Measuring and Specifying Combinatorial Coverage of Test Input Configurations

D. Richard Kuhn[1], Raghu N. Kacker[1], Yu Lei[2]

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

[2]*Computer Science and Engineering*
Univ. of Texas Arlington
Arlington, TX 76019
ylei@uta.edu

***Abstract*—** A key issue in testing is how many tests are needed for a required level of coverage or fault detection. Estimates are often based on error rates in initial testing, or on code coverage. For example, tests may be run until a desired level of statement or branch coverage is achieved. Combinatorial methods present an opportunity for a different approach to estimating required test set size, using characteristics of the test set. This paper describes methods for estimating the coverage of, and ability to detect, *t*-way interaction faults of a test set based on a covering array. We also develop a connection between (static) combinatorial coverage and (dynamic) code coverage, such that if a specific condition is satisfied, 100% branch coverage is assured. Using these results, we propose practical recommendations for using combinatorial coverage in specifying test requirements.

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

I. INTRODUCTION

Specifying test coverage requirements is typically a difficult and imprecise process for "black box" testing, where no source code is used. A test goal may be to positively demonstrate a collection of specified features, often by a single test for each feature or option. Such a process is not adequate for robustness or reliability testing, because simply showing that a particular input can demonstrate the feature does little to prove that it is adequate for the wide range of inputs likely to be encountered in real-world use.

A more thorough approach involves exercising the system with a broader range of inputs, often through methodologies such as "fuzz testing" or other use of random input data. While useful for discovering errors, this approach still does not give a sound measure of the extent to which a system is capable of operating correctly for all inputs. Alternatively, an operational profile may be developed which tests the system according to the statistical distribution of inputs that occur in operational use. This process can provide reasonable confidence for the system's behavior in normal operation, but may miss the rare input configurations that can result in a failure. A common approach for high assurance in these cases is to supplement testing with tests designed to exercise the system with rare scenarios, based on experience or engineering judgment.

This approach is clearly dependent on the skill of testers, and it may leave a large proportion of the possible input space untested. It also provides no quantitative measure of the proportion of significant input combinations that have been tested. Therefore, if test services are to be contracted out, there is little sound basis for developers to specify the level of testing required, or for testers to prove that testing has been adequate for the required assurance level. This paper describes measurement methods derived from combinatorial testing that can be used in analyzing the thoroughness of a test set, based on characteristics of the test set separate from its coverage of executable code.

II. COMBINATORIAL COVERAGE

Combinatorial coverage measures the proportion of *t*-way combinations of variable settings included in a test set, for specified levels of *t*. For example, with three binary variables *a*, *b*, and *c*, there are 12 possible 2-way settings: *ab* = 00, 01, 10, or 11, and likewise for *ac* and *bc*. A set of two tests, *abc* = 000 and *abc* = 001, covers five of the 12 possible 2-way settings: *ab*=00, *ac*=00, *bc*=00, *ac*=01, and *bc*=01, for total 2-way coverage of 5/12. In addition to the total combinatorial coverage in this example, other combinatorial measures are meaningful for software testing, as explained in this paper. To understand the significance of these measures, it is helpful to review the basics of combinatorial methods in testing.

Combinatorial testing [1][2][3][4][5] is based on the observation that not every parameter contributes to every failure and most failures are triggered by a single parameter value, or interactions between a small number of parameters, generally two to six [5], a relationship known as the *interaction rule*. An example of a single-value fault might be a buffer overflow that occurs whenever the length of an input string exceeds a particular limit. Only a single condition must be true to trigger the fault: *input length* > *buffer size*. A 2-way interaction fault is more complex, because two particular input values are needed to trigger the fault. One example is a search/replace function that only fails if both the search string and the replacement string are single characters. If one of the strings is longer than one character, the code does not fail, thus we refer to

this as a 2-way interaction fault. The effectiveness of a software testing technique, including combinatorial testing, depends on whether test settings corresponding to the actual faults are included in the test sets. When test sets do not include settings corresponding to actual faults, the faults will not be detected. Matrices known as *covering arrays* can be computed to cover all *t*-way combinations of variable values, up to a specified level of *t* (typically $t \leqslant 6$), making it possible to efficiently test all such *t*-way interactions [3][6]. As with all testing, it is necessary to select a subset of values for variables with a large number of values, and test effectiveness is also dependent on the values selected, but combinatorial testing has been shown to be highly effective.

Combinatorial coverage [7][8][9][10][11] measures address the question of what proportion of possible settings of any *t* variables are covered by a test set. If the test set is a *t*-way covering array, then *t*-way 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 a high percentage of 2-way coverage or better, and thus may be a high quality test set from the standpoint of exercising interactions. These measures have been applied on a pilot project basis to IV&V for NASA software, with successful results indicating further investigation [11].

The effectiveness of a test set in detecting interaction faults clearly depends on tests covering *t*-way combinations, but not necessarily on the method of producing the tests. A *t*-way covering array is guaranteed to produce 100% coverage of combinations containing up to *t* variables, but a randomly generated test set may also produce 100% *t*-way combination coverage if enough tests are generated. Note that the combination coverage of random tests increases with the number of variables [4]. Thus in many ways, comparisons of "combinatorial vs. random testing" present a false dichotomy – all tests provide some degree of combinatorial coverage, and randomly generated tests can cover a high proportion of combinations for some configurations of variables and number of values per variable [12]. The definitions below are useful in measuring combinatorial coverage [8]:

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, i.e., the variable-value configuration is a particular setting 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.

Example. Table I shows 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 2/6 = 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 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

Simple *t*-way coverage measures the proportion of combinations of variables for which *all* configurations of *t* variables are fully covered, or 33% for Table I. It is also useful to measure the number of combinations covered out of all possible combinations.

Definition. *Total variable-value configuration coverage*: For a given combination of *t* variables, total variable-value configuration coverage is the proportion of all *t*-way 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.

The number of *t*-way combinations in an array of *n* variables is $C(n,t) = n!/(n\text{-}t!)t!$, or "*n* choose *t*" in combinatorics, the number of ways of taking *t* out of *n* things at a time. Suppose each variable has *v* values, then each set of *t* variables has v^t configurations, so the total number of possible combination settings is $v^t \times C(n, t)$. Any test set covers at least some fraction of this amount. For the array in Table I, there are C(4,2) = 6 possible variable combinations and $C(4,2)\times 2^2 = 24$ 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, so the total variable-value configuration coverage is 19/24 = 79%. But only two, *bd* and *cd*, are covered with all 4 value pairs. So for simple *t*-way coverage, we have only 33% (2/6) coverage, but 79% (19/24) for total variable-value configuration coverage. 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.

Figure 1 shows a graph of the coverage data for the tests in Table 1. Coverage is given as the Y axis, with the percentage of combinations reaching a particular coverage level as the X axis.

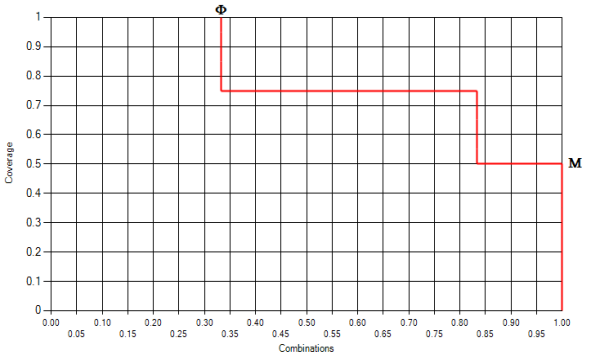


Figure 1. Graph of coverage from test data

Note from Figure 1 that all of the six 2-way combinations of variables are covered to at least the .50 level, 83% are covered to the .75 level or higher, and a third have 100% of variable-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%. 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; here, $t = 2$. 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). Since all variable combinations are covered to at least the level of M, we will refer to M as the "*t-way minimum coverage*". Where the value of *t* is not clear from the context, these measures are designated Φ_t and M_t.

