Combinatorial Coverage Difference Measurement

D. Richard Kuhn
M. S. Raunak
Computer Security Division
Information Technology Laboratory

Raghu N. Kacker
Applied and Computational Mathematics Division
Information Technology Laboratory

June 22, 2021

This publication is available free of charge from:
https://doi.org/10.6028/NIST.CSWP.06222021-draft

National Institute of
Standards and Technology
U.S. Department of Commerce

Abstract

Structural coverage criteria are widely used tools in software engineering, useful for measuring aspects of test execution thoroughness. However, in many cases structural coverage may not be applicable, either because source code is not available, or because processing is based on neural networks or other black-box components. Vulnerability and fault detection in such cases will typically rely on large volumes of tests, with the goal of discovering flaws that result in system failures or security weaknesses. This publication explains combinatorial coverage difference measures that have been applied to problems that include fault identification and autonomous systems validation, and documents functions of research tools for computing these measures. The metrics and tools described are introduced as research tools; later work will be useful in determining which are of value in assurance and testing or simulation.

Keywords

combinatorial coverage; combinatorial coverage difference measures; combinatorial methods; combinatorial testing; fault location; software testing; structural coverage

Disclaimer

Any mention of commercial products or reference to commercial organizations is for information only; it does not imply recommendation or endorsement by NIST, nor does it imply that the products mentioned are necessarily the best available for the purpose.

Additional Information

For additional information on NIST's Cybersecurity programs, projects and publications, visit the Computer Security Resource Center. Information on other efforts at NIST and in the Information Technology Laboratory (ITL) is also available.

Public Comment Period: *June 22, 2021 through August 20, 2021*

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: acts-project@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

Contents

List of Figures

1 Introduction

This publication explains concepts of combinatorial coverage and coverage difference measures that have utility for software verification and testing, or for security vulnerability detection, and documents functions of research tools for computing these measures. These measures have been applied to problems that include fault identification, vulnerability detection, and autonomous systems verification and validation. The metrics and tools described are introduced as research tools, with examples of their application in assurance and testing, simulation, autonomous systems, intrusion detection, and other domains.

Among these applications, autonomy and artificial intelligence are of particular interest. As noted in the National Security Commission on Artificial Intelligence (NSCAI), test, evaluation, verification, and validation of traditional software is not sufficient for the type of assurance needed for AI and autonomous systems [1]. The NSCAI report notes that "agencies lack common metrics to assess trustworthiness that AI systems will perform as intended. To minimize performance problems and unanticipated outcomes, an entirely new type of Test and Evaluation, Verification and Validation (TEVV) will be needed. This is a priority task, and a challenging one."

A fundamental problem is that many or most AI systems rely on black box functions, such as deep neural networks, where a single algorithm is used, and the behavior of the system is 'programmed' by the data used in training it. Measures of the adequacy and completeness of the training data are therefore essential, to ensure that the resulting model is sufficiently representative of the real world. As environments evolve, or the system is applied to new environments, it is also necessary to know how many, and what kind of, differences exist with the new environment. This publication provides a number of quantitative measures that can be applied in meeting these requirements.

1.1 Software Verification and Testing

Verification of complex software systems is an important, yet challenging task. Testing is the most common method for assuring that software meets its specification and is defect free. To claim that software is defect-free, one has to show that it produces the "correct" output or "behaves" according to specification under *all* possible parameter values and configuration. In the software verification world, this is known as *exhaustive testing.* For any software of reasonable size and complexity, exhaustive testing is completely infeasible. Thus, during the testing process, a small subset of parameter values and configurations is selected to ensure that the software is producing its output or maintaining its behavior as "expected" . The selected parameter values are called *Test Cases*, and the set of all test cases is called a *Test Suite.* The essence of software testing, therefore, lies in effective ways of identifying the test cases and building the test suite. Two overarching questions related to this process include

1) how to select the test cases, and
2) how to decide when enough test cases have been selected.

Over the years, researchers have proposed different *Test Adequacy* criteria for answering these two questions.

1.2 Test Adequacy and Coverage Criteria

Many test adequacy criteria are based on the runtime knowledge of the internal structure of the software, which requires access to the source code. These adequacy criteria are referred to as *structural coverage* criteria, and an extensive body of theory has been developed to clarify their effectiveness and the relationships among different criteria [2, 3]. For example, it can be shown that branch coverage subsumes statement coverage; i.e., if full branch coverage is achieved, then full statement coverage is guaranteed. Depending on which lines of code are getting executed by *test cases*, different test adequacy criteria such as *statement coverage* or *branch coverage* may be applied. The motivation here is that if all the test cases in a test suite miss executing a statement or a branch in a piece of software, then a fault in the unexecuted code will likely be not revealed. Thus, test suites that achieve higher coverage scores are desired. Organizations often set goals for test suite selection such as achieving 85% or 90% statement coverage or branch coverage.

For large, complex software, achieving 100% statement coverage may not be feasible due to unreachable code. Specifying coverage goals in this way help testers choose the test cases. The goal is to maximize the coverage using the smallest number of test cases such that a desired level is reached. This approach, in turn, also helps to answer the second testing challenge: deciding how many test cases to select, i.e., knowing when to stop. If the goal is to achieve 90% branch coverage, then any test suite that ensures execution of 90% of all the branches in the code will meet the goal.

Note that if there is a fault/error/bug in a statement, simply having that statement executed by a test case does not guarantee discovering the fault. Thus high structural coverage may be considered a necessary, but not sufficient, condition for software assurance. Yet, this is the most common approach for providing some level of confidence about reasonably tested software.

The other major approach to test case selection, often called the black-box approach, does not rely on the runtime execution information. Instead the input parameter and configuration space is divided into groups (e.g., equivalence classes), and test cases are selected such as that the parameter values and configuration space is *covered* by the test suite. Here again, simply designing a test suite that covers the equivalence classes does not provide strong assurance about discovering all the faults in a program. Gaining any confidence or assurance on the fault-free operation of a software usually remain as an art for the test suite designer. One objective of this publication is to provide measures and criteria that can be used in quantifying the thoroughness of a test suite.

There is empirical evidence regarding software faults being caused by specific interactions of parameter values. If one approaches the selection of test cases based on systematically discretizing the parameter/configuration space, judiciously identifying the combinations of values that are of interest, and *covering* those interactions of parameter values, a high-level assurance of fault discovery can be provided. *Combinatorial coverage metrics* are designed precisely to achieve this goal.

2 Combinatorial Coverage Measurement

Combinatorial methods offer an approach to coverage measurement that provides a measure directly related to fault detection. A series of studies have shown that most software bugs and failures are caused by one or two parameters, with progressively fewer by three or more [4, 5, 6, 7, 8, 9]. This finding means that testing parameter combinations can provide more efficient fault detection than conventional methods. In this section, we review concepts of measuring the combinatorial coverage of an input space [10, 11, 12], for use in testing or in other applications where it is important to ensure inclusion of combinations of input parameter values.

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

Figure 1: Example test array for a system with four binary components

Combinatorial coverage measurement concepts can be illustrated using the example in Figure 1, which shows a test array that does not contain all 2-way combinations. To facilitate discussion, it is helpful to establish terminology for two related, but distinct, concepts that will be used in this publication:

- *t-way combination*: a set of t parameters or variables. For example, using the parameters in figure 1, (b, d) is a 2-way combination, (a, c) is a different 2-way combination, and (a, b, d) is a 3-way combination.
- *t-way value-combination*: a combination for which the parameters have specific values. (Note: in the original definition from [10], this was referred to as a variable-value combination.) For example, (b=0, d=0) is one value combination, and (b=1, d=0) is a different value combination for the same parameter combination.

In discussing covering arrays and combinatorial testing, it is conventional to denote the configuration of parameters and values in an exponential notation as ${v_1}^{n_1}{v_2}^{n_2}...{v_k}^{n_k}$, where k is the number of different counts of values, v are values, and n are counts of the parameters with each value. For example, a configuration of 10 parameters, including five boolean, two with three values each, one with five values, and two with 6 values, would be shown as $2^5 3^2 5^1 6^2$.

2.1 Basic Combinatorial Coverage Measures

We define two important measures related to basic combinatorial coverage following [4]:

- *CIPC(n, t)*: *Combinatorial Input Parameter Coverage*, also known as *variable-value configuration coverage* [11, 10], is the proportion of the total $C(n,t) \times v^t$ t-way value combinations that are covered by at least one test in a test suite. Here n is the number of input parameters or variables, t is the number of interactions within those parameters being covered, and v represents the number of possible values the parameters can take.
- *PT-completeness* or (p, t)-completeness: This measures the proportion of $C(n,t)$ combinations that has at least a $CIPC(n,t)$ value of p [10, 13].

