Combinatorial Coverage Measurement Concepts and Applications

D. Richard Kuhn[1], Itzel Dominguez Mendoza[2], Raghu N. Kacker[1], Yu Lei[3]

[1] National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{kuhn, raghu.kacker}@nist.gov

[2]Centro Nacional de Metrologia
Santiago de Querétaro,
Mexico
itzel.dominguezmendoza@nist.gov

[3]Computer Science and Engineering
Univ. of Texas
Arlington, TX, USA
ylei@uta.edu

***Abstract*—Empirical data demonstrate the value of *t*-way coverage, but in some testing situations, it is not practical to use covering arrays. However any set of tests covers at least some proportion of *t*-way combinations. This paper describes a variety of measures of combinatorial coverage that can be used in evaluating aspects of *t*-way coverage of a test suite. We also provide lower bounds on *t*-way coverage of several widely-used testing strategies, and describe a tool that analyzes test suites using the measures discussed in the paper.**

Keywords-component; combinatorial testing; factor covering array; state-space coverage; verification and validation (V&V); t-way testing; configuration model; t-way testing;

I. INTRODUCTION

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

Short of creating new test suites from scratch, one approach to obtaining the advantages of combinatorial testing is to measure the combinatorial coverage of existing tests, then supplement as needed. 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 an existing test suite covers almost all 3-way combinations, for example, then it may be sufficient for the level of assurance that is required. Determining the level of input or configuration state space coverage can also help in understanding the degree of risk that remains after testing. (In the remainder of the paper, the term "state space" may refer to either input or configuration space, since coverage measures apply to both.) If 90% - 100% of the relevant state space has been covered, then risk is likely to be smaller than would remain after testing that covers a much smaller portion of the state space.

This paper describes a set of measures of combinatorial coverage [1][2] and illustrates how these measures can be used in evaluating and comparing test suites or test strategies. We prove certain properties of coverage for various test strategies in the case where all parameters have the same number of values, and show how the measurement tool can be used to analyze test sets with mixed level parameters and constraints.

II. COMBINATORIAL 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 possible that it provides 2-way coverage or better. For example, random input generation may produce tests covering a high proportion of combinations [3]. In addition to evaluating structural metrics such as statement or branch coverage, for software assurance it would be helpful to know what percentage of 2-way, 3-way, etc. coverage has been obtained. The fault detection effectiveness of combinatorial testing clearly depends on tests covering *t*-way combinations [4][5], but not necessarily on the method of producing the tests.

Definition. *Variable-value configuration:* 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*, for a selection of three variables *a, c*, and *d* the set {*a*=0, *c*=1, *d*=0} is a variable-value configuration, and the set {*a*=1, *c*=1, *d*=0} is a different variable-value configuration.

Definition. *Simple t-way combination coverage:* 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 valid variable-values configurations are fully covered. In this and related definitions, "valid" configurations are those which are determined by user-

defined constraints to be relevant to the test problem, i.e., those that are not excluded by constraints. Not considered in this paper are mixed-strength arrays, where some subsets of variables may be covered to different strengths than others.

Example. Table I shows an example with four binary variables, *a*, *b*, *c*, and *d*, where each row represents a test. Of the six possible 2-way variable 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 Table 1 is 1/3 = 33.3%. There are four 3-way variable 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%. As shown later, test sets may provide strong coverage for some measures even if simple combinatorial coverage is very low.

TABLE I. TEST ARRAY WITH FOUR BINARY COMPONENTS

a	b	c	d
0	0	0	0
0	1	1	0
1	0	0	1
0	1	1	1

A test set that provides 100% simple combinatorial coverage for *t*-way combinations will also provide some degree of coverage for (*t*+1)-way combinations, (*t*+2)-way combinations, etc. For some applications, it may be useful to break out the coverage for *t*+*k*, for a particular level of *k*.

Definition. (*t* + *k*)-way simple combination coverage: For a given test set that provides 100% simple t-way coverage for *n* variables, (*t*+*k*)-way simple combination coverage is the proportion of valid (*t*+*k*)-way combinations of *n* variables for which all variable-value configurations are fully covered.

Simple *t*-way coverage measures the proportion of combinations of variables 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 possible variable-value 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 some measures with respect to such variable-value configurations. These measures may be more significant for fault detection than simple coverage.

***Definition*. Total variable-value configuration coverage:** *For a given combination of t variables, total variable-value configuration coverage is the proportion of variable-value configurations that are covered by at least one test case in a test set. This measure may also be referred to as total t-way coverage.*

For total (*t*+*k*)-way coverage where *k* = 1, Chen and Zhang [6] have proposed the *tuple density* metric. A special metric for (*t*+1)-way coverage is useful because (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 [4].

Definition. Tuple Density: Sum of *t* and the fraction of the covered (*t*+1)-tuples out of all possible (*t*+1)-tuples [6].

A related measure is (*p*,*t*)-completeness:

***Definition*. (*p, t*)-completeness:** *For a given set of n variables, (p, t)-completeness is the proportion of valid combinations that have configuration coverage of at least p* [7].

TABLE II. COVERAGE OF TEST SET IN TABLE I.

