Ensuring reliability through combinatorial coverage measures[1]

M S Raunak, D R Kuhn, and R N Kacker
National Institute of Standards and Technology
Gaithersburg, MD 20899

J Y Lei
University of Texas at Arlington
TX 76019, USA

Abstract:

A key question in software assurance is, "How much testing is enough?" Coverage criteria such as statement or branch coverage were developed to help answer this question, but we also need to measure whether tests are sufficiently representative of inputs that will be encountered in the real world. This article describes methods that provide quantitative measures of test input space coverage, to ensure that the system responds correctly and reliably to user interactions and the environmental conditions.

Background

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 and thus *reliable*, one has to show that it produces the "correct" output or "behaves" according to specification without failing under *all* possible parameter values and configurations. 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 for one test execution is called a *Test Case*, and the set of test cases selected for testing a system 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.

Many *Test Adequacy* criteria are based on the 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. For example, it can be shown that branch coverage subsumes statement coverage, i.e., if full branch

[1] This article is adapted from NIST cybersecurity white paper referenced in [1]

coverage is achieved, then full statement coverage is guaranteed. Different test adequacy criteria such as *statement coverage* or *branch coverage* may be applied to achieve some level of coverage of the code structure. 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, and coverage requirements may be tailored according to system risks. For example, 100% coverage of a criterion known as modified condition decision coverage is required for life-critical aviation software. For most commercial software, organizations often set goals for test suite selection such as achieving 85% or 90% statement coverage or branch coverage. This is because achieving 100% coverage may not be feasible within cost and resource constraints due to complex and unreachable code.

Specifying coverage goals in this way helps testers to systematically choose the test cases. The objective 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. The smaller the size of such a test suite, the more efficient it will be in terms of time and computational cost.

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 surely not sufficient, condition for software assurance. This is especially true for such systems that are based on cryptographic algorithms, simulation models, or artificial intelligence and machine learning. Yet structural coverage based test suite selection is still 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 are 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.

In our earlier article [2], we introduced the concept of combinatorial testing, and its effectiveness in building reliable system. In this article, we present *combinatorial coverage* metrics and their use in measuring and ensuring the reliability of software and other systems.

Combinatorial Coverage Measures

Combinatorial testing offers 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 [3-8]. This finding means that testing parameter combinations can provide more efficient fault detection than conventional methods. In this article, we introduce concepts of measuring the combinatorial coverage of an input space [9-11], 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

Fig. 1. Example test array for a system with four binary components

Combinatorial coverage measurement concepts can be illustrated using the example in Fig. 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 Fig. 1, (*b*,*d*) is a 2-way combination, (*a*,*c*) is a different 2-way combination, and (*a*,*c*,*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 [9], this is 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.

Two basic measures are the following [1]:

- *CIPC(n, t)*: Combinatorial Input Parameter Coverage, also known as variable-value configuration coverage [9][10], is the proportion of the total $C(n,t)^i \times v^t$ *t*-way value combinations that are covered by at least one test case. [2] 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, which is 2 in this case.
- *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].

Example. For Fig. 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 configurations are covered and the only ones missing 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 suite, we can compute the configuration coverage for each of the six parameter combinations, as shown in Fig. 2. For this test suite, one of the parameter 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 Fig. 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 a test suite in which parameters have differing numbers of values (example from [3]).

Vars	Configurations	Coverage
a b	00, 01, 10	.75
a c	00, 01, 10	.75

[2] The notation C(n,t) stands for *n choose t*

a d	00, 01, 11	.75
b c	00, 11	.50
b d	00, 01, 10, 11	1.0
c d	00, 01, 10, 11	1.0

*total 2-way cove*rage = 19/24 = 0.79167
(.50, 2)-completeness = 6/6 = 1.0
(.75, 2)-completeness = 5/6 = 0.83333
(1.0, 2)-completeness = 2/6 = 0.33333

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

To make the data in Fig. 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 Fig. 3. If the list of configurations (column 2) is sorted in descending order, then rotated, a graph as seen in Fig. 4 results. We also define two additional measures:

- $\mathbf{M_t}$ = the *minimum* t-way coverage, i.e., the proportion of coverage for which *(p, t)-completeness* = 1.0
- $\mathbf{\Phi_t}$ = the proportion of full t-way combinations for which (1.0, t)-completeness is achieved.

b d	00, 01, 10, 11
c d	00, 01, 10, 11
a b	00, 01, 10
a c	00, 01, 10
a d	00, 01, 11
b c	00, 11

b d	c d	a b	a c	a d	b c
00, 01, 10, 11	00, 01, 10, 11	00, 01, 10	00, 01, 10	00, 01, 11	00, 11

Fig. 3. Table mapped to graph.

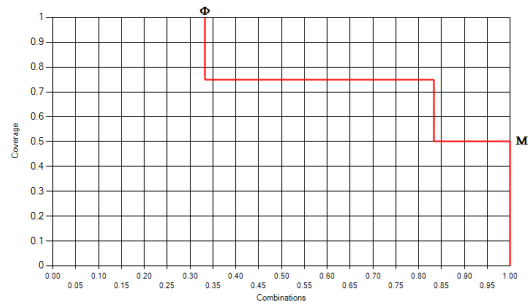

Fig. 4. Example coverage graph for t = 2.