Example. As shown in Figure 1 above, there are $C(4,2) = 6$ possible parameter combinations: [$\{a,b\}$, $\{a,c\}$, $\{a,d\}$, $\{b,c\}$, $\{b,d\}$, $\{c,d\}$] and $C(4,2) \times 2^2 = 24$ possible value-combinations. For notational shorthand, we will refer to these parameter combinations as {*ab*, *ac*, *ad*, *bc*, *bd*, and *cd*}. Of these, 19 value-combinations are covered. The missing value-combinations are $ab = 11$, $ac = 11$, $ad = 10$, $bc = 01$, $bc = 10$. Note that only two, *bd* and *cd*, are covered with all 4 possible value pairs. So we have 79% (19/24) for the value combination coverage metric. For a better understanding of this test set, we can compute the configuration coverage for each of the six parameter combinations, as shown in Fig. 2. For this test set, one of the combinations *bc* is covered at the 50% level, three *ab*, *ac*, *ad* are covered at the 75% level, and two (*bd*, *cd*) are covered at the 100% level. And, as noted above, for the whole set of tests, 79% of variable-value configurations are covered. All 2-way combinations have at least 50% configuration coverage, so $(.50, 2)$-completeness for this set of tests is 100%. Although the example in figure 1 uses parameters 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. (example from [4]).

Vars	Configurations	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

total 2-way coverage = 19/24 = .79167

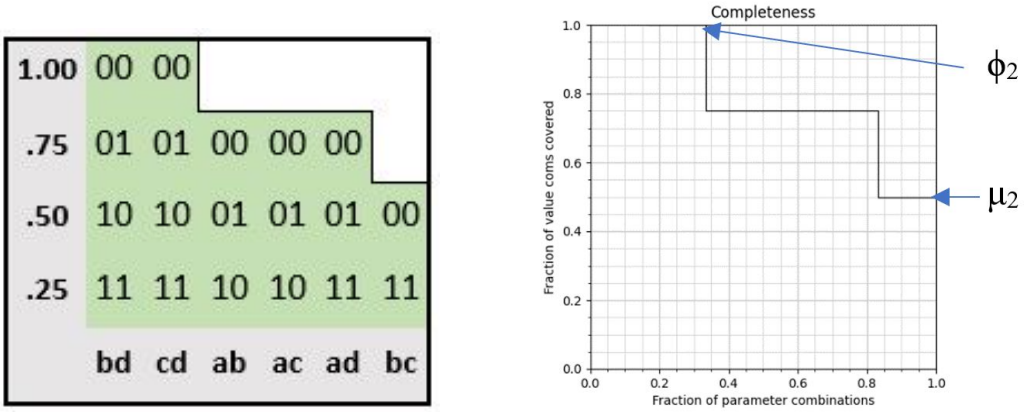
(.50, 2)-completeness = 6/6 = 1.0
(.75, 2)-completeness = 5/6 = 0.83333
(1.0, 2)-completeness = 2/6 = 0.33333

Figure 2: A test array covering 2 way combinations of a, b, c, and d to different levels.

To make the data in figure 2 more understandable, it will help to produce a graph whose components are easily tied to the data. One way to do this is shown in figure If the value combinations are listed as columns and sorted by the level of coverage for each combination, a graph as seen in figure 3a results.

The area under the curve represents value combinations covered, and the area above the curve represents combinations that have *not been covered* in the test set. It is also easy to see that the value combinations in figure 3a cover 19/24 = 79% of the 2-way combinations that are possible. Based on the ideas just presented, we define three measures S_t, μ_t, and ϕ_t. The measures are illustrated for 2-way covering in figure 3b.

- S_t: fraction of possible value combinations covered, i.e, the area under the curve in a completeness graph, as shown in figure 3b.



(a) Table Mapped to Graph (b) The concept of μ_t and ϕ_t

Figure 3: Graphical representation of 2-way input space covering

- μ_t: the minimum t-way coverage, i.e., minimum (p, t)-completeness
- ϕ_t: the proportion of full t-way combinations for which $(1.0, t)$-completeness is achieved.

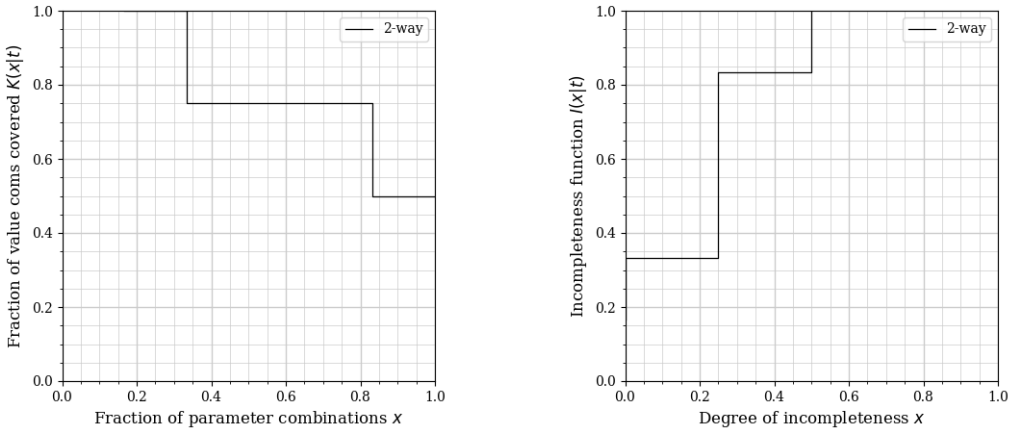
Because the graph represents a unit square, it can be seen that $S_t \geq \mu_t + \phi_t - \mu_t\phi_t$. Graphing the quantities μ and ϕ provides an intuitive approach to understanding combinatorial coverage. The value of μ is particularly important. If μ is low, then there is at least one combination that is not covered well, so there may be insufficient evidence that the Software Under Test (SUT) will process inputs containing this combination correctly.

The combinations covered are represented by the area below the curve in Fig. 3 (b). Combinations that have not been included in any test are those above the curve. In the case of the example, these are $ab = 11$, $ac = 11$, $ad = 10$, and $bc = 01$, $bc = 10$. Even if code has been tested and tests demonstrated to provide full statement and branch coverage, the combinations in the uncovered region of the input space, in the upper right corner, have not been included in any test. Therefore we cannot say with confidence what will happen if these combinations are encountered in system operation, raising the possibility of vulnerabilities or system failures or errors. Furthermore, the value combinations that are missing would be likely triggers of errors that are encountered, as they are inputs that have not been tested.

2.2 Completeness and Incompleteness Functions

The fraction of the input space that has not been tested is of course $1 - S_t$, which is the level of incompleteness of the full test set. We can define functions that give the degree of completeness or incompleteness. The completeness function shows the strength of the test suite created from the chosen parameter value combinations. Incompleteness, on the other hand, shows the weakness that still remains in terms of the combinations of parameter values with which a software has yet not been tested. In other

words, the completeness is useful for answering the question, "How thorough is this test set?", while incompleteness answers the opposite question, "How much is missing from this test set?"



(a) Completeness function (b) Incompleteness function

Figure 4: Completeness and incompleteness functions.

A *combinatorial t-way completeness-function* $K(x \mid t)$ of a test suite is the (p, t)-completeness, or minimum value combination coverage, of the fraction x of highest covered parameter combinations (see Fig. 4 (a). The graph of $K(x \mid t)$ is a step function where $0 < x \leq 1.00$, for a given value of t. A point $K(x \mid t)$ on the vertical axis, for point x on the horizontal axis, gives the minimum fraction of value combinations covered in all of the fraction x of highest-covered parameter combinations. The area below the graph of $K\ (x \mid t)$ is S_t, the overall combinatorial t-way completeness. The area above the step-function graph of $K\ (x \mid t)$ is overall combinatorial t-way incompleteness or 1 - S_t.

A *combinatorial t-way incompleteness-function* $I(x \mid t)$ of a test suite is the fraction of t-way parameter-combinations, which are incomplete (relative to a test suite of strength t) to the degree x or less, The graph of $I(x \mid t)$ is a step-function. Fig. 4 (b) displays a graph of the t-way incompleteness-function $I(x \mid t)$ of the test suite in Fig. 1 for $t = 2$. For any point x, $I(x \mid t)$ on the graph, the abscissa (horizontal axis) indicates the degree of incompleteness x, where $0 \leq x < 1$, and the ordinate (vertical axis) $I(x \mid t)$ indicates the fraction of t-way parameter-combinations, out of $C(k, t)$, which are incomplete to the degree x or less. The area above the step-function graph of $I(x \mid t)$ is overall combinatorial t-way incompleteness. The area below the graph of $I(x \mid t)$ is overall combinatorial t-way completeness.

