the values of all but one variable at their nominal values, and letting that variable assume its extreme values. The boundary value analysis test cases for our function F of two variables (illustrated in Figure 5.2) are

$$\{<x_{1nom}, x_{2min}>, <x_{1nom}, x_{2min+}>, <x_{1nom}, x_{2nom}>, <x_{1nom}, x_{2max-}>, <x_{1nom}, x_{2max}>,$$
$$<x_{1min}, x_{2nom}>, <x_{1min+}, x_{2nom}>, <x_{1max-}, x_{2nom}>, <x_{1max}, x_{2nom}>\}$$

5.1.1 Generalizing Boundary Value Analysis

The basic boundary value analysis technique can be generalized in two ways: by the number of variables and by the kinds of ranges. Generalizing the number of variables is easy: if we have a function of n variables, we hold all but one at the nominal values and let the remaining variable assume the

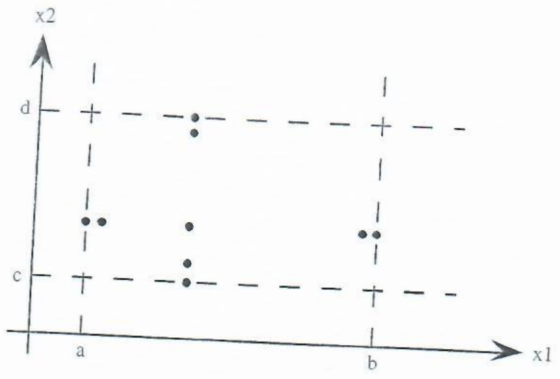

Figure 5.2 Boundary value analysis test cases for a function of two variables.

min, min+, nom, max–, and max values, repeating this for each variable. Thus, for a function of n variables, boundary value analysis yields 4n + 1 unique test cases.

Generalizing ranges depends on the nature (or more precisely, the type) of the variables themselves. In the NextDate function, for example, we have variables for the month, the day, and the year. In a Fortran-like language, we would most likely encode these, so that January would correspond to 1, February to 2, and so on. In a language that supports user-defined types (like Pascal or Ada), we could define the variable month as an enumerated type {Jan., Feb.,…, Dec.}. Either way, the values for min, min+, nom, max–, and max are clear from the context. When a variable has discrete, bounded values, as the variables in the commission problem have, the min, min+, nom, max–, and max are also easily determined. When no explicit bounds are present, as in the triangle problem, we usually have to create artificial bounds. The lower bound of side lengths is clearly 1 (a negative side length is silly), but what might we do for an upper bound? By default, the largest representable integer (called MAXINT in some languages) is one possibility, or we might impose an arbitrary upper limit such as 200 or 2000.

Boundary value analysis does not make much sense for Boolean variables; the extreme values are TRUE and FALSE, but no clear choice is available for the remaining three. We will see in Chapter 7 that Boolean variables lend themselves to decision table-based testing. Logical variables also present a problem for boundary value analysis. In the ATM example, a customer's PIN is a logical variable, as is the transaction type (deposit, withdrawal, or inquiry). We could "go through the motions" of boundary value analysis testing for such variables, but the exercise is not very satisfying to the tester's intuition.

5.1.2 Limitations of Boundary Value Analysis

Boundary value analysis works well when the program to be tested is a function of several independent variables that represent bounded physical quantities. The key words here are *independent* and *physical quantities*. A quick look at the boundary value analysis test cases for NextDate (in Section 5.5) shows them to be inadequate. Very little stress occurs on February and on leap years, for example. The real problem here is that interesting dependencies exist among the month, day, and year variables. Boundary value analysis presumes the variables to be truly independent. Even so, boundary value analysis happens to catch end-of-month and end-of-year faults. Boundary value analysis test cases are derived from the extrema of bounded, independent variables that refer to physical quantities, with no consideration of the nature of the function, or of the semantic meaning of the variables. We see boundary value analysis test cases to be rudimentary because they are obtained with very little insight and imagination. As with so many things, you get what you pay for.

The physical quantity criterion is equally important. When a variable refers to a physical quantity, such as temperature, pressure, air speed, angle of attack, load, and so forth, physical boundaries can be extremely important. (In an interesting example of this, Sky Harbor International Airport in Phoenix had to close on June 26, 1992, because the air temperature was 122°F. Aircraft pilots were unable to make certain instrument settings before take-off: the instruments could only accept a maximum air temperature of 120°F.) In another case, a medical analysis system uses stepper motors to position a carousel of samples to be analyzed. It turns out that the mechanics of moving the carousel back to the starting cell often causes the robot arm to miss the first cell.

