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. 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. Input domain testing (also called "boundary value testing") is the best-known specification-based testing technique. Historically, this form of testing has focused on the input domain; however, it is often a good supplement to apply many of these techniques to develop range-based test cases.

There are two independent considerations that apply to input domain testing. The first asks whether or not we are concerned with invalid values of variables. Normal boundary value testing is concerned only with valid values of the input variables. Robust boundary value testing considers invalid and valid variable values. The second consideration is whether we make the "single fault" assumption common to reliability theory. This assumes that faults are due to incorrect values of a single variable. If this is not warranted, meaning that we are concerned with interaction among two or more variables, we need to take the cross product of the individual variables. Taken together, the two considerations yield four variations of boundary value testing:

- Normal boundary value testing
- Robust boundary value testing
- Worst-case boundary value testing
- Robust worst-case boundary value testing

For the sake of comprehensible drawings, the discussion in this chapter refers 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:

$$a \le x_1 \le b$$

$$c \le x_2 \le d$$

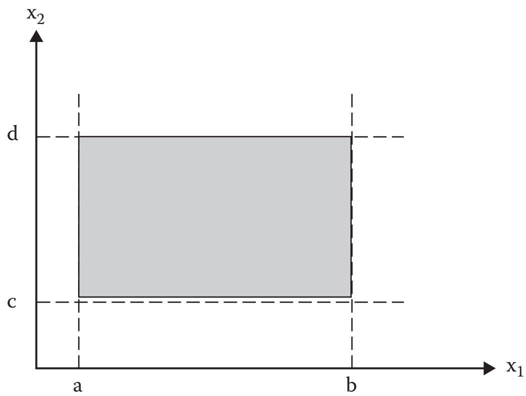

Figure 5.1 Input domain of a function of two variables.

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 these languages. The input space (domain) of our function F is shown in Figure 5.1. Any point within the shaded rectangle and including the boundaries is a legitimate input to the function F.

5.1 Normal Boundary Value Testing

All four forms of boundary value testing focus 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 ≤, and counters often are "off by one." (Does counting begin at zero or at one?) 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 at 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 [part of Atego]; for more information, see http://www.aonix.com/pdf/2140-AON.pdf). The T tool refers to these values as min, min+, nom, max–, and max. The robust forms add two values, min– and max+.

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. The All Pairs testing approach (described in Chapter 20) contradicts this, with the observation that, in software-controlled medical systems, almost all faults are the result of interaction between a pair of variables. Thus, the normal and robust variations cases are obtained by holding the values of all but one variable at their nominal

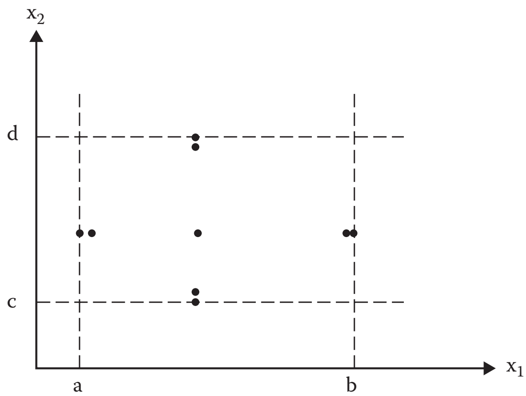

Figure 5.2 Boundary value analysis test cases for a function of two variables.

values, and letting that variable assume its full set of test values. The normal 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 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. For other data types, as long as a variable supports an ordering relation (see Chapter 3 for a definition), we can usually infer the min, min+, nominal, max–, and max values. Test values for alphabet characters, for example, would be {a, b, m, y, and z}.