The incompleteness-function $I(x \mid t)$ for a test suite of strength t is a horizontal line at the ordinate value $I(x \mid t) = 1.00$, which means that the fraction of parameter-combinations for which no t-tuple of values is missing is 1.00. That is, no t-way parameter-combination has missing t-tuple of values.

Example: $K(0.4 \mid 2)$ is 0.75, which is the lowest value 2-way combination coverage for the top 40% of parameter combinations. It can also be seen in the graph that 83% of the 2-way combinations have

coverage of at least 0.75, and all have coverage of 0.50 or greater.

2.3 Applications of Measurement

Spacecraft software is subject to extensive testing prior to deployment. The combinatorial coverage tools described in this section were initially applied to a large test set developed software for a NASA mission [13]. The test set contained nearly 7,500 tests, so strong t-way coverage was expected to be present, but there were no methods for measuring to ensure this coverage.

We used our Combinatorial Coverage Measurement (CCM) tool which outputs both the *CIPC(n,t)* as well as (p, t)-completeness of a test suite. After applying the CCM test coverage tool, it was found that reasonably strong coverage was present in the tests developed entirely by engineering judgement (see Figure 5), but even 2-way combinations, shown using the red line, were not fully covered, and many additional tests would have been needed for 3-way (blue line) or higher strength (e.g., green line representing 4-way coverage. Details of coverage are provided in Fig. 6.

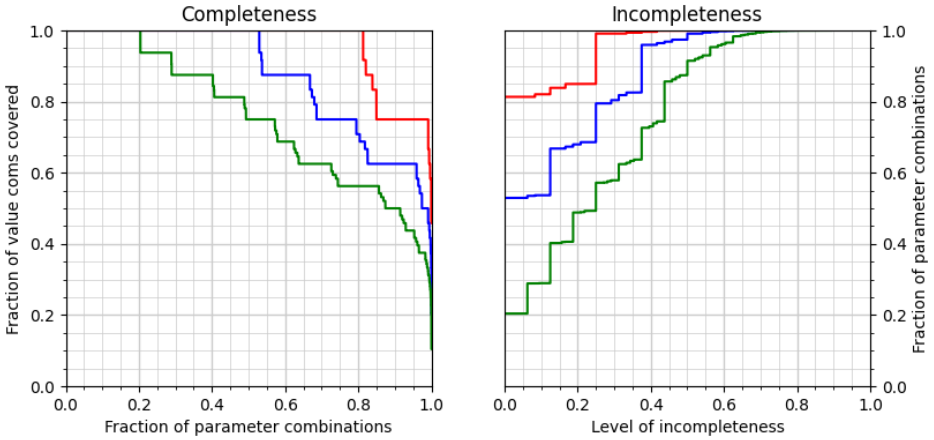


Figure 5: Combinatorial coverage of spacecraft software

CCM has been applied in a number of projects to improve existing regression tests, which in some cases were based on recording ongoing transactions for replay after modifications were made to the code. Measuring the coverage of these transaction sets allowed for improvements to more thoroughly check functions. Security in particular is an area where ultra-rare and unusual combinations of inputs may lead to problems, especially since attackers look for these rare inputs to discover vulnerabilities [14][15]. CCM has been helpful in evaluating the thoroughness of tests for secure protocols and other security-critical applications [16] [17]. Industry examples include the following:

- Adobe used CCM to measure the effectiveness of a validation framework for an analytics tool in its Marketing Cloud product. This was the first usable approach to quantitatively measur-

```
_____________________________________________________________
Summary coverage report,   file = Stereo7489x82.csv
Input model configuration:
2⁷⁵ 4² 6² 6²
_____________________________________________________________
n = nr of parameters                       =             82
r = nr of rows                             =          7,489
C_2 = nr 2-way coms                        =          3,321
M_2 = nr possible 2-way value coms         =         14,761
V_2 = nr covered 2-way value coms          =         13,882
S_2 = total 2-way coverage                 =         0.9405
F_2 = fraction 2-way value coms full cov   =         0.8133
L_c2 = min cov of 2-way value coms         =         0.4583
U_i2 = max 2-way incompleteness            =         0.5417
_____________________________________________________________
C_3 = nr 3-way coms                        =         88,560
M_3 = nr possible 3-way value coms         =        828,135
V_3 = nr covered 3-way value coms          =        687,979
S_3 = total 3-way coverage                 =         0.8308
F_3 = fraction 3-way value coms full cov   =         0.5292
L_c3 = min cov of 3-way value coms         =         0.2014
U_i3 = max 3-way incompleteness            =         0.7986
_____________________________________________________________
C_4 = nr 4-way coms                        =      1,749,060
M_4 = nr possible 4-way value coms         =     34,364,130
V_4 = nr covered 4-way value coms          =     23,651,484
S_4 = total 4-way coverage                 =         0.6883
F_4 = fraction 4-way value coms full cov   =         0.2041
L_c4 = min cov of 4-way value coms         =         0.1042
U_i4 = max 4-way incompleteness            =         0.8958
_____________________________________________________________
```

Figure 6: Spacecraft software coverage report

ing assurance for this type of analytics product, and results showed the combinatorial coverage measurement was effective [18].

- Siemens included CCM in its Zero Defect program to measure aspects of test sets to identify where improvements could be made, where "the goal was to craft the leanest and most impactful formal testing and devise optimal configurations for the system under test." [19].
- An analysis of tests for Bombardier Transportation discovered that manually created tests by engineers did not achieve strong combinatorial coverage, with an average of 78.6% for 2-way interactions, 57% for 3-way interactions, 40.2% for 4-way interactions, 20.2% for 5-way interactions and 13% for 6-way interactions.. The measures were used to determine improvements to test sets for critical code [20].

3 Combinatorial Coverage Difference Measures

Combinatorial Coverage Difference Measure or CCDM has so far been applied to several problem domains. Initially this approach was used in fault identification, specifically to determine the particular combination(s) of parameter values that would trigger a fault. Another example problem where there is a need to distinguish one class of elements from another is anomaly-based intrusion detection, which seeks to determine if a particular exchange of packets represents an attempted network intrusion. Thus it is useful to generalize the approach to find combinations that are present in one class or set and absent or rare in another, and in order to distinguish one set from another. Note that this is simply a generalized version of the fault location problem, where the class whose distinguishing features are to be identified is the set of failing tests. In this publication, we will refer to sets being distinguished as either *Class* or *Non-class* files or sets. A *Class file* or a *Class set* is the one where certain value combinations of parameters, inputs, or configuration are present while in a *Non-class file* or *Non-class set*, these value combinations are absent or rare.

3.1 Related Definitions

The measures defined in this section have been implemented in research tools that can be applied to a broad range of problems. These tools take as input the two files of *Class* and *Non-class* instances, and produce measures of their similarity or difference. The first define the t-way combination sets in the two input sets.

- C_t = set of t-way value combinations in Class file
- N_t = set of t-way value combinations in Non-class file

The terms *Class* and *Non-class* are used as generic terms for sets of objects that can be distinguished on the basis of some property or properties. For example, in earlier applications, set differences of value

combinations have been used to identify the causes of failures [12][21]. In a machine learning context, however, these sets may refer to concepts that are to be learned, such as distinguishing one animal species from others. In both cases, the process is the same: set differencing is used to identify combinations that occur in the *class set* that do not occur, or are rare, in the *non-class set*. If this difference is computed on value-combinations in failed tests vs. passed tests, then the difference contains value combinations that have triggered the failure (in a deterministic system). In machine learning, the difference represents properties or attributes that occur in the class (e.g., a particular animal species) that do not occur, or are rare, in the non-class examples (other species).

3.1.1 Distinguishing Combinations

We define a t-way combination c_t as a *distinguishing combination* for the class C if it is present in class instances C, and absent in non-class instances N, or if it is more common in C than N as determined by a threshold value.

A threshold T determines if a particular t-way value combination c_t is common in set C_t and rare in set N_t, and thus distinguishes one set from the other.

Definition: A combination x_t for a class C is *distinguishing* iff $occurrences(x_t, C_t) > T \times occurrences(c_t, N_t)$, where $occurrences(x, Y)$ = frequency of value combination x in set of value combinations Y.