As an example of logical (versus physical) variables, we might look at PINs or telephone numbers. It is hard to imagine what faults might be revealed by testing PIN values of 0000, 0001, 5000, 9998, and 9999.

5.2 Robustness Testing

Robustness testing is a simple extension of boundary value analysis: in addition to the five boundary value analysis values of a variable, we see what happens when the extrema are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min–). Robustness test cases for our continuing example are shown in Figure 5.3.

Most of the discussion of boundary value analysis applies directly to robustness testing, especially the generalizations and limitations. The most interesting part of robustness testing is not with the inputs, but with the expected outputs. What happens when a physical quantity exceeds its maximum? If it is the angle of attack of an airplane wing, the aircraft might stall. If it is the load capacity of a public elevator, we hope nothing special would happen. If it is a date, like May 32, we would expect an error message. The main value of robustness testing is that it forces attention on exception handling. With strongly typed languages, robustness testing may be very awkward. In Pascal, for example, if a variable is defined to be within a certain range, values outside that range result in runtime errors that abort normal execution. This raises an interesting question of implementation philosophy: Is it better to perform explicit range checking and use exception handling to deal with robust values, or is it better to stay with strong typing? The exception handling choice mandates robustness testing.

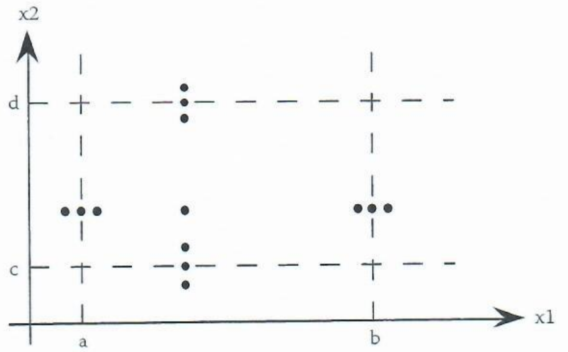

Figure 5.3 Robustness test cases for a function of two variables.

5.3 Worst-Case Testing

Boundary value analysis, as we said earlier, makes the single fault assumption of reliability theory. Rejecting this assumption means that we are interested in what happens when more than one variable has an extreme value. In electronic circuit analysis, this is called worst-case analysis; we use that idea here to generate worst-case test cases. For each variable, we start with the five-element set that contains the min, min+, nom, max–, and max values. We then take the Cartesian product (see Chapter 3) of these sets to generate test cases. The result of the two-variable version of this is shown in Figure 5.4.

Worst-case testing is clearly more thorough in the sense that boundary value analysis test cases are a proper subset of worst-case test cases. It also represents much more effort: worst-case testing for a function of n variables generates 5^n test cases, as opposed to $4n + 1$ test cases for boundary value analysis.

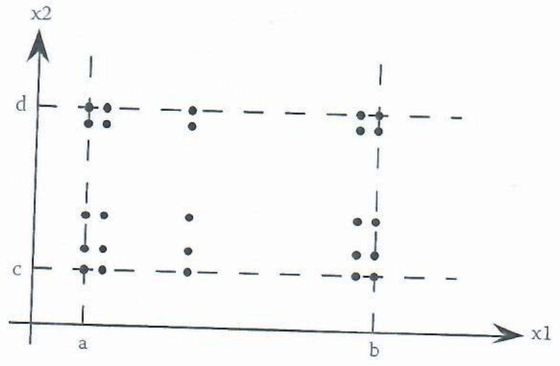

Figure 5.4 Worst-case test cases for a function of two variables.

Worst-case testing follows the generalization pattern we saw for boundary value analysis. It also has the same limitations, particularly those related to independence. Probably the best application for worst-case testing is where physical variables have numerous interactions, and where failure of the function is extremely costly. For really paranoid testing, we could go to robust worst-case testing. This involves the Cartesian product of the seven-element sets we used in robustness testing resulting in 7^n test cases. Figure 5.5 shows the robust worst-case test cases for our two variable functions.

