COMBINATORIAL TESTING

D. Richard Kuhn*, National Institute of Standards and Technology, kuhn@nist.gov

Raghu N. Kacker, National Institute of Standards and Technology, raghu.kacker@nist.gov

Yu Lei, University of Texas Arlington, ylei@uta.edu

Keywords: combinatorial testing; covering arrays; design of experiments; pairwise testing; pseudo-exhaustive testing; software assurance; software testing; verification;

ABSTRACT

Combinatorial testing is a method that can reduce cost and improve test effectiveness significantly for many applications. The key insight underlying this form of testing is that not every parameter contributes to every failure, and empirical data suggest that nearly all software failures are caused by interactions between relatively few parameters. This finding has important implications for testing because it suggests that testing combinations of parameters can provide highly effective fault detection. This article introduces combinatorial testing and how it evolved from statistical Design of Experiments approaches, explains its mathematical basis, where this approach can be used in software testing, and measurements of combinatorial coverage for existing test data.

INTRODUCTION

Recognizing that system failures can result from the interaction of conditions that might be innocuous individually, software developers have long used "pairwise testing", in which all possible pairs of parameter values are covered by at least one test. However, many faults will be triggered only by an unusual combinatorial interaction of more than two parameters. A medical device study found one case in which a failure involved a four-way interaction between parameter values (1). Subsequent investigations (2, 3, 4, 5) found a similar distribution of failure-triggering conditions: usually, many were caused by a single parameter value, a smaller proportion resulted from an interaction between two parameter values, and progressively fewer were triggered by 3, 4, 5, and 6-way interactions. These results suggest that combinatorial testing which exercises high degree (4-way or above) combinatorial interactions is necessary to reach a higher level of software assurance. If we know from experience that t or fewer variables are involved in failures for a particular application type and we can test all t-way combinations of discrete variable settings, then we can have reasonably high confidence that the application will function correctly.

The key ingredient for this form of testing is known as a covering array, a mathematical object in which all t-way combinations of parameter values are covered at least once (6, 7, 8). Generating covering arrays for complex interactions (beyond pairwise) is a difficult problem, but algorithms have been developed that make it possible to generate covering arrays several orders of magnitude faster than previous methods, making up to 6-way covering arrays tractable for many applications. While the most basic form of combinatorial testing – pairwise – is well developed, it is only in the past few years that efficient algorithms for complex covering arrays – for up to 6-way coverage – have become available. New algorithms, coupled with fast, inexpensive processors, are making sophisticated combinatorial testing a practical approach that may hold the promise of better software testing at a lower cost. This article reviews the empirical and mathematical basis for combinatorial testing and explains its benefits and tradeoffs in practice. The first section reviews the development of combinatorial testing from the statistical field of Design of Experiments (DoE), differences with traditional DoE, and the empirical basis for the application of combinatorial methods to software testing. The second section reviews practical use of these methods and their application in different types of testing problems. A concluding section discusses recent developments and some active research areas.

THE DEVELOPMENT OF COMBINATORIAL SOFTWARE TESTING

Combinatorial testing for software has its roots in the statistical field of design of experiments (DoE). DoE is a statistical methodology for conducting controlled experiments in which a system is exercised with carefully chosen parameter values that allow the calculation of useful statistical information about the relationship between input factors and a parameter of interest known as the response variable. Underlying every DoE plan is an assumed statistical model for the relationship between input factors and the response variable. Generally the purpose of DoE is to improve the performance of the same or similar systems. DoE is most commonly used for systems such as agricultural production, medical treatments, and industrial manufacturing which are subject to uncontrolled variables. Frequently, the effects of many test factors each having multiple test values are investigated at the same time, so combinatorial mathematics is often used in planning DoE.

DESIGN OF EXPERIMENTS

The body of knowledge we call DoE today is based largely on the 1920s works of Fisher and Yates (9) for agricultural experiments. In agriculture, the test factors are treatments (such as fertilizer types and amounts) which are applied to experimental units (such as plots of land) and the yield of crop (response variable) is measured. Frequently the yield from different experimental units can be quite different and since statistical conclusions are often needed for conditions where some factors in the experiment cannot be controlled. Assignment of treatments to experimental units using a mechanical randomization device made it possible to regard the effects of uncontrolled factors as random error. Thus random assignment of treatments to the experimental units made it possible to use statistical methods to interpret the data from DoE. The types of DoE plans used in agriculture were randomized block designs, balanced incomplete block designs, Latin square designs, and their extensions.