If $occurrences(x_t, N_t) = 0$, then the combination x_t is unique to the class file, and if $occurrences(x_t, N_t) = 0$ for all distinguishing combinations, then the set of distinguishing combinations is equal to the set difference $C_t \backslash N_t$. (We will abbreviate C_t and N_t as C and N, where interaction level t is clear or is not needed for discussion.) If T is some multiple, then the combination is rare in N and common in C, therefore it is more strongly associated with C. The higher the value of T, the more strongly distinguishing combinations are associated with C. This threshold value is chosen based on the application, but in general a threshold value of $T = 0$ can be used for deterministic software, the most common case, and $T > 0$ for non-deterministic problems, such as may be encountered in anomaly-based intrusion detection. The earliest uses of the methods described here assumed deterministic functions, where a particular combination would always trigger a fault if it was present. In some applications, it may be more important to base decisions on the proportion of combinations in class versus non-class instances. For example, a particular combination of packet field values may be much more common in server interactions that are part of an intrusion attempt, but the combination's presence or absence is not uniquely determinative of the type of transaction or class of interaction. We will refer to the set of distinguishing t-way combinations for class set C as U_{Ct}, and for N as U_{Nt}.

- U_{Ct} - The t-way combination that is present in the Class and rare or absent in the Non-class set.
- U_{Nt} – The t-way combination that is present in the Non-class and rare or absent int Class set.

For some applications, a distinguishing combination will guarantee recognition of a class member within a particular database. For non-deterministic processes, distinguishing combinations may be evidence for membership in a class but not guarantee this. For example, in disease diagnosis, combinations of

various conditions may be strongly indicative of a particular disease, but tests looking at other attributes (outside of the existing data set) may be needed.

A second problem with determining if a combination indicates class membership is the large number of combinations that are possible, often too large for the number of observations in a data set. For example, if the number of *t-way* combinations in class set C and Non-class set N is smaller than the number of possible *t-way* combinations, then some combinations may be associated with class membership only by chance. Future versions of the tool described here will include data on the number of combinations in the input files compared with the number needed for assurance that a combination is reasonably associated with a class.

3.1.2 Minimal Distinguishing Combinations

Any t-way combination contains t different $(t-1)$-way combinations of parameters, since $\binom{t}{t-1} = t$; for example abc contains ab, ac, bc. The number of distinguishing combinations will increase with increasing t, since any t-way combination that is distinguishing will be included when a $(t+1)$-way combination is produced by joining an additional attribute to the t-way combination. So if c_t is distinguishing, then c_u is distinguishing for any higher strength value u. We can define a *minimal distinguishing t-way combination* by the lowest value of t for which c_t is distinguishing.

Example:

Class File (C) =

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

Non-class File (N) =

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

Then $abc = 011$ is distinguishing, $ab = 01$ is a minimal distinguishing combination, and $ac = 01$ is not distinguishing because it occurs in both C and N. When computing sets of combinations for various measures, it will be important to separate minimal distinguishing combinations from those that are not minimal. For example, if $ab = 01$ is a minimal distinguishing combination, then any 3-way (or higher strength) combinations that include $ab = 01$ will also be counted as distinguishing. In many applications, as discussed later in this paper, it is not appropriate to include these higher strength combinations for some measures. We define a *filtered* set of t-way distinguishing combinations as one in which combinations containing lower strength distinguishing combinations have been filtered out. For example, if $ab = 01$ is distinguishing, and set X contains 3-way combinations $abd = 010$, $abe = 011$, and $bce = 000$ then $abd = 010$ and $abe = 011$ will not be included in the filtered set X.

3.2 Applications of Differencing

The notion of distinguishing combinations has a natural application whenever it is useful to distinguish elements of two or more sets, based on differences in their attributes or values. Initially applied to fault location for combinatorial testing, the method can also be used in explaining decisions of artificial intelligence or machine learning algorithms, and in the important problem of transfer learning in AI.

3.2.1 Fault Location

If a large test set is run, and a few tests fail, then it becomes necessary to locate the cause of the failure. In particular, it will be useful to identify what specific combinations triggered the failure. A variety of methods have been developed to accomplish this task, but the general objective is to identify combinations that occur in failing tests that do not occur in passing tests [21] [22] [23] [24] [25] [26] [12]. The reasoning is that if a combination is in a failing test, then it is clearly a possible proximate cause of the failure. However, if the same combination occurred in a passing test, then it does not always lead to a failure, and in deterministic systems it can be ruled out as a sufficient failure-triggering combination. Hence there is a need to identify combinations that occur in failing tests, but not in passing tests. These are distinguishing combinations as defined above, because their presence in a test identifies that test as one in the failure set.

This process is illustrated in Fig. 7, where the large blue set represents combinations in passing tests, and the small yellow set represents combinations in failing tests. If P_t is the set of t-way combinations in passing tests, and F_t is the set of t-way combinations in failing tests, then $F_t \backslash P_t = \{$*combinations in failing tests not also in passing tests*$\}$. Note that this set can be large, since each test includes $C(n,t)$ t-way combinations, so computing this difference does not immediately locate failure-triggering combinations. In general, the difference $F_t \backslash P_t$ is a starting point for determining a set of suspect combination(s) that may have resulted in a failure or error. A variety of methods are available for determining which combinations in the suspect set will trigger a failure, often including generation of new tests to separate out results produced by inclusion of the suspect combinations, until individual combinations can be identified that trigger the failure or error.

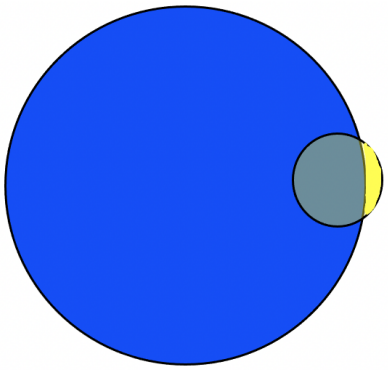


Figure 7: Passing and failing tests, with distinguishing combinations.

3.2.2 Explainable Artificial Intelligence

The fault location process described above maps directly to the problem of explaining a decision in classification problems [27]. This extension is in fact a generalization of the process, identifying a particular class of object by its features versus non-class objects that have different combinations of features. In fault location, the class is associated with failures and non-class instances with passing or non-failures. Instead of narrowing down a set of suspect combinations, however, a generalized use of combination differencing produces a set of t-way combinations each of which are always associated with the class and never, or rarely, associated with the non-class instances. For example, a certain 3-way combination of temperature, humidity, and CO_2 air content may be associated with occupied rooms, but never occur in unoccupied rooms, a problem that may occur in building access control. The levels of these factors in the 3-way combination may occur individually or in pairs in unoccupied rooms, but not the unique 3-way combination. As such, the identification of the 3-way combination serves as an explanation or justification of the conclusion.

3.2.3 Transfer Learning

Transfer learning in the field of AI deals with predicting the performance and accuracy of a model that has been trained on one data set when applied to different data. Lanus et al. [28] defines a set of difference metrics that can be used in evaluating whether a training data set (source) is sufficiently representative of a second set (target) for a model trained on the first one to apply successfully. The intuition is that the smaller the difference between source and target, as measured by combination differences, the greater accuracy that can be expected using the model trained on the source.

The differencing method is also shown to be useful in training set design. Conventionally, machine learning applications use a training data set and a test data set, selected randomly. Random selection helps to ensure that the training set contains a representative sample of the field of application, and the

training set is generally larger than the test set (e.g., 2:1 or 3:1 ratio). Combinatorial coverage can be used to determine the degree to which combinations in the training set cover combinations in the test set. This approach may not only improve machine learning training processes, but also provide practical measures of data set characteristics, for use in AI/ML experiments and algorithm development.

3.3 Extending the Combination Differencing Method

In some applications, it will be desirable to compute not only the $C \backslash N$ (Class \Nonclass) set difference, but $N \backslash C$ (Nonclass \Class) as well. The tool described in this section computes various measures related to these differences in a single run, generating results that can then be used in verification and testing, explainable AI, fault location, or other problems where combinatorial coverage measurement may be relevant.

3.3.1 Available Measures

NIST's combinatorial differencing tool produces a variety of measures that can be applied to a broad range of problems. Two matrices, represented as comma-separated value files, are processed by a command line tool, which outputs a detailed set of measures of the input files and their combination set differences. The number of parameters, or parameters/factors, must be the same in both. It is assumed that any possible individual value of each parameter occurs in either C, or N, or both.

Two assumptions are possible regarding combinations of the values of input parameters. In some applications, it is reasonable to assume that the universe of relevant combinations is represented by $C \cup N$. This assumption may be suitable when there are many physical or other constraints on possible combinations, and **the data set is very large** such that every relevant combination occurs somewhere in C or N. (Note that as t is increased, it is always possible to find some t-way combination of values that does not occur in either C or N, unless the union of these sets is exhaustive. *Relevant* indicates a level of t that is estimated to represent the maximum number of factors that affect system function, such as 5-way or 6-way.) Alternatively, it may be more realistic to assume that some combinations of values may eventually be encountered in actual use cases, but they are not present in either C or N, because of insufficient simulation or run time. For example, a small autonomous vehicle simulation may include observations where the vehicle is operated in snow and cloudy skies, and in dry weather and sunny skies, but may not include the condition where the vehicle is in snow with sunny skies. The choice of assumption regarding the universe of possible parameter combinations affects computation of some measures, as discussed below. A number of measures and ratios of measures can be computed based on sets of combinations, as illustrated in Fig. 8. .