Graphing the quantities M and Φ provides an intuitive approach to understanding combinatorial coverage. The point M in this graph is particularly important. If M is low, then there is at least one combination that is not covered well, i.e., has not been tested for the system. This means there is no guarantee that the SUT will process the omitted input combination correctly.

To see the significance of this measurement, consider Fig. 5. The combinations covered, represented by the area below the curve, are shown in green. Combinations that have not been included in any test are those above the curve, shown in orange. 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 orange 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.

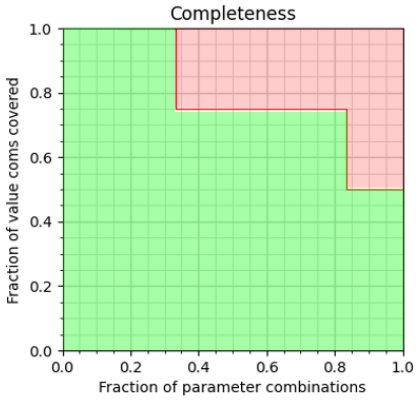

Fig. 5. Covered and non-covered regions of the input space.

```
Summary coverage report,   file = Stereo7489x82.csv
Input model configuration:
2^5 4^2 6^2 6^2

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
```

Fig. 6. Combinatorial coverage of spacecraft software

Application of Combinatorial Coverage Measures

Spacecraft software is subject to extensive testing prior to deployment. The combinatorial coverage measures and tool built on this idea were initially applied to a large test suite developed software for a NASA mission [12]. The test suite contained more than 8,500 tests, so strong *t*-way coverage was expected to be present, but there were no established methods and tools 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 Fig. 6), but even 2-way combinations were not fully covered, and many additional tests would have been needed for 3-way or higher strength.

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 check functions more thoroughly. 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 [15][16]. CCM has been helpful in evaluating the thoroughness of tests for secure protocols and other security-critical applications [17][18]. 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 measuring assurance for this type of analytics product, and results showed the combinatorial coverage measurement was effective [19].
- Siemens included CCM in its Zero Defect program to measure aspects of a test suite 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." [16]

- 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 only 13% for 6-way interactions. The measures were used to determine improvements to test suite for critical code [1].

DISCUSSION

Exhaustively testing a system with all possible inputs and configuration settings to ensure its reliability is nearly always an infeasible endeavor. System developers therefore rely on methodologies to select a small subset of tests that are capable of discovering any potential bugs in the system. Measuring the fault detecting capability of a test suite, however, is challenging. Combinatorial coverage measure allows us to systematically look at the t-way combinations of variable values that are present in the test suite and the ones that are absent. It provides both a numeric as well as visual representation of the strength of the test suite. Test engineers can then strengthen the test suite based on this coverage measures. Our CCM method and tools have been effectively used in large organizations to not only measure the strength of their test suite, but also to improve their fault discovering capability and thus to improve reliability of their systems. This demonstrates that combinatorial measures presented here are an effective and useful way to understand and improve the reliability of software systems.

Structural coverage based adequacy criterion such as statement or branch coverage may not be suitable for fault discovery in such systems as the ones based on cryptography, simulation models, or artificial intelligence and machine learning algorithms [20][21]. They are often tested with randomly selected set of test cases, where no measurement of coverage is measured leading towards not having any quantification of the strength of the test suite. The situation is the same with Fuzz testing, which is also common in many testing scenarios. Alternatively, Combinatorial Coverage Measurements (CCM) provide an effective set of metrics and tools to gauge the coverage of the input space. Studies have shown that this approach not only provides a quantitative measure of a particular test suite's strength of discovering faults, it also helps to identify where the gaps are present in testing. Thus, CCM methods lead towards strengthening the overall test suite, which in turn, improves the reliability of the system.

One of the challenges with combinatorial coverage based methods is the need for discretization of the variable values. However, such discretization is also needed in traditional testing practices while constructing a test suite. Another challenge is the test suite getting too large for systems with a very large number of input variables. The parameter explosion problem is not unique to combinatorial methods, it is present in all testing. On the other hand, the logarithmic relationship between the number of input variables and the number of test cases in combinatorial test suite makes it attractive. The strongest benefit is of course derived from the intuitive and quantitative measure of the input space being covered by the selected test cases. As presented in our preceding article [2], with t-way coverage (t=1..6), one can achieve very high reliability of a system.

DISCLAIMER

U.S. Government work. Not subject to U.S. copyright. Commercial products may be identified in this document, but such identification does not imply recommendation or endorsement by NIST, nor that the products identified are necessarily the best available for the purpose.

