CHAPTER – 3
3.1 Introduction

Testing is a critical component of modern software development. The problem of designing a suite of test cases is superficially similar to that of designing an experiment to estimate main effects and interactions, but there are crucial differences. Additive models are unhelpful, and classical design criteria are also. We propose a new class of models, and new measures of effectiveness. We compare several designs.

In testing one must generate test cases, administer them, correct faults when they are found, and manage the whole enterprise. Thus, there are (at least) three ways productivity can be improved, namely, generation of more efficient sets of test cases, automation of the testing process, and improvement of the management process. The last two issues are hard to influence since they require organizational changes and commitments (for process improvement paradigms [HUMP 89]). Here we concentrate on the first issue, that of designing test cases.

The process of software testing is typically divided into various phases: Unit testing (testing of small pieces of code written typically by one programmer), Integration testing (testing of several subsystems, each of which is comprised of many units) and System testing (testing of combination of subsystems). There may be still further phases such as Acceptance testing (when the software is first delivered to a client), Alpha testing (testing in developers environment), Beta testing (testing in real environment), FOA (First Office Application) etc.

Besides these stages of testing, there are many different methods of testing. Structural testing or, White box testing, refers to the type of testing in
which tests are designed on the basis of detailed architectural knowledge of the software under test. Functional testing, or Black Box testing, [JORG 95] refers to the type of testing where only the knowledge of the functionality of the software is used for testing; knowledge of the detailed architectural structure, or of the procedures used in coding, is not utilized. Structural testing is typically used during unit testing, where the tester (usually the developer who created the code) knows the internal structure and tries to exercise it based on detailed knowledge of the code. Functional testing is used during integration and system test, where the emphasis is on the user’s perspective and not on the internal workings of the software. Functional testing tries to test the functionality of the software as it is perceived by the end users (based on user manuals) and the requirements writers. Thus, functional testing consists of subjecting the system under test to various user controlled inputs, and watching its performance and behavior. The primary focus of this paper is on functional testing. Beizer [BEI 95] provides an excellent introduction to this subject and talks about various objectives of testing in this context.

Since the number of possible inputs is typically very large, testers need to select a sample, commonly called a suite, of test cases, based on effectiveness and adequacy. Much functional testing is done in an intuitive and less formal manner. Typically, testers, working from their knowledge of the system under test and of the prospective users, decide on a set of specific inputs. Clearly there is the possibility that important interactions among the inputs will be missed. Herein lie significant opportunities for a systematic approach, based on ideas from sampling and experimental design theory. Consider the following example.

22
3.1.1 Example: Testing an Air to Ground Missile System

Consider a software system controlling the state of an air to ground missile. The key inputs for the software are the altitude, attack and bank angles, speed, pitch, roll, and yaw. (There are many more—e.g. ambient temperature, pressure, wind velocity, etc. We will consider these later on.) Typically, these variables do not have any joint constraints as far as the software is concerned.

To test this software system, combinations of all these inputs must be provided and the output from the software system checked against the corresponding physics. Each combination tested is called a test case. One would like to generate test cases that include inputs over a broad range of permissible values.

Since in this example, we have continuous variables as inputs, the total number of possible test cases is unlimited. To reduce the number of test cases, testers have developed a number of heuristic strategies. Two guiding principles are i) non-redundancy (test cases are chosen so that for each test case that is included, the number of test cases which remain to be tried is reduced by at least one), and ii) generality (the outcome of the test case is generalizable beyond the specific inputs used). To implement these principles a number of concepts have been developed which can be applied in conjunction with each other.

One of these relates to the notion of Equivalence Partitioning [MYER 79]. It is assumed that the range of each of the input variables can be divided into a number of mutually exclusive classes, called equivalence classes, with the property, that the outcome of a test is generalizable to the entire equivalence class. That is, the same outcome would be expected regardless of a specific input value.
from that class. Myers states: "one can reasonably assume that a test of a representative value of each class is equivalent to a test of any other value." Since one cannot "reasonably assume" unless there is only one member in an equivalence class, in practice testers divide the input domain into a number of possibly overlapping classes (but usually with very little overlap) and select from 1 to 3 distinct inputs as representatives from each class. Typically there is much freedom in choosing the partitioning.