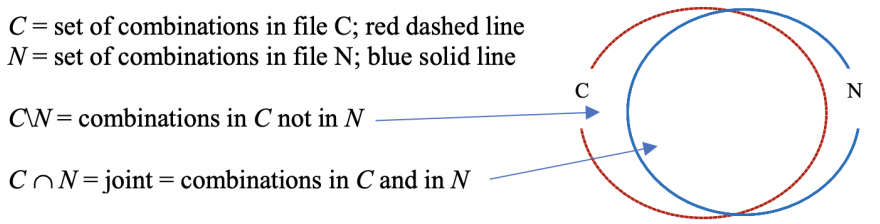


Figure 8: Class and non-class file combinations

The basic values and ratios described below are computed and reported in order to compare sets of combinations in files C and N. These quantities are given for combinations of values in C and N. In most cases they reflect *sets* of value combinations. That is, if a given combination such as $ab = 01$ appears more than once in file C, it will be included only once in the set C_2 of 2-way combinations in file C. However, multisets including counts of t-way combinations are also maintained and may be used in computing some ratios, as detailed in the definitions below.

Symbols:

- n = number of parameters or factors/parameters
- r_C = number of instances/rows in Class array/file
- r_N = number of instances/rows in Non-class array/file
- C_t = number of t-way combinations in $C = \binom{n}{t}$
- N_t = number of t-way combinations in $N = \binom{n}{t}$
- M_{Ct} = number of t-way value-combinations in $C = v^t\binom{n}{t}$
- M_{Nt} = number of t-way value combinations in $N = v^t\binom{n}{t}$
- M_t = number of possible value combinations in $C \cup N$
- V_t = number of value combinations covered in $C \cup N$ (if assumed that observations cover all value combinations, then $M_t = V_t$)
- J_t = number of combinations covered jointly, in $C \cap N$
- U_{Ct} = number of distinguishing combinations for C (combinations present in C but absent in N)
- U_{Nt} = number of distinguishing combinations for N (combinations present in N but absent in C)
- F_{Ct} = number of filtered distinguishing combinations for C
- F_{Nt} = number of filtered distinguishing combinations for N
- *Counts* of quantities above for numbers of combinations are also available

3.3.2 Basic ratios

- **Similarity Ratio (SR):** J_t/V_t represents intersection over union, or "Jaccard similarity", which is a common measure of the similarity of two sets. For class, C, and non-class, N, sets of combinations, this ratio gives the proportion of combinations that are jointly found in C and N. If the

similarity is low, then it is relatively easy to distinguish C from N, because they are very different (e.g., turtles vs. wolves). Conversely, if C and N are similar, then there are fewer distinguishing combinations to identify instances as members of the class C (e.g., dogs vs. wolves).

- **Total Coverage Ratio (CR):** V_t/M_t = Proportion of total possible t-way combinations covered. If it is assumed that all combinations of interest occur in either C or N, then this ratio will be 1.0. If it is not assumed that all combinations of interest occur in either C or N, then this ratio provides a measure of the degree to which observations cover combinations that may be encountered in practice. If it is high, then there is greater confidence that the test or simulation has included situations that may occur in operation. Conversely, if low, then more test or simulation may be required.

- **Distinguishing Combination Density (DCD):** U_{Ct}/M_t : Proportion of distinguishing combinations for C out of possible combinations. This value suggests how "easy" it may be to identify members of a class; a higher value means more attributes or traits associated with a class.

- **Filtered Distinguishing Combination Density (FDCD):** F_{Ct}/M_t: Proportion of filtered distinguishing combinations for C out of possible combinations. This value suggests how "easy" it may be to identify members of a class, and also indicates the degree to which distinguishing t-way combinations are of smaller size, i.e., lower values of t.

- **Distinguishing Combinations Per Combination**: U_{Ct}/C_t: Another measure of how common distinguishing combinations are among combinations in class and non-class members.

- **Distinguishing Combinations per Instance**: U_{Ct}/r_C: How common are differences among members of the class?

- **Filtered Distinguishing Combinations per Combination:** F_{Ct}/C_t: Another measure of how common distinguishing combinations are *among combinations* in class members.

- **Filtered Distinguishing Combinations per Instance:** F_{Ct}/r_C: How common are differences *among members of the class*?

3.3.3 Additional metrics

- **Ratios of distinguishing combinations:** U_{Ct}/U_{Nt}, F_{Ct}/F_{Nt} give potentially useful information on whether class or non-class objects have more distinguishing combinations, which may indicate the degree to which a class differs from non-class objects. This differs from the basic similarity ratio, which compares class objects with all objects; instead it compares class with non-class objects, potentially a sharper distinction.

- ***Counts* of quantities for numbers of combinations:** In some cases it is useful to know how frequently particular combinations occur among class and non-class instances. This is particularly true with non-deterministic applications, where certain combinations may be strongly indicative of a particular conclusion, but do not guarantee the conclusion in all cases.

3.3.4 Example

To illustrate these measures and their potential use, we can consider a small example. Suppose we have a database of animals as shown below (a $2^4 3^1 4^1$ configuration), and the task is to determine levels of similarity and differences. Consider differences between birds and non-birds in the database. What t-way combinations are associated with these two classes?

1) fur: y, n
2) eggs: y, n
3) legs: 2, 4
4) wings: y, n
5) size: s, m, l
6) color: b(lack), w(hite), g(ray), r(ed)

	fur	eggs	legs	wings	size	color
dog1	y	n	4	n	m	b
dog2	y	n	4	n	s	w
bat1	y	n	2	y	s	g
bat2	y	n	2	y	s	b
bird1	n	y	2	y	s	r
bird2	n	y	2	y	m	w
bear1	y	n	4	n	l	b
bear2	y	n	4	n	l	w

Table 1: Example animal identification task

Measuring 1-way through 4-way combinations in Table 1 produces the result shown in Fig. 9. Among the measures reported, the distinguishing combination density (raw and filtered) and similarity measures are important in evaluating similarity and difference. As can be seen in the table, there are five single values that are common between the bird and non-bird classes, or 33.3% of the total number of single values. At $t = 4$, there are no 4-way combinations shared by the two classes, so 4-way distinguishing combinations for the bird class could be used to identify this class. To make this easier to visualize, Fig. 10 shows Venn diagrams of the combinations of features in the two classes.

3.4 Visualizing Coverage of Value combinations

Recall from Section 1 that we refer to the parameters in a t-way tuple as a combination, and the particular values for the t-way tuple as a value combination. For example, if parameters are a, b, ... , g, then $\{a, b, e\}$ is a 3-way combination containing the parameters a, b, and e. If the three parameters are binary, then there are eight possible value combinations: $\{0, 0, 0\}$, $\{0, 0, 1\}$ etc. Fig. 3b illustrated how to graph overall combinatorial coverage of a data set, but it is important to be able to evaluate combinatorial coverage more closely. The minimum coverage, μ_t, (for which (p, t)-completeness = 1.0)

```
Summary coverage report, F1 = bird.csv   F2 = notbird.csv
Nvars:     6    Nrows F1:       2    Nrows F2:        6
Full data set = F1 union F2
Measures by interaction level t = 1, 2, 3, 4
```

t ->	1	2	3	4
Combinations	6	15	20	15
Value coms	15	66	113	99
Covered	15	66	113	99
Tot cover ratio	1.000	1.000	1.000	1.000
F1\F2 number	3	21	35	29
Dcom density	0.200	0.318	0.310	0.293
F1\F2 filtered	3	5	0	0
Filt dcom dens	0.200	0.076	0.000	0.000
Joint number	5	3	1	0
Similarity	0.333	0.045	0.009	0.000
Dcom/com	0.500	1.400	1.750	1.933
Dcom/row	1.200	4.773	6.195	4.394
Filt dcom/com	0.500	0.333	0.000	0.000
Filt dcom/row	1.200	1.136	0.000	0.000

Figure 9: Distinguishing combinations for Class and Non-class

gives the smallest proportion of value combinations covered for all t-way combinations in a data set, but there can be great variability among a large number of combinations.

Suppose we have a set of training data for a machine learning application, and there is a need to determine the degree to which the training set reflects combinations of values in real-world environments. This is essentially the same as the problem of determining if a set of tests is adequately reflective of inputs that will be encountered in daily use for the SUT. If possible values and constraints are known in advance, a covering array can be constructed to the desired strength, to ensure that suitable coverage is achieved. However, it is not always practical to use this approach.