Suppose S_t = total variable-value coverage (i.e., the proportion of variable-value configurations that are covered by at least one test). It can be shown that [8]:

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

If a test set has only one test, then it covers $C(n, t)$ combinations, so the total variable-value coverage S_t of a test set containing one test is $C(n, t)/ v^t \times C(n, t)$. Thus for any test set, $M_t \geq 1/v^t > 0$. Note that a test set which provides 100% simple combinatorial coverage for *t*-way combinations will also provide some degree of higher strength, $(t+k)$-way coverage (the interaction level *t* in *t*-way combinations is referred to as *strength*). It can be shown that for a *t*-way covering array, $M_{t+1} \geq 1/v^t$ [8].

III. Fault Coverage

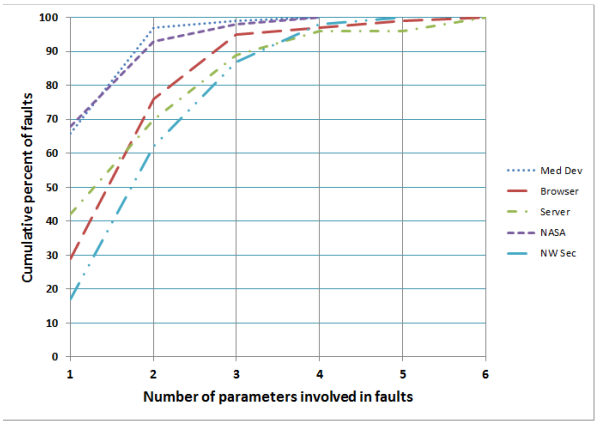
As described in Section II, *combinatorial coverage* measures the extent to which *t*-way combination settings have been included in a test set. Combinatorial coverage is useful in a variety of testing problems, but estimating the usefulness of *t*-way testing also requires some understanding of the complexity of test value combinations that are needed. For example, an application that has been tested and used extensively is likely to have few single-factor faults, because these would have already been detected. But a new, untested application may have a fairly high proportion of 1-way and 2-way faults. In this case, we may confine initial testing of the new application to 2-way or 3-way covering arrays, since we are likely to detect faults with even limited testing. That is, the 2-way and 3-way arrays are likely to cover the combinations that trigger faults for this example, but less likely to cover the remaining faults in the extensively tested application. We can think of the relationship between *fault distribution* and *combinatorial coverage* as *fault coverage*. Fault coverage is useful in gauging the effectiveness of a test set because it measures coverage of combinations related to fault detection, allowing testers to estimate if tests are sufficient or if more should be produced to cover relevant portions of the input space.

Common approaches to determining when to stop testing often involve code coverage requirements or tracking error detection rates. As errors are discovered and removed, projections are made to estimate the number of remaining faults and number of tests required to find them, based on assumptions that fault discovery can be predicted by statistical models such Rayleigh or Wiebull distributions [13]. Alternatively, tests may be run until a desired level of code coverage is reached, when source code is available. In this paper we describe a different approach, using combinatorial coverage measurement of test set characteristics in estimating required test set size.

A significant factor in fault detection effectiveness is the distribution of *t*-way faults, which is not known prior to testing. However based on past experience, an approximate distribution of faults at different interaction strengths may be known. For example, for a particular class of application the fraction of 1-way faults may be $F_1 = 60\%$, 2-way faults $F_2 = 25\%$, 3-way faults $F_3 = 10\%$, and 4-way faults $F_4 = 5\%$. Such information could be used in estimating the required strength *t* for *t*-way covering array from which test values will be derived.

We assume deterministic software that computes the same output for a given set of input parameters and values. Faults are also deterministic in that we assume a failure-triggering combination of input values will always produce a failure if it is present in the input. Under these assumptions, two factors in fault detection effectiveness are the fault distribution within the SUT, and combinatorial coverage of the tests. A range of probability of detection can be estimated using the *t*-way coverage of tests and an approximate distribution of *t*-way faults.

Table II. Cumulative faults, Est. upper and lower bounds



t	LB	UB	cumulative lower and upper	
1	0.17	0.68	0.17	0.68
2	0.45	0.29	0.62	0.97
3	0.25	0.02	0.87	0.99
4	0.09	0.01	0.96	1.00
5	0	0	0.96	
6	0.04	0	1.00	

Figure2.Cumulative fault distribution

Figure 2 shows the cumulative percentage of faults at t = 1 to 6 for various applications [5]. We refer to the distribution of faults as shown in Figure 2 as the *fault profile.* Figure 2 shows the fault profile for a variety of fielded products in different application domains, and results for initial testing of a NASA distributed database system. As shown in Figure 2, the fault detection rate increases rapidly with interaction strength, up to t=4. With the medical device applications, for example, 66% of the failures were triggered by only a single parameter value, 97% by 2-way combinations, and 99% by 3-way combinations. The detection rate curves for the other applications studied are similar, reaching 100% detection with 4 to 6-way interactions. Studies by other researchers have been consistent with these results [14][15]. (It is interesting that the fault profile for the medical devices, which were fielded products, is nearly the same as the fault profile for initial testing of the NASA database software.) Note that 100% of the medical device faults were 4-way or lower strength, but the browser faults included some 6-way faults. In other words, the browser faults were rarer and harder to detect than those of the medical devices.

To plan a level of testing appropriate for assurance needs and resource budget, it is helpful to estimate the fault detection that can be achieved with a given level of combinatorial coverage. Although it is impossible to know the fault distribution in advance, approximate lower and upper bounds for fault detection at different interaction levels can be approximated using data from similar applications, a range of various applications relevant to the problem, or for a hypothesized fault distribution. Table II provides an example, showing lower and upper bounds for cumulative fault detection at interaction levels of 1 to 6 based on the fault distribution shown in Figure 2. For example, it can be seen that single values ("1-way" interaction) account for between roughly 17% to 68% of faults. Table II shows that 1-way or 2-way interactions together account for roughly 62% to 97% of faults. Figure 3 shows the incremental growth in cumulative detection rate for 1-way to 6-way interactions, using values from Table II. Thus in the "best case", upper bound line, 68% of faults are discovered with tests covering all single values, 1-way interactions, and an additional 29% may be found by covering all 2-way interactions for a cumulative total of 97%. In the lower bound, or "worst case", 17% and 45% of faults are 1-way and 2-way respectively, for 62% detection by covering all 2-way combinations.

Note that the estimated lower and upper bounds for fault detection converge rapidly with increasing interaction strength. Applications with simple, easily discoverable faults tend to have many single-value or 2-way combinations that trigger failure, while for extensively tested applications, the easy faults have been discovered. Heavily used and tested applications tend to have a higher proportion of 3-way to 6-way faults, and so far, faults involving more than six variables have not been reported. Thus testing 4-way to 6-way combinations can provide strong assurance.