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. Mathematically, the variables need to be described by a true ordering relation, in which, for every pair <a, b> of values of a variable, it is possible to say that a ≤ b or b ≤ a. (See Chapter 3 for a detailed definition of ordering relations.) Sets of car colors, for example, or football teams, do not support an ordering relation; thus, no form of boundary value testing is appropriate for such variables. 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. 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, nor 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 takeoff: 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 (vs. 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 Robust Boundary Value Testing

Robust boundary value testing is a simple extension of normal boundary value testing: 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–). Robust boundary value 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

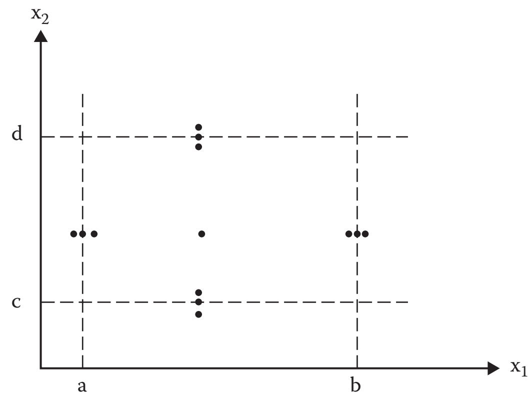

Figure 5.3 Robustness test cases for a function of two variables.

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 run-time 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.

5.3 Worst-Case Boundary Value Testing

Both forms of boundary value testing, as we said earlier, make the single fault assumption of reliability theory. Owing to their similarity, we treat both normal worst-case boundary testing and robust worst-case boundary testing in this subsection. Rejecting single-fault 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 boundary value 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.

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 function.

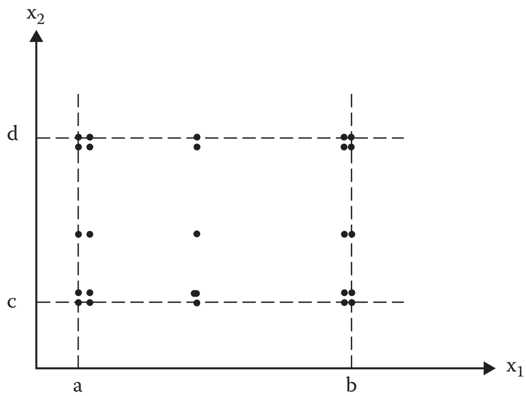

Figure 5.4 Worst-case test cases for a function of two variables.

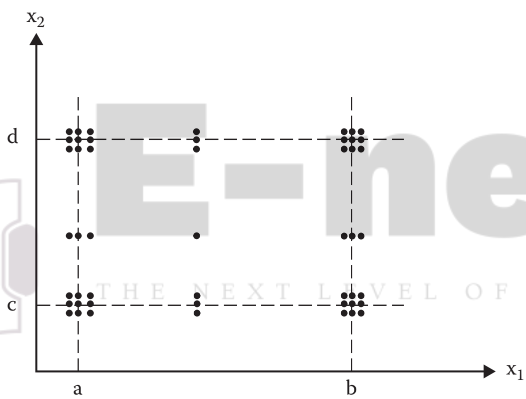

Figure 5.5 Robust worst-case test cases for a function of two variables.

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 the 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. No guidelines are used other than "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. Special value test cases for NextDate will include 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 boundary value methods—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 just have selected examples for worst-case boundary value and robust worst-case boundary value testing.

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. For each side, the test values are {1, 2, 100, 199, 200}. Robust boundary value test cases will add {0, 201}. 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. Further, there is no test case for scalene triangles.

The cross-product of test values will have 125 test cases (some of which will be repeated)—too many to list here. The full set is available as a spreadsheet in the set of student exercises. Table 5.2 only lists the first 25 worst-case boundary value test cases for the triangle problem. You can picture them as a plane slice through the cube (actually it is a rectangular parallelepiped) in which a = 1 and the other two variables take on their full set of cross-product values.

Table 5.1 Normal Boundary Value 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

Table 5.2 (Selected) Worst-Case Boundary Value 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

All 125 worst-case test cases for NextDate are listed in Table 5.3. Take some time to examine it for gaps of untested functionality and for redundant testing. For example, would anyone actually want to test January 1 in five different years? Is the end of February tested sufficiently?