In the 1940s and 1950s, DoE methods were adapted to improve performance of industrial processes and pharmaceutical development. The factors were process variables which could be classified as primary test factors, background factors, factors held constant, and uncontrolled factors. In the 1960s, Taguchi investigated and promoted the use of DoE methods to develop robust products and manufacturing processes. Taguchi determined that to develop high quality products it was necessary to determine product parameters at which the variation in product performance from uncontrollable factors (such as environmental factors, deterioration, and manufacturing variations) was as small as possible. Taguchi used DoE to improve the main function of a product (by amplifying effect of signal factor and attenuating effect of noise factors). He promoted the use of mathematical objects called orthogonal arrays (OAs) for his adaptations of DoE methods, and his ideas became popular among manufacturers.

ORTHOGONAL ARRAYS

An orthogonal array, $OA_\lambda(N;t, k, v)$ is an N x k array. In every N x t subarray, each t-tuple occurs exactly λ times. We refer to t as the strength of the coverage of interactions, k as the number of parameters or components (degree), and v as the number of possible values for each parameter or component (order).

Example. Suppose we have a system with three on-off switches, controlled by an embedded processor. The following table tests all pairs of switch settings exactly once each. Thus $t = 2$, $\lambda = 1$, $v = 2$. Note that there are $v^t = 2^2$ possible combinations of values for each pair: 00, 01, 10, 11. There are $\binom{3}{2} = 3$ ways to select switch pairs: (1,2), (1,3), and (2,3), and each test covers three pairs, so the four tests shown in Table 1 cover a total of 12 combinations, which implies that each combination is covered exactly once. As one might suspect, it can be very challenging to fit all combinations to be covered into a set of tests exactly the same number of times.

Test	**Sw 1**	**Sw 2**	**Sw 3**
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Table 1. An orthogonal array with three factors of two levels each

The benefits of DoE based on orthogonal arrays are (1) the main effects of different factors are not confounded with each other and are uncorrelated, (2) the main effects have higher precision (smaller variance) relative to effects evaluated from one-factor-at-a-time DoE, and (3) each main effect is the average effect for all test values of the other test factors. Two factors are said to interact if their effects are not additive. A two-factor interaction effect is a measure of how the main effect of one factor changes with the value of another factor. In the late1980s, with the advent of computer and communication systems with embedded software, the problem of testing software and hardware-software systems became important. Some researchers who were familiar with DoE started to investigate the use of OAs for testing software and systems.

Initial applications of OAs used orthogonal arrays of strength 2 to test computer and communication systems, a methodology known as pairwise testing. The objective in pairwise testing is to exercise all possible pairs of test values and check whether the software or the system under test behaves as expected. A test suite based on an OA of strength 2 satisfies the criterion of pairwise testing that for every pair of test factors, all possible pairs of the test values are exercised. Advantages of test suites based on OAs can be stated in relation to exhaustive testing. A test suite of strength 2 for four binary variables and one 4-value variable, i.e., the combinatorial test structure $2^4 \times 4^1$, based on OA(8, $2^4 \times 4^1$, 2), would require 8 test cases while exhaustive testing would require 64 test cases.

Two limitations of test suites based on OAs for pairwise testing are that an OA matching the required combinatorial test structure may not exist, and the test cases determined using OAs frequently contain invalid pairs of test values. For example, suppose the required combinatorial test structure for five test factors is $2^4 \times 3^1$(four factors have 2 test values each and one has three test values). It can be shown that OA of strength 2 matching the combinatorial test structure $2^4 \times 3^1$ does not exist. In such cases a suitable OA is modified to fit the need. The array in Table 2 is not an orthogonal array but it covers all pairs of test settings and it can be used to construct a pairwise testing suite for the combinatorial test structure $2^4 \times 3^1$.

Suppose the five factors were (1) operating system (OS) with test values {XP, Linux}, (2) browser with test values {Internet Explorer (IE), Firefox}, (3) protocol with test values {IPv4, IPv6}, (4) CPU type with test values {Intel, AMD}, and (5) database management system (DBMS) with test values {MySQL, Sybase, Oracle}. Then the corresponding test suite based on Table 2 is shown in Table 3.

Table 2: Array for combinatorial test structure $2^4 \times 3^1$

	1	2	3	4	5
1	0	0	0	0	0
2	1	1	1	1	0
3	0	0	1	1	1
4	1	1	0	0	1
5	0	1	0	1	2
6	1	0	1	0	2
7	0	1	1	0	2
8	1	0	0	1	2

Table 3: Pairwise testing suite based on Table 2

Test cases	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	Linux	Firefox	IPv6	AMD	MySQL
3	XP	IE	IPv6	AMD	Sybase
4	Linux	Firefox	IPv4	Intel	Sybase
5	XP	Firefox	IPv4	AMD	Oracle
6	Linux	IE	IPv6	Intel	Oracle
7	XP	Firefox	IPv6	Intel	Oracle
8	Linux	IE	IPv4	AMD	Oracle

Since the browser Internet Explorer (IE) does not run on the operating system Linux, the pair {Linux, IE} appearing in test cases 6 and 8 are invalid; therefore, these two test cases are not legitimate. If the test case 6 and 8 were deleted from the test suite then we will lose other pairs covered by these two test cases, such as the pairs {AMD, Oracle} and {Intel, Oracle}. Then the test suite shown in Table 4 would not cover all valid pairs of test values. This is an example of a constraint among test values. Constraints are an important consideration in covering array generation, discussed later in this article.

COVERING ARRAYS

Since OAs do not exist for many combinatorial test structures, and since test suites based on OAs may contain invalid test cases (see Tbl. 3), researchers in the early 1990s (10) developed methods to generate pairwise test suites which covered all valid pairs of test values in as-few-as-possible test cases. In 1993, Sloan (11) formally defined the concept of covering arrays as follows. A fixed-value covering array denoted by $CA(N, v^k, t)$ is an N x k matrix of entries from the set {0, 1, …, (v – 1)}such that that every set of t-columns contains each possible t-tuple of entries at least once. A fixed value covering array is also denoted by CA(N, k, v, t). A mixed-value covering array denoted by $CA(N,v_1^{k1}v_2^{k2}\ldots v_n^{kn}, t)$ is an extension of fixed value CA where k = k1 + k2 +… + kn; k1 columns have v_1 distinct values, k2 columns have v_2 distinct values, ... , and kn columns have v_n distinct values.

All orthogonal arrays are covering arrays but not all covering array are orthogonal arrays. It can also be shown that given any orthogonal array, a covering array of equal or fewer rows exists that covers the same combinations. Consider the covering array in Table 4, which includes all pairwise combinations of the five variables shown in the orthogonal array in Table 3. In this case, only six rows are needed to cover all 2-way combinations, rather than eight. Some 2-way combinations in the CA are covered more than once, rather than exactly one each as in the OA, but for software testing this does not matter if we are only trying to determine whether the output is correct for each set of inputs.

Table 4. Covering array for combinatorial test structure $2^4 \times 3^1$

	1	2	3	4	5
1	0	0	0	0	0
2	1	1	1	1	0
3	0	1	0	1	1
4	1	0	1	0	1
5	0	0	1	1	2
6	1	1	0	0	2

The connection between pairwise testing and covering arrays was made by Dalal and Mallows (7) who observed that DoE plans based on OAs enable evaluations of the main effect of each test factor (the average effect for all test values of the other test factors). Evaluation of main effects is important in DoE. In pairwise testing for software there is no need to evaluate main effects of test factors such as, for example, the effect of different levels of fertilizer and water on crop yield. Instead interest lies in covering all pairs of test values to determine if the software responds correctly to inputs. Thus CAs rather than OAs are better suited for pairwise testing of software and systems.

For software failures triggered by a single parameter value or interactions between parameters, pairwise testing can be effective. For example, a router may be observed to fail only for a particular protocol when packet volume exceeds a certain rate, a 2-way interaction between protocol type and packet rate. Figure 1 illustrates how such a 2-way interaction may happen in code. Note that the failure will only be triggered when both pressure < 10 and volume > 300 are true.

```
if (pressure < 10) {
        // do something
        if (volume > 300)  {
                faulty code!  BOOM!
        }
        else {
                good code, no problem
        }
}
else {
        // do something else
}
```

Figure 1. 2-way interaction failures are triggered when two conditions are true.

Limitations of Pairwise Testing

What if some failure is triggered only by a very unusual combination of 3, 4, or more sensor values? It is very unlikely that pairwise tests would detect this unusual case; we would need to test 3-way and 4-way combinations of values. But is testing all 4-way combinations enough to detect all errors? What degree of interaction occurs in real failures in real systems? NIST studies beginning in 1999 showed that across a variety of domains, all failures could be triggered by a maximum of 4-way to 6-way interactions (1, 2, 3, 4). As shown in Figure 2, the failure detection rate increased rapidly with interaction strength (the interaction level t in t-way combinations is often referred to as strength). With the NASA application, for example, 67% of the failures were triggered by only a single parameter value, 93% by 2-way combinations, and 98% by 3-way combinations. The detection rate curves for the other applications studied are similar, reaching 100% detection with 4 to 6-way interactions. Studies by other researchers (5) have been consistent with these results.

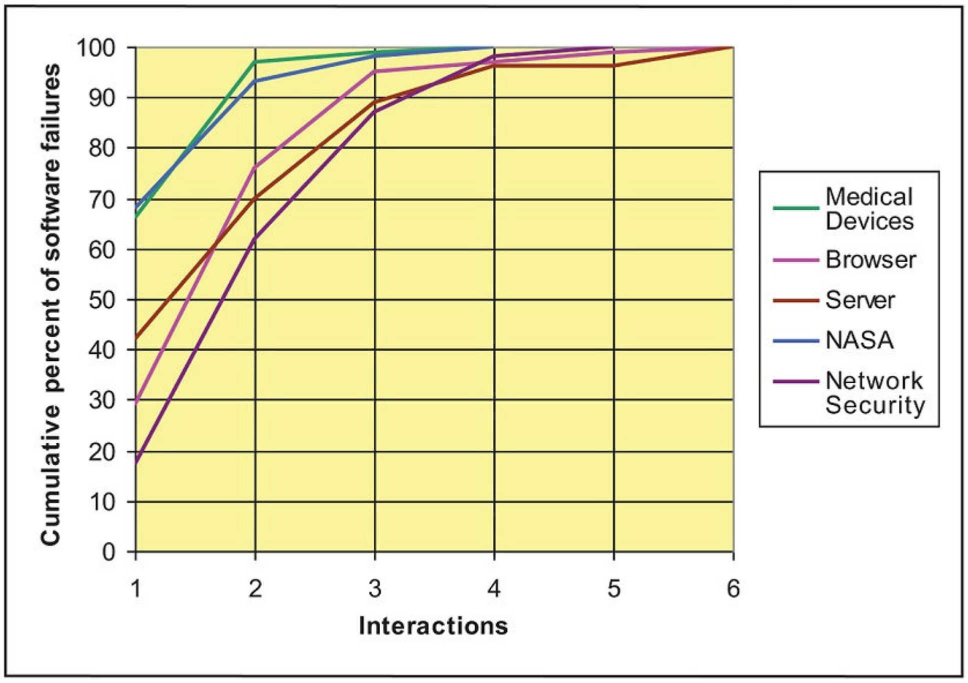


Figure 2. Most failures are triggered by one or two parameters interacting, with progressively fewer by 3, 4, or more.

Software Failures and the Interaction Rule

These results are interesting because they suggest that, while pairwise testing is not sufficient, the degree of interaction involved in failures is relatively low. This result is summarized in an empirical observation called the Interaction Rule: most failures are caused by one or two parameters interacting, with progressively fewer by 3, 4, or more parameter interactions.

Testing all 4-way to 6-way combinations may therefore provide reasonably high assurance. As with most issues in software, however, the situation is not that simple. Efficient generation of test suites to cover all t-way combinations is a difficult mathematical problem that has been studied for nearly a century. In addition, most parameters are continuous variables which have possible values in a very large range (+/- 2^{32} or more). These values must be discretized to a few distinct values. Most glaring of all is the problem of determining the correct result that should be expected from the system under test for each set of test inputs. Generating 1,000 test data inputs is of little help if we cannot determine what the system under test (SUT) should produce as output for each of the 1,000 tests.

With the exception of combination covering test generation, these challenges are common to all types of software testing, and a variety of good techniques have been developed for dealing with them. What has made combinatorial testing practical today is the development of efficient algorithms to generate tests covering t-way combinations, and effective methods of integrating the tests produced into the testing process.

Algorithms for High-strength Covering Arrays

A variety of approaches for covering array construction algorithms have been developed, but most can be categorized as algebraic, search-based, or greedy. Algebraic methods produce compact covering arrays very quickly for certain classes of problems (8). Their disadvantage is that they are not practical across a broad range of problems. Search-based approaches include simulated annealing, genetic algorithms, ant-colony optimization, and other methods that iteratively improve a candidate solution. They can produce highly optimized results, i.e., small test suites, but run time is often excessive. One of the most successful search-based methods for covering array construction is simulated annealing (11).

Greedy algorithms have been used for problems in many domains, and their general framework applies well to combinatorial testing (12, 13) A greedy algorithm constructs a set of possible solution candidates, selects the best one according to some metric, then repeats until the problem is solved. To construct a covering array for testing, the greedy approach generates a large number of candidates, selects the one that covers the most previously uncovered combinations, and continues until all combinations have been covered by the test suite. Greedy algorithms are widely used because they tend to be efficient in terms of run time, and generate small covering arrays. It can be shown that the number of tests produced by a greedy algorithm is proportional to $v^t \log n$, where v = number of discrete variable values, t = interaction strength, and n = number of variables (14).

COMBINATORIAL TESTING IN PRACTICE

There are basically two approaches to combinatorial testing – use combinations of configuration parameter values, or combinations of input parameter values. In the first case, we select combinations of values of configurable parameters. For example, a server might be tested by setting up all 4-way combinations of configuration parameters such as number of simultaneous connections allowed, memory, OS, database size, etc., with the same test suite run against each configuration. The tests may have been constructed using any methodology, not necessarily combinatorial coverage. The combinatorial aspect of this approach is in achieving combinatorial coverage of configuration parameter values.

In the second approach, we select combinations of input data values, which then become part of complete test cases, creating a test suite for the application. In this case combinatorial coverage of input data values is required for tests constructed. A typical ad hoc approach to testing involves subject matter experts setting up use scenarios, then selecting input values to exercise the application in each scenario, possibly supplementing these tests with unusual or suspected problem cases. In the combinatorial approach to input data selection, a test data generation tool is used to cover all combinations of input values up to some specified limit.

CONFIGURATION TESTING

Many, if not most, software systems have a large number of configuration parameters. Many of the earliest applications of combinatorial testing were in testing all pairs of system configurations. For example, telecommunications software may be configured to work with different types of call (local, long distance, international), billing (caller, phone card, 800), access (ISDN, VOIP, PBX), and server for billing (Windows Server, Linux/MySQL, Oracle). The software must work correctly with all combinations of these, so a single test suite could be applied to all pairwise combinations of these four major configuration items. Any system with a variety of configuration options is a suitable candidate for this type of testing.

For example, suppose we had an application that is intended to run on a variety of platforms comprised of five components: an operating system (Windows XP, Apple OS X, Red Hat Enterprise Linux), a browser (Internet Explorer, Firefox), protocol stack (IPv4, IPv6), a processor (Intel, AMD), and a database (MySQL, Sybase, Oracle), a total of 3x2x2x2x2 = 48 possible platforms. With only 10 tests, shown in Table 5, it is possible to test every component interacting with every other component at least once, i.e., all possible pairs of platform components are covered.

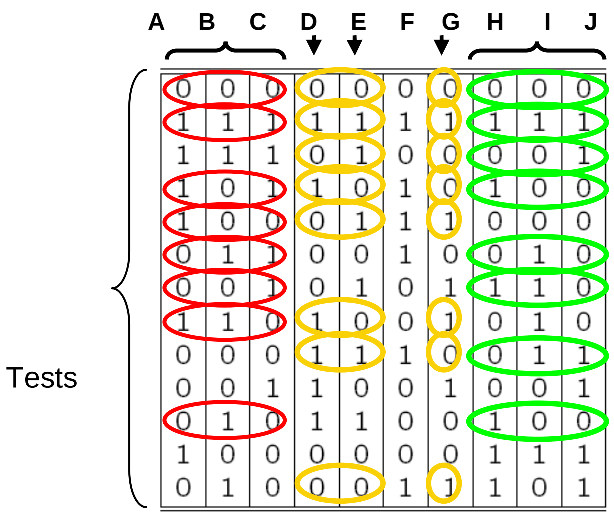
Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHEL	IE	IPv6	AMD	MySQL
8	RHEL	Firefox	IPv4	Intel	Sybase
9	RHEL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

Table 5. Pairwise test configurations

INPUT TESTING

Even if an application has no configuration options, some form of input will be processed. For example, a word processing application may allow the user to select 10 ways to modify some highlighted text: subscript, superscript, underline, bold, italic, strikethrough, emboss, shadