# 1 Measuring Combinatorial Coverage

Since it is nearly always impossible to test all possible combinations, combinatorial testing is a reasonable alternative. But it is not always practical to re-design an organization's testing procedures to use tests based on covering arrays. Testing procedures often develop over time, and employees have extensive experience with a particular approach. Units of the organization may even be structured around established, documented test procedures. This is particularly true in organizations that must test according to contractual requirements, or government standards. And because much software assurance involves testing applications that have been modified to meet new specifications, an extensive library of legacy tests will exist. The organization can save time and money by re-using existing tests, which generally have not been developed as covering arrays.

How can combinatorial methods be useful in such an environment? Short of creating new test suites from scratch, one of the best approaches to obtaining the advantages of methods described in this book is to measure the combinatorial coverage of existing tests, then supplement as needed. As introduced in Chapter 1, these methods help take advantage of the Interaction Rule, by testing combinations up to a suitable level. Depending on the budget and criticality of the software, 2-way through 5-way or 6-way testing may be appropriate. Building covering arrays for some specified level of $t$ is one way to provide $t$-way coverage. However, many large test suites naturally cover a high percentage of $t$-way combinations. If our existing test suite covers almost all 3-way combinations, for example, then it may be sufficient for the level of assurance that we need. Determining the level of input or configuration state space coverage can also help in understanding the degree of risk that remains after testing. If 90% - 100% of the relevant state space has been covered, then presumably the risk is small, but if coverage is much smaller, then the risk may be substantial. This chapter describes some measures of combinatorial coverage that can be helpful in evaluating the degree of $t$-way coverage of any test suite, even if it was not constructed for combinatorial coverage.

## *1.1 Software Test Coverage*

Test coverage is one of the most important topics in software assurance. Users would like some quantitative measure to judge the risk in using a product. For a given test set, what can we say about the combinatorial coverage it provides? With physical products, such as light bulbs or motors, reliability engineers can provide a probability of failure within a particular time frame. This is possible because the failures in physical products are typically the result of natural processes, such as metal fatigue.

With software the situation is more complex, and many different approaches have been devised for determining software test coverage. With millions of lines of code, or only with a few thousand, the number of paths through a program is so large that it is impossible to test all paths. For each *if* statement, there are two possible branches, so a sequence of $n$ *if* statements will result in $2^n$ possible paths. Thus even a small program with only 270 *if* statements in an execution trace may have more possible paths than there are atoms in the universe, which is on the order of $10^{80}$. With loops (*while* statements) the number of possible paths is literally infinite. Thus a variety of measures have been developed to gauge the degree of test coverage. The following are some of the better-known coverage metrics:

*Commonly used coverage measures do not apply well to combinatorial testing.*

- **Statement coverage:** This is the simplest of coverage criteria – the percentage of statements exercised by the test set. While it may seem at first that 100% statement coverage should provide good confidence in the program, in practice, statement coverage is a relatively weak criterion. At best, statement coverage represents a sanity check: unless statement coverage is close to 100%, the test set is probably inadequate.

- **Decision or branch coverage:** The percentage of branches that have been evaluated to both *true* and *false* by the test set.

- **Condition coverage:** The percentage of conditions within decision expressions that have been evaluated to both true and false. Note that 100% condition coverage does not guarantee 100% decision coverage. For example, "`if (A || B) {do something} else {do something else}`" is tested with [0 1], [1 0], then A and B will both have been evaluated to 0 and 1, but the *else* branch will not be taken because neither test leaves both A and B false.

- **Modified condition decision coverage (MCDC):** This is a strong coverage criterion that is required by the US Federal Aviation Administration for Level A (catastrophic failure consequence) software; i.e., software whose failure could lead to complete loss of life. It requires that every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision outcome, and that each entry and exit point have been invoked at least once.

## *1.2 Combinatorial Coverage*

Note that the coverage measures above depend on access to program source code. Combinatorial testing, in contrast, is a black box technique. Inputs are specified and expected results determined from some form of specification. The program is then treated as simply a processor that accepts inputs and produces outputs, with no knowledge expected of its inner workings.

Even in the absence of knowledge about a program's inner structure, we can apply combinatorial methods to produce precise and useful measures. In this case, we measure the state space of inputs. Suppose we have a program that accepts two inputs, *x* and *y*, with 10 values each. Then the input state space consists of the $10^2 = 100$ pairs of *x* and *y* values, which can be pictured as a checkerboard square of 10 rows by 10 columns. With three inputs, *x*, *y*, and *z*, we would have a cube with $10^3 = 1{,}000$ points in its input state space. Extending the example to *n* inputs we would have *n* dimensions with $10^n$ points. Exhaustive testing would require inputs of all $10^n$ combinations, but combinatorial testing could be used to reduce the size of the test set.

How should state space coverage be measured? Looking closely at the nature of combinatorial testing leads to several measures that are useful. We begin by introducing what will be called a *variable-value configuration.*

**Definition.** For a set of *t* variables, a variable-value configuration is a set of *t* valid values, one for each of the variables.

**Example.** Given four binary variables, *a, b, c,* and *d*, *a*=0, *c*=1, *d*=0 is a variable-value configuration, and *a*=1, *c*=1, *d*=0 is a different variable-value configuration for the same three variables *a, c,* and *d*. Choosing a different 3-way combination of the variables, *b*=0, *c*=1, *d*=0 and *b*=1, *c*=1, *d*=0 are two different variable-value configuration for variables, *b, c,* and *d*.

