Chapter 12

Test Generation from Combinatorial Designs

The purpose of this chapter is to introduce techniques for the generation of test configurations and test data using the combinatorial design techniques with program inputs and their values as, respectively, factors and levels. These techniques are useful when testing a variety of applications. They allow selection of a small set of test configurations from an often impractically large set, and are effective in detecting faults arising out of factor interactions.

12.1. Combinatorial designs

Software applications are often designed to work in a variety of environments. Combinations of factors such as the operating system, network connection, and hardware platform, lead to a variety of environments. Each environment corresponds to a given set of values for each factor, known as a *test configuration*. For example, Windows XP, Dial-up connection, and a PC with 512 MB of main memory, is one possible configuration. To ensure high reliability across the intended environments, the application must be tested under as many test configurations, or environments, as possible. However, as illustrated in examples later in this chapter, the number of such test configurations could be exorbitantly large making it impossible to test the application exhaustively.

An analogous situation arises in the testing of programs that have one or more input variables. Each test run of a program often requires at least one value for each variable. For example, a program to find the greatest common divisor of two integers x and y requires two values, one corresponding to x and the other to y. In earlier chapters we have seen how program inputs can be selected using techniques such as equivalence partitioning and boundary value analysis. While these techniques offer a set of guidelines to design test cases, they suffer from two shortcomings: (a) they raise the possibility of a large number of sub-domains in the

partition of the input space and (b) they lack guidelines on how to select inputs from various sub-domains in the partition.

The number of sub-domains in a partition of the input domain increases in direct proportion to the number and type of input variables, and especially so when multidimensional partitioning is used. Also, once a partition is determined, one selects at random a value from each of the sub-domains. Such a selection procedure, especially when using uni-dimensional equivalence partitioning, does not account for the possibility of faults in the program under test that arise due to specific interactions amongst values of different input variables. While boundary value analysis leads to the selection of test cases that test a program at the boundaries of the input domain, other interactions in the input domain might remain untested.

This chapter describes several techniques for generating test configurations or test sets that are small even when the set of possible configurations, or the input domain, and the number of sub-domains in its partition, is large and complex. The number of test configurations, or the test set so generated, has been found to be effective in the discovery of faults due to the interaction of various input variables. The techniques we describe here are known by several names such as design of experiments, combinatorial designs, orthogonal designs, interaction testing, and pairwise testing.

12.1.1. Test configuration and test set

In this chapter we use the terms *test configuration* and *test set* interchangeably. Even though we use the terms interchangeably, they do have different meaning in the context of software testing. However, the techniques described in this chapter apply the generation of both test configurations as well as test sets, we have taken the liberty of using them interchangeably. One must be aware that a test configuration is usually a static selection of factors, such as the hardware platform or an operating system. Such selection is completed prior to the start of the test. In contrast, a test set is a collection of test cases used as input during the test process.

12.1.2. Modeling the input and configuration spaces

The input space of a program P consists of k-tuples of values that could be input to P during execution. The configuration space of P consists of all possible settings of the environment variables under which P could be used.

EXAMPLE 12.1. Consider program P that takes two integers $x > 0$ and $y > 0$ as inputs. The input space of P is the set of all pairs of positive non-zero integers. Now suppose that this program is intended to be executed under the Windows and the Mac OS operating system, through the Netscape or Safari browsers, and must be able to print to a local or a networked printer. The configuration space of P consists of triples (X, Y, Z) where X represents an operating system, Y a browser, and Z a local or a networked printer. ■

Next, consider a program P that takes n inputs corresponding to variables $X_1, X_2, \ldots, X_n$. We refer to the inputs as *factors*. The inputs are also referred to as *test parameters* or as *values*. Let us assume that each factor may be set at any one from a total of $c_i, 1 \leq i \leq n$ values. Each value assignable to a factor is known as a *level*. The notation $\mid F \mid$ refers to the number of levels for factor F.

The environment under which an application is intended to be used, generally contributes one or more factors. In Example 12.1, the operating system, browser, and printer connection are three factors that will likely affect the operation and performance of P.

A set of values, one for each factor, is known as a *factor combination*. For example, suppose that program P has two input variables X and Y. Let us say that during an execution of P, X and Y may each assume a value from the set $\{a, b, c\}$ and $\{d, e, f\}$, respectively. Thus we have 2 factors and 3 levels for each factor. This leads to a total of $3^2 = 9$ factor combinations, namely $(a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, d), (c, e)$, and (c, f). In general, for k factors with each factor assuming a value from a set of n values, the total number of factor combinations is n^k.

Suppose now that each factor combination yields one test case. For many programs, the number of tests generated for exhaustive testing could be exorbitantly large. For example, if a program has 15 factors with 4 levels each, the total number of tests is $4^{15} \approx 10^9$. Executing a billion tests might be impractical for many software applications.

There are special combinatorial design techniques that enable the selection of a small subset of factor combinations from the complete set. This sample is targeted at specific types of faults known as *interaction* faults. Before we describe how the combinatorial designs are generated, let us look at a few examples that illustrate where they are useful.

EXAMPLE 12.2. Let us model the input space of an online Pizza Delivery Service (PDS) for the purpose of testing. The service takes orders online, checks for their validity, and schedules Pizza for delivery. A customer is required to specify the following four items as part of the online order: Pizza size, Toppings list, Delivery address, and a home phone number. Let us denote these four factors by `S`, `T`, `A`, and `P`, respectively.

Suppose now that there are three varieties for size: `Large`, `Medium`, and `Small`. There is a list of 6 toppings from which to select. In addition, the customer can customize the toppings. The delivery address consists of customer name, one line of address, city, and the zip code. The phone number is a numeric string possibly containing the dash ("–") separator.