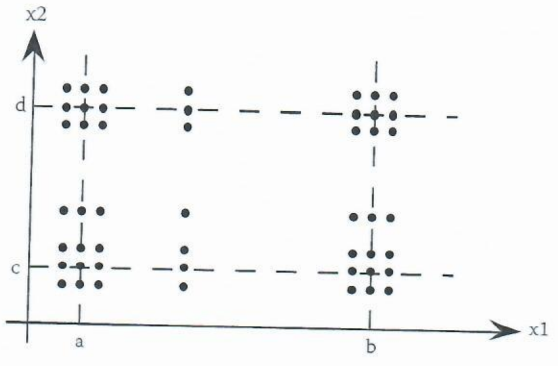

Figure 5.5 Robust worst-case test cases for a function of two variables.

5.4 Special Value Testing

Special value testing is probably the most widely practiced form of functional testing. It also is the most intuitive and least uniform. Special value testing occurs when a tester uses domain knowledge, experience with similar programs, and information about "soft spots" to devise test cases. We might also call this ad hoc testing or "seat-of-the-pants (or -skirt)" testing. No guidelines are used other than to use "best engineering judgment." As a result, special value testing is very dependent on the abilities of the tester.

Despite all the apparent negatives, special value testing can be very useful. In the next section, you will find test cases generated by the methods we just discussed for three of our examples. If you look carefully at these, especially for the NextDate function, you find that none is very satisfactory. If an interested tester defined special value test cases for NextDate, we would see several test cases involving February 28, February 29, and leap years. Even though special value testing is highly subjective, it often results in a set of test cases that is more effective in revealing faults than the test sets generated by the other methods we have studied—testimony to the craft of software testing.

5.5 Examples

Each of the three continuing examples is a function of three variables. Printing all the test cases from all the methods for each problem is very space consuming, so we have just selected examples.

Table 5.1 Triangle Problem Boundary Value Analysis Test Cases

Case	*a*	*b*	*c*	*Expected Output*
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a Triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a Triangle

lso is the
n knowl-
est cases.
elines are
y depen-

t section,
es. If you
isfactory.
test cases
ghly sub-
e test sets
g.

ases from
nples.

5.5.1 Test Cases for the Triangle Problem

In the problem statement, no conditions are specified on the triangle sides, other than being integers. Obviously, the lower bounds of the ranges are all 1. We arbitrarily take 200 as an upper bound. Table 5.1 contains boundary value test cases using these ranges. Notice that test cases 3, 8, and 13 are identical; two should be deleted.

Table 5.2 shows the worst case test cases in just "one corner" of the input space cube. (The remaining test cases are in the spread sheet for Exercise 5 at the end of this chapter.)

Table 5.2 Triangle Problem Worst-Case Test Cases

Case	*a*	*b*	*c*	*Expected Output*
1	1	1	1	Equilateral
2	1	1	2	Not a Triangle
3	1	1	100	Not a Triangle
4	1	1	199	Not a Triangle
5	1	1	200	Not a Triangle
6	1	2	1	Not a Triangle
7	1	2	2	Isosceles
8	1	2	100	Not a Triangle
9	1	2	199	Not a Triangle
10	1	2	200	Not a Triangle
11	1	100	1	Not a Triangle
12	1	100	2	Not a Triangle
13	1	100	100	Isosceles
14	1	100	199	Not a Triangle
15	1	100	200	Not a Triangle
16	1	199	1	Not a Triangle
17	1	199	2	Not a Triangle
18	1	199	100	Not a Triangle
19	1	199	199	Isosceles
20	1	199	200	Not a Triangle
21	1	200	1	Not a Triangle
22	1	200	2	Not a Triangle
23	1	200	100	Not a Triangle
24	1	200	199	Not a Triangle
25	1	200	200	Isosceles

5.5.2 Test Cases for the NextDate Function

Table 5.3 contains the worst case test cases for the NextDate function. As before only one corner of the input space cube B is shown.

Table 5.3 NextDate Worst-Case Test Cases

Case	Month	Day	Year	Expected Output
1	1	1	1812	January 2, 1812
2	1	1	1813	January 2, 1813
3	1	1	1912	January 2, 1912
4	1	1	2011	January 2, 2011
5	1	1	2012	January 2, 2012
6	1	2	1812	January 3, 1812
7	1	2	1813	January 3, 1813
8	1	2	1912	January 3, 1912
9	1	2	2011	January 3, 2011
10	1	2	2012	January 3, 2012
11	1	15	1812	January 16, 1812
12	1	15	1813	January 16, 1813
13	1	15	1912	January 16, 1912
14	1	15	2011	January 16, 2011
15	1	15	2012	January 16, 2012
16	1	30	1812	January 31, 1812
17	1	30	1813	January 31, 1813
18	1	30	1912	January 31, 1912
19	1	30	2011	January 31, 2011
20	1	30	2012	January 31, 2012
21	1	31	1812	February 1, 1812
22	1	31	1813	February 1, 1813
23	1	31	1912	February 1, 1912
24	1	31	2011	February 1, 2011
25	1	31	2012	February 1, 2012

5.5.3 Test Cases for the Commission Problem

Instead of going through 125 boring test cases again, we will look at some more interesting test cases for the commission problem. This time, we will look at boundary values for the output range, especially near the threshold points of $1000 and $1800. The output space of the commission is shown in Figure 5.6. The intercepts of these threshold planes with the axes are shown.