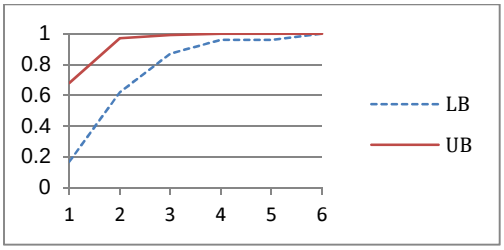


Figure 2. Fault distribution range estimates

We refer to the proportion of t-way combinations covered in a test set as S_t. Thus a t-way covering array has S_t = 1.0, since by definition it covers all t-way combinations. The t-way covering array also has S_i = 1.0 for $i < t$ because a covering array of strength t also covers all combinations that include less than t variables. A covering array of strength t can detect t-way faults because all t-way combinations are covered, but the array will always include other combinations beyond t-way as well. Thus a proportion of (t+1)-way faults can be detected, as well as those of higher strength up to n-way for n variables. We will refer to the proportion of combination settings covered beyond t as *ancillary coverage* for a t-way covering array. For example, a particular 2-way covering array may cover 80% of 3-way combinations and 55% of 4-way combinations, so it has S_2 = 1.0, S_3 = .80, S_4 = .55. It should detect 2-way interaction faults and any 3-way

interaction faults that happen to be among the 80% of 3-way combinations covered.

The effectiveness of fault detection clearly depends on both the proportion of *t*-way combinations covered and the distribution of faults, at each level of *t*. That is, to detect *t*-way faults, the test set must include relevant *t*-way combinations. Although the distribution of *t*-way faults is normally not known for a particular system under test, empirical data on similar systems or software in general may provide reasonable estimates of fault distribution. For example, the fault distribution for similar systems may be 1-way faults F_1 = 60%, 2-way faults F_2 = 25%, 3-way faults F_3 = 10% and 4-way faults F_4 = 5%. Such information could be used in determining the required strength *t* needed for testing, with the objective of covering as many of the *t*-way combinations relevant to the system as possible, within a given resource budget. Under this model, we approximate the detection effectiveness using fault coverage where *k* = maximum interaction strength in failures, F_t = proportion of faults that are *t*-way, and S_t = *t*-way coverage, as

$$fault\ coverage = \mathrm{C} = \sum_{1 \le t \le k} F_t \times S_t \qquad (2)$$

Thus for the example above, if we have a 2-way covering array that also provides ancillary 3-way coverage of S_3 = .80 and 4-way coverage of S_4 = .40, then fault coverage is .60(1.0) + .25(1.0) + .1(.80) + .05(.40) = .95.

Fault coverage can provide an approximation of fault detection effectiveness. It is only an approximation because faults are not necessarily uniformly distributed across the input space. For example, we may have 80% of 3-way faults covered, but the failure-triggering faults may by chance be in the 20% of 3-way combinations not included in the test array. But for answering the key question of what strength covering array is needed to achieve a fault detection rate goal, fault coverage can be a reasonable approximation given available information.

Fault coverage may also be viewed as an estimator of the proportion of relevant input space for which correct operation of the software has been verified (assuming fully passing tests). As such, it is a quantitative measure of testing thoroughness. For instance, in the example above, for which fault coverage is computed as .95, 100% of the 1-way and 2-way combinations have been covered, and these are estimated in the fault distribution to account for 85% of the total set of faults. The test set also provides 80% coverage of 3-way combinations, which are 10% of the faults, and 40% coverage of the 4-way combinations, which are 5% of the faults. Thus the relevant input space is covered to 95%, based on the estimated fault distribution. A variety of possible fault distributions can be studied in this way to evaluate a test set coverage of fault-triggering combinations in the input space. Appendix IV provides an example of this for spacecraft test software.

The ancillary (*t*+*k*)-way coverage of a *t*-way covering array varies depending on input configuration, but in general will increase with increasing *n* and decline with increasing *v* and *t*. Appendix I shows (*t*+1) and (*t*+2)-way coverage for 2-way through 4-way covering arrays of *n* variables, for *n* = 10, 20, 30, 40, 50; and *v* = 2, 4, 6, 8. The number of tests, *N*, for a given covering array is proportional to $v^t \log n$. Thus ancillary coverage can be seen in Appendix I to increase with log *n*. It can be shown [8] that ancillary (*t*+*k*)-way coverage is proportional to $1/v^k$, which can also be seen clearly in coverage at *v*=2, 4, 6, 8 in Appendix I and accompanying graphs.

Using ancillary coverage data for various covering arrays, and empirical data on failures, we can compute approximate upper and lower bounds for fault coverage from expression (2). We will use the data from Figure 1 to illustrate the development of a fault coverage model, then consider specialized cases in later sections. Fault distributions in Table II are used to compute the fault coverage shown in Appendix II. An example is reproduced below in Figure 4. Lower and upper bound fault coverage computations are given for covering arrays of *t*=2 through *t*=5. Table III below shows that a 2-way covering array includes 100% coverage for 1-way and 2-way combinations, 76.8% for 3-way, and 46.1% for 4-way combinations. This results in lower and upper bound estimates of .853 and .998 respectively.

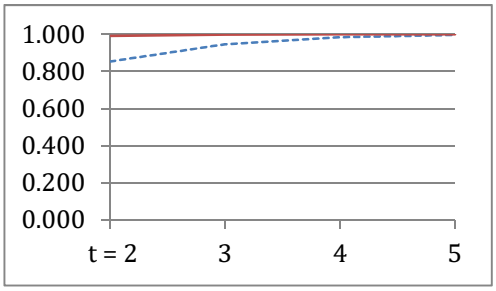


Figure 4.

The calculations in Table III and Appendix II are based on averaged fault distributions for a variety of applications. Individual programs can be expected to have significant variation from these averages. For example, in [16], 2-way tests detect 38% of faults, and fault detection does not increase with 3-way tests, although 4-way tests obtain 100% fault detection. For applications that have been thoroughly tested or used such that the "easy" bugs have been found, 1-way or 2-way faults may be rare or non-existent. Such a distribution can change the fault coverage profile substantially. Appendix III gives two examples of this effect.

TABLE II. FAULT COVERAGE AT T+1 AND T+2 FOR 10 VARIABLES, 2 VALUES, STRENGTH 2 TO 5

covering array → / t, coverage ↓	2-way	3-way	4-way	5-way
1	1.000	1.000	1.000	1.000
2	1.000	1.000	1.000	1.000
3	0.768	1.000	1.000	1.000
4	0.461	0.835	1.000	1.000
5		0.535	0.883	1.000
6			0.594	0.895
LB	**0.853**	**0.945**	**0.984**	**0.996**
UB	**0.990**	**0.998**	**1.000**	**1.000**

In Appendix IV we estimate the fault coverage for a test set for a NASA spacecraft. Previous work [9] introduced combinatorial coverage and analyzed coverage for this test set, which was not originally designed as a *t*-way covering array. The test set had been developed using conventional methods, and the goal of the previous research was to determine if these methods produced good combinatorial coverage. As can be seen in Appendix IV, coverage for 2-way through 4-way combinations is quite good, although not a full covering array at any of these levels.

To estimate the fault coverage, we consider different fault distributions. For example, in previously untested software, single value or 2-way faults may predominate, similar to the upper bound range in Table II. As faults are removed, single value faults become less common. As the number of faults is reduced, more complex faults, involving more parameters, may represent a larger proportion of the total. Appendix IV illustrates the fault coverage for various possible fault distributions.