*Simple t-way combination coverage*

Of the total number of *t*-way combinations for a given collection of variables, what percentage will be covered by the test set? If the test set is a covering array, then coverage is 100%, by definition, but many test sets not based on covering arrays may still provide significant *t*-way coverage. If the test set is large, but not designed as a covering array, it is very possible that it provides 2-way coverage or better. For example, random input generation may have been used to produce the tests, and good branch or condition coverage may have been achieved. In addition to the structural coverage figure, for software assurance it would be helpful to know what percentage of 2-way, 3-way, etc. coverage has been obtained.

**Definition**: For a given test set for *n* variables, simple *t*-way combination coverage is the proportion of *t*-way combinations of *n* variables for which all variable-values configurations are fully covered.

**Example.** Figure 53 shows an example with four binary variables, *a*, *b*, *c*, and *d*, where each row represents a test. Of the six 2-way combinations, *ab, ac, ad, bc, bd, cd*, only *bd* and *cd* have all four binary values covered, so simple 2-way coverage for the four tests in Figure 53 is 2/6 = 1/3 = 33.3%. (Reading down the columns headed *c* and *d*, we see all four possible binary pairs 00,10,01, and 11, as is also true for *b* and *d*, but no other pair of columns contains all four.) There are four 3-way combinations, *abc, abd, acd, bcd*, each with eight possible configurations: 000, 001, 010, 011, 100, 101, 110, 111. Of the four combinations, none has all eight configurations covered, so simple 3-way coverage for this test set is 0%.

| a | b | c | d |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 |

**Figure 1.** An example test array for a system with four binary components

*(t + k)-way combination coverage*

A test set that provides full combinatorial coverage for t-way combinations will also provide some degree of coverage for (*t*+1)-way combinations, (*t*+2)-way combinations, etc. This statistic may be useful for comparing two combinatorial test sets. For example, different algorithms may be used to generate 3-way covering arrays. They both achieve 100% 3-way coverage, but if one provides better 4-way and 5-way coverage, then it can be considered to provide more software testing assurance. Somewhat paradoxically, a *t*-way covering array that is considered "better" in the sense of containing fewer tests may end up detecting fewer faults because it covers a smaller proportion of (*t*+1)-way combinations. More discussion of this consideration is provided in Sect. 9.2.

*A test set for t-way interactions will also cover some higher strength interactions at t+1, t+2, etc.*

**Definition.** For a given test set for *n* variables, (*t*+*k*)-way combination coverage is the proportion of (*t*+*k*)-way combinations of *n* variables for which all variable-values configurations are fully covered. (Note that this measure would normally be applied only to a *t*-way covering array, as a measure of coverage beyond *t*).

**Example.** If the test set in Figure 53 is extended as shown in Figure 54, we can extend 3-way coverage. For this test set, *bcd* is covered, out of the four 3-way combinations, so 2-way coverage is 100%, and (2+1)-way = 3-way coverage is 25%.

| a | b | c | d |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |

**Figure 2.** Eight tests for four binary variables.

*Tuple Density*

For ($t$+$k$)-way coverage where $k$ = 1, Chen and Zhang [38] have proposed the *tuple density* metric. A special metric for ($t$+1)-way coverage is useful because [38] (1) the coverage of higher strength tuples for $t' > t$+1 is much lower (because the number of $t$-way combinations to be covered grows exponentially with $t$), (2) the coverage at $t$+1 provides some information for coverage at $t' > t$+1 because ($t$+1)-way tuples are subsumed by higher strength tuples, and (3) the number of additional faults triggered by $t$-way combinations drops rapidly with $t > 2$ (this is the Interaction Rule).

**Definition.** Tuple density is the sum of $t$ and the percentage of the covered ($t$+1)-tuples out of all possible ($t$+1)-tuples [38].

**Example.** As shown in the previous example, the test set in Figure 54 provides 100% coverage of 2-way combinations and 25% coverage of 3-way combinations, so the tuple density of this test set is 2.25.

*Variable-Value Configuration coverage*

So far we have only considered measures of the proportion of combinations for which *all configurations* of $t$ variables are fully covered. But when $t$ variables with $v$ values each are considered, each $t$-tuple has $v^t$ configurations. For example, in pairwise (2-way) coverage of binary variables, every 2-way combination has four configurations: 00, 01, 10, 11. We can define two measures with respect to configurations:

**Definition.** For a given combination of $t$ variables, variable-value configuration coverage is the proportion of variable-value configurations that are covered.

**Definition.** For a given set of $n$ variables, ($p$, $t$)-completeness is the proportion of the C($n$, $t$) combinations that have configuration coverage of at least $p$ [115].

**Example.** For Figure 54 above, there are C(4, 2) = 6 possible variable combinations and $C(4,2)2^2 = 24$ possible variable-value configurations. Of these, 19 variable-value configurations are covered and the only ones missing are *ab*=11, *ac*=11, *ad*=10, *bc*=01, *bc*=10. But only two, *bd* and *cd*, are covered with all 4 value pairs. So for the basic definition of simple $t$-way coverage, we have only 33% (2/6) coverage, but 79% (19/24) for the configuration coverage metric. For a better understanding of this test set, we can compute the configuration coverage for each of the six variable combinations, as shown in 0. So for this test set, one of the combinations (*bc*) is covered at the 50% level, three (*ab*, *ac*, *ad*) are covered at the 75% level, and two (*bd*, *cd*) are covered at the 100% level. And, as noted above, for the whole set of

tests, 79% of variable-value configurations are covered. All 2-way combinations have at least 50% configuration coverage, so (.50, 2)-completeness for this set of tests is 100%.