Table 5.4 Commission Problem Output Boundary Value Analysis Test Cases

Case	Locks	Stocks	Barrels	Sales	Commission	Comment
1	1	1	1	100	10	Output minimum
2	1	1	2	125	12.5	Output minimum +
3	1	2	1	130	13	Output minimum +
4	2	1	1	145	14.5	Output minimum +
5	5	5	5	500	50	Midpoint
6	10	10	9	975	97.5	Border point –
7	10	9	10	970	97	Border point –
8	9	10	10	955	95.5	Border point –
9	10	10	10	1000	100	Border point
10	10	10	11	1025	103.75	Border point +
11	10	11	10	1030	104.5	Border point +
12	11	10	10	1045	106.75	Border point +
13	14	14	14	1400	160	Midpoint
14	18	18	17	1775	216.25	Border point –
15	18	17	18	1770	215.5	Border point –
16	17	18	18	1755	213.25	Border point –
17	18	18	18	1800	220	Border point
18	18	18	19	1825	225	Border point +
19	18	19	18	1830	226	Border point +
20	19	18	18	1845	229	Border point +
21	48	48	48	4800	820	Midpoint
22	70	80	89	7775	1415	Output maximum –
23	70	79	90	7770	1414	Output maximum –
24	69	80	90	7755	1411	Output maximum –
25	70	80	90	7800	1420	Output maximum

Table 5.4 shows the worst case test cases in the output space for the commission problem. These test cases exercise the points at which the commission percentage changes.

The volume below the lower plane corresponds to sales below the $1000 threshold. The volume between the two planes is the 15% commission range. Part of the reason for using the output range to determine test cases is that cases from the input range are almost all in the 20% zone. We want to find input variable combinations that stress the boundary values: $100, $1000, $1800, and $7800. These test cases were developed with a spreadsheet, which saves a lot of calculator pecking. The minimum and maximum were easy, and the numbers happen to work out so that the border points are easy to generate. Here is where it gets interesting: test case 9 is the $1000 border point. If we tweak the input variables, we get values just below and just above the border (cases 6 to 8 and 10 to 12). If we wanted

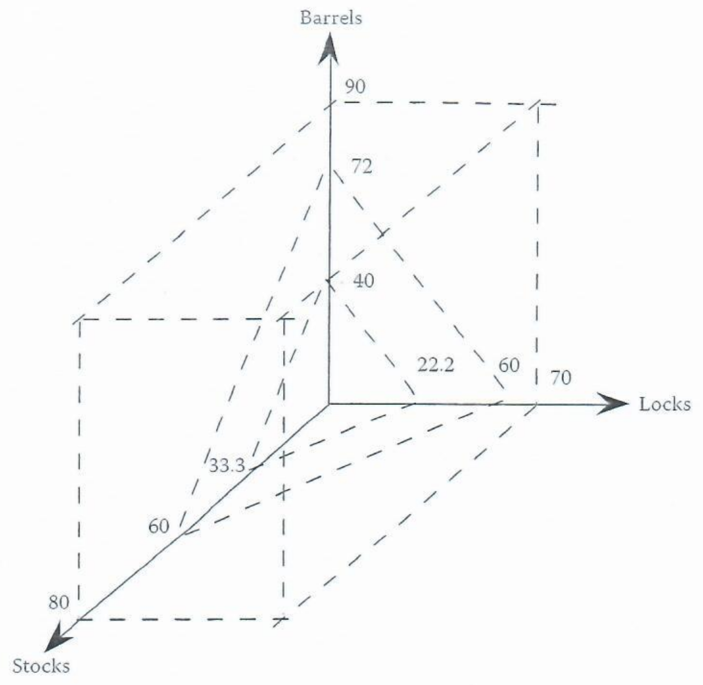

Figure 5.6 Input space of the commission problem.

to, we could pick values near the intercepts such as (22, 1, 1) and (21, 1, 1). As we continue in this way, we have a sense that we are exercising interesting parts of the code. We might claim that this is really a form of special value testing, because we used our mathematical insight to generate test cases.