The table below lists one model of the input space for the PDS. Note that while for `Size` we have selected all three possible levels, we have constrained the other factors to a smaller set of levels. Thus we are concerned with only one of two types of values for `Toppings`, Custom or Preset, and one of the two types of values for factors `Address` and `Phone`, namely `Valid` and `Invalid`.

Factor	Levels		
`Size`	Large	Medium	Small
`Toppings`	Custom	Preset	
`Address`	Valid	Invalid	
`Phone`	Valid	Invalid	

The total number of factor combinations is $2^4 + 2^3 = 24$. However, as an alternate to the table above, we could consider $6 + 1 = 7$ levels for `Toppings`. This would increase the number of combinations to $2^4{+}5{\times}2^3{+}2^3{+}5{\times}2^2 = 84$. We could also consider additional types of values for `Address` and `Phone` which would further increase the number of distinct combinations. Notice that if we were to consider each possible valid and invalid string of characters, limited only by length, as a level for `Address`, we will arrive at a huge number of factor combinations. ■

Table 12.1: Factors and levels for the Unix `sort` utility.

Factor	Meaning	Levels			
-	Forces the source to be the standard input.	Unused	Used		
-c	Verify that the input is sorted according to the options specified on the command line.	Unused	Used		
-m	Merge sorted input	Unused	Used		
-u	Suppress all but one of the matching keys.	Unused	Used		
-o *output*	Output sent to a file.	Unused	Valid file	Invalid file	
-T*directory*	Temporary directory for sorting.	Unused	Exists	Does not exist	
-y*kmem*	Use *kmem* kilobytes of memory for sorting.	Unused	Valid *kmem*	Invalid *kmem*	
-zrecsize	Specfies record size to hold each line from the input file.	Unused	Zero size	Large size	
-d*fiMnr*	Perform dictionary sort	Unused	fi	Mnr	fiMnr

Later in this section we explain the advantages and disadvantages of limiting the number of factor combinations by partitioning the set of values for each factor into a few subsets. Notice also the similarity of this approach with equivalence partitioning. The next example illustrates factors in a GUI.

EXAMPLE 12.3. The Graphical User Interface of application T consists of three menus labeled `File`, `Edit`, and `Format`. Each menu contains several items listed below.

Factor	Levels			
`File`	New	Open	Save	Close
`Edit`	Cut	Copy	Paste	Select
`Typeset`	LaTex	BibTex	PlainTeX	MakeIndex

We have three factors in T. Each of these three factors can be set to any of four levels. Thus we have a total $4^3 = 64$ factor combinations.

Note that each factor in this example corresponds to a relatively smaller set of levels when compared to factors `Address` and `Phone` in the previous example. Hence the number of levels for each factor is set equal to the cardinality of the set of the corresponding values. ■

EXAMPLE 12.4. Let us now consider the Unix `sort` utility for sorting ASCII data in files or obtained from the standard input. The utility has several options and makes an interesting example for the identification of factors and levels. The command line for `sort` is as given below.

`sort` [-cmu] [-o output] [-T directory] [-y [kmem]] [-z recsz] [-dfiMnr] [- b] [t char] [-k keydef] [+pos1 [-pos2]] [file...]

Table 12.2: Factors and levels for the Unix `sort` utility (continued).

Factor	Meaning	Levels			
-f	Ignore case.	Unused	Used		
-i	Ignore non-ASCII characters.	Unused	Used		
-M	Fields are compared as months.	Unused	Used		
-n	Sort input numerically.	Unused	Used		
-r	Reverse the output order.	Unused	Used		
-b	Ignore leading blanks when using +pos1 and -pos2.	Unused	Used		
-tc	Use character c as field separator.	Unused	c_1	$c_1\,c_2$	
-k*keydef*	Restricted sort key definition.	Unused	start	end	start*type*
+*pos1*	Start position in the input line for comparing fields.	Unused	f.c	f	0.c
-*pos2*	End position for comparing fields.	Unused	f.c	f	0.c
file	File to be sorted.	Not specified	Exists	Does not exist	

Tables 12.1 and 12.2 list all the factors of `sort` and their corresponding levels. Note that the levels have been derived using equivalence partitioning for each option and are not unique. We have decided to limit the number of levels for each factor to 4. You could come up with a different, and possibly a larger or a smaller, set of levels for each factor.

In Table **??** level *Unused* indicates that the corresponding option is not used while testing the `sort` command. *Used* means that the option is used. Level *Valid File* indicates that the file specified using the `-o` option exists whereas *Invalid File* indicates that the specified file does not exist. Other options can be interpreted similarly.

We have identified a total of 20 factors for the `sort` command. The levels listed in Table **??** lead to a total of approximately 1.9×10^9 combinations. ■

EXAMPLE 12.5. There is often a need to test a web application on different platforms to ensure that any claim such as "Application X can be used under Windows and OS X." Here we consider a combination of hardware, operating system, and a browser as a platform. Such testing is commonly referred to as *compatibility* testing.

Let us identify factors and levels needed in the compatibility testing of application X. Given that we would like X to work on a variety of hardware, OS, and browser combinations, it is easy to obtain three factors, i.e. hardware, OS, and browser. These are listed in the top row of Table 12.3. Notice that instead of listing factors in different rows, we now list them in different columns. The levels for each factor are listed in rows under the corresponding columns. This has been done to simply the formatting of the table.

A quick examination of the factors and levels in Table 12.3 reveals that there are 75 factor combinations. However, some of these combinations are infeasible. For example, OS 10.2 is an OS for the Apple computers and not for the Dell Dimension series PCs. Similarly, the Safari browser is used on Apple computers and not on the PC in the Dell Series. While various editions of the Windows OS can be used on an Apple computer using an OS bridge such as the Virtual PC or Boot Camp, we assume that this is not the case for testing application X.