Although the example in Figure 53 uses variables with the same number of values, this is not essential for the measurement. Coverage measurement tools that we have developed compute coverage for test sets in which parameters have differing numbers of values, as shown in Figure 60.

| Vars | Configurations | Config |
|---|---|---|
| a b | 00, 01, 10 | .75 |
| a c | 00, 01, 10 | .75 |
| a d | 00, 01, 11 | .75 |
| b c | 00, 11 | .50 |
| b d | 00, 01, 10, 11 | 1.0 |
| c d | 00, 01, 10, 11 | 1.0 |

- *total 2-way cove*rage = 19/24 = .79167
- *(.50, 2)-completeness* = 6/6 = 1.0
- *(.75, 2)-completeness* = 5/6 = 0.83333
- *(1.0, 2)-completeness* = 2/6 = 0.33333

**Figure 3.** The test array covers all possible 2-way combinations of a, b, c, and d to different levels.

The graph in Figure 56 shows a graphical display of the coverage data for the tests in Figure 54. Coverage is given as the Y axis, with the percentage of combinations reaching a particular coverage level as the X axis. Note from Fig. 1 that 100% of the combinations are covered to at least the .50 level, 83% are covered to the .75 level or higher, and a third covered 100%. Thus the rightmost horizontal line on the graph corresponds to the smallest coverage value from the test set, in this case 50%. Thus (.50, 2)-completeness = 100%.

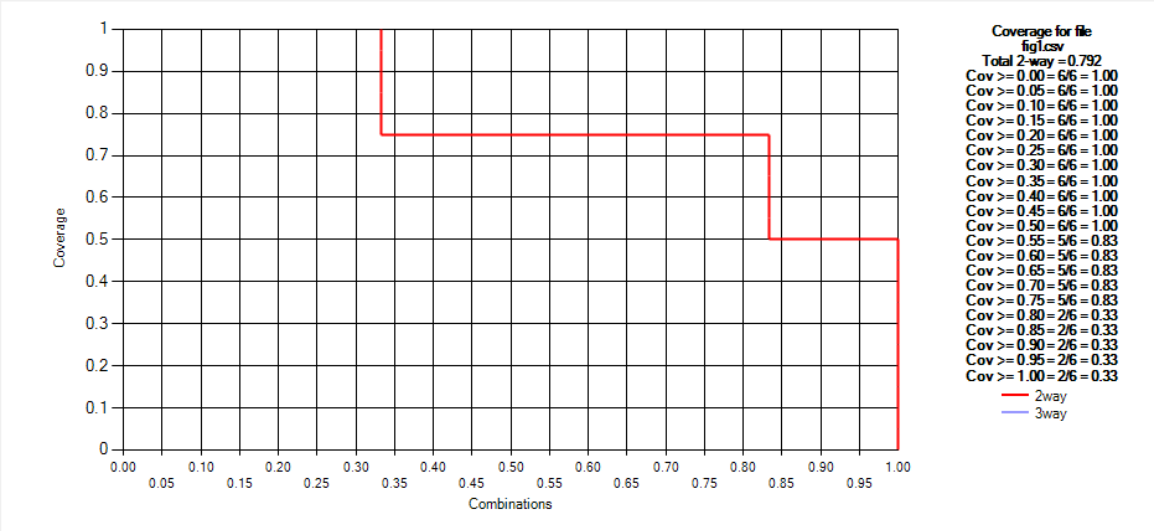



**Figure 4.** Graph of coverage from example test data above.

Note that the total 2-way coverage is shown as 0.792. This figure corresponds approximately to the area under the curve in Figure 56. (The area is not exactly equal to the true figure of 0.792 because the curve is plotted in increments of 0.05.) Counting the squares and partial squares, we see that there are approximately 40 squares in the upper right corner of the graph, so the area below the curve is roughly 160/200. Additional tests can be added to provide greater coverage. Suppose an additional test [1,1,0,1] is added. Coverage then increases as shown in Figure 57. Now we can see immediately that all combinations are covered to at least the 75% level, as shown by the rightmost horizontal line (in this case there is only one horizontal line). The leftmost vertical line reaches the 1.0 level of coverage, showing that 50% of combinations are covered to the 100% level.

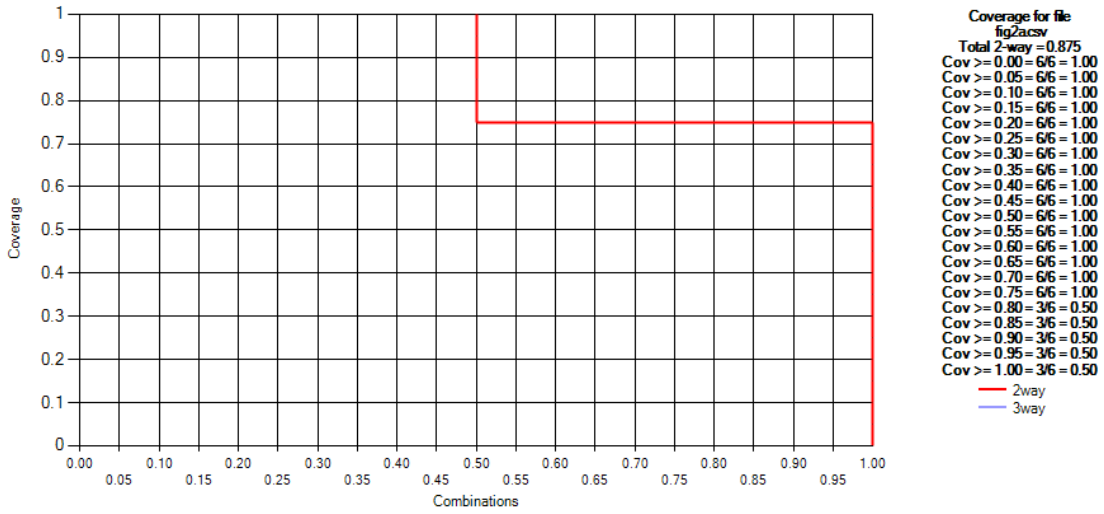



**Figure 5.** Coverage after test [1,1,0,1] is added.

Note that the upper right corner includes roughly 25 squares, so the area under the curve is 175/200, or 87.5%, matching the Total 2-way coverage figure. Adding another test, [1,0,1,1] results in the coverage shown in Figure 58.

