Thus, we get MOLS(5) = $\{M_1, M_2, M_3, M_4\}$. It is easy to check that indeed the elements of MOLS(5) are mutually orthogonal by superimposing them pairwise. For example, superimposing M_2 and M_4 leads to the following matrix where each element occurs exactly once.

```
11  22  33  44  55
35  41  52  13  24
54  15  21  32  43
23  34  45  51  12
42  53  14  25  31
```

The maximum number of MOLS(n) is $n - 1$ when n is not prime.

The method illustrated in the previous example is guaranteed to work only when constructing MOLS(n) for n that is prime or a power of prime. For other values of n, the maximum size of MOLS(n) is $n - 1$. However, different methods are used to construct MOLS(n) when n is not a prime or a power of prime. Such methods are beyond the scope of this book. There is no general method available to construct the largest possible MOLS(n) for n that is not a prime or a power of prime. The CRC Handbook of Combinatorial Designs (see the Bibliographic Notes section) lists a large set of MOLS for various values of n.

6.6 Pairwise Design: Binary Factors

A factor is considered binary if it can assume one of the two possible values.

We now introduce the techniques for selecting a subset of factor combinations from the complete set. First, we focus on programs whose configuration or input space can be modeled as a collection of factors where each factor assumes one or two values. We are interested in selecting a subset from the complete set of factor combinations such that all pairs of factor levels are covered in the selected subset.

Each factor combination selected will generate at least one test input or test configuration for the program under test. As discussed earlier, the complete set of factor combinations might be too large and impractical to drive a test process, and hence the need for selecting a subset.

To illustrate the process of selecting a subset of combinations from the complete set, suppose that a program to be tested requires three inputs, one corresponding to each input variable. Each variable can take only one of two distinct values. Considering each input variable as a factor, the total number of factor combinations is 2^3. Let X, Y, and Z denote the three input variables and $\{X_1, X_2\}$, $\{Y_1, Y_2\}$, and $\{Z_1, Z_2\}$, their

asy to check
y orthogonal
imposing M_2
ement occurs

eed to work only
er of prime. For
1. However, dif-
not a prime or a
s book. There is
ble MOLS(n) for
dbook of Com-
sts a large set of

of factor com-
rograms whose
tion of factors
sted in selecting
h that all pairs

one test input
discussed ear-
too large and
for selecting a

tions from the
s three inputs,
can take only
able as a fac-
X, Y, and Z
$\{Z_1, Z_2\}$, their

respective sets of values. All possible combinations of these three factors are given below.

(X_1, Y_1, Z_1) (X_1, Y_1, Z_2)
(X_1, Y_2, Z_1) (X_1, Y_2, Z_2)
(X_2, Y_1, Z_1) (X_2, Y_1, Z_2)
(X_2, Y_2, Z_1) (X_2, Y_2, Z_2)

However, we are interested in generating tests such that each pair of input values is covered, i.e. each pair of input values appears in at least one test. There are 12 such pairs, namely (X_1, Y_1), (X_1, Y_2), (X_1, Z_1), (X_1, Z_2), (X_2, Y_1), (X_2, Y_2), (X_2, Z_1), (X_2, Z_2), (Y_1, Z_1), (Y_1, Z_2), (Y_2, Z_1), and (Y_2, Z_2). In this case, the following set of four combinations suffices.

The number of tests needed to cover all binary factors is significantly less than what is obtained using exhaustive enumeration.

(X_1, Y_1, Z_2) (X_1, Y_2, Z_1)
(X_2, Y_1, Z_1) (X_2, Y_2, Z_2)

The above set of four combinations is also known as a *pairwise design*. Further, it is a *balanced* design because each value occurs exactly the same number of times. There are several sets of four combinations that cover all 12 pairs (see Exercise 6.8).

Notice that being content with pairwise coverage of the input values reduces the required number of tests from eight to four, a 50% reduction. The large number of combinations resulting from 3-way, 4-way, and higher order designs often force us to be content with pairwise coverage.