IV. RELATIONSHIP WITH CODE COVERAGE

Suppose we have two test sets, T_1 and T_2, that both provide 100% 1-way coverage and 80% 2-way coverage, with coverage statistics T_1: S_2 = 0.80, M_2 = 0.70 and T_2: S_2 = 0.80, M_2 = 0.3. Their fault coverage as computed from expression (2) will be the same, but the different values of M suggest that there may be differences in fault detection capability. Suppose the code contains the following segment:

```
if (x <= 0 && y <= 0){faulty code}
else                 {good code}
```

and the input model partitions of values for `x` and `y`:

```
x = {-9999, -1, 0, 1, 9999}
y = {-9999, -1, 0, 1, 9999}
```

Then for the 25 pairs of input values for *x* and *y*, 9 will trigger the fault. Therefore if at least 17/25 = 68% of input combinations are covered in a test set, at least one will trigger the fault. T_1 ensures this, because all 2-way combinations are covered to at least 70% (M_2 = .70). Although T_2 has the same level of overall simple coverage, S_2 = .80, its minimum coverage, M_2, is only 30%, so pair $\{x, y\}$ may have less than 17/25 coverage.

We label as B_t the minimum proportion of *t*-way settings triggering a branch for a given segment of code. Thus for the example, 9/25 of 2-way settings trigger the true branch, and 16/25 settings trigger the false branch. Thus, for the true branch, the proportion of 2-way settings triggering the branch is 9/25 = .36; and for the false branch, 16/25 of the 2-way settings. Thus B_2 = .36 for this example code. For the example above, it is clear that at least one combination in test set T_1 will branch to the faulty code, because minimum coverage M_2 = .70 exceeds the proportion of settings needed to ensure that at least one will cause the predicate `x <= 0 && y <= 0` to be true (64%).

We can show that if $B_t + M_t > 1$, to be referred to as the branch coverage condition, then the test set will provide 100% branch coverage, where all variables included in decision predicates have values in the variable set with minimum coverage characteristic M_t. This makes sense intuitively because it ensures that for every *t*-way variable combination, there is an intersection between the set of covered value configurations and the set of value configurations that trigger a branch, as the example above demonstrates.

Branch Coverage Condition: A test set provides 100% branch coverage for *t*-way conditionals if $M_t + B_t > 1$, where M_t = minimum combinatorial coverage at level *t*, and B_t = minimum proportion of *t*-way combinations that trigger a branch within the code, where all variables in decision predicates have values from the variable set with minimum coverage characteristic M_t.

Proof: For minimum *t*-way coverage of M_t, let $k = M_t v^t$ = number of *t*-way combinations (out of v^t) covered. That is, for the *t*-way combination(s) with the lowest coverage, *k* different settings are covered, so *k* or more settings are covered for all *t*-way combinations in the test set. Let $m = B_t v^t$ = minimum number of combinations triggering a branch within the code. Thus any test set containing a test with one of these *m* combination settings will trigger the branch. So any test set will trigger the branch if $k > v^t - m = v^t(1-B_t) < M_t v^t$ or $M_t + B_t > 1$. □

Example. If t=2 and M = .5 for a decision predicate containing two binary parameters, then there are $.5(2^2)=2$ settings covered in the test set. There are C(4,2) =6 ways in which two settings of two parameters can be included in a test set: 00,01 | 00,10 | 00,11 | 01,10 | 01,11 | 10,11. If every decision predicate is satisfied by at least one setting, then B_t = .25, and there are three of the four settings that

do not satisfy the predicate. From these, there are C(3,2)=3 ways in which the non-satisfying settings can be included in a test set, so half of the possible test sets will include at least one test that satisfies the predicate. If B_t is increased to .5, two of the two-parameter settings will satisfy the predicate, with C(2,2) = 1 possible test set without a test that will satisfy the predicate. If B_t = .75, then three settings will satisfy the predicate and the possible test sets without a test containing a satisfying parameter setting is C(1,2) = 0.

Note the constraint in the branch coverage condition that all variables in decision predicates have values from the variable set with minimum coverage characteristic M_t. This may be a relatively strong assumption in practice. For example, it will hold if (a) the values in decision predicates are only those from test inputs, rather than including computed values for internal (non-input) variables; or (b) values for variables in decision predicates meet the combinatorial coverage requirement, even where they include computed values. Some decision predicates may meet requirement (a), but others may include input variables whose value has changed, or internal variables not included in the test set. In the latter case, the coverage requirement may be validated by inspecting internal state prior to execution of each decision predicate, through means such as a debugger or assertions that write out decision predicate variable values. The latter option may in most cases be impractical. The branch coverage condition therefore has most practical utility for decision predicates containing input variables, but it also helps in understanding the effectiveness of test sets with good combinatorial coverage, though not necessarily a full covering array.

A corollary to the condition is that if every decision can be satisfied by more than one parameter combination setting, a full covering array is not needed for 100% branch coverage. Again, this makes sense intuitively because if two or more combination settings trigger each branch, then a test set that that covers v^t-1 settings for each *t*-way combination must include at least one of the two settings that will trigger the branch. We can generalize to compute a level that M_t must exceed to provide full branch coverage.

Branch Coverage Corollary: If *k* or more *t*-way settings satisfy every decision predicate, then the branch coverage condition is obtained with $M_t > 1 - \frac{k}{v^t}$, where all variables in decision predicates have values from the variable set with minimum coverage characteristic M_t.
Proof: If *k* or more *t*-way settings satisfy every predicate, then $B_t v^t \geq k$, and $M_t + \frac{k}{v^t} > 1$ implies $M_t + B_t > 1$. □

Note that if *k*=1, i.e., some decision predicates are satisfied by only one of v^t *t*-way combinations, then a full covering array is needed for branch coverage because coverage minimum M increases only with increments of $1/v^t$.

Example. If the condition in the example in the first paragraph of this section were `x == 0 && y == 0 || x == 1 && y == 1`, then two of the 2-way settings would ensure that the decision predicate was satisfied, so $B_t v^t \geq 2$. If M_t is increased to 96% (24/25), then $M_t + B_t = 1 - \frac{1}{v^t} + \frac{2}{v^t} > 1$. For 10 variables with 5 values each, the IPOG algorithm [3] generates a 2-way covering array of 309 tests, but 96% coverage is reached after only 225 tests, a reduction in test set size of more than 25%.

V. Implications for Testing

Combinatorial fault coverage can supplement, or provide an alternative to, conventional methods of specifying test requirements. Combinatorial coverage provides a direct measure of the proportion of the relevant input space covered by a test set, and incorporating fault distribution data makes it possible to develop fault coverage figures that approximate the proportion of faults that the test set can detect. Thus, fault coverage can be used for estimating the fault detection capacity of a test set. It provides more useful information than raw combinatorial coverage figures, because it takes account of the approximate distribution of faults. By estimating lower bounds on the distribution of *t*-way faults at each level of t, we can estimate the number of tests needed to reach a desired detection rate, or approximate fault detection capacity for tests that can be produced within a given resource budget. A number of considerations come into play when applying this approach.

Number of variable values: As can be seen in Appendix II and III, fault coverage varies inversely with the number of values per variable, so two covering arrays do not necessarily provide the same fault detection capacity even though they both cover 100% of *t*-way combinations. With some fault distributions, a *t*-way array of boolean variables may provide better fault coverage than a (*t*+1)-way array with more values per variable. For example, in Appendix III, fault coverage for Distribution 1 at *t*=3, *v*=2 is .843, but only .832 for *t*=4 where *v*=8.