Table 5.3 Worst-Case Test Cases

Case	*Month*	*Day*	*Year*	*Expected Output*
1	1	1	1812	1, 2, 1812
2	1	1	1813	1, 2, 1813
3	1	1	1912	1, 2, 1912
4	1	1	2011	1, 2, 2011
5	1	1	2012	1, 2, 2012
6	1	2	1812	1, 3, 1812
7	1	2	1813	1, 3, 1813
8	1	2	1912	1, 3, 1912
9	1	2	2011	1, 3, 2011
10	1	2	2012	1, 3, 2012
11	1	15	1812	1, 16, 1812
12	1	15	1813	1, 16, 1813
13	1	15	1912	1, 16, 1912
14	1	15	2011	1, 16, 2011
15	1	15	2012	1, 16, 2012
16	1	30	1812	1, 31, 1812
17	1	30	1813	1, 31, 1813
18	1	30	1912	1, 31, 1912
19	1	30	2011	1, 31, 2011
20	1	30	2012	1, 31, 2012
21	1	31	1812	2, 1, 1812
22	1	31	1813	2, 1, 1813
23	1	31	1912	2, 1, 1912
24	1	31	2011	2, 1, 2011
25	1	31	2012	2, 1, 2012
26	2	1	1812	2, 2, 1812
27	2	1	1813	2, 2, 1813
28	2	1	1912	2, 2, 1912

(*continued*)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
29	2	1	2011	2, 2, 2011
30	2	1	2012	2, 2, 2012
31	2	2	1812	2, 3, 1812
32	2	2	1813	2, 3, 1813
33	2	2	1912	2, 3, 1912
34	2	2	2011	2, 3, 2011
35	2	2	2012	2, 3, 2012
36	2	15	1812	2, 16, 1812
37	2	15	1813	2, 16, 1813
38	2	15	1912	2, 16, 1912
39	2	15	2011	2, 16, 2011
40	2	15	2012	2, 16, 2012
41	2	30	1812	Invalid date
42	2	30	1813	Invalid date
43	2	30	1912	Invalid date
44	2	30	2011	Invalid date
45	2	30	2012	Invalid date
46	2	31	1812	Invalid date
47	2	31	1813	Invalid date
48	2	31	1912	Invalid date
49	2	31	2011	Invalid date
50	2	31	2012	Invalid date
51	6	1	1812	6, 2, 1812
52	6	1	1813	6, 2, 1813
53	6	1	1912	6, 2, 1912
54	6	1	2011	6, 2, 2011
55	6	1	2012	6, 2, 2012
56	6	2	1812	6, 3, 1812
57	6	2	1813	6, 3, 1813

(continued)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
58	6	2	1912	6, 3, 1912
59	6	2	2011	6, 3, 2011
60	6	2	2012	6, 3, 2012
61	6	15	1812	6, 16, 1812
62	6	15	1813	6, 16, 1813
63	6	15	1912	6, 16, 1912
64	6	15	2011	6, 16, 2011
65	6	15	2012	6, 16, 2012
66	6	30	1812	7, 1, 1812
67	6	30	1813	7, 1, 1813
68	6	30	1912	7, 1, 1912
69	6	30	2011	7, 1, 2011
70	6	30	2012	7, 1, 2012
71	6	31	1812	Invalid date
72	6	31	1813	Invalid date
73	6	31	1912	Invalid date
74	6	31	2011	Invalid date
75	6	31	2012	Invalid date
76	11	1	1812	11, 2, 1812
77	11	1	1813	11, 2, 1813
78	11	1	1912	11, 2, 1912
79	11	1	2011	11, 2, 2011
80	11	1	2012	11, 2, 2012
81	11	2	1812	11, 3, 1812
82	11	2	1813	11, 3, 1813
83	11	2	1912	11, 3, 1912
84	11	2	2011	11, 3, 2011
85	11	2	2012	11, 3, 2012
86	11	15	1812	11, 16, 1812