Having formed the equivalence partitioning, one still needs to decide which members be considered as representative members. This is where another notion, that of Boundary Value Analysis is applied. This is based on the experience that “test cases that explore boundary conditions have a higher payoff than test cases that do not" [MYER 79]. Boundary conditions are described as “those situations directly on, or above, and beneath the edges of input equivalence classes”. Thus, this concept is similar to that of a minimax strategy. Let us illustrate these concepts in context of the example we discussed above.

3.1.1 Example (continued): Testing an Air to Ground Missile System

To determine the test cases, we can go back to the requirement document. Suppose we are interested in testing the response during attack maneuvering. We will know the maximum and minimum possible values for each variable. Thus, we could choose to have a set of equivalence classes, each corresponding to the range of one variable. The concept is similar to that of a factor in standard design theory. Now we have the problem of selecting representative values. Following the boundary value heuristic, we will select the maximum and minimum as the two representative values for each of the 7 input variables. Since normal
maneuvering lies within these limits, we may want to include one or more intermediate values, for example a mid-point. Let the lower, middle, and upper values be input states 1, 2, 3 respectively. Then in the language of statistical experimental design, we have seven factors, \( A, \ldots, G \) (altitude, attack angle, bank angle, speed, pitch, roll, and yaw), each at three levels. To test all the possible combinations, one would need a complete factorial experiment, which would have \( 3^7 = 2187 \) test cases consisting of all possible sequences of 1's, 2's, and 3's. With only two levels per factor, we need \( 2^7 = 128 \) test cases.

The notions of equivalence partitioning and boundary value analysis have achieved a tremendous reduction from the effectively infinite problem we would face in exhaustive testing. In this example, it may be possible to test all 128 or 2187 test cases, but, this may be only one of many systems that need to be tested. Further, if we want to increase the scope of the testing by including three more variables (for example \( H \): ambient temperature, \( I \): pressure, and \( J \): wind velocity) then we would have \( 2^{10} = 1024 \) or even \( 3^{10} = 59049 \) test cases. Many of the dimensions are categorical, and thus, there is no reduction feasible due to boundary value analysis and even using automated testing, the number of cases is impossibly large. We illustrate this by giving two more examples.

3.1.2 Example: Screen Testing

Typically, users of business systems interact with software via a succession of screens, each of which has a number of fields. It is not uncommon to have 50 or more fields, for example Cohen, Dalal, Kajla, and Patton [COHE 94] give an example of a screen with 76 fields. Assuming only 2 values per field, (for example "valid" and "missing"), one has \( 2^{76} \) test cases. At a rate of a
million cases per second (impossible to achieve even today with automation), this would require $2 \times 10^{15}$ years to test.

### 3.1.3 Example: Interoperability Testing

Periodically, software companies update their products, and sell them as new versions (e.g. Windows 95 vs. Windows 3.1). When these products come out, it is essential that they work with a number of versions of other software products. Thus, the issue of interoperability testing is critical. For example, one would want Windows95 and 3.1, Word 6.0 and 2.0, Excel 5.0 and 4.0, etc. all to work with one another. Thus, suppose one had four software products, and wanted to support two versions of each, then we must study 28 interoperability problems. However, each interoperability problem represents a large number of sub problems, and detailed analysis may result in a huge number of factors to be studied. Also in each version of the software, there may be a large number of parameter settings, which may have unpredictable influences on the results of a test. In such a case we may choose to regard the test environment as being random, so that the outcome of a test is not deterministic, but will identify a fault only with a certain probability that is strictly between 0 and 1.

### 3.2 Covering Designs

In a statistical experiment if one is interested in only main effects, then one can get away with a highly fractionated factorial design. However in the case of software testing, there is no interest in estimating additive effects. Interest lies in covering the test space as completely as possible and checking whether the test cases pass or fail. However, it is certainly possible to use standard statistical designs. For example consider the set of test cases given in Table 3.1.
This is an orthogonal array of strength two, with 7 factors and 2 levels. It requires 8 test cases instead of 128. It is clear that all possible pairwise combinations of levels of two factors are covered in a balanced way (exactly twice each in this example). Thus testing according to this design will protect against any incorrect implementation of the code involving any pairwise interaction, and also whatever other higher order interactions happen to be represented in the table.