Vars	Configurations covered	Config coverage
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

simple 2-way coverage = 2/6 = 0.333
*total 2-way cove*rage = 19/24 = 0.791
(*.50, 2*)*-completeness* = 6/6 = 1.0
(*.75, 2*)*-completeness* = 5/6 = 0.833
(*1.0, 2*)*-completeness* = 2/6 = 0.333

Example. For Table 1 above, there are C(4, 2) = 6 possible variable combinations and C(4, 2) × 2^2 = 24 possible variable-value configurations, where C(*n*,*t*) is the number of combinations of *n* things taken *t* at a time, or "*n* choose *t*". 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 total variable-value 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 Table 1. 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 Table 1 uses variables with the same number of values, this is not essential for the measurement, and the same approach can be used to compute coverage for test sets in which parameters have differing numbers of values.

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

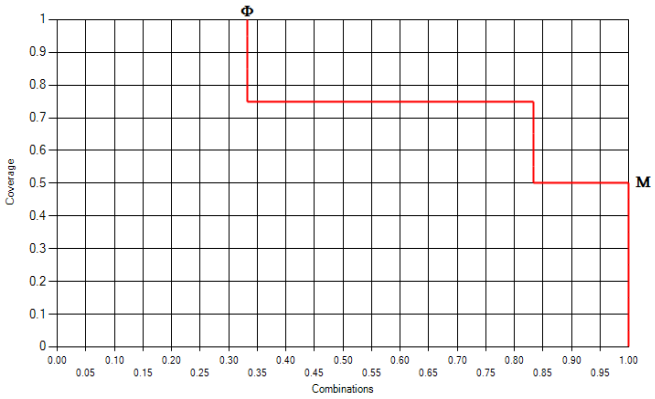


Figure 1. Graph of coverage from test data

The symbol Φ in Figure 1 indicates the proportion of combinations with 100% variable-value coverage, and M indicates the minimum proportion of coverage for all *t*-way variable combinations. In this case 33% (Φ) of the variable combinations have full variable-value coverage, and all variable combinations are covered to at least the 50% level (M). Note that Φ is the level of *simple t-way coverage*. Since all variable 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 Φ_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.

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

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

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 $C(n,t) \times v^t$, so the coverage of one test is $1/v^t$. Thus, $M_t \geq 1/v^t > 0$.

Example. The methods described in this paper were originally developed to analyze the input space coverage of spacecraft software [7]. A very thorough set of over 7,000 tests had been developed for each of three systems. At that time 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.

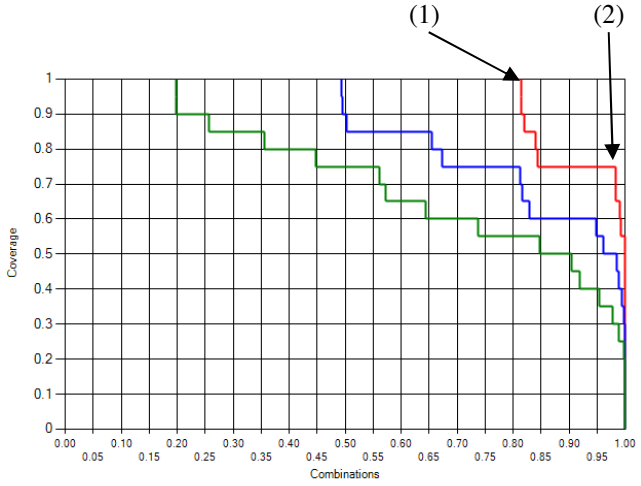


Figure 2. Configuration coverage for $1^3 2^{75} 4^2 6^2$ inputs.

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 the 82 variable configuration $1^3 2^{75} 4^2 6^2$ (three 1-value, 75 binary, two 4-value, and two 6-value). Figure 2. 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, because 82% of the 2-way combinations have 100% of possible variable-value configurations covered (1) and about 98% of the 2-way combinations have at least 75% of possible variable-value configurations covered (2).

III. Analysis of Test Strategies

These coverage metrics can be used to analyze various testing strategies by measuring the combinatorial coverage they provide. To illustrate this type of analysis, some examples are discussed in this section. The objective here is to understand the coverage aspects of test strategies, to aid testers in choosing among them or in determining additional tests that may be needed. For example, if no errors have been found with testing up to a certain level of t, how likely is it that (t+1)-way combinations will detect a fault? Measuring the level of total variable-value coverage for t+1 will show what proportion of (t+1)-way combinations have been covered so far, which may help in deciding whether to run a full (t+1)-way covering array. For example, if we have already covered more than 75% of the combinations at the next level of t, it may not be cost-effective to do additional testing. In addition to these practical considerations, this type of analysis helps to explain why some test strategies are effective.

A. All Values

Consider the t-way coverage from one of the most basic test criteria, all-values, also called "each-choice". 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 their first values, Test 2 to their second values, 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. As shown above, each test covers $1/v^t$ of the variable-value configurations, and no combination appears in more than one test, so with v values per parameter and thus v tests, we have

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

Therefore, for the all-values criterion, where all values are covered at least once, minimum coverage $M \geq 1/v^{t-1}$. We can also reach this result by noting that each test covers $C(n, t)$ combinations, so with v values the proportion of combinations covered is $vC(n,t)/C(n,t)v^t = 1/v^{t-1}$. This relationship can be seen in Figure 3. which shows coverage for two tests with ten binary variables; 2-way minimum coverage = .5, and 3-way coverage = .25.

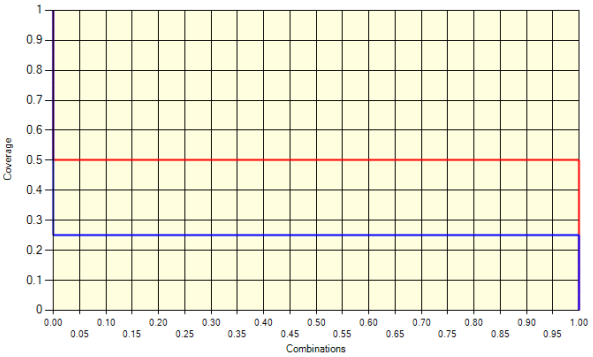


Figure 3. t-way coverage, 2 tests with binary values.

Base Choice

Base-choice testing [8] 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 "special interest" 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 Table III, with four binary parameters.

TABLE III. Base choice tests for 2^4 configuration

	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

The base choice strategy can be highly effective, despite its simplicity. In one study of five programs seeded with 128 faults [10], it was found that "although the Base Choice strategy requires fewer test cases than Orthogonal Arrays and AETG, it found as many faults." In that study, AETG [9] 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 test in which each parameter takes its base choice 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*, *ac*, and *ad* have new values in test 2, while *bc* and *bd* are unchanged). This must be done for each parameter, so we will have the original base 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:

$$M_t = \frac{n(n-1)/2 + n(n-1)}{C(n,2)2^2}$$
$$= \frac{C(n,2) + 2C(n,2)}{C(n,2)2^2}$$
$$= 3/4.$$

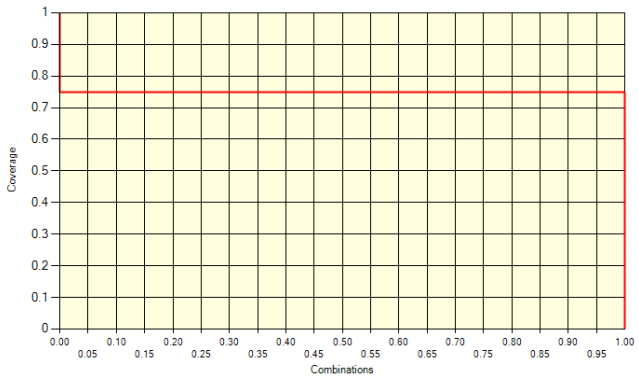


Figure 4. 2-way coverage for test set in Table III.

This can be seen in the graph in Figure 4. of coverage for Table III. Note that the 75% coverage level is independent of n.

This equation can be generalized to higher interaction strengths. Base choice testing requires $n(v\text{-}1)$ additional tests beyond the initial one, so for any n, t with $n \geq t$

$$M_t\,(base\text{-}choice) = \frac{C(n,t) + n(v-1)C(n-1,t-1)}{C(n,t)v^t}$$
$$= \frac{1 + t(v-1)}{v^t}$$

B. Modified condition decision coverage

Modified condition decision coverage (MCDC) is a test strategy required by the US Federal Aviation Administration for life critical software [11]. It is important to emphasize that MCDC is a coverage criterion for test suites normally applied to an entire program. In the analysis below, we analyze test sets with respect to one expression at a time. This analysis helps to explain why MCDC is effective. In practical applications, an MCDC test set applied to an entire program would be likely to have better combinatorial coverage than the results for individual expressions shown below.

MCDC Lower Bounds

An exhaustive analysis of all Boolean expressions of up to six variables has shown [11] that of n+1 tests can provide MCDC coverage for nearly all expressions, where n is the number of variables involved in a Boolean expression. A natural question to ask is then, what level of combinatorial coverage is provided by MCDC tests? This question can be addressed in two ways, by empirical data on combinatorial coverage for MCDC test sets, and by evaluating the MCDC test construction using methods described in NISTIR 7878. Shown below is one result on minimum coverage from MCDC tests, followed by data on combinatorial coverage of MCDC test sets for various numbers of variables.

An MCDC test set is constructed with *independence pairs*, where the boolean expression being tested evaluates to 0 for one test in the pair and to 1 for the other test. Because there are n+1 tests in the test set, the independence pairs overlap, such that each test belongs to an independence pair for two variables. An example is shown in Table IV [12]. Tests 0 and 1 are the pair showing independent effect of a, 1 and 2 for b, 2 and 3 for d, and 3 and 4 for c. Note that only the variable concerned changes value between the two tests of a pair, which changes the value of the expression and thus shows the effect of that particular variable or condition. (MCDC can be applied either to boolean variables or to conditions that evaluate to true/false.)

In the *unique-cause* form of MCDC, it must be shown that the expression changes value as one variable value is switched while others remain fixed. For example, $(a+b)(c+d)$ = 1 for the first test. Changing the value of a from 1 to 0 while other values remain fixed causes the expression to evaluate to 0. In this manner, one of the n variables is changed with each additional test, for a total of n+1 tests.