Uncertainty and range of estimates: The cumulative distribution of faults in Figure 2 shows wide variation at *t*=1 to *t*=3. This produces a similar variation in fault coverage estimates, as can be seen in Appendix II. At *t*=4, lower and upper estimates converge much more closely. The variation and uncertainty for $t \leqslant 3$ suggest that 4-way coverage criterion should be the minimum for high-assurance software. As noted in Section III, the number of tests increases exponentially with *t*, so using 4-way coverage, rather than 5-way or 6-way, may also represent a sensible tradeoff between cost and assurance level in some applications. For some applications, 4-way testing may be

optimal in that higher strength arrays become prohibitive in time or resources.

Interaction with branch condition: The branch coverage condition can be used with information from static analysis of the source code to determine a level of minimum combinatorial coverage that will provide full branch coverage, a moderately strong (dynamic) code coverage criterion. This code coverage goal can be achieved with substantially fewer tests than would be required for a full *t*-way covering array (although a full array would provide stronger testing). The fact that branch coverage can be obtained with many fewer tests than a full covering array also helps to explain the effectiveness of randomly generated tests in some cases.

The branch coverage condition suggests that it is generally best to keep M as high as possible when executing tests if the code contains relational expressions, which is nearly always the case. When we discretize variable values, we end up with combinations like those in the example, where multiple settings of a particular variable combination can trigger branches. When multiple combinations can trigger a branch to faulty code, we are better off including tests that increase M evenly, rather than covering all settings of a subset of combinations more quickly, so that we reach the branch condition faster. For example, adding a previously uncovered 2-way combination in Figure 1 could either increase Φ to .5, or M to .75. The first option makes no progress towards reaching the branch condition, so the second option is preferable.

Impact of ancillary coverage: The cumulative distribution of interaction failures shown in Fig. 1 is based on analysis of failure reports, identifying the number of factors involved, rather than failures reported in *t*-way testing. That is, the distribution was not developed by running 2-way through 6-way tests and counting the failures discovered at each level, because such a procedure could not accurately determine failures triggered by each level of *t*-way combinations. As shown in previous sections, a *t*-way covering array also includes a significant (often high) percentage of *t*+1, *t*+2, etc. coverage. So, for example, if we run a 3-way covering array of tests and discover n faults, it does not mean that all n faults were triggered by single-value, 2-way, or 3-way combinations. It is highly likely that some resulted from 4-way, 5-way, or conceivably higher strength combinations.

When reviewing case studies of combinatorial testing, we often see that for some applications, all failures were discovered by relatively low strength covering arrays, including 3-way and 4-way [16]. In other words, a significant number of testing studies seemed to find failures a bit "easier" to detect than the distribution in Figure 1 might suggest. The high levels of ancillary coverage for *t*-way arrays mean that test effectiveness can be higher than might be expected considering only the interaction strength *t*. This is especially true for applications with a high proportion of boolean inputs, because coverage beyond t, with a t-way array, is much higher for boolean variables than with larger numbers of values per variable.

Estimating residual risk: One benefit of measuring the combinatorial coverage of a test set is that it provides information on risk through combinations not covered, one of the original motivations for development of combinatorial coverage ideas [8][9]. Knowing the proportion of *t*-way combination settings not covered for different values of *t* provides useful information for testers and decision-makers, as it helps in estimating the risk that the application will encounter a set of inputs for which its behavior was not verified. Fault coverage allows a tighter estimate of the risk of undetected faults, by factoring in the fault distribution.

Test requirements specification: One application of fault coverage is in supplementing requirements for black-box testing. When specifying test requirements, if source code is available, a variety of coverage measures may be used. For example, it may be required that 100% statement or branch coverage be achieved. Without source code, test goals may be based on criteria such as a level of inter-module call coverage and specific requirements based criteria. Tests are typically required to trace back to requirements, and conversely it must be shown that all requirements have been tested with one or more tests. However, this just insures that code works for a few inputs, and is a relatively weak measure of how thorough the requirements-based testing has been. Just because we have multiple tests for all requirements, there may be no indication of the range of input space for which the code satisfies requirements. Even if two requirements-based test sets "fully cover" all requirements, one may be better than another if one evaluates the application on a broader range of input configurations. The range of input space covered is a separate dimension beyond a simple count of tests per requirement. Fault coverage gives a more precise measure of the degree to which requirements are covered.

VI. Summary and Conclusions

The objective of this work was to build on previous results to develop a relationship between the (static) distribution of combinations in input data and (dynamic) executable code coverage. The fault coverage estimation introduced in Section III can be used for initial estimation of test set size, using measures of combinatorial coverage that can be computed with measurement tools such as described in [8][10][11]. Appendices II and III illustrate results of this computation for a variety of test problem configurations. Results can be used in scoping the number of tests and level of effort, and estimating residual risk from complex combinations not tested. In the future, we

may also measure combinatorial coverage of the input distribution, which could supplement operational profile methods.

When source code is available, the method introduced in Section IV can be used to the minimum level of (static) combination coverage to ensure 100% (dynamic) branch coverage (and therefore statement coverage also) of executable code, where all variables in decision predicates have values from test inputs. This approach may be useful where cost considerations make it difficult to use a full covering array in testing. Branch coverage is a reasonably strong test criterion, and Section IV shows how it may be achieved at substantial savings compared with full *t*-way testing. Future work might evaluate fault detection rates of partial versus full covering arrays, to enable better cost-benefit tradeoffs in testing.

Certain products may be identified in this document, but such identification does not imply recommendation by NIST nor that the products identified are necessarily best for the purpose.

REFERENCES

[1] S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, A. Iannino. Applying design of experiments to software testing, *Proc. Intl. Conf. on Software Engineering, (ICSE '97)*, 1997, pp. 205-215

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

[3] 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.

[4] Kuhn, D. R., Kacker, R. N., & Lei, Y. *Practical combinatorial testing*. NIST SP 800-142, Oct. 2010.

[5] D.R. Kuhn, D.R. Wallace, Jr. A.M. Gallo, Software fault interactions and implications for software testing, *IEEE Trans. Software Engineering*, vol. 30, no. 6, June, 2004.

[6] 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).

[7] D.R. Kuhn, R. Kacker, Y. Lei. *Combinatorial Coverage Measurement*, NIST IR 7878, Sept. 2012.

[8] Kuhn, D. R., Dominguez Mendoza, I., Kacker, R. N., & Lei, Y. Combinatorial Coverage Measurement Concepts and Applications. *Proc. IEEE Sixth Intl Conf on Software Testing, Verification and Validation Workshops (IWCT), 2013* pp. 352-361. IEEE.

[9] 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.

[10] D.R. Kuhn, R.N. Kacker "Measuring Combinatorial Coverage of System State-space for IV&V", *NASA IV&V Workshop*, 2012.

[11] C. Price, R. Kuhn, R. Forquer, A. Lagoy, R. Kacker, "Evaluating the *t*-way Combinatorial Technique for Determining the Thoroughness of a Test Suite", *NASA IV&V Workshop*, 2013.

[12] 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.

[13] Lyu, M. R. (1996). *Handbook of software reliability engineering* (Vol. 222). CA: IEEE computer society press.