Brownlie, Prowse and Phadke [BROW 92] have suggested the use of orthogonal array designs of strength two for testing.

<table>
<thead>
<tr>
<th></th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.1 An orthogonal array

However, in software testing repeating a run with exactly the same inputs will give exactly the same output, so exact replication is unhelpful, and wasteful. The problem is not that of estimating an unknown response function, but rather that of determining whether the software functions correctly under all relevant input conditions; the response is either “O.K.”, in which case nothing needs to be done, or “failure” in which case the test case can be analyzed to determine the cause (or causes) of the failure. The efficiency of a test design is measured by the degree to which it covers the relevant input space. If we insist on coverage of all pairs, but give up the restriction of balance, we can do a lot better. For example, Table 3.2 gives a design we call R6, which achieves 2-coverage of 10 two-level factors, in just 6 test cases.
Table 3.2 The design \( R_n \)

<table>
<thead>
<tr>
<th>Test</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The structure of this design is clear: the first test case sets all factors to level 1; in the last five runs, all combinations of two 1's and three 2’s are used, once each. Then in each pair of columns, we see (1, 1) in the first row, and (2,2) at least once in the last five rows (since the sets of three 2’s must overlap); and since the columns are not identical and each contains three 1’s and three 2’s, both (1,2) and (2,1) must occur also. Thus in every pair of columns, all four possibilities (1,1),(1,2),(2,1),(2,2) each occur at least once. All pairs of values are covered in only 6 test cases.

Thus, in the missile example, besides incorporating the 7 factors already mentioned, one can now include three more factors and still guarantee protection against any pairwise interaction, using only 6 test cases. Unlike orthogonal arrays, for which the number of cases grows at least linearly with the number of factors, covering designs grow at only a logarithmic rate. The savings are thus more dramatic when the number of factors is large; for example with 126 binary factors we can cover all pairwise interactions with only 10 runs, whereas an orthogonal array would require at least 128 runs.

### 3.3 A Model for Software Faults

Suppose we have defined \( k \) factors, with the \( i \)-th having \( q_i \) levels. A \( t \)-factor fault is one which is triggered whenever some set of \( t \) factors is held at some set of levels. Thus there are \( \sum_1 q_i \) possible 1-factor faults.
(1/2) \sum_{1 \leq j < i} q_i q_j

possible 2-factor faults. \ldots \pi_i q_i possible k-factor faults. Each possible fault can be specified by giving the relevant factors and levels; we use the notation f = (f_1, f_2, \ldots, f_k) where 0 \leq f_i \leq q_i and f_i = 0 means that the i-th factor setting is irrelevant. Thus for example if we have six factors, each with 3 levels, a possible fault is \( f = (2, 3, 0, 0, 1, 0) \), so that three of the factors are irrelevant but the other three must be set at particular levels to trigger the fault. Let \( F_i \) be the set of all such fault-vectors having exactly t non-zero elements. i.e. corresponding to t-factor faults, and let \( F \) be the union \( \bigcup_{t=1}^{k} F_t \). The number of elements in \( F \) is \( \pi_i(1 + q_i) \).

This notation is not completely satisfactory. For example, if \( k = 4 \) and \( q_1 = q_2 = 2 \), the 1-factor fault \( (1,0,0,0) \) can be regarded as a pair of 2-factor faults, \( (1,1,0,0) \), and \( (1,2,0,0) \). This might be entirely appropriate, if there are two distinct errors in the software. Clearly the 1-factor description is more parsimonious, but in general parsimony is not uniquely defined: for example the triad of 2-factor faults \( (1,1,0),(1,2,0),(2,1,0) \) can be expressed as the union of a 1-factor fault and a 2-factor fault (in two distinct ways). We ignore this difficulty since in practice faults occur sparsely and minimal descriptions are usually unique.

Identifying faults using the standard notation for additive models is possible, but very clumsy. For example to identify the single three-factor fault \( (1, 1, 1, 0) \) we need three main effects, three 2-factor interactions, and one 3-factor interaction. Also we need a non-linear transformation of the response function into the set \{0, 1\}. 

29
A probability model for the faults in a software module must specify the probability that each possible fault is present. Thus a general model would have $|F|$ parameters. Simplifications can be made if we assume symmetry; thus one simple model depends on a set of $k$ parameters $p_1, p_2, \ldots, p_k$, and assumes that each fault in $F_i$ occurs with probability $p_i$, with different faults occurring independently. Note that in this model, ignoring the multiple description difficulty alluded to in the previous paragraph, the probability that the $i$th factor figures in a 1-factor fault is proportional to $q_i$. An alternative model would make these probabilities all equal. In the following, we will assume that all the $p$'s are small, so that at most one fault appears. Also we will assume that at most one $p$ is nonzero.

In the absence of a priori knowledge, a symmetry assumption seems reasonable. However, if we do not have a stochastic environment, the independence aspect of this model is not empirically verifiable, since it is impossible in principle to obtain an empirical estimate of the probability that a particular fault occurs. We can interpret the parameters $p_i$ as being subjective probabilities. If the levels of the various factors have been randomized, the symmetry assumption is natural. If we accept the independence assumption, the parameter $p_i$ can be estimated straightforwardly (by testing all $t$-factor settings).

If our problem does involve a stochastic environment, we may assume that a test case corresponding to a particular fault $f$ will result in a failure only with some nontrivial probability $r_f$. For simplicity, we may choose to assume that these probabilities are all equal. Mallows [MALL 97] discuss the stochastic-environment problem.
If we use this model, experience with a class of similar systems may lead to useful prior estimates of the parameters. In a later section we report some empirical evidence.

3.4 Factor-Covering Designs - Definitions, Preliminaries and Properties

3.4.1 Definition. Suppose we have $n$ runs and $k$ factors, with the $i$-th factor having $q_i$ levels. Then the design is called a $(n, Q)$ design, where $Q$ stands for the word $q_1, q_2, \ldots, q_k$.

Thus the design $R_6$ displayed in Table 3.2, is a $(6, 2^{10})$ design. (one should be aware that our notation, while perhaps natural for a statistician, differs from that used in many previous papers on this subject. Also the term “covering design” has been used in other senses from ours.) Such a design is a collection of $n$ row-vectors of length $k$, with the $i$-th element of each vector being drawn from the alphabet $\{1, \ldots, q_i\}$.

3.4.2 Definition. If an $(n, Q)$ design has the property that the projection onto any $t$ coordinates exhibits all $\pi q_i$ possibilities, we say such a design is $t$-covering. A $t$-covering design is optimal if $n$ is minimal for fixed $Q$, $t$.

Other names that have been used for these designs are $t$-surjective array, or (for the transposed array) a qualitatively $t$-independent family of vectors. Box and Tysseal [TYSS 96] call a $t$-covering $(n, 2^k)$ design a $(n, k, t)$ screen, and investigate the coverage properties of certain orthogonal arrays.

$R_6$ is a 2-covering design of type $(6, 2^{10})$. In fact it is the optimal 2-covering design. Of course, it is not a 3-covering design; for 20 triads the setting $(1, 1, 1)$ occurs twice, for another 10 triads $(2, 2, 2)$ occurs twice, and for another
30 triads (2, 2, 1) (or some permutation of this) occurs twice: for the remaining 60 triads of factors, all six runs give distinct settings. Thus this design covers only 660 of the 960 possible triad settings.

This example raises some general questions. How well can a t-covering design cover the possible combinations of t+1, t+2... factors? (Mallows [MALL 97] gives an asymptotic answer to this question for random designs.) More generally, how can we measure the efficiency of an (n, Q) design? We propose two indices of merit of an (n, Q) design.