¢ REFERENCES

1. D R Kuhn, M S Raunak, R N Kacker "Combinatorial Coverage Difference Measurement", *CSRC white paper*, published on June 22, 2021, National Institute of Standards and Technology (NIST), https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.06222021-draft.pdf
2. M S Raunak, D R Kuhn, R N Kacker, J Y Lei, "Combinatorial Testing for Building Reliable Systems", *IEEE Reliability Magazine*, vol. 1, no. 1, Mar 2024.

3. D R Kuhn, D R Wallace, A.M. Gallo, Jr., "Software Fault Interactions and Implications for Software Testing", *IEEE Transactions on Software Engineering*, vol. 30, no. 6, June 2004, pp. 418-421.
4. D R Kuhn and M J Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing", *27th Annual NASA Goddard/IEEE Software Engineering Workshop (SEW '02)*, Greenbelt, Maryland, Dec 5-6, 2002, pp. 91-95
5. D R Kuhn, V Okun, "Pseudo-exhaustive Testing for Software", *30th Annual IEEE/NASA Software Engineering Workshop (SEW-30)*, Columbia, Maryland, April 24-28, 2006, pp. 153-158
6. D Cotroneo, R Pietrantuono, S Russo, & K Trivedi (2016). "How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation", *Journal of Systems and Software (JSS)*, *113*, pp. 27-43, 2016.
7. Z Ratliff, D R Kuhn, R N Kacker, Y.Lei, K Trivedi, "The Relationship Between Software Bug Type and Number of Factors Involved in Failures", *Intl Workshop on Combinatorial Testing* (ICST), 2016.
8. X Li, R Gao, W E Wong, C Yang, and D Li, "Applying combinatorial testing in industrial settings." *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)* (pp. 53-60).
9. D M Cohen, S R Dalal, J Parelius, and G C Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, 9 1996
10. D R Kuhn, I D Mendoza, R N. Kacker, and Y Lei. "Combinatorial coverage measurement concepts and applications", *IEEE 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2013, pages 352–361.
11. D R Kuhn, R N Kacker, and Y Lei. "Practical combinatorial testing", *NIST special Publication, 800 142*, 2010.
12. D R Kuhn, R N Kacker, and Y Lei. "Combinatorial coverage as an aspect of test quality.", *CrossTalk*, vol. 28, no. 2, pg. 19–23, 2015.
13. A H Patil, N Goveas, K Rangarajan, et al. "Test suite design methodology using combinatorial approach for internet of things operating systems", *Journal of Software Engineering and Applications*, vol. 8, no. 7, pg 303, 2015.
14. D E Simos, Kr Kleine, A G Voyiatzis, D R Kuhn, and R N Kacker. "TLS cipher suites recommendations: a combinatorial coverage measurement approach", *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pg 69–73, 2016.
15. M Leithner, K Kleine, and D E Simos, "Cametrics: A tool for advanced combinatorial analysis and measurement of test sets", *IEEE International Conference on Software Testing, Verification and Validation Workshops* (ICSTW), pages 318–327. IEEE, 2018.
16. D E Simos, D R Kuhn, A G Voyiatzis, and R N Kacker. "Combinatorial methods in security testing". *IEEE Computer*, vol. 49 no. 10:80–83, 2016.
17. D R Kuhn and R N Kacker, "An application of combinatorial methods for explainability in artificial intelligence and machine learning", *Technical report, National Institute of Standards and Technology*, 2019.
18. M Ozcan. "Applications of practical combinatorial testing methods at siemens industry inc., building technologies division", *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pg 208–215, 2017.
19. M Fifo, E Enoiu, and W 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.
20. D R Kuhn, M S Raunak, R N Kacker, "Cominatorial Coverage for Assured Autonomy", *IEEE International Symposium on Software Reliability Engineering Workshops* (ISSREW), 2022.
21. J R Toohey, M S Raunak, D W Binkley, "From neuron coverage to steering angle: Testing autonomous vehicles effective", IEEE Computer, vol. 54, no. 8, pg 77-85, Aug. 2021.

M S Raunak is a computer scientist at the National Institute of Standards and Technology. He has Ph.D. in computer science from University of Massachusetts Amherst. His research interest includes verification, validation, and assurance of "difficult-to-test" systems. Dr. Raunak is a senior member of IEEE. Contact him/her at raunak@nist.gov

D R Kuhn is a computer scientist at the National Institute of Standards and Technology. He is one of the pioneering researchers in combinatorial testing. Mr. Kuhn received an M.S. in computer science from the Univeristy of Maryland College Park and is a Fellow of the IEEE. Contact him at kuhn@nist.gov.

R N Kacker is a Mathematician at the National Institute of Standards and Technology, and one of the pioneering researchers in combinatorial testing. Contact him at raghu.kacker@nist.gov.

Jeff Y Lei is a professor of Computer Science and Engineering at the University of Texas Arlington. He received his Master's degree from Institute of Software, Chinese Academy of Sciences, and his Ph.D. from North Carolina State University. His research is in the area of automated software analysis, testing and verification, with a current focus on building trustworthy AI-based software systems. contact him at ylei@cse.uta.edu

[i] C(n,t) notation stands for *n choose t*