[14] K. Z. Bell and Mladen A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. *Proceedings of the ITI Third IEEE Intl Conf. Information & Communications Technology*, pages 221–235, Cairo, Egypt, December 2005.

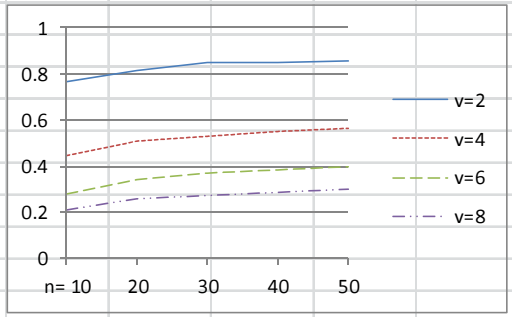
[15] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, J.-L. Lanet, A case study in JML-based software validation. *Proc. 19th Int. IEEE Conf. on Automated Sofware Engineering*, pp. 294-297, Linz, Sep. 2004

[16] Montanez, C., Kuhn, D. R., Brady, M., Rivello, R. M., Reyes, J., & Michael, K. (2012). Evaluation of fault detection effectiveness for combinatorial and exhaustive selection of discretized test inputs. *Software Quality Professional Magazine*, *14*(3).

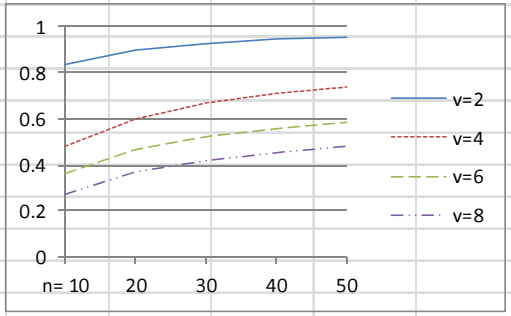
Appendix I. Ancillary coverage, (t+1)-way and (t+2)-way for 10-variable covering arrays

t=2	t+1					t=3	t+1					t=4	t+1				
	n= 10	20	30	40	50		n = 10	20	30	40	50		n = 10	20	30	40	50
v = 2	0.768	0.815	0.848	0.851	0.856	v = 2	0.835	0.897	0.926	0.949	0.953	v = 2	0.883	0.945	0.967	0.979	0.984
4	0.444	0.509	0.531	0.551	0.565	4	0.483	0.598	0.671	0.712	0.741	4	0.548	0.709	0.794	0.834	0.861
6	0.28	0.339	0.367	0.38	0.396	6	0.364	0.466	0.522	0.558	0.586	6	0.423	0.576	0.65	0.696	0.727
8	0.208	0.257	0.272	0.289	0.3	8	0.272	0.367	0.419	0.453	0.478	8	0.326	0.469	0.541	0.587	

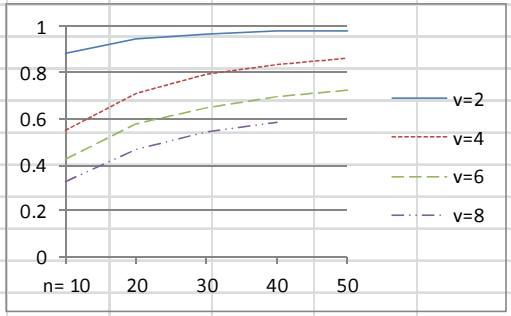
t=2	t+2					t=3	t+2					t=4	t+2				
	n = 10	20	30	40	50		n = 10	20	30	40	50		n = 10	20	30	40	50
v = 2	0.461	0.521	0.572	0.571	0.576	v = 2	0.535	0.628	0.684	0.734	0.747	v = 2	0.594	0.715	0.777	0.818	0.841
4	0.126	0.153	0.162	0.172	0.178	4	0.14	0.19	0.228	0.253	0.272	4	0.167	0.251	0.312	0.349	0.377
6	0.049	0.063	0.07	0.074	0.078	6	0.068	0.095	0.111	0.123	0.132	6	0.082	0.128	0.156	0.176	0.191
8	0.027	0.035	0.038	0.04	0.042	8	0.037	0.054	0.064	0.071	0.076	8	0.046	0.074	0.091	0.103	



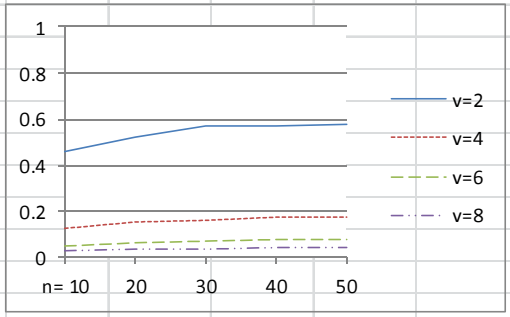
t=2 t+1 coverage



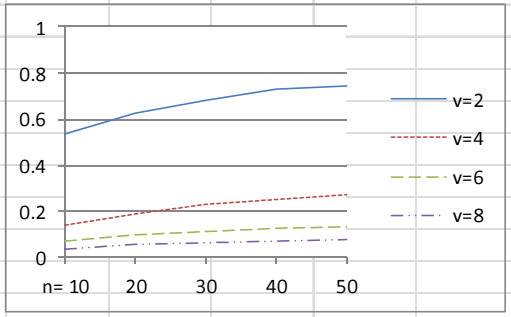
t=3 t+1 coverage



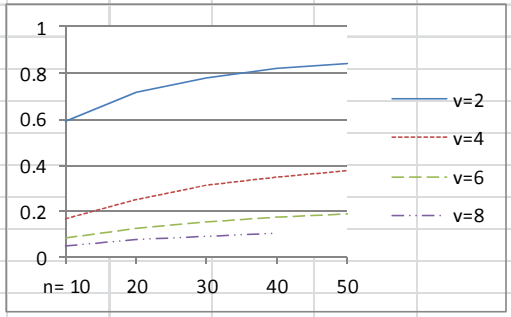
t=4 t+1 coverage



t=2 t+2 coverage



t=3 t+2 coverage



t=4 t+2 coverage

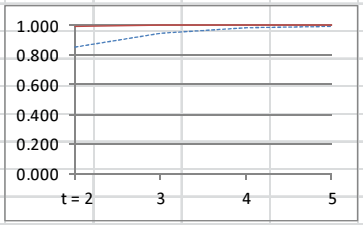
Appendix II.

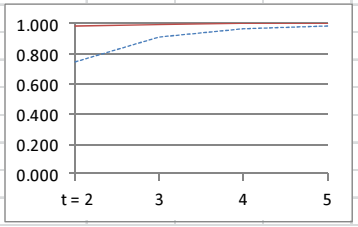
Explanation of charts: Lower and upper bounds on fault coverage are estimated for n-variable covering arrays, using ancillary (t+1)-way and (t+2)-way coverage for *t*-way arrays, for t = 2 through 5. For example, the first column below indicates that a 2-way covering array provides 100% 1-way and 100% 2-way coverage, 76.8% 3-way coverage, and 46.1% 4-way coverage. Similarly, the second column shows 100% coverage of 1-way through 3-way for a 3-way array, with 83.5% and 53.5% 4-way and 5-way coverage. Lower and upper bounds are then computed using the empirical data from Table 1 (reproduced right) and expression (2).