Table 5.5 contains some typical special value test cases for the commission problem.

Table 5.5 Output Special Value Test Cases

Case	Locks	Stocks	Barrels	Sales	Commission	Comment
1	10	11	9	1005	100.75	Border point +
2	18	17	19	1795	219.25	Border point –
3	18	19	17	1805	221	Border point +

5.6 Random Testing

At least two decades of discussion of random testing are included in the literature. Most of this interest is among academics, and in a statistical sense, it is interesting. Our three sample problems lend themselves nicely to random testing. The basic idea is that, rather than always choose the min, min+, nom, max–, and max values of a bounded variable, use a random number

T

generato
question
coverage
results o
picks val

where th
generate
until at l
ended w
percenta
percenta

Table 5.6 Random Test Cases for the Triangle Program

Test Cases	*Nontriangles*	*Scalene*	*Isosceles*	*Equilateral*
1289	663	593	32	1
15436	7696	7372	367	1
17091	8556	8164	367	1
2603	1284	1252	66	1
6475	3197	3122	155	1
5978	2998	2850	129	1
9008	4447	4353	207	1
Percentage	49.83%	47.87%	2.29%	0.01%

generator to pick test case values. This avoids a form of bias in testing. It also raises a serious question: How many random test cases are sufficient? Later, when we discuss structural test coverage metrics, we will have an elegant answer. For now, Tables 5.6 to Table 5.8 show the results of randomly generated test cases. They are derived from a Visual Basic application that picks values for a bounded variable $a \le x \le b$ as follows:

$$x = \text{Int}((b - a + 1) * \text{Rnd} + a)$$

where the function Int returns the integer part of a floating point number, and the function Rnd generates random numbers in the interval [0, 1]. The program keeps generating random test cases until at least one of each output occurs. In each table, the program went through seven cycles that ended with the hard-to-generate test case. In Table 5.6 and Table 5.7, the last line shows what percentage of the random test cases was generated for each column. In the table for NextDate, the percentages are very close to the computed probability given in the last line of Table 5.8.

Table 5.7 Random Test Cases for the Commission Program

Test Cases	*10%*	*15%*	*20%*
91	1	6	84
27	1	1	25
72	1	1	70
176	1	6	169
48	1	1	46
152	1	6	145
125	1	4	120
Percentage	1.01%	3.62%	95.37%

Table 5.8 Random Test Cases for the NextDate Program

Test Cases	*Days 1–30 of 31-Day Months*	*Day 31 of 31-Day Months*	*Days 1–29 of 30-Day Months*	*Days 30 of 30-Day Months*
913	542	17	274	10
1101	621	9	358	8
4201	2448	64	1242	46
1097	600	21	350	9
5853	3342	100	1804	82
3959	2195	73	1252	42
1436	786	22	456	13
Percentage	56.76%	1.65%	30.91%	1.13%
Probability	56.45%	1.88%	31.18%	1.88%

Days 1–27 of Feb.	*Feb. 28 of a Leap Year*	*Feb. 28 of a Non-Leap Year*	*Feb. 29 of a Leap Year*	*Impossible Days*
45	1	1	1	22
83	1	1	1	19
312	1	8	3	77
92	1	4	1	19
417	1	11	2	94
310	1	6	5	75
126	1	5	1	26
7.46%	0.04%	0.19%	0.08%	1.79%
7.26%	0.07%	0.20%	0.07%	1.01%

5.7 Guidelines for Boundary Value Testing

With the exception of special value testing, the test methods based on the input domain of a function (program) are the most rudimentary of all functional testing methods. They share the common assumption that the input variables are truly independent, and when this assumption is not warranted, the methods generate unsatisfactory test cases (such as June 31, 1912, for NextDate). These methods have two other distinctions: normal versus robust values, and the single fault versus the multiple fault assumption. Just using these distinctions carefully will result in better testing. Each of these methods can be applied to the output range of a program, as we did for the commission problem.

Another useful form of output-based test cases is for systems that generate error messages. The tester should devise test cases to check that error messages are generated when they are appropriate, and are not falsely generated. Boundary value analysis can also be used for internal variables, such as loop control variables, indices, and pointers. Strictly speaking, these are not input variables, but errors in the use of these variables are quite common. Robustness testing is a good choice for testing internal variables.