3.4.3 Definition. For each t, the t-coverage of a design is the ratio of the number of t-factor settings that are covered to the total possible number of t-factor settings. Consider a particular (n, Q) design, and suppose there are \( n_t(j) \) t-factor settings that are each covered exactly \( j \) times, \( j = 0, 1, 2 \ldots \). Then the number of different t-factor settings that are covered is \( \sum_j n_t(j) \), and the total number of t-factor settings that are exercised by the design (counting multiplicities) is \( \sum_j n_t(j) \). For an \( (n, q^k) \) design we thus have

\[
\text{t-coverage} = \frac{\sum_j n_t(j)}{q^k \cdot C(k,t)}
\]

where, \( C(k,t) \) is the binomial coefficient "k choose t".

The "t-coverage" tells what fraction of the possible t-factor settings are covered by the design; 100% is perfect, all possible t-factor settings are covered.

3.4.4 Definition. For each t, the t-diversity of a design is the ratio of the number of t-factor settings that are covered to the total number of t-factor settings that occur in the design.

Thus this index tells to what degree the design avoids replication; 100% says that all t-factor settings that appear in the design are different, so the design
does a perfect job in avoiding repetitions: 50% says that each setting that is covered is exercised twice (on the average). This suggests that we might be able to do better, i.e. achieve higher coverage, by working towards higher diversity. If the t-coverage of a \((n, q^k)\) design is \(c\), the t-diversity is \(cq^k / n\), which is a number independent of the detailed structure of the design, and even independent of \(k\).

If two designs have the same t-factor coverage but one has fewer runs than the other, the difference can be measured in two (equivalent) ways: first, simply by the ratio of the numbers of runs, and second by the fact that the smaller design has larger diversity; the larger design has more t-factor settings altogether, but the extra ones are wasted on replicating settings that are already covered. Again, if the t-diversity is 100%, the t-coverage for a \(q^k\) design must be \(n/q\), and there is no way to get any higher value; this design does a perfect job of spreading its effort. However if the t-diversity is only 50%, there might be another design (of the same size) with higher coverage (and higher diversity). Computation of these coverage and diversity measures is tedious but straightforward; for large designs with much symmetry they pose some interesting combinatorial challenges.

<table>
<thead>
<tr>
<th>(k)</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>13</th>
<th>16</th>
<th>18</th>
<th>56</th>
<th>126</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 3.3 Sizes of the Best Known 2-Covering \((n,3^k)\) Designs

For the R6 design in Table 3.2, the 3-coverage is \(660/960 = 68.8\%\), and the 3-diversity is \(660/720 = 91.7\%\) suggesting that this is close to the best we can do for a 6 run design and that to increase the coverage we will have to increase \(n\). It is believed that we need 12 runs to cover all triads completely.
3.5 Coverage Properties of Some Designs

3.5.1 Orthogonal Arrays

Taguchi [TAGU 86] and many other writers have promoted the use of orthogonal arrays, which are balanced fractions of complete factorial designs. In classical terms they are main-effect plans with proportional balance, and hence are \( t = 2 \) designs. For each such design we can count how many of the possible three-way and four-way combinations are covered. We find that some of these designs are \( t = 3 \) designs, i.e. they cover all three-factor combinations. However in some cases there are designs that have better coverage properties than these orthogonal designs. For example, there is an 18-run orthogonal array which is a 2-covering \((18, 2.3^7)\) design.

<table>
<thead>
<tr>
<th>K</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>11</th>
<th>14</th>
<th>16</th>
<th>20</th>
<th>22</th>
<th>28</th>
<th>30</th>
<th>32</th>
<th>40</th>
<th>44</th>
<th>56</th>
<th>64</th>
<th>70</th>
<th>80</th>
<th>121</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>31</td>
<td>32</td>
<td>34</td>
<td>35</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 3.4 Sizes of the Best Known 3-Covering \((n,2^3)\) Designs

However Table 3.3 above shows that a 2-covering \((18, 3^{16})\) design exists, and it happens that this can be augmented to form a 2-covering \((18, 2.3^{16})\) design (which we call \(C_{18}\)); in fact it can be augmented much further (see below). The \((18, 2.3^7)\) orthogonal array \(L_{18}\) (as given by Taguchi) is not a sub design of \(C_{18}\). We see from Table 3.4 that \(L_{18}\) covers 70.1% of all possible triads (of its eight factors), while \(C_{18}\) covers 57.8% of the possible triads of its 17 factors. The orthogonal array \(L_{16}\) is a 2-covering \((16, 2^{15})\) design, which does not cover all three-way possibilities; Table 3.4 shows that there is a design with \(n = 16\) that covers all triads of 14 two-level factors.