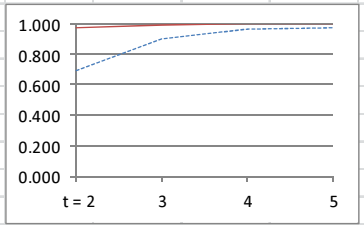
t	LB	UB
1	0.17	0.68
2	0.45	0.29
3	0.25	0.02
4	0.09	0.01
5	0	0
6	0.04	0

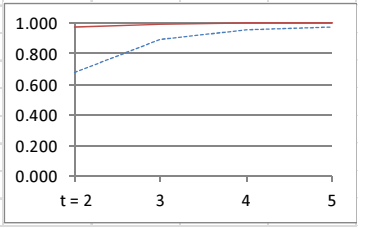
Estimated fault coverage range for 10-variable covering arrays

10 vars	x axis:			t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5
	LB	UB		2 values per variable				4 values per variable				6 values per variable				8 values per variable			
1	0.17	0.68	1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	0.45	0.29	2	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
3	0.25	0.02	3	0.768	1.000	1.000	1.000	0.444	1.000	1.000	1.000	0.280	1.000	1.000	1.000	0.208	1.000	1.000	1.000
4	0.09	0.01	4	0.461	0.835	1.000	1.000	0.126	0.483	1.000	1.000	0.049	0.364	1.000	1.000	0.027	0.272	1.000	1.000
5	0	0	5		0.535	0.883	1.000		0.140	0.548	1.000		0.068	0.423	1.000		0.037	0.326	1.000
6	0.04	0	6			0.594	0.895			0.167	0.573			0.082	0.451			0.046	0.353
LB			**LB**	**0.853**	**0.945**	**0.984**	**0.996**	**0.742**	**0.913**	**0.967**	**0.983**	**0.694**	**0.903**	**0.963**	**0.978**	**0.674**	**0.894**	**0.962**	**0.974**
UB			**UB**	**0.990**	**0.998**	**1.000**	**1.000**	**0.980**	**0.995**	**1.000**	**1.000**	**0.976**	**0.994**	**1.000**	**1.000**	**0.974**	**0.993**	**1.000**	**1.000**



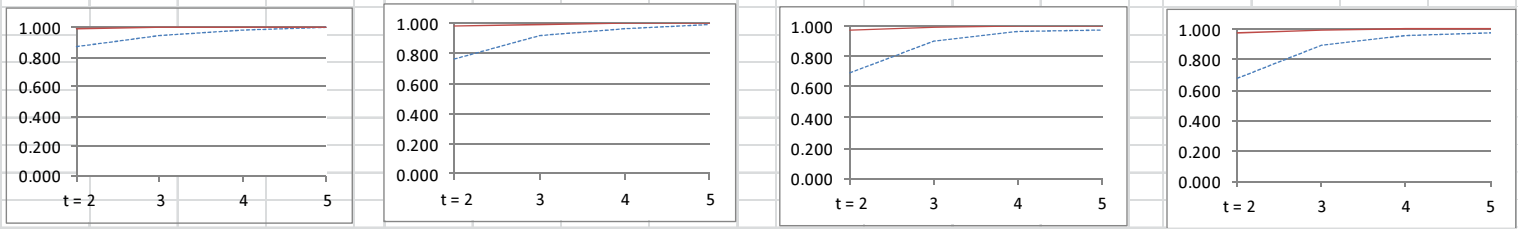






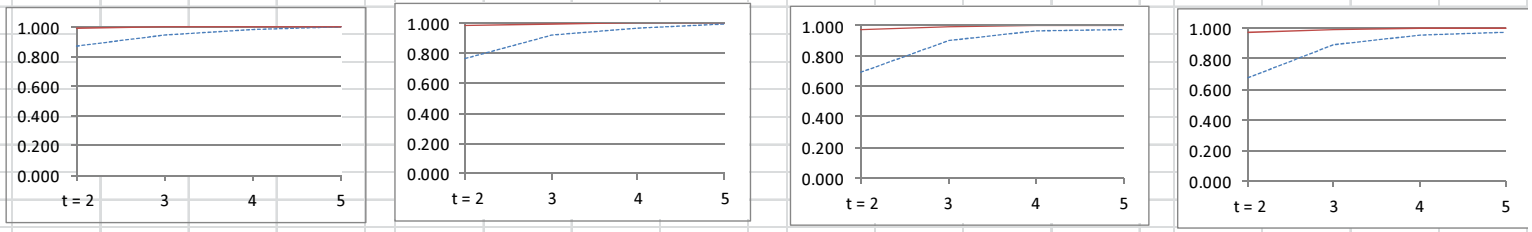
Estimated fault coverage range for 20-variable covering arrays

20 vars				t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5
	LB	UB		2 values per variable				4 values per variable				6 values per variable				8 values per variable			
1	0.17	0.68	1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	0.45	0.29	2	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
3	0.25	0.02	3	0.815	1.000	1.000	1.000	0.509	1.000	1.000	1.000	0.339	1.000	1.000	1.000	0.257	1.000	1.000	1.000
4	0.09	0.01	4	0.521	0.897	1.000	1.000	0.153	0.598	1.000	1.000	0.063	0.466	1.000	1.000	0.035	0.367	1.000	1.000
5	0	0	5		0.628	0.945	1.000		0.19	0.709	1.000		0.095	0.576	1.000		0.054	0.469	1.000
6	0.04	0	6			0.715	0.972			0.251	0.784			0.128	0.655			0.074	0.544
LB			**LB**	**0.871**	**0.951**	**0.989**	**0.999**	**0.761**	**0.924**	**0.970**	**0.991**	**0.710**	**0.912**	**0.965**	**0.986**	**0.687**	**0.903**	**0.963**	**0.982**
UB			**UB**	**0.992**	**0.999**	**1.000**	**1.000**	**0.982**	**0.996**	**1.000**	**1.000**	**0.977**	**0.995**	**1.000**	**1.000**	**0.975**	**0.994**	**1.000**	**1.000**



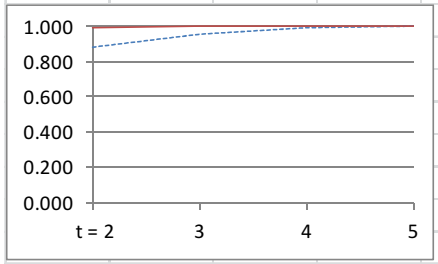
Estimated fault coverage range for 30-variable covering arrays

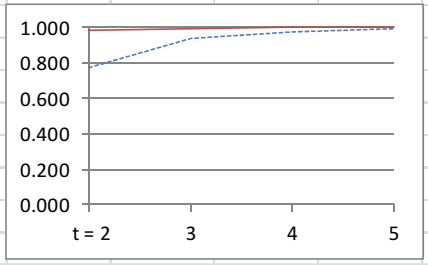
30 vars				t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5
	LB	UB		2 values per variable				4 values per variable				6 values per variable				8 values per variable			
1	0.17	0.68	1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	0.45	0.29	2	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
3	0.25	0.02	3	0.848	1.000	1.000	1.000	0.531	1.000	1.000	1.000	0.367	1.000	1.000	1.000	0.272	1.000	1.000	1.000
4	0.09	0.01	4	0.572	0.926	1.000	1.000	0.162	0.671	1.000	1.000	0.07	0.522	1.000	1.000	0.038	0.419	1.000	1.000
5	0	0	5		0.684	0.967	1.000		0.228	0.794	1.000		0.111	0.65	1.000		0.064	0.541	1.000
6	0.04	0	6			0.777	0.986			0.312	0.868			0.156	0.739			0.091	
LB			**LB**	**0.883**	**0.953**	**0.991**	**0.999**	**0.767**	**0.930**	**0.972**	**0.995**	**0.718**	**0.917**	**0.966**	**0.990**	**0.691**	**0.908**	**0.964**	**0.960**
UB			**UB**	**0.993**	**0.999**	**1.000**	**1.000**	**0.982**	**0.997**	**1.000**	**1.000**	**0.978**	**0.995**	**1.000**	**1.000**	**0.976**	**0.994**	**1.000**	**1.000**
	0.17	0.68																	
	0.62	0.97																	
	0.87	0.99																	
	0.96	1																	
	0.96	1																	
	1	1																	