(*continued*)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
87	11	15	1813	11, 16, 1813
88	11	15	1912	11, 16, 1912
89	11	15	2011	11, 16, 2011
90	11	15	2012	11, 16, 2012
91	11	30	1812	12, 1, 1812
92	11	30	1813	12, 1, 1813
93	11	30	1912	12, 1, 1912
94	11	30	2011	12, 1, 2011
95	11	30	2012	12, 1, 2012
96	11	31	1812	Invalid date
97	11	31	1813	Invalid date
98	11	31	1912	Invalid date
99	11	31	2011	Invalid date
100	11	31	2012	Invalid date
101	12	1	1812	12, 2, 1812
102	12	1	1813	12, 2, 1813
103	12	1	1912	12, 2, 1912
104	12	1	2011	12, 2, 2011
105	12	1	2012	12, 2, 2012
106	12	2	1812	12, 3, 1812
107	12	2	1813	12, 3, 1813
108	12	2	1912	12, 3, 1912
109	12	2	2011	12, 3, 2011
110	12	2	2012	12, 3, 2012
111	12	15	1812	12, 16, 1812
112	12	15	1813	12, 16, 1813
113	12	15	1912	12, 16, 1912
114	12	15	2011	12, 16, 2011
115	12	15	2012	12, 16, 2012

(*continued*)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
116	12	30	1812	12, 31, 1812
117	12	30	1813	12, 31, 1813
118	12	30	1912	12, 31, 1912
119	12	30	2011	12, 31, 2011
120	12	30	2012	12, 31, 2012
121	12	31	1812	1, 1, 1813
122	12	31	1813	1, 1, 1814
123	12	31	1912	1, 1, 1913
124	12	31	2011	1, 1, 2012
125	12	31	2012	1, 1, 2013

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 derived from the output range, especially near the threshold points of \$1000 and \$1800 where the commission percentage changes. The output space of the commission is shown in Figure 5.6. The intercepts of these threshold planes with the axes are shown.

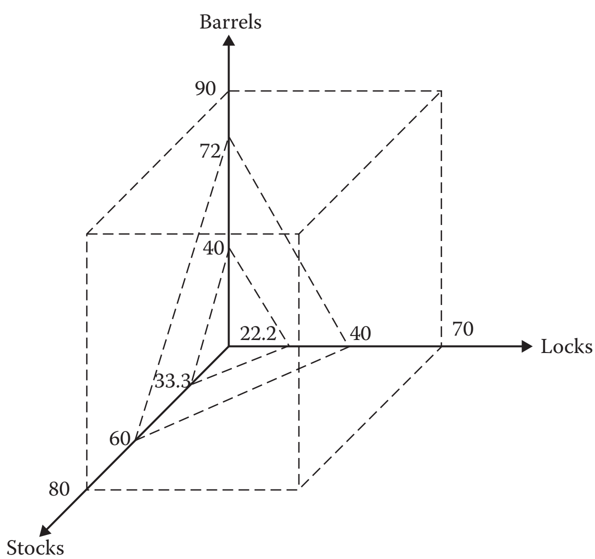

Figure 5.6 Input space of the commission problem.

Table 5.4 Output Boundary Value Analysis Test Cases

Case	*Locks*	*Stocks*	*Barrels*	*Sales*	*Comm*	*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

The volume between the origin and 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 sales/commission boundary values: $100, $1000, $1800, and $7800. The minimum and maximum were easy, and

Table 5.5 Output Special Value Test Cases

Case	*Locks*	*Stocks*	*Barrels*	*Sales*	*Comm*	*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 +

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 6–8 and 10–12). If we wanted to, we could pick values near the borders such as (22, 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 5.4 contains test cases derived from boundary values on the output side of the commission function. Table 5.5 contains special value test cases.