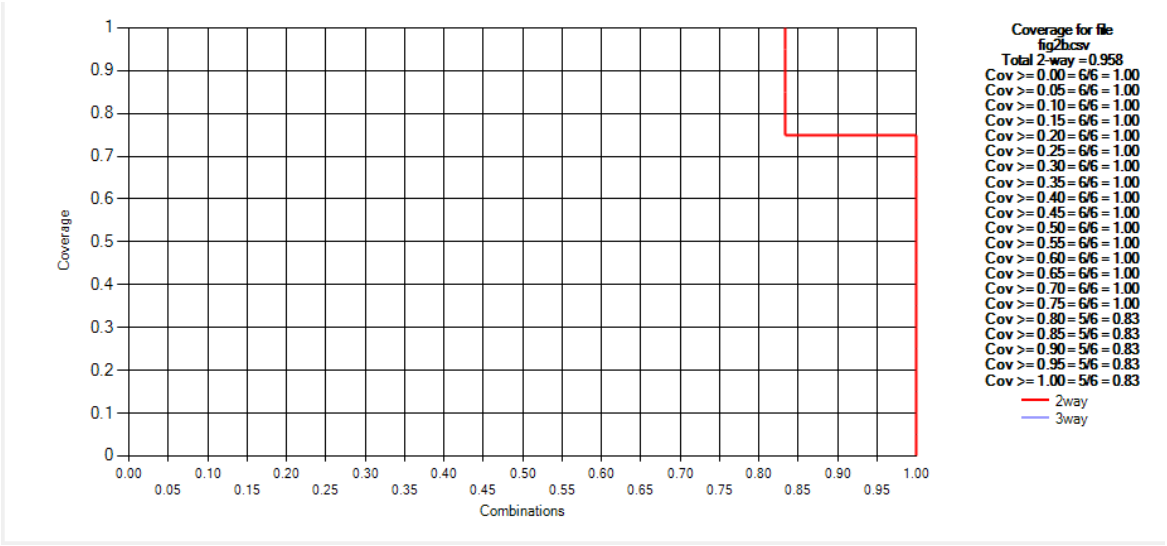



**Figure 6.** Coverage after test [1,0,1,1] is added.

If a test [1,0,1,0] is then added, 100% coverage for all combinations is reached, as shown in Figure 59. The graph can thus be thought of as a "coverage strength meter", where the red indicator line moves to the right as higher coverage levels are reached. A covering array, which by definition covers 100% of variable-value configurations will always produce a figure as below, with full coverage for all combinations.

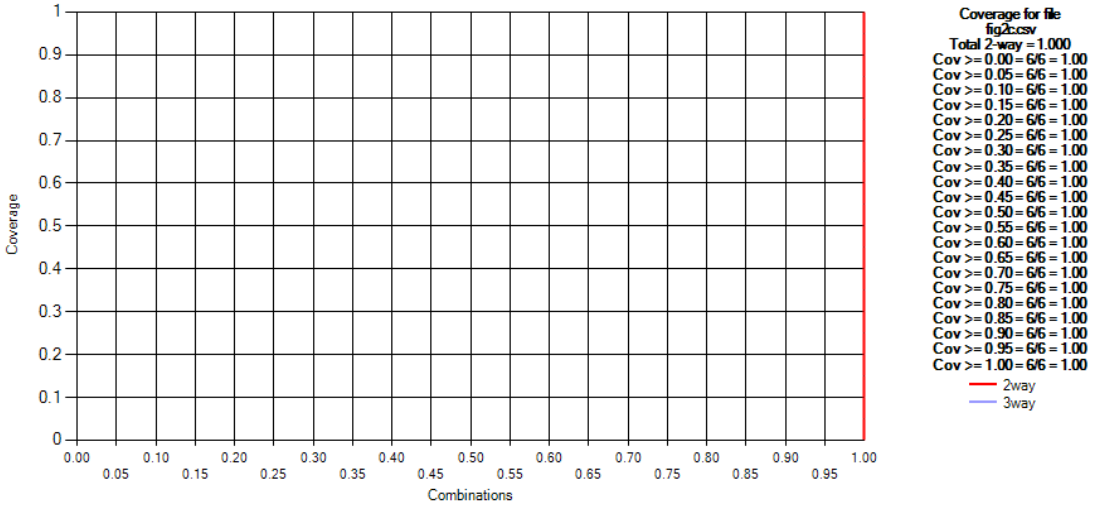
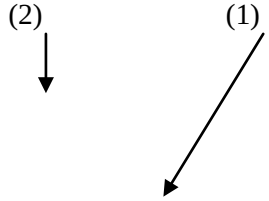



**Figure 7.** Adding test [1,0,1,0] provides 100% coverage, i.e., a covering array.

## *1.3 Using Combinatorial Coverage*

The methods described in this chapter were originally developed to analyze the state space coverage of spacecraft software [115]. As with many assurance efforts, a very thorough set of over 7,000 tests had been developed for each of three systems, but combinatorial coverage was not the goal. With such a large test suite, it seemed likely that a huge number of combinations had been covered, but how many? Did these tests provide 2-way, 3-way, or even higher degree coverage? If an existing test suite is relatively thorough, it may be practical to supplement it with a few additional tests to bring coverage up to the desired level.