In many cases, it is not clear what inputs will be encountered in use. In a testing application, this problem may arise when a system is deployed in a new environment. This situation also occurs in machine learning, where it is referred to as the transfer learning problem, where an algorithm has been trained for one environment and it is to be applied to another. If the distribution of inputs is nearly the same in the two environments, then the application may function well, otherwise there may be a risk of errors (in machine learning) or failures (in conventional applications).

Note that it is possible that differences between two environments may include constraints on combinations of values that may be possible, not only on the values themselves. In many cases it may not be possible to analytically determine the constraints that exist. It may be possible to infer some from data collected during use, but in many cases this is not practical. Any constraints among attribute values are inherent in data collected, so an alternative to specifying constraints is to determine if combinations are similar between two data sets. That is, are there some combinations in the test or training set that

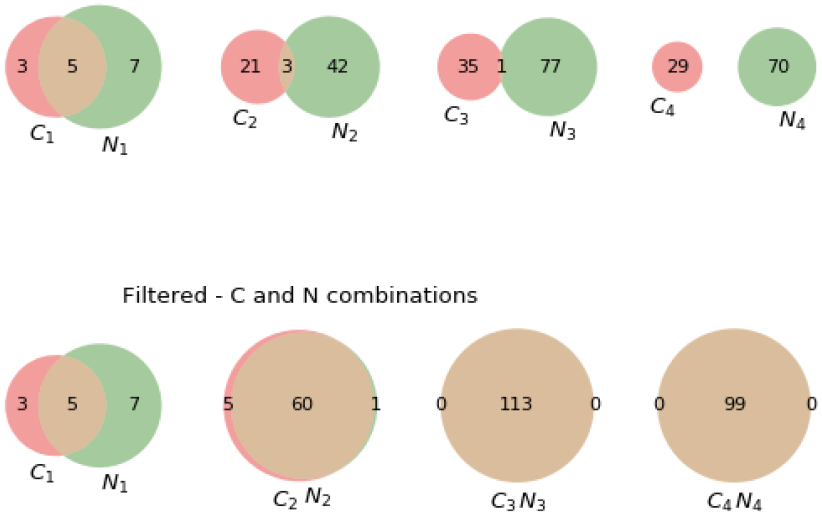


Figure 10: Raw and filtered set distinguishing combinations for classes C and N

are not present in the data from actual use, or vice versa? Thus it is essential to measure the degree to which two data sets are similar.

3.4.1 Basic Measures

A variety of measures can be applied to understand similarity between data sets using combinatorial coverage, as introduced previously. Suppose we wish to evaluate the degree to which a set of inputs is representative of a full set. In this case, we will assume that the full set has unknown constraints, so the problem is not as simple as generating a covering array of t-way combinations of values observed in the full data set. Instead, we want to consider the degree to which the small subset reflects the values of a much larger set, as a test set (in a testing context) or for training (in a machine learning context).

Fig.11 (a) shows a test (or training) set and Fig. 11 (b) shows the larger set which includes all observations, including those in (a). Even with this small array, it is hard to say at a glance how well test array (a) covers the combinations in (b). Analyzing the array coverage, we can produce useful measures shown in Fig. 12, where file F1 is the two row array in (a) and F2 is the four row array in (b). Because we are assuming in this case that the larger array is representative of the real-world environment, the combinations covered in the full set represent 100% of possible combinations (by assumption). Also provided are the following, for each level of t-way interaction:

- *coms* = number of combinations of the six attributes, for the given level of t
- *value combinations* = number of value combinations total, for given level of t (N value combinations = N covered because of the assumption above).

a	b	c	d	e	f
1	0	1	0	1	1
1	0	1	1	1	1

(a) Test set

a	b	c	d	e	f
1	0	1	0	1	1
1	0	1	1	1	1
1	0	0	1	0	1
1	0	1	0	1	1

(b) All known observations

Figure 11: Six parameter test set

- $F1 \backslash F2$ = number of combinations in $F1$ that are not in $F2$, always *zero* in this case because $a \subset b$
- *joint* = number of value combinations for the given level of t that appear in both $F1$ and $F2$
- *jointpct* = percent of value combinations for the given level of t that appear in both $F1$ and $F2$

Note that the report shows 9 single values covered for the full set of four tests. For six binary parameters there would be of course 12 possible single-value value combinations, but constraints among values may limit possible values, so there are three parameters that show only one value. Of these nine values, the two tests in (a) include seven, so the joint coverage is $7/9 = 78\%$. Coverage is also shown for combinations up to $t = 4$-way. For example, the joint coverage for $t = 2$, i.e., 2-way combinations that occur in both array (a) and (b) is 69% , with 20 of 29 combinations covered. For this analysis, the set difference of $F1 \backslash F2 = 0$, because there are no combinations in $F1$ that are not in $F2$. For other types of problems, there may be sets that overlap but do not have a subset relationship.

```
Summary coverage report,   F1 = test5.csv   F2 = all5.csv
Nvars:     6    Nrows F1:       2    Nrows F2:        4
Full data set = F1 union F2
Measures by interaction level t = 1, 2, 3, 4
```

t ->	1	2	3	4
Combinations	6	15	20	15
Settings	9	29	46	39
Covered	9	29	46	39
Tot cover ratio	1.000	1.000	1.000	1.000
F1\F2 number	0	0	0	0
Dcom density	0.000	0.000	0.000	0.000
F1\F2 filtered	0	0	0	0
Filt dcom dens	0.000	0.000	0.000	0.000
Joint number	7	20	30	25
Similarity	0.778	0.690	0.652	0.641
Dcom/com	0.000	0.000	0.000	0.000
Dcom/row	0.000	0.000	0.000	0.000
Filt dcom/com	0.000	0.000	0.000	0.000
Filt dcom/row	0.000	0.000	0.000	0.000

Figure 12: Summary coverage report

3.4.2 Per-combination Coverage

The values shown in the coverage report are useful for determining the degree to which a test (or training) set is representative of real-world observations, but these values only indicate overall coverage. To really understand the utility of the tests in (a), we need to look at coverage on a per-combination basis. Fig. 13 shows views of 2-way and 3-way coverage for the example above. As shown in the report, the overall coverage for 2-way and 3-way is indicated as S2 = 0.69 and S3 = 0.65 respectively. Minimum coverage, μ_2 and μ_3 is also indicated (as M2 and M3), both at 0.50. Each block in a coverage chart shows the level of coverage for a combination. Thus there are 15 blocks in the 2-way coverage chart, because there are 15 2-way combinations, and similarly 20 blocks for the 3-way combinations. For readability, combination coverage figures are computed as a list comprehension, but presented as a square chart. Thus blocks for 2-way coverage show (reading from left to right, top to bottom) coverage for $(a, b), (a, c), (a, d), (a, e), (a, f), (b, c), \ldots, (e, f)$. Referring back to Fig. 13, it can be seen that (a) contains all 2-way value combinations for $(a, b), (a, d), (a, f), (b, d)$, etc. This is indicated below with black squares for complete coverage. Similarly, lower levels of coverage for other combinations can be seen in other blocks.

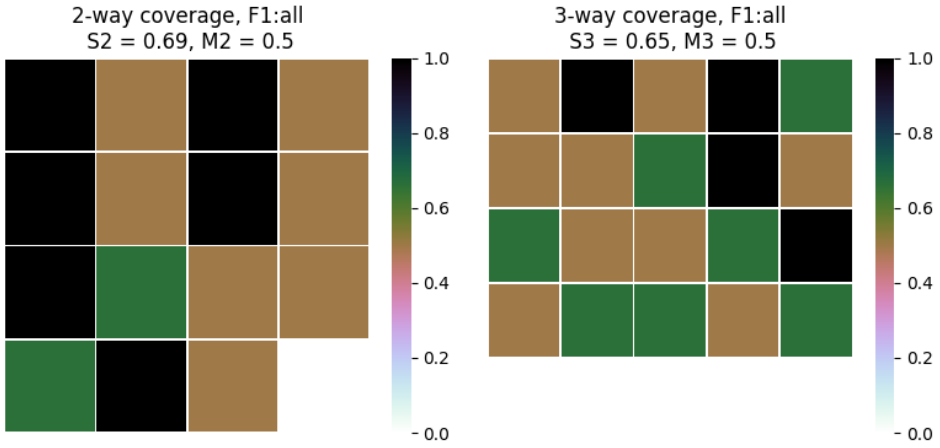


Figure 13: Per-combination coverage

Vulnerability and fault detection often depend on specific sequences of inputs that establish states which eventually lead to a failure. That is, most software processes are not pure functions where input produces the same output whenever the process is invoked. The response of the process to a particular set of input values may depend on its current state, which is established by a sequence of inputs. Determining initial state for a particular test, and generating inputs to produce that state, or set of conditions, is a difficult problem in practical software testing.