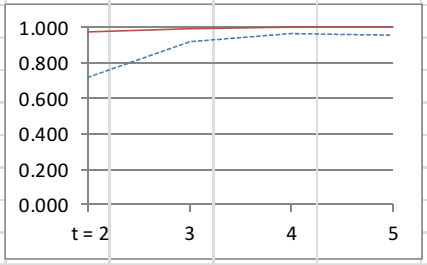


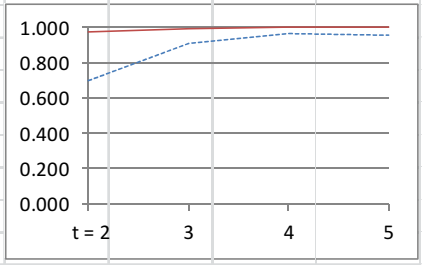
Estimated fault coverage range for 40-variable covering arrays

t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5
2 values per variable				4 values per variable				6 values per variable				8 values per variable			
1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
0.851	1.000	1.000	1.000	0.551	1.000	1.000	1.000	0.380	1.000	1.000	1.000	0.289	1.000	1.000	1.000
0.571	0.949	1.000	1.000	0.172	0.712	1.000	1.000	0.074	0.558	1.000	1.000	0.040	0.453	1.000	1.000
	0.734	0.979	1.000		0.253	0.834	1.000		0.123	0.696	1.000		0.071	0.587	1.000
		0.818	0.991			0.349	0.903			0.176				0.103	
0.884	**0.955**	**0.993**	**1.000**	**0.773**	**0.934**	**0.974**	**0.996**	**0.722**	**0.920**	**0.967**	**0.960**	**0.696**	**0.911**	**0.964**	**0.960**
0.993	**0.999**	**1.000**	**1.000**	**0.983**	**0.997**	**1.000**	**1.000**	**0.978**	**0.996**	**1.000**	**1.000**	**0.976**	**0.995**	**1.000**	**1.000**









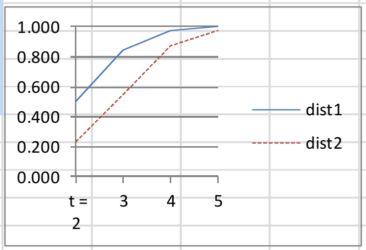
Appendix III.

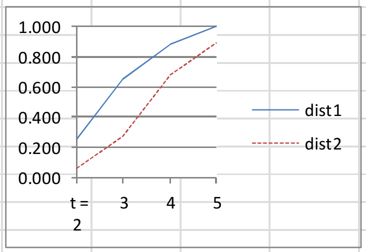
Explanation of charts: Comparison of fault coverage for two different hypothetical distributions of faults in 10-variable covering arrays, using (t+1)-way and (t+2)-way coverage for *t*-way arrays, for t = 2 through 5. For example, the first column below indicates that a 2-way covering array provides 100% 1-way and 100% 2-way coverage, 76.8% 3-way coverage, and 46.1% 4-way coverage, resulting in 2-way fault coverage of 49.9% for distribution 1 (right) and 23.1% for distribution 2.

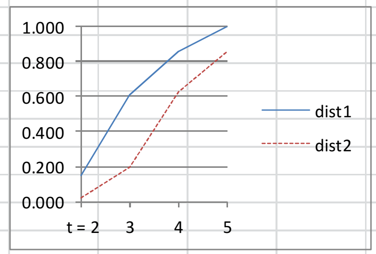
	dist1	dist2
1	0	0
2	0	0
3	0.5	0
4	0.25	0.5
5	0.25	0.25
6	0	0.25

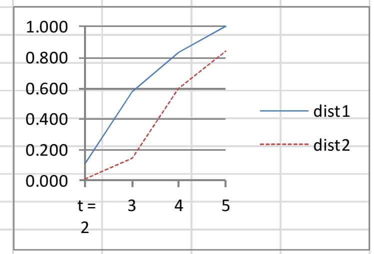
Estimated fault coverage range for two different distributions of faults in 10-variable covering arrays

	t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5	t = 2	3	4	5
	2 values per variable				4 values per variable				6 values per variable				8 values per variable			
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
3	0.768	1.000	1.000	1.000	0.444	1.000	1.000	1.000	0.280	1.000	1.000	1.000	0.208	1.000	1.000	1.000
4	0.461	0.835	1.000	1.000	0.126	0.483	1.000	1.000	0.049	0.364	1.000	1.000	0.027	0.272	1.000	1.000
5		0.535	0.883	1.000		0.140	0.548	1.000		0.068	0.423	1.000		0.037	0.326	1.000
6			0.594	0.895			0.167	0.573			0.082	0.451			0.046	0.353
dist1	**0.499**	**0.843**	**0.971**	**1.000**	**0.254**	**0.656**	**0.887**	**1.000**	**0.152**	**0.608**	**0.856**	**1.000**	**0.111**	**0.577**	**0.832**	**1.000**
dist2	**0.231**	**0.551**	**0.869**	**0.974**	**0.063**	**0.277**	**0.679**	**0.893**	**0.025**	**0.199**	**0.626**	**0.863**	**0.014**	**0.145**	**0.593**	**0.838**









Appendix IV.

The left panel of Table IV shows the combinatorial coverage of 7,489 tests for a NASA spacecraft documented in [9]. The test set was developed using conventional methods and analyzed to determine the level of combinatorial coverage. The right panel shows fault coverage estimated using expression (2) under various possible fault profiles for three hypothetical systems under test.

Profile P_1 is approximately the upper bound from Table II, representing an average of fault distributions for previously reported data [5]. P_2 assumes an application for which single value faults have been removed. P_3 assumes an application has been thoroughly tested and all single value and 2-way faults have been removed. Fault coverage declines because *t*-way combinatorial coverage decreases with increasing *t*. A fault profile such as P_1 might be expected in an average application, while P_2 and P_3 might be seen in more well-tested applications. Note that these figures estimate only the *proportion* of fault coverage; a previously untested application could be expected to have a higher absolute number of faults than those that have been used and tested extensively.

Test combination coverage for *t*=1..6			**Fault distribution at *t*=1..6 for three fault profiles**		
***t* =**	**coverage**		**P_1**	**P_2**	**P_3**
1	1.0		.65	.00	.00
2	.94		.25	.65	.00
3	.83		.05	.10	.45
4	.68		.02	.10	.20
5	.53		.02	.10	.20
6	.39		.01	.05	.15
		fault coverage	**0.95**	**0.84**	**0.68**

Table IV. Fault coverage under various assumptions.