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 generator to pick test case values. This avoids any 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 through 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 $a \leq x \leq b$ as follows:

Table 5.6 Random Test Cases for Triangle Program

Test Cases	*Nontriangles*	*Scalene*	*Isosceles*	*Equilateral*
1289	663	593	32	1
15,436	7696	7372	367	1
17,091	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%

Table 5.7 Random Test Cases for 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%

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

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 Tables 5.6 and 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.

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 specification-based 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). 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; however, errors in the use of these variables are quite common. Robustness testing is a good choice for testing internal variables.

There is a discussion in Chapter 10 about "the testing pendulum"—it refers to the problem of syntactic versus semantic approaches to developing test cases. Here is a short example given both ways. Consider a function F of three variables, a, b, and c. The boundaries are $0 \leq a < 10{,}000$, $0 \leq b < 10{,}000$, and $0 \leq c < 18.8$. The function F is $F = (a - b)/c$; Table 5.9 shows the normal boundary value test cases. Absent semantic knowledge, the first four test cases in Table 5.9 are what a boundary value testing tool would generate (a tool would not generate the expected output values). Even just the syntactic version is problematic—it does not avoid the division by zero possibility in test case 11.

Table 5.8 Random Test Cases for NextDate Program

Test Cases	*Days 1–30 of 31-Day Months*	*Day 31 of 31-Day Months*	*Days 1–29 of 30-Day Months*	*Day 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%

When we add the semantic information that F calculates the miles per gallon of an automobile, where a and b are end and start trip odometer values, and c is the gas tank capacity, we see more severe problems:

1. We must always have a ≥ b. This will avoid the negative values of F (test cases 1, 2, 9, and 10).
2. Test cases 3, 8, and 12–15 all refer to trips of length 0, so they could be collapsed into one test case, probably test case 8.
3. Division by zero is an obvious problem, thereby eliminating test case 11. Applying the semantic knowledge will result in the better set of case cases in Table 5.10.
4. Table 5.10 is still problematic—we never see the effect of boundary values on the tank capacity.

Table 5.9 Normal Boundary Value Test Cases for F = (a – b)/c

Test Case	a	b	c	F
1	0	5000	9.4	–531.9
2	1	5000	9.4	–531.8
3	5000	5000	9.4	0.0
4	9998	5000	9.4	531.7
5	9999	5000	9.4	531.8
6	5000	0	9.4	531.9
7	5000	1	9.4	531.8
8	5000	5000	9.4	0.0
9	5000	9998	9.4	–531.7
10	5000	9999	9.4	–531.8
11	5000	5000	0	Undefined
12	5000	5000	1	0.0
13	5000	5000	9.4	0.0
14	5000	5000	18.7	0.0
15	5000	5000	18.8	0.0

Table 5.10 Semantic Boundary Value Test Cases for F = (a – b)/c

Test Case	*End Odometer*	*Start Odometer*	*Tank Capacity*	*Miles per Gallon*
4	9998	5000	9.4	531.7
5	9999	5000	9.4	531.8
6	5000	0	9.4	531.9
7	5000	1	9.4	531.8
8	5000	5000	9.4	0.0

EXERCISES

1. Develop a formula for the number of robustness test cases for a function of n variables.
2. Develop a formula for the number of robust worst-case test cases for a function of n variables.
3. Make a Venn diagram showing the relationships among test cases from boundary value analysis, robustness testing, worst-case testing, and robust worst-case testing.
4. What happens if we try to do output range robustness testing? Use the commission problem as an example.

5. If you did exercise 8 in Chapter 2, you are already familiar with the CRC Press website for downloads (http://www.crcpress.com/product/isbn/9781466560680). There you will find an Excel spreadsheet named specBasedTesting.xls. (It is an extended version of Naive.xls, and it contains the same inserted faults.) Different sheets contain worst-case boundary value test cases for the triangle, NextDate, and commission problems, respectively. Run these sets of test cases and compare the results with your naive testing from Chapter 2.
6. Apply special value testing to the miles per gallon example in Tables 5.9 and 5.10. Provide reasons for your chosen test cases.