The original test suites had been developed to verify correct system behavior in normal operation as well as a variety of fault scenarios, and performance tests were also included. Careful analysis and engineering judgment were used to prepare the original tests, but the test suite was not designed according to criteria such as statement or branch coverage. The system was relatively large, with 82 variables in a $1^3 2^{75} 4^2 6^2$ (three 1-value, 72 binary, two 4-value, and two 6-value). Figure 60 shows combinatorial coverage for this system (red = 2-way, blue = 3-way, green = 4-way). This particular test set was not a covering array, but pairwise coverage is still relatively good, with about 99% of the 2-way combinations having at least 75% of possible variable-value configurations covered (1), and 82% have 100% of possible variable-value configurations covered (2).



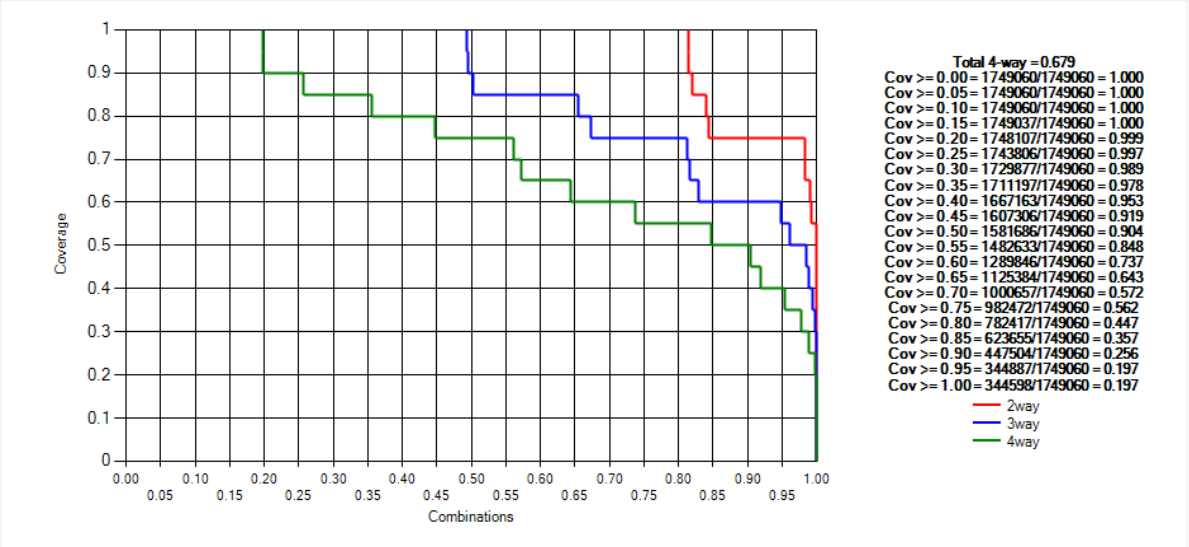



**Figure 8.** Configuration coverage for $1^3 2^{75} 4^2 6^2$ inputs.

Consider the graph in Figure 61. The label Φ indicates the percentage of combinations with 100% variable-value coverage, and M indicates the percentage of coverage for 100% of the *t*-way combinations. In this case 33% (Φ) of the combinations have full variable-value coverage, and all combinations are covered to at least the 50% level (M). So (1.0, 2)-completeness = Φ and (M, 2)-completeness = 100%, but it is helpful to have more intuitive terms for the points Φ and M. Note that Φ is the level of simple *t*-way coverage. Since all combinations are covered to at least the level of M, we will refer to M as the "*t-way minimum coverage*", keeping in mind that "coverage" refers to a proportion of variable-value configuration values. Where the value of *t* is not clear from the context, these measures are designated $\Phi_t$ and $M_t$. Using these terms we can analyze the relationship between total variable-value configuration coverage, *t*-way minimum coverage and simple *t*-way coverage.

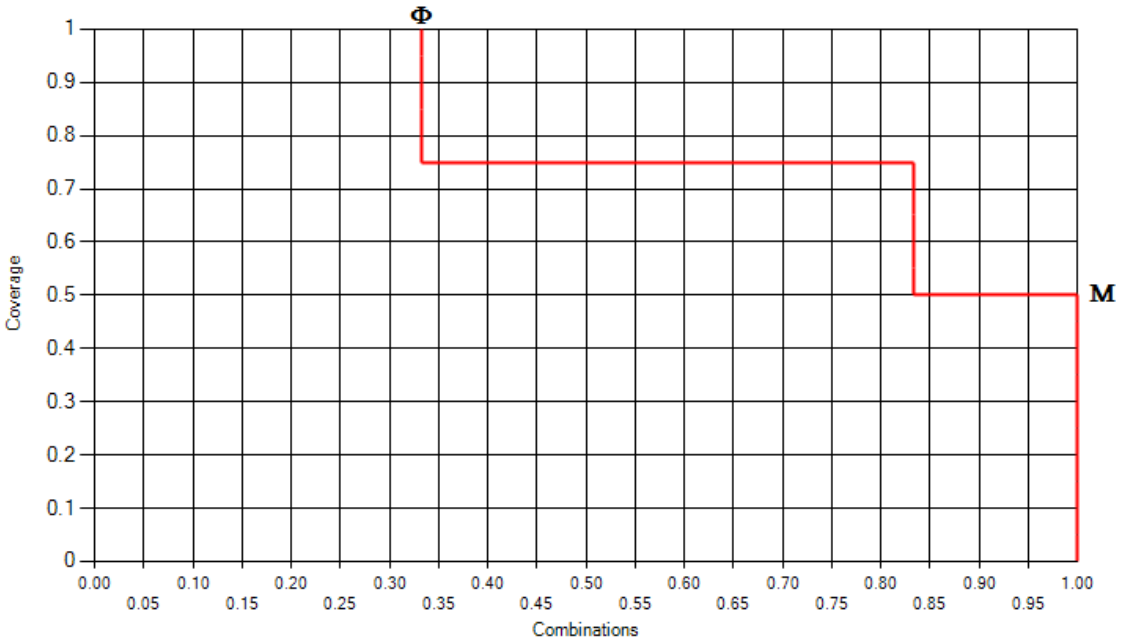



**Figure 9.** Example coverage graph for t = 2.

Let $S_t$ = total variable-value coverage, the proportion of variable-value configurations that are covered by at least one test. If the area of the entire graph is 1(i.e., 100% of combinations), then

$$S_t \geq 1 - (1- \Phi_t)(1- M_t)$$
$$S_t \geq \Phi_t + M_t - \Phi_t M_t \quad (1)$$

If a test suite has only one test, then it covers C($n$, $t$) combinations. The total number of combinations that must be covered is C($n$,$t$) × $v^t$, so the coverage of one test is $1/v^t$. Thus,

$$t\text{-way minimum coverage}: \quad M_t \geq \frac{1}{v^t} > 0. \quad (2)$$

Thus, for any non-empty test suite, $t$-way minimum coverage ≥ $1/v^t$. This can be seen in Figure 62, which shows coverage for a single test with binary variables, where 2-way minimum coverage is 25%. With only one test, 2-way full coverage is of course 0, and $S$ = total variable-value coverage (denoted "Total t-way" on the graph legend) is 25%.

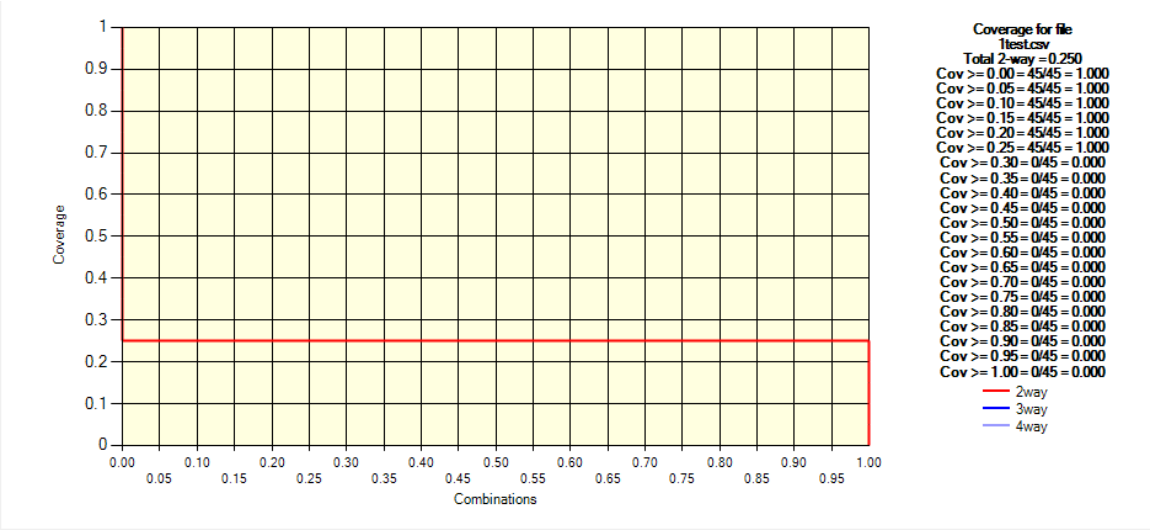



**Figure 10.** Coverage for one test, 10 binary variables.

## *1.4 Cost and Practical Considerations*

An important cost advantage introduced by coverage measurement is the ability to use existing test sets, identify particular combinations that may be missing, and supplement existing tests. In some cases, as in the example of Figure 60, it may be discovered that the existing test set is already strong with respect to a particular strength $t$ (in this case 2-way), and tests for $t$+1 generated. The tradeoff in cost of applying coverage measurement is the need to map existing tests into discrete values that can be analyzed by the coverage measurement tools (see Appendix C). For example, if the test set includes a field with various dollar amounts, it may be necessary to establish equivalence classes such as 0, 1..1000, >1000, etc.

These concepts can be used to analyze various testing strategies by measuring the combinatorial coverage they provide.

*All-values*: Consider the *t*-way coverage from one of the most basic test criteria, all-values, also called "each-choice" [2]. This strategy requires that every parameter value be covered at least once. If all parameters have the same number of values, $v$, then only $v$ tests are needed to cover all. Test 1 has all parameters set to $v_1$, Test 2 to $v_2$, and so on. If parameters have different numbers of values, where $p_1$ .. $p_n$ have $v_i$ values each, the number of tests required is at least $Max_{i=1,n}\, v_i$.

*Example*: If there are three values, 0, 1, and 2, for five parameters, then 0,0,0,0,0; 1,1,1,1,1; and 2,2,2,2,2 will test all values once. Since each test covers $1/v^t$ variable-value configurations, and no combination appears in more than one test, with $v$ values per parameter, we have

$$M_t\,(\textit{all-values}) \;\geq\; v\frac{1}{v^t} = \frac{1}{v^{t-1}}$$

So if all values are covered at least once, for all-values, minimum coverage $M \geq 1/v^{t-1}$. This relationship can be seen in Figure 63, which shows coverage for two tests with binary variables; 2-way minimum coverage = .5, and 3-way and 4-way coverage are .25 and .125 respectively.

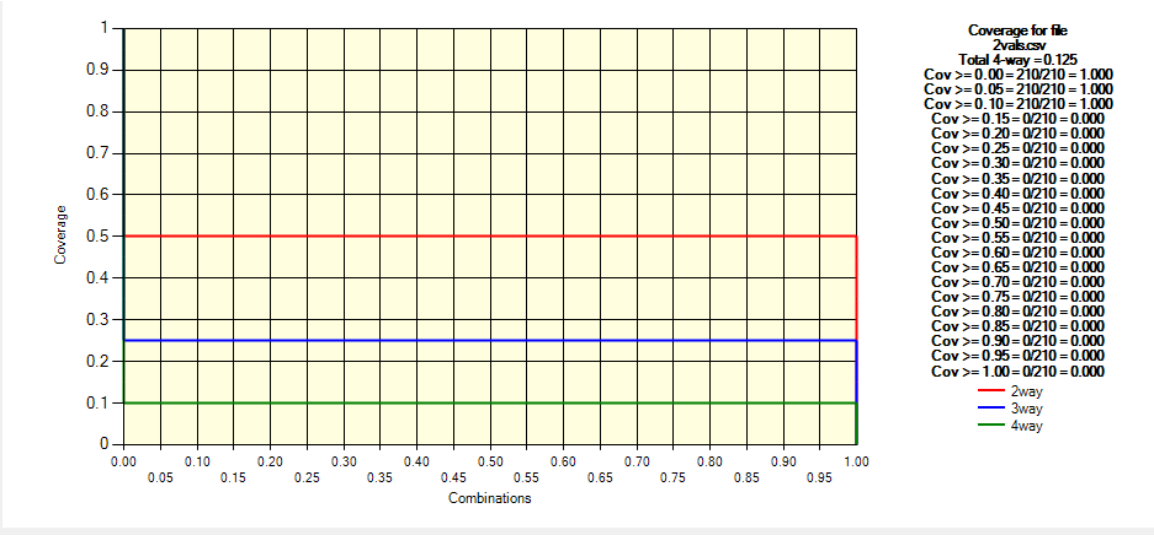



**Figure 11.** t-way coverage for two tests with binary values.

*Base choice*: Base-choice testing [2] requires that every parameter value be covered at least once and in a test in which all the other values are held constant. Each parameter has one or more values designated as base choices. The base choices can be arbitrary, but can also be selected as "more important" values, e.g., default values, or values that are used most often in operation. If parameters have different numbers of values, where $p_1$ .. $p_n$ have $v_i$ values each, the number of tests required is at least $1 + \Sigma_{i=1,n}\,(v_i - 1)$, or where all $n$ parameters have the same number of values $v$, the number of tests is $1+n(v-1)$. An example is shown below in Figure 64, with four binary parameters.

| | *a* | *b* | *c* | *d* |
|---|---|---|---|---|
| base: | 0 | 0 | 0 | 0 |
| test 2 | 1 | 0 | 0 | 0 |
| test 3 | 0 | 1 | 0 | 0 |

| | | | | |
|---|---|---|---|---|
| test 4 | 0 | 0 | 1 | 0 |
| test 5 | 0 | 0 | 0 | 1 |

**Figure 12.** Base choice test suite for a $2^4$ configuration

The base choice strategy can be highly effective, despite its simplicity. In one study of five programs seeded with 128 faults [79, 80], it was found that "although the Base Choice strategy requires fewer test cases than Orthogonal Arrays and AETG, it found as many faults [65]." In that study, AETG [52] was used to generate 2-way (pairwise) test arrays. We can use combinatorial coverage measurement to help understand this finding. For this example of analyzing base choice, we will consider $n$ parameters with 2 values each. First, note that the base choice test covers C($n$, $t$) combinations, so for pairwise testing this is C($n$, 2) = $n$($n$-1)/2. Changing a single value of the base test to something else will cover $n$-1 new pairs (in our example, *ab* and *ac* have new values in test 2, while *bc* is unchanged). This must be done for each parameter, so we will have the original base choice test combinations plus $n$($n$-1) additional combinations. The total number of 2-way combinations is C($n$, 2) × $2^2$, so for $n$ binary parameters:

$$\mathrm{M}_t\ (\textit{2-way binary base-choice}) = \frac{n(n-1)/2 + n(n-1)}{C(n,2)2^2} = \frac{C(n,2) + 2C(n,2)}{C(n,2)2^2}$$
$$= 3/4.$$

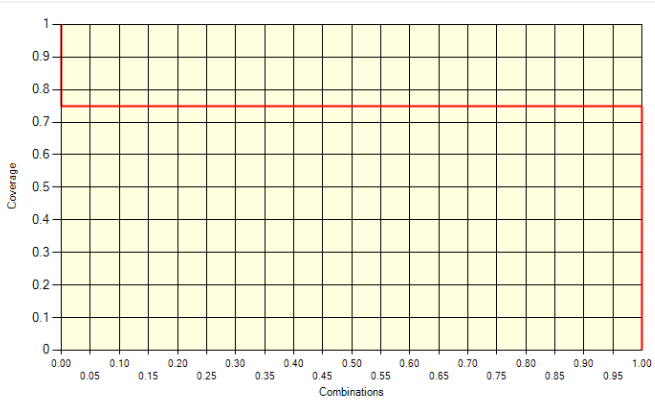



**Figure 13.** Graph of 2-way coverage for test set in Figure 64.

This can be seen in the graph in Figure 65 of coverage for Figure 64. Note that the 75% coverage level is independent of $n$. For $v > 2$, the analysis is a little more complicated. The base choice test covers C($n$, 2), and each new test covers $n$-1 new combinations, but we need $v$-1 additional tests beyond the base choice for $v$ values. So the proportion of 2-way combinations covered by the base choice strategy in general is:

$$\mathrm{M}_t\ (\textit{base-choice}) = \frac{C(n,2) + 2(v-1)C(n,2)}{C(n,2)v^2}$$

$$= \frac{1 + 2(v-1)}{v^2}$$

For example, with $v$ = 3 values per parameter, we still cover 55.6% of combinations, and with $v$ = 4, coverage is 43.75%. This analysis helps explain why the base choice strategy compares favorably with pairwise testing.

This equation can be generalized to higher interaction strengths. The base choice test covers C($n$,$t$) combinations, and each new test covers C($n$-1,$t$-1) new combinations, since the parameter value being varied can be combined with $t$-1 other parameters for each $t$-way combination. Base choice requires $n(v-1)$ additional tests beyond the initial one, so for any $n$, $t$ with $n \geq t$

$$\mathrm{M}_t\ (\textit{base-choice}) = \frac{C(n,t) + n(v-1)C(n-1,t-1)}{C(n,t)v^t} = \frac{C(n,t) + t(v-1)C(n,t)}{C(n,t)v^t}$$
$$= \frac{1 + t(v-1)}{v^t}$$

### *1.5 Analysis of ($t$+1)-way Coverage*

A $t$-way covering array by definition provides 100% coverage at strength $t$, but it also covers some ($t$+1)-way combinations (assuming $n \geq t$+1). Using the concepts introduced previously we can investigate minimal coverage levels for ($t$+1)-way combinations in covering arrays. Given a $t$-way covering array, for any set of $t$+1 parameters, we know that any combination of $t$ parameters is fully covered in some set of tests. Joining any other parameter with any combination of $t$ parameters in the tests will give a ($t$+1)-way combination, which has $v^{t+1}$ possible settings. For any set of tests covering all $t$-way combinations, the proportion of ($t$+1)-way combinations covered is thus $v^t/v^{t+1}$, so if we designate ($t$+1)-way variable-value configuration coverage as $S_{t+1}$, then

$$S_{t+1} \geq \frac{1}{v} \text{ for any } t\text{-way covering array with } n \geq t+1. \qquad (5)$$

Note that variable-value configuration coverage is not the same as simple $t$-way coverage, which gives the proportion of $t$-way configurations that are 100% covered. Clearly no set of ($t$+1)-way combinations will be fully covered with a $t$-way covering array if $N < v^{t+1}$, where $N$ = number of tests. For most levels of $t$ and $v$ encountered in practical testing, this condition will hold. For example, if $v$=3, then a 2-way covering array with less than $3^3$=27 tests can be computed (using IPOG-F) for any test problem with less than 60 parameters. So the proportion of combinations with full variable-value coverage, designated $\Phi$, will be zero for 3-way coverage for this example. And in general, designating ($t$+1)-way full variable-value configuration coverage as $\Phi_{t+1}$, if $N < v^{t+1}$, then $\Phi_{t+1} = 0$, for any $t$-way covering array with $n \geq t$+1.

As an additional illustration, we show that expression (5) can also be reached by noting that with a $t$-way covering array, unique ($t$+1)-way combinations can be identified as follows: traversing the covering array, for the first appearance of each $t$-way combination, append each of the $n$-$t$ parameters not contained in the $t$-way combination to create a ($t$+1)-way combination. This procedure counts each ($t$+1)-way combination C($t$+1, $t$) = $t$+1 times. The covering array contains $C(n,t)v^t$ $t$-way combinations, so the proportion of ($t$+1)-way combinations, $S_{t+1}$, in the $t$-way array is at least

$$\frac{C(n,t)v^t(n-t)/(t+1)}{C(n,t+1)v^{t+1}} = \frac{1}{v}$$

In most practical cases, the ($t$+1)-way coverage of a $t$-way array will be higher than 1/v .

### *1.6 Chapter Summary*

1. Many coverage measures have been devised for code coverage, including statement, branch or decision, condition, and modified condition decision coverage. These measures are based on aspects of source code and are not suitable for combinatorial coverage measurement.

2. Measuring configuration-spanning coverage can be helpful in understanding state space coverage. If we do use combinatorial testing, then configuration-spanning coverage will be 100% for the level of $t$ that was selected, but we may still want to investigate the coverage our test set provides for $t$+1 or $t$+2. Calculating this statistic can help in choosing between t-way covering arrays generated by different algorithms. As seen in the examples above, it may be relatively easy to produce tests that provide a high degree of spanning coverage, even if not 100%. In many cases it may be possible to generate additional tests to boost the coverage of a test set.

3. The following properties hold:

$$M_t \geq \frac{1}{v^t} > 0$$

$$S_t \geq \Phi_t + M_t - \Phi_t M_t$$

$S_{t+1} \geq \frac{1}{v}$ for any $t$-way covering array with $n \geq t+1$

if $N < v^{t+1}$, then $\Phi_{t+1} = 0$, for any $t$-way covering array with $n \geq t+1$