Let us now generalize the problem of pairwise design to $n \geq 2$ factors where each factor takes on one of two values. To do so, we define S_{2k-1} to be the set of all binary strings of length $2k-1$ such that each string has exactly k 1s. There are exactly $\binom{2k-1}{k}$ strings in S_{2k-1}. Recall that $\binom{n}{k} = \frac{!n}{!(n-k)\ !k}$.

Example 6.15 For $k = 2$, S_3 contains $\binom{3}{2} = 3$ binary strings of length 3, each containing exactly two 1s. All strings are laid out in the following with column number indicating the position within a string and row number indicating the string count.

$$\begin{array}{c|ccc}
 & 1 & 2 & 3 \\
\hline
1 & 0 & 1 & 1 \\
2 & 1 & 0 & 1 \\
3 & 1 & 1 & 0
\end{array}$$

For $k = 3$, S_5 contains $\binom{5}{3} = 10$ binary strings of length 5, each containing three 1s.

$$\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 0 & 1 & 1 & 1 \\
2 & 0 & 1 & 1 & 1 & 0 \\
3 & 1 & 1 & 1 & 0 & 0 \\
4 & 1 & 0 & 1 & 1 & 0 \\
5 & 0 & 1 & 1 & 0 & 1 \\
6 & 1 & 1 & 0 & 1 & 0 \\
7 & 1 & 0 & 1 & 0 & 1 \\
8 & 0 & 1 & 0 & 1 & 1 \\
9 & 1 & 1 & 0 & 0 & 1 \\
10 & 1 & 0 & 0 & 1 & 1
\end{array}$$

Given the number of two-valued parameters, the following procedure can be used to generate a pairwise design. We have named the procedure as the "SAMNA procedure" after its authors (see the Bibliographic Notes section for details).

```
Procedure for generating pairwise designs.

Input:      n: Number of two-valued input variables (or factors).
Output:     A set of factor combinations such that all pairs of input
            values are covered.
Procedure:  SAMNA

/*  $X_1, X_2, \ldots, X_n$ denote the $n$ input variables.
    $X_i^*$ denotes one of the two possible values of variable $X_i$,
    for $1 \le i \le n$.
    One of the two values of each variable corresponds to a
    0 and the other to 1.
*/

Step 1  Compute the smallest integer $k$ such that $n \le |S_{2k-1}|$.
Step 2  Select any subset of $n$ strings from $S_{2k-1}$. Arrange these to form a
        $n \times (2k-1)$ matrix with one string in each row while the columns
        contain different bits in each string.
Step 3  Append a column of 0s to the end of each string selected. This will
        increase the size of each string from $2k-1$ to $2k$.
Step 4  Each one of the $2k$ columns contains a bit pattern from which we
        generate a combination of factor levels as follows. Each combina-
        tion is of the kind $(X_1^*, X_2^*, \ldots, X_n^*)$, where the value of each variable
        is selected depending on whether the bit in column $i$, $1 \le i \le n$ is a 0
        or a 1.

End of procedure SAMNA
```

procedure can
procedure as
graphic Notes

ctors).
s of input

these to form a
le the columns

ected. This will

from which we
Each combina-
of each variable
, $1 \leq i \leq n$ is a 0

Example 6.16 Consider a simple Java applet named `ChemFun` that allows a user to create an in-memory database of chemical elements and search for an element. The applet has five inputs listed below with their possible values. We refer to the inputs as factors. For simplicity we have assumed that each input has exactly two possible values.

Factor	Name	Levels	Comments
1	Operation	{Create, Show}	Two buttons
2	Name	{Empty, Non-empty}	Data field, string
3	Symbol	{Empty, Non-empty}	Data field, string
4	Atomic number	{Invalid, Valid}	Data field, data > 0
5	Properties	{Empty, Non-empty}	Data field, string

The applet offers two operations via buttons labeled `Create` and `Show`. Pressing the `Create` button causes the element related data, e.g. its name and atomic number, to be recorded in the database. However, the applet is required to perform simple checks prior to saving the data. The user is notified if the checks fail and data entered is not saved. The `Show` operation searches for data that matches information typed in any of the four data fields.

Notice that each of the data fields could take on an impractically large number of values. For example, a user could type almost any string using characters on the keyboard in the `Atomic number` field. However, using equivalence class partitioning, we have reduced the number of data values for which we want to test `ChemFun`.

Testing `ChemFun` on all possible parameter combinations would require a total of $2^5 = 32$ tests. However, if we are interested in testing against all pairwise combinations of the five parameters, then only six tests are required. We now show how these six tests are derived using procedure SAMNA.

Given five binary factors (or parameters), exhaustive testing would require 32 tests. However, pairwise testing requires only six tests.

Input: $n = 5$ factors.
Output: A set of factor combinations such that all pairs of input values are covered.

Step 1 Compute the smallest integer k such that $n \leq |S_{2k-1}|$.
In this case we obtain $k = 3$

Step 2 Select any subset of n strings from S_{2k-1}. Arrange these to form a $n \times (2k - 1)$ matrix with one string in each row while the columns contain bits from each string.

We select first five of the 10 strings in S_5 listed in Example 6.15. Arranging them as desired gives us the following matrix.

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 1 & 1 & 1 \\ 2 & 0 & 1 & 1 & 1 & 0 \\ 3 & 1 & 1 & 1 & 0 & 0 \\ 4 & 1 & 0 & 1 & 1 & 0 \\ 5 & 0 & 1 & 1 & 0 & 1 \end{array}$$

Step 3 Append 0s to the end of each selected string. This will increase the size of each string from $2k - 1$ to $2k$.

Appending a column of 0s to the matrix above leads to the following 5×6 matrix.

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 1 & 1 & 1 & 0 & 0 \\ 3 & 1 & 1 & 1 & 0 & 0 & 0 \\ 4 & 1 & 0 & 1 & 1 & 0 & 0 \\ 5 & 0 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Step 4 Each of the $2k$ columns contains a bit pattern from which we generate a combination of factor values as follows. Each combination is of the kind $(X_1^*, X_2^*, \ldots, X_n^*)$, where the value of each variable is selected depending on whether the bit in column i, $1 \leq i \leq n$, is a 0 or a 1.

The following factor combinations are obtained by replacing the 0s and 1s in each column of the matrix listed above by the corresponding values of each factor. Here we have assumed that the first of two factor levels listed earlier corresponds to a 0 and the second level corresponds to a 1.

	1	2	3	4	5	6
1	Create	Create	Show	Show	Show	Create
2	Empty	NE	NE	NE	Empty	Empty
3	Non-empty	NE	NE	Empty	Empty	Empty
4	Valid	Invalid	Valid	Valid	Invalid	Invalid
5	Empty	NE	NE	Empty	NE	Empty

NE: Non-empty.

The matrix above lists six factor combinations, one corresponding to each column. Each combination can now be used to generate one or more tests as explained in Section 6.2. The following is a set of six tests for the `ChemFun` applet.

Each factor combination leads to one test.

T = { t_1: < Button = Create, Name = “”, Symbol = “C”, Atomic number = 6, Properties = “” >

t_2: < Button = Create, Name = “Carbon”, Symbol = “C”, Atomic number = −6, Properties = “Non-metal” >

t_3: < Button = Show, Name = “Hydrogen”, Symbol = “C”, Atomic number = 1, Properties = “Non-metal” >

t_4: < Button = Show, Name = “Carbon”, Symbol = “C”, Atomic number = 6, Properties = “” >

t_5: < Button = Show, Name = “”, Symbol = “”, Atomic number = −6, Properties = “Non-metal” >

t_6: < Button = Create, Name = “″, Symbol = “”, Atomic number = −6, Properties = “” >

}

The 5×6 array of 0s and 1s in Step 2 of Procedure SAMNA, is sometimes referred to as a *combinatorial object*. While Procedure SAMNA is one method for generating combinatorial objects, several other methods appear in the following sections.

Each method for generating combinatorial objects has its advantages and disadvantages that are often measured in terms of the total number of test cases, or test configurations, generated. In the context of software testing, we are interested in procedures that allow the generation of a small set of test cases or test configurations, while covering all interaction tuples. Usually, we are interested in covering interaction pairs, though in some cases interaction triples or even quadruples might be of interest.

In software testing, we are interested in generating a small set of tests that will likely reveal all t-way interaction faults.