TABLE IV. (A+B)(C+D)

	a	b	c	d	$(a+b)(c+d)$
Test0	1	0	0	1	1
Test1	0	0	0	1	0
Test2	0	1	0	1	1
Test3	0	1	0	0	0
Test4	0	1	1	0	1

We can now show the following:

Theorem 1. If an MCDC test set exists for an expression, total combinatorial coverage for unique-cause tests = $\frac{1+t}{2^t}$.

Proof: Choose one test arbitrarily as the base test. The base test, test0, contains n variables with Boolean values $v_{01}..v_{0n}$. The base test, $test_0$, covers C(n,t) t-way combinations. With a total of n+1 tests, there are n tests in addition to the base test. Because MCDC requires that all variables take on both values, the negation of the base test values, $\bar{v}_{01}..\bar{v}_{0n}$ must appear in one other of the additional tests, of which there are n. To construct test 1, MCDC requires that one variable value is changed while others held fixed, which will

increase the number of covered combinations by n-1 pairs or C(n-1,t-1) t-way combinations. Continuing in this manner for each of the n variables adds a total of n×C(n-1,t-1) combinations covered to the initial coverage of C(n,t), so coverage $S_t = \dfrac{C(n,t) + nC(n-1,t-1)}{C(n,t)2^t}$

$= \dfrac{1+t}{2^t}$

□

Another definition of MCDC, *masking* MCDC, "allows any number of conditions to change so long as only the condition of interest has influence on the outcome of the expression" [11]. Masking MCDC is easier to satisfy, because there is more flexibility in the choice of values for the variables or conditions. For example, consider values in columns of an MCDC test set (see Figure 5.), where each row is a test and each column gives values for a particular variable. For each t-way combination of variables, test 0 has one set of values, $v_{01}..v_{0t}$. An independence pair of tests must exist for each parameter to show that changing that value of the parameter also changes the value of the expression, and that the particular parameter has independent effect on the expression value. Thus if the expression has value z for test0, there must be another test, i, such that parameter $p1$ has value v_{01} and the expression has value $\bar{z}$. Similarly for p_2 there must be a third test, j, such that p_2 has value v_{i2} and the expression value switches back to z. Thus for any pair of parameters there are at least three distinct value pairs on at least three rows: v_{01}, v_{02} in test 0, v_{01},v_{i2} in test i, and v_{i2}, v_i* in some other row j, where * is any of the other parameters $\neq p_2$. Similarly there are at least t+1 t-way combinations on t+1 rows for any t parameters.

	p1	*p2*	. . .
Test0	v_{01}	v_{02}	. . .
.. . .			
Testi	$\bar{v}_{01}$	v_{i2}	. . .
. . .			
Testj	v_{i1}	$\bar{v}_{i2}$	. . .
. . .			

Figure 5. MCDC tests.

Theorem 2. With the masking form of MCDC, coverage may exceed that for unique-cause form, so in general,

MCDC total coverage $\geqslant \dfrac{1+t}{2^t}$.

Proof: Because masking MCDC does not require all values other than the parameter under consideration to remain fixed, there may be more than t+1 distinct combinations. Therefore because there are 2^t possible combination settings for t parameters, the proportion of total coverage is at least $\frac{1+t}{2^t}$. □

MCDC Coverage Upper Bound

We can easily derive an upper bound on combinatorial coverage by noting that each test covers C(n,t) combinations, so with n+1 tests the proportion of covered combinations is at most

$$\frac{(n+1)C(n,t)}{2^t C(n,t)} = \frac{n+1}{2^t}$$

Note that if $n+1 < 2^t$, then the number of tests is insufficient to cover all 2^t values for any t parameters, so $\Phi_t = 0$. So for MCDC total combinatorial coverage S_t, $\dfrac{t+1}{2^t} \leq S_t \leq \dfrac{n+1}{2^t}$.

C. (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). Given a t-way covering array, 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 total (t+1)-way variable-value configuration coverage as S_{t+1}, then $S_{t+1} \geq 1/v$, for any t-way covering array with $n \geq t$+1.

For practical testing, this observation means that when v is small, we may gain a lot of efficiency by extending a t-way test suite to t+1, rather than re-run a (t+1)-way test suite from scratch. For example, with binary variables, if we have run a 3-way covering array then we have tested half of the 4-way combinations as well. With a mixed-level array, a coverage measurement tool will be needed to identify the level of t-way coverage achieved.

Clearly, where Φ_{t+1} = (t+1)-way full variable-value configuration coverage, if $N < v^{t+1}$, then $\Phi_{t+1} = 0$, for any t-way covering array with $n \geq t$+1 where N = number of tests. For many 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 $\Phi = 0$ for 3-way coverage for this example. Alternatively, note that if $N > v^{t+1}$ then Φ_{t+1} may exceed 0. For example, with n = 360 parameters, t = 2, Φ_{t+1} = 0.002, for a covering array of 39 tests computed by ACTS. In this case, Φ_{t+1} is low despite 360 parameters, but with 39 tests, 13,233 of C(360,3)=7,711,320 3-way variable combinations are covered with all 27 values purely by chance.

D. *Very Large Covering Arrays*

Covering array construction is a difficult problem [13], but good algorithms are available now for many or most practical testing applications [14][15][16][17][18]. However, for applications with hundreds of parameters, the most commonly used algorithms require long computation times, or may not complete at all. This is particularly true for higher-strength coverage, above 3-way. The Combinatorial Coverage Measurement (CCM) tool is currently being used to evaluate coverage for a problem with 358 factors, in a $2^{352}3^{2}4^{1}5^{2}14^{1}$ design. Test arrays are generated randomly, then evaluated using CCM and tests supplemented to provide an adequate level of coverage (generally exceeding 98%) up to 6-way. Random test generation is trivial and completes in less than 1 second, and CCM evaluation times range from seconds to several hours for higher strength levels.

File:	apl70.csv
No. Tests:	70
No. Parameters:	358

t-way	Combinations	Var/Val	Var/Val cov.	Invalid	Total
2-way	63,903	271,478	270,459	0	99.6246473 %
3-way	7,583,156	66,370,316	65,100,347	0	98.0865422 %
4-way	673,005,095	12,131,841,873	11,366,077,530	0	93.6879774 %

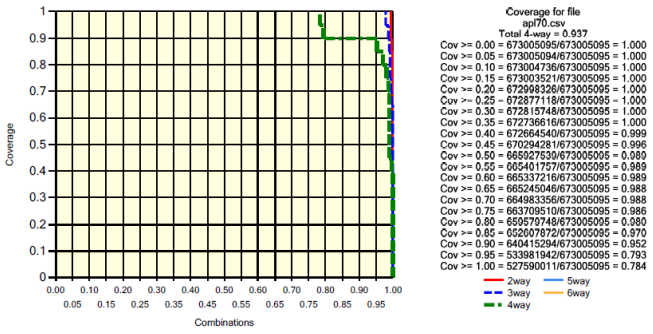


Figure 6. Coverage for 70 tests, $2^{352}3^{2}4^{1}5^{2}14^{1}$ design

IV. Coverage Measurement Tool

Measures described above have been implemented in a feature-rich tool, Combinatorial Coverage Measurement (CCM), which is designed to handle large files with mixed level parameters and constraints. The input file is a comma-separated values (CSV) file where each row is a test and each column represents a parameter. Constraints can be specified at the beginning of the input file, or through a graphical user interface. Coverage measurements can be displayed as graphs or heat maps, and detailed reports generated. An option allows for test sets to be automatically supplemented with additional tests to bring coverage up to a specified level (may be less than 100% if desired to reduce test set size). Combinations that are not covered in the input test set can be exported for post-processing. CCM has the ability to automatically detect parameter values for discrete values, or the user may define equivalence classes for continuous-value parameters. Input test files can be very large, limited only by system resources, and the tool has been applied to test sets with up to 625 binary parameters on a basic laptop with dual-core processor and 8 GB of memory.

A. *Features*

The main screen contains controls to load the file containing the set test; it will show the result of the measurement and charts.

If all tests and parameters in the input file should be loaded, just click on "Load input file" and select the test file to be analyzed. If just some tests or parameters are needed, before loading the input file, the number of tests and parameters should be specified using the numeric fields above on the left and pressing "Set number of tests and parameters". Note that if all tests are to be loaded, it is not necessary to set the number of tests and parameters, provided that parameters have values (no more than as indicated by the "Max values per parameter" field). The tool will read in all tests and discover parameter values that will be used in the measurement process. The tool can also process continuous-valued parameters such as account balances, distances, or others with a large range of possible values. See the "Specifying boundaries" discussion below. The parameters and their values are shown; they can be modified specifying boundaries, and adding or removing values. Clicking in a column will show the values below in order to be modified.

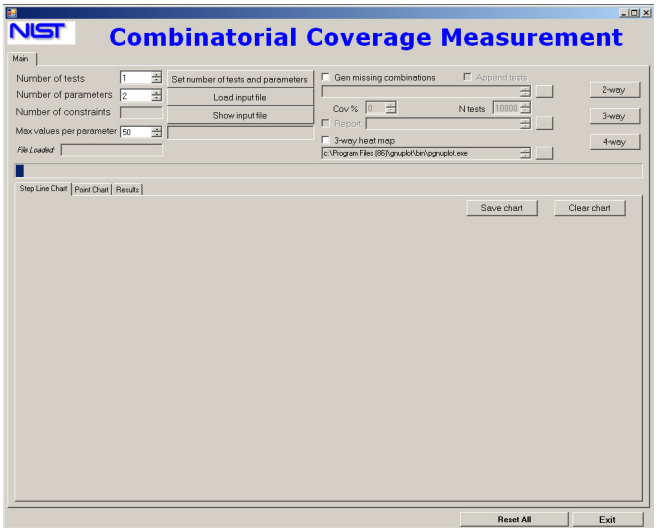


Figure 7. CCM main screen

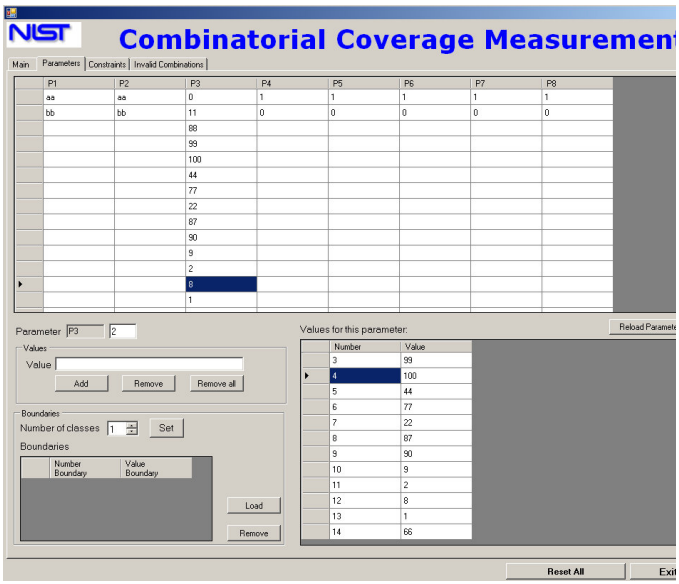
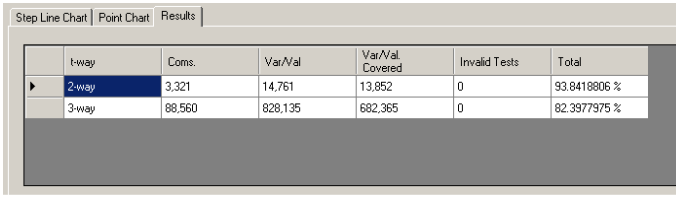


Figure 8. CCM parameter data screen.

Charts display coverage measurements described previously and may be saved to an image file. After the file loads, coverage measures may be computed by clicking on the appropriate button.



t-way	Coms.	Var/Val	Var/Val Covered	Invalid Tests	Total
2-way	3,321	14,761	13,852	0	93.8418806 %
3-way	88,560	828,135	682,365	0	82.3977975 %

Figure 9. CCM coverage report screen.

Combinatorial Coverage Measurement Tool

12/12/2012

File:	apl-set2.csv
No. Tests:	7617
No. Parameters:	82

t-way	Combinations	Var/Val	Var/Val cov.	Invalid	Total
2-way	3,321	14,761	13,852	0	93.8418806 %
3-way	88,560	828,135	682,365	0	82.3977975 %

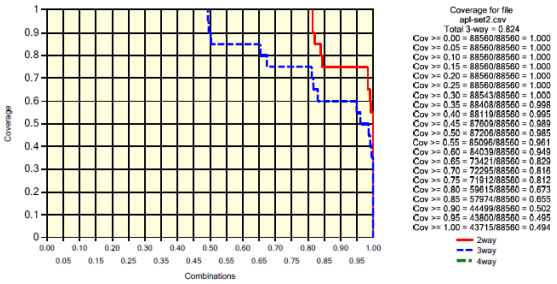


Figure 10. CCM coverage chart.

B. Boundaries and continuous-value parameters

For continuous-valued parameters, equivalence classes are specified by indicating the number of value classes and boundaries between the classes. Boundaries may include decimal values. Where the boundary between two classes c_1 and c_2 is x, the system places input values $< x$ into c_1 and values $\geq x$ in c_2.

	Number	Value	ValueBoundary	ReferenceBound
	0	0	0	0 <= 25
	1	11	0	11 <= 25
	2	88	2	88 >= 51
	3	99	2	99 >= 51
▶	4	100	2	100 >= 51
	5	44	1	26 <= 44<=50
	6	77	2	77 >= 51
	7	22	0	22 <= 25
	8	87	2	87 >= 51
	9	90	2	90 >= 51
	10	9	0	9 <= 25
	11	2	0	2 <= 25

Figure 11. CCM range variable boundaries.

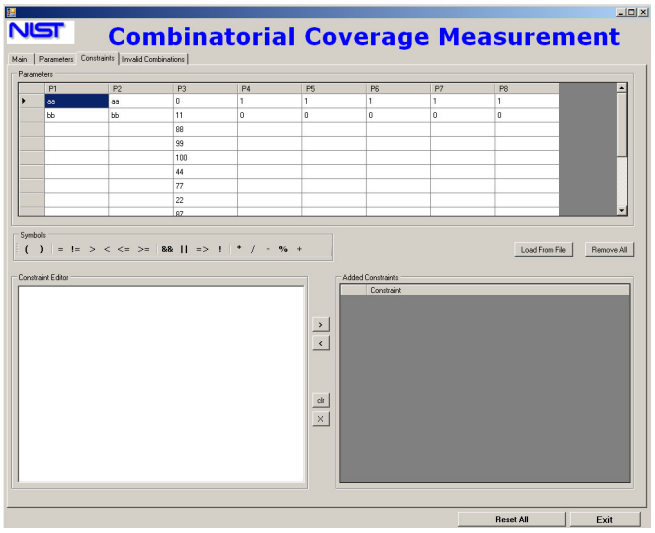


Figure 12. CCM constraint screen.

C. Constraints

Constraints may be included in the input file or entered interactively. Each line will be considered a separated constraint. The image below shows an example of an input file with constraints. The first two lines are the constraints; followed by the tests, where each column corresponds to a parameter.

	A	B	C	D	E
1	P1 = "Linux" => (P2 != "Explorer")				
2	P5="MySQL"=>(P1="Windows")				
3	Apple OS	Explorer	Ipv4	AMD	MySQL
4	Apple OS	Explorer	Ipv4	AMD	Oracle
5	Apple OS	Explorer	Ipv4	AMD	Sybase
6	Apple OS	Explorer	Ipv4	Intel	MySQL
7	Apple OS	Explorer	Ipv4	Intel	Sybase
8	Apple OS	Explorer	Ipv6	AMD	Oracle
9	Linux	FireFox	Ipv4	Intel	Oracle
10	Linux	FireFox	Ipv4	Intel	Sybase
11	Linux	FireFox	Ipv6	AMD	MySQL
12	Linux	FireFox	Ipv6	AMD	Oracle
13	Linux	FireFox	Ipv6	AMD	Sybase

Figure 13. CCM input file layout.

The constraints are shown in the main window, if no constraints are specified in the input file, they can be added interactively either by typing or selecting operators from the tool bar. Three types of operators can be used: (1) Boolean operators including &&, ||, =>; (2) Relational operators including =, !=, >, <, >=, <=; and (3) Arithmetic operators including +, -, *, /, %.

In addition to incorporating constraints in computing measures, the tool will identify any tests that do not satisfy the specified constriants. The following syntax can be used to specify constraints:

```
<Constraint>  ::= <Simple_Constraint>
                | <Constraint> <Boolean_Op> <Constraint>
<Simple_Constraint> ::= <Term> <Relational_Op> <Term>
<Term> := <Parameter>
        | <Parameter> <Arithmetic_Op> <Parameter>
        | <Parameter> <Arithmetic_Op> <Value>
<Boolean_Op>   := "!" | "&&" | "||" | "=>"
<Relational_Op> := "=" | "!=" | ">" | "<" | ">=" | "<="
<Arithmetic_Op>:= "+" | "-" | "*" | "/" | "%"
<Value> := <Integer_Value> | <Boolean_Value> | <Enum_Value>
```

Figure 14. CCM constraint syntax.

Examples of constraints that can be specified include:

- `(P1 = "Windows")=>(P2 = "IE" || P2 = "FireFox" || P2 = "Netscape")`, where P1 is a parameter for OS and P2 is a parameter for Browser. *If OS is Windows, then Browser must be IE, FireFox, or Netscape.*

- `(P1 > 100) || (P2 > 100)`, where P1 and P2 are two parameters of type Number or Range. *P1 or P2 must be greater than 100.*

- `(P1 > P2)=>(P3 > P4)`, where P1, P2, P3, and P4 are parameters of type Number or Range. *If P1 is greater than P2, then P3 must be greater than P4.*

Constraint handling is implemented using an open source constraint solver called Choco [19]. Choco is used to determine if a test satisfies all constraints by converting this check to a constraint satisfaction problem. Choco is designed to handle arbitrary constraint objects, so test parameters are encoded into this form before invoking the constraint solver. The Choco solver is an independent module as used in CCM, and could be replaced by a different constraint solver.

D. *Invalid Combinations*

The invalid combinations will be shown if constraints are specified. If any coverage measurement has been specified, the invalid combinations will be generated. When all the invalid combinations have been generated they will be shown in a CCM window. Combinations determined to be invalid by constraints should not appear in the test set, so these are identified by marking in the leftmost column of the display as shown below.

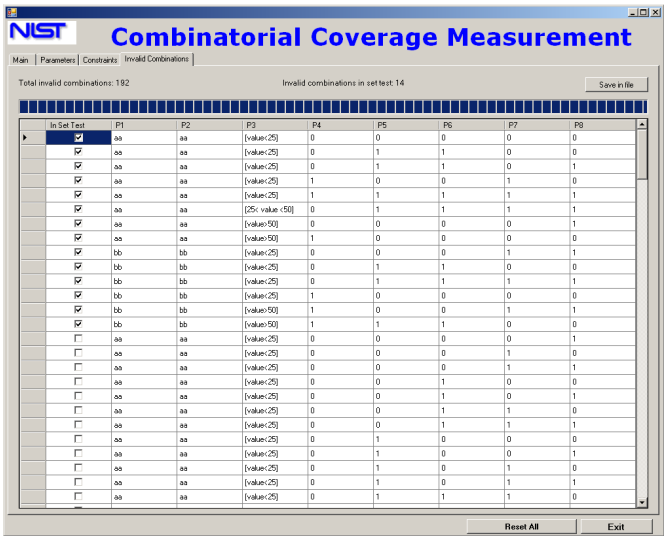


Figure 15. CCM invalid combinations screen.

A report may be produced that will include the following quantities:

- *Total invalid combinations*: Number of all invalid combinations based on possible parameter values and constraints specified.
- *Invalid combinations in set test*: Number of invalid combinations that occur in the set test.

V. Related Work

Although relatively new, combinatorial coverage has been discussed in some earlier papers. Some of the concepts discussed in this paper were introduced in [7], which also included an application of an early version of the tool to the analysis of large test suites; additional

measurement concepts were covered in [2]. A NIST tech report [1] extended this work to include the measures described in Sect. II. Tuple density is described by [6].

Also relevant are methods and tools for extending an array to provide *t*-way coverage. This problem was considered in [20], and several currently available covering array generators provide the capability, including PICT [21] and ACTS [13]. A significant difference with these tools is that they evaluate only whether all variable-value configurations are covered for each combination, which we have referred to as simple *t*-way coverage. The measures introduced in this paper can thus be considered to provide a more "fine grained" set of combinatorial coverage measures.

VI. Conclusions

An extensive body of empirical work shows that combinatorial testing can be a very efficient component of software assurance. The key aspect of combinatorial methods is to cover *t*-way combinations sufficiently well to detect faults, but it is not essential that tests be generated as covering arrays. Although covering arrays are generally the most compact way of achieving t-way coverage, they are not always practical. For example, regulations or contractual requirements may specify a particular type of testing, such as MCDC, or existing test sets may be used to reduce cost. In such circumstances, it may be desirable to compute the *t*-way coverage provided by the test set. CCM is an easy to use, practical tool to compute combinatorial coverage, which accommodates parameter constraints and mixed level variables.

The most basic measure is simple combinatorial coverage – the proportion of combinations for which t-way coverage is achieved. A more useful measure is total coverage – the proportion of t-way combination settings covered. A test set may have a relatively low level of simple coverage despite good total coverage, such as the example in Table I, in which only 33% of the 2-way combinations were covered but total coverage exceeded 79%. Using CCM to measure total combinatorial coverage for a test set, then supplementing tests to achieve a desired level of coverage, can provide strong interaction testing in situations where practical considerations rule out construction of tests from scratch using covering arrays.

References

[1] D.R. Kuhn, R. Kacker, Y. Lei. Combinatorial Coverage Measurement, NIST IR 7878, Sept. 2012. http://dx.doi.org/10.6028/NIST.IR.7878

[2] Kuhn, D. R., Kacker, R. N., & Lei, Y. (2010). Practical combinatorial testing. *NIST Special Publication*, *800*, 142.

[3] A. Arcuri, L. Briand, "Formal Analysis of the Probability of Interaction Fault Detection Using Random Testing," *IEEE Trans. Software Engineering*, 18 Aug. 2011. IEEE Computer Society, http://doi.ieeecomputersociety.org/10.1109/TSE.2011.85

[4] D.R. Kuhn, D.R. Wallace, Jr. A.M. Gallo, Software fault interactions and implications for software testing, Software Engineering, IEEE Transactions on, 2004

[5] D.R. Kuhn, M.J. Reilly. An investigation of the applicability of design of experiments to software testing. Proceedings of 27th NASA/IEEE Software Engineering Workshop, Greenbelt, Maryland, 2002; 91–95.

[6] B. Chen, J. Zhang, Tuple Density: A New Metric for Combinatorial Test Suites, *Proceedings of the 33rd International Conference on Software Engineering*, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. ACM 2011, ISBN 978-1-4503-0445-0

[7] J.R. Maximoff, M.D. Trela, D.R. Kuhn, R. Kacker, "A Method for Analyzing System State-space Coverage within a t-Wise Testing Framework", *IEEE International Systems Conference 2010*, Apr. 4-11, 2010, San Diego.

[8] Ammann, P. E. & Offutt, A. J. (1994). Using formal methods to derive test frames in category-partition testing, *Proc. Ninth Annual Conf. Computer Assurance* (COMPASS'94),Gaithersburg MD, IEEE Computer Society Press, pp. 69-80.

[9] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. IEEE Transactions On Software Engineering, 23(7):437–444, 1997.

[10] M. Grindal, J. Offutt, S.F. Andler, Combination Testing Strategies: a Survey, *Software Testing, Verification, and Reliability*, v. 15, 2005, pp. 167-199.

[11] J. J. Chilenski, An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion, *Report DOT/FAA/AR-01/18*, April 2001, 214 pp.

[12] Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., & Rierson, L. K. (2001). *A practical tutorial on modified condition/decision coverage*. National Aeronautics and Space Administration, Langley Research Center, p. 74.

[13] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG: a General Strategy for t-way Software Testing, *Proc., IEEE Engineering of Computer Based Systems 2007*, pp. 549 – 556.

[14] R. Bryce, C.J. Colbourn. The Density Algorithm for Pairwise Interaction Testing, *Journal of Software Testing, Verification and Reliability*, August 2007

[15] Bryce, R. C.J. Colbourn, M.B. Cohen. *A Framework of Greedy Methods for Constructing Interaction Tests*. The 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, pages 146-155. (May 2005).

[16] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, W. B. Mugridge, Constructing test suites for interaction testing. Proceedings of 25th IEEE International Conference on Software Engineering, 2003.

[17] Charles J. Colbourn , Myra B. Cohen, A Deterministic Density Algorithm for Pairwise Interaction Coverage, Proc. of the IASTED Intl. Conference on Software Engineering, 2004

[18] Pairwise Testing Home Page: http://pairwise.org

[19] The Choco Constraint Solver, http://www.emn.fr/z-info/choco-solver/index.html.

[20] A. Hartman and L. Raskin. Problems and Algorithms for Covering Arrays. Discrete Mathematics, 284(1-3):149–156, 2004.

[21] J. Czerwonka, *"Pairwise testing in real world: Practical extensions to test case generator"*, Proceedings of 24th Pacific Northwest Software Quality Conference, October 9–11, 2006, Portland, Oregon, USA, pp. 419–430, (2006).

Note: Identification of certain commercial products in this article does not imply recommendation by NIST, nor does it imply that the products identified are necessarily the best available for the purpose.