3.5.2 Random and Balanced Random Designs

In a random design, in each run, for each factor we simply assign the level.
at random (independently for each run). Clearly, deleting any repeated rows will give greater efficiency, but this will not help much since such repetitions are very unlikely. It is helpful to assign the levels at random subject to the constraint that in each column each level occurs as nearly as possible the same number of times. We recall that Satterthwaite [SATT 59] proposed to use random constructions for classical purposes, where they are not very effective. However we will see that they perform well as coverage designs as long as complete t-coverage is not required. Lin [LIN 95] has used the balanced random idea to construct supersaturated designs.

3.6 Constructing Factor-Covering Designs

We have seen above that factor-covering designs with moderate efficiency can be constructed at random. Sherwood [SHE 94] has written (in C) a program called CATS (Constrained Array Test System) which finds covering designs using a greedy search algorithm. Given Q, a list of all feasible runs is generated. (there are |Q| of them). If some runs are prohibited, they are excluded from this list. Also, given t, a list of all possible t-factor settings is generated. This is \( F_t \) in the notation of Section 3. We want a design that covers each of the settings in \( F_t \). At each stage, after some number of runs has been chosen, so that some fraction of these settings have been covered, each of the unused runs is examined, and the number of additional t-factor settings that are covered is computed. The algorithm selects the run that maximizes the number of t-way settings that are covered. (If more than one run achieves the maximum, the first one encountered is chosen.) This procedure continues until all the t-factor settings have been covered.

Since the possible runs are tested in sequence, the user can start with a list
of runs that takes the importance of the factors into account, so that the more
critical settings will be covered early. To handle large problems where exhaustive
search of all the possibilities is not feasible, the algorithm starts by choosing a
subset of the factors, say \( f_1 \) of them, and a covering design \( D_1 \) for these factors is
constructed. Then an additional subset of \( f_2 \) factors is considered, and a list of
feasible runs is generated by concatenating the runs of \( D_1 \) with all possible
settings of these new factors. A covering design \( D_2 \) for the \( f_1 + f_2 \) factors is
generated by selecting from this list. This construction is iterated until all the
factors have been entered. This strategy ensures that the size of the list of runs that
need to be considered remains manageable. The present version of CATS does not
use randomization. It is not clear how much might be gained (in terms of number
of runs) by using randomization throughout the algorithm.

The AETG system seems to be more efficient than CATS. For example, for 20 factors,
each with 10 levels, CATS found a \((240, 10^{20})\) design; the AETG system found a \((181, 10^{20})\) design [COHE 96].

3.7 Summary

We believe our modeling and test-generation approach satisfies the goal of
usability by testers. In our experience, testers found the activity of specifying a
software component’s inputs to be natural and straightforward. By using the
RATG software system [GOPA 05], testers required minimal training to write
their first data model and generate test tuples with pair wise combinations. As
noted above, these tuples offer immediate value when used as test data sets (inputs
for hand-crafted tests). Of course a significantly greater investment, mostly in
software and script development, is required to develop the infrastructure such as
an oracle that will allow the tests to be run wholly automatically.

Our Thesis work reports about systems that generate, document, execute, and evaluate thousands of test cases.

**Model of the test data is fundamental.** The model is comparable with an executable specification; like a specification, model development requires considerable domain expertise. For example, permissible data values and complex constraints among data values must be discovered and represented. Although a model-based test-generation system will require far more effort to develop than the model, development of the model should be allocated a significant portion of the up-front effort.

**Model-based testing is a development project.** The development, application and ongoing maintenance of a test automation system requires expertise from software developers and professional testers. This mix of skill sets is difficult to find in either a development or a testing organization.