A larger challenge in testing is ensuring that tests have been sufficiently broad and diverse, reflecting potential use conditions, to ensure high confidence in system correctness, safety, and security. However, beyond basic structural coverage metrics, it is often difficult to determine if sufficient diversity of inputs has been achieved. Measures are needed to ensure that relevant combinations of input values and input sequences have been tested and verified for correct operation. Combinatorial coverage measures provide an effective method of quantifying the thoroughness of test input values. A number of measures have been defined for the coverage of (static) input value combinations. These measures quantify the degree to which input values cover the potential space of parameter value combinations, without regard to the sequence in which these inputs occur in a test set, or in normal operation. Because system state is affected or determined by the sequence of inputs, even thorough coverage of the input space may not detect some failure conditions. Thus it is desirable to supplement measures of input space coverage with measures of the sequences of input value combinations. This publication documents input sequence coverage measures and a research tool that can be applied in practical applications.

3.4.3 Within-combination Coverage and Variability

It is also useful to consider the coverage associated with individual combinations, as shown in Fig. 14 below. These charts show the percentage of values covered for each 2-way combination which contains one of the six attribute parameters. Thus the first chart shows coverage for attribute a within 2-way combinations: (a,b), (a,c), (a,d), (a,e), (a,f). Note that for attribute d, coverage is very good, with either 0.75 (green) or 100% (black) coverage for every pair of attributes containing d. The overall coverage for

these five 2-way combinations containing d is 0.83. Conversely, coverage is not good for e, as shown in the next chart, where overall coverage for these five is S2 = 0.55. The same type of charts can be produced for 3-way and 4-way combinations, subject to practical limits on the number of combinations that can be displayed.

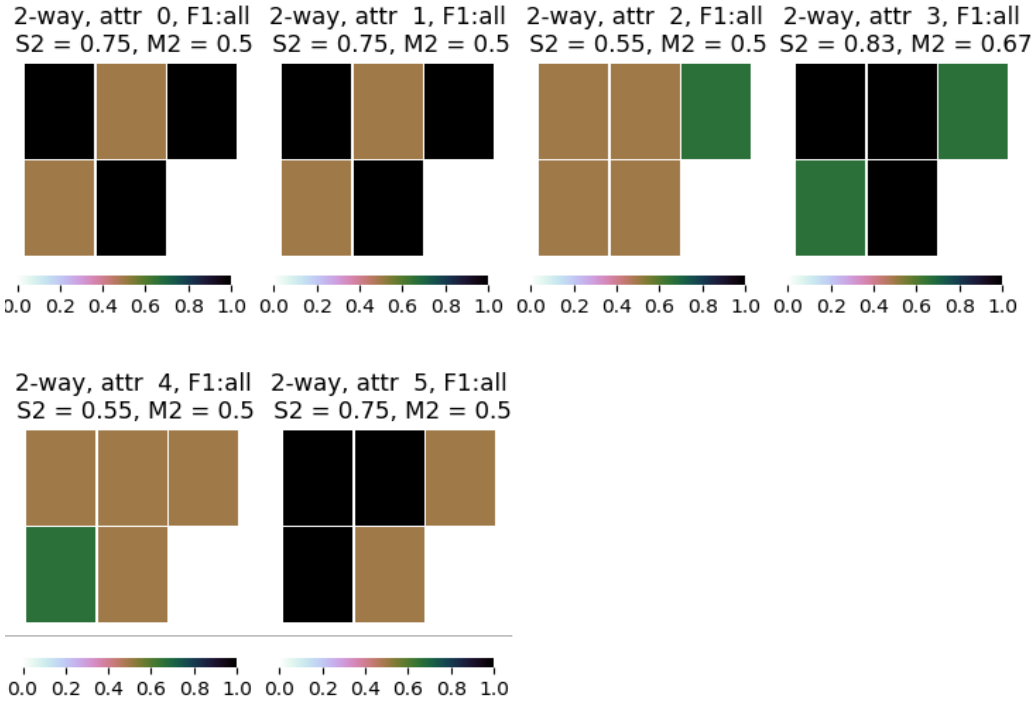


Figure 14: Within-combination coverage

3.4.4 Example

For a real-world example of this visualization approach, see Fig. 15, which shows graphs of 2-way and 3-way combinations in a 22-attribute mushroom identification machine learning problem. The objective of this task is to identify combinations that are strongly or weakly associated with edible or poisonous mushrooms. The left chart shows summary coverage for 231 2-way combinations, and the right chart shows 1,540 3-way combinations.

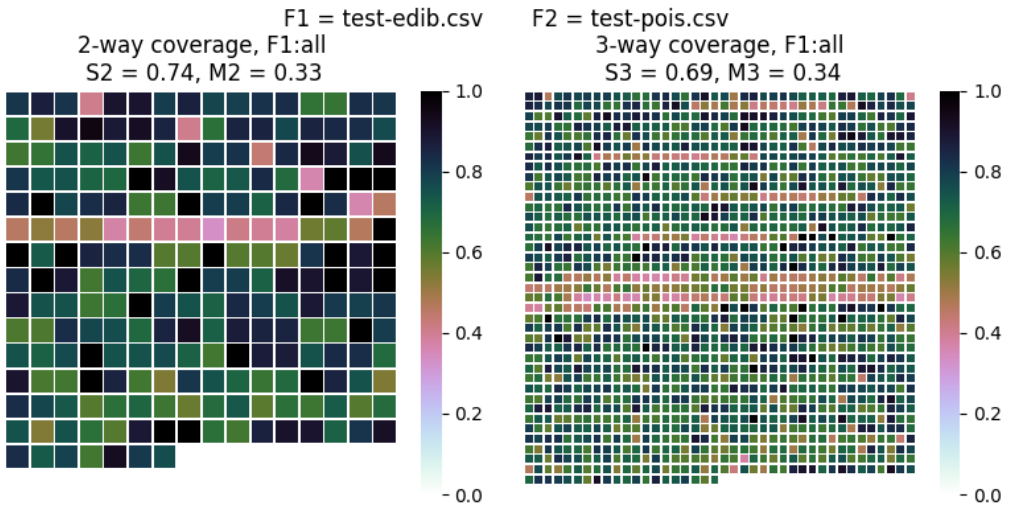


Figure 15: Summary coverage, 2-way and 3-way combinations of 22 attributes

The utility of this type of graph can be seen in Fig. 16, which shows the within-combination coverage of 3-way combinations for each of the 22 attributes in the mushroom identification problem. In this case, attribute 4 has much lower coverage than other attributes. That is, there is much less variability among 3-way combinations containing attribute 4 than for the other attributes. For these combinations, the total coverage, S3, is only 0.51, so roughly half of the value combinations of combinations containing attribute 4 are not associated with this mushroom class. The relevance of this observation depends on the goal of our analysis:

- In a class identification problem, this would suggest that combinations containing this attribute are useful in identifying the mushroom class, since many value combinations do not occur in the edible class (as shown by the low value of S).

- In a testing situation, a low value for S would indicate that this attribute is not well covered in tests, so the test set should be inspected and improved.

If all attributes are covered to about the same level, then the matrices in Fig. 15 will have a fairly uniform color, with coverage level indicated by the heatmap color bar on the right side. For testing, coverage would ideally be at or close to 1.0 for all combinations. That is, all value combinations would be included in tests. In a machine learning context, the attribute coverage heatmaps could be used in comparing model training and test sets, to check that distribution of combination coverage is nearly the same for both sets. Note that in machine learning it is expected that combination coverage varies with attributes, and they will normally not have the same level of value coverage. But if training and test sets are nearly the same, then the heatmaps should be very similar. This approach thus gives us a quick and visual way to identify weak spots in a test set, or confirm similarity between data sets for AI/ML or other applications.

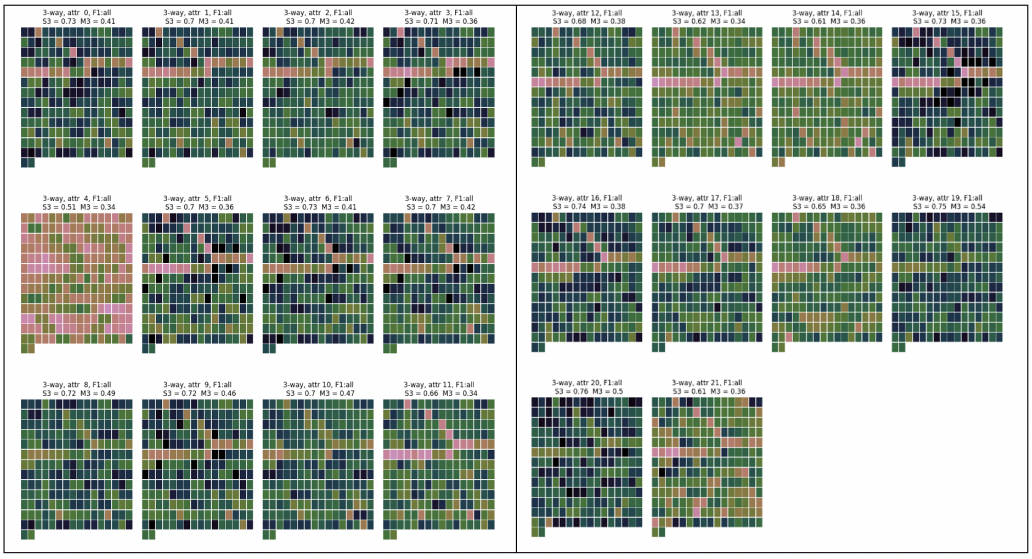


Figure 16: 3-way combination coverage of combinations containing each of 22 attributes

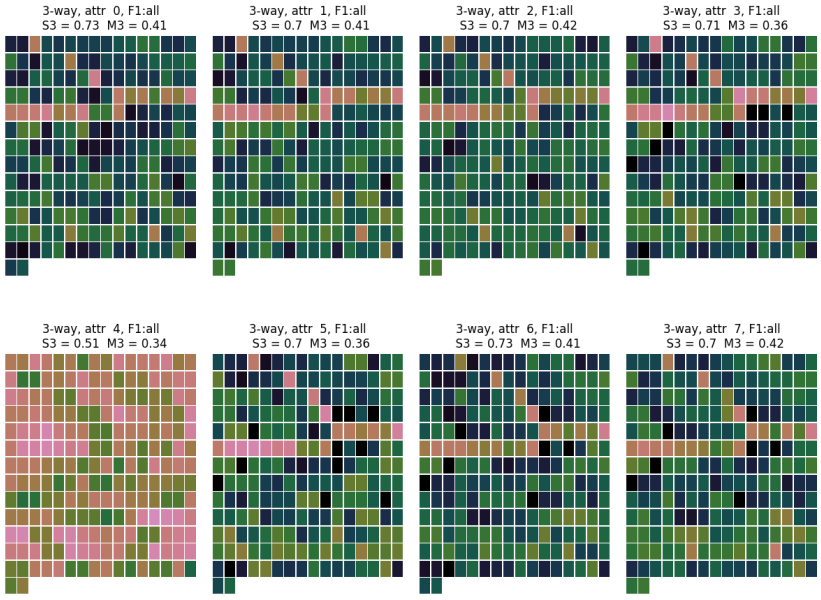


Figure 17: Magnified view of first eight heatmaps

4 Summary

This publication documents a number of measures and ratios that are computed by research tools for use with combinatorial testing. Using t-way combinations of parameter or attribute values provides a new approach for measuring similarity and difference between data sets. Additionally, it is shown that t-way combinations may be used to distinguish between data sets that may appear very similar based only on single-attribute values. These measures have been shown to be useful in test design and evaluation, fault location, explainable artificial intelligence, and transfer learning. The tools described have been introduced as research tools, and additional work may identify other practical application.

References

[1] Eric Schmidt, Bob Work, Safra Catz, Steve Chien, Chris Darby, Kenneth Ford, Jose-Marie Griffiths, Eric Horvitz, Andrew Jassy, William Mark, et al. National security commission on artificial intelligence (ai). Technical report, National Security Commission on Artificial Intellegence, 2021.

[2] E. J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, November 1982.

[3] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[4] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Journal of Software Engineering*, 30(6):418–421, 2004.

[5] D Richard Kuhn and Michael J Reilly. An investigation of the applicability of design of experiments to software testing. In *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*, pages 91–95. IEEE, 2002.

[6] D Richard Kuhn and Vadim Okun. Pseudo-exhaustive testing for software. In *2006 30th Annual IEEE/NASA Software Engineering Workshop*, pages 153–158. IEEE, 2006.

[7] Domenico Cotroneo, Roberto Pietrantuono, Stefano Russo, and Kishor Trivedi. How do bugs surface? a comprehensive study on the characteristics of software bugs manifestation. *Journal of Systems and Software*, 113:27–43, 2016.

[8] Zachary B Ratliff, D Richard Kuhn, Raghu N Kacker, Yu Lei, and Kishor S Trivedi. The relationship between software bug type and number of factors involved in failures. In *2016 IEEE international symposium on software reliability engineering workshops (ISSREW)*, pages 119–124. IEEE, 2016.

[9] Xuelin Li, Ruizhi Gao, W Eric Wong, Chunhui Yang, and Dong Li. Applying combinatorial testing in industrial settings. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 53–60. IEEE, 2016.

[10] D Richard Kuhn, Itzel Dominguez Mendoza, Raghu N Kacker, and Yu Lei. Combinatorial coverage measurement concepts and applications. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 352–361. IEEE, 2013.

[11] D. Richard Kuhn, Itzel Dominguez Mendoza, Raghu N. Kacker, and Yu Lei. Combinatorial coverage measurement concepts and applications. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 352–361. IEEE, 2013.

[12] D Richard Kuhn, Raghu N Kacker, and Yu Lei. Practical combinatorial testing. *NIST special Publication*, 800(142):142, 2010.

[13] D Richard Kuhn, Raghu N Kacker, and Yu Lei. Combinatorial coverage as an aspect of test quality. *CrossTalk*, 28(2):19–23, 2015.

[14] Abhinandan H Patil, Neena Goveas, Krishnan Rangarajan, et al. Test suite design methodology using combinatorial approach for internet of things operating systems. *Journal of Software Engineering and Applications*, 8(07):303, 2015.

[15] Dimitris E Simos, Kristoffer Kleine, Artemios G Voyiatzis, Rick Kuhn, and Raghu Kacker. Tls cipher suites recommendations: A combinatorial coverage measurement approach. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 69–73. IEEE, 2016.

[16] Manuel Leithner, Kristoffer Kleine, and Dimitris E Simos. Cametrics: A tool for advanced combinatorial analysis and measurement of test sets. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 318–327. IEEE, 2018.

[17] Dimitris E Simos, Rick Kuhn, Artemios G Voyiatzis, and Raghu Kacker. Combinatorial methods in security testing. *IEEE Computer*, 49(10):80–83, 2016.

[18] Richard Kuhn and Raghu Kacker. An application of combinatorial methods for explainability in artificial intelligence and machine learning (draft). Technical report, National Institute of Standards and Technology, 2019.

[19] Murat Ozcan. Applications of practical combinatorial testing methods at siemens industry inc., building technologies division. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 208–215. IEEE, 2017.

[20] Miraldi Fifo, Eduard Enoiu, and Wasif Afzal. On measuring combinatorial coverage of manually created test cases for industrial software. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 264–267. IEEE, 2019.

[21] Laleh Sh Ghandehari, Jaganmohan Chandrasekaran, Yu Lei, Raghu Kacker, and D Richard Kuhn. Ben: A combinatorial testing-based fault localization tool. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4. IEEE, 2015.

[22] Laleh Sh Ghandehari, Yu Lei, Raghu Kacker, Richard Kuhn, Tao Xie, and David Kung. A combinatorial testing-based approach to fault localization. *IEEE Transactions on Software Engineering*, 46(6):616–645, 2018.

[23] Charles J Colbourn and Violet R Syrotiuk. On a combinatorial framework for fault characterization. *Mathematics in Computer Science*, 12(4):429–451, 2018.

[24] Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Efficient and guaranteed detection of t-way failure-inducing combinations. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 200–209. IEEE, 2019.

[25] Rekha Jayaram and R Krishnan. Approaches to fault localization in combinatorial testing: A survey. In *Smart Computing and Informatics*, pages 533–540. Springer, 2018.

[26] Dale Blue, Andrew Hicks, Ryan Rawlins, and Rachel Tzoref-Brill. Practical fault localization with combinatorial test design. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 268–271. IEEE, 2019.

[27] Joshua R Maximoff, D Richard Kuhn, Michael D Trela, and Raghu Kacker. A method for analyzing system state-space coverage within a t-wise testing framework. In *2010 IEEE International Systems Conference*, pages 598–603. IEEE, 2010.

[28] Erica Lanus, Laura Freeman, D Richard Kuhn, Raghu N Kacker, and Yu Lei. Combinatorial testing metrics for machine learning. In *2021 IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2021.

Acknowledgments:

Many thanks to...

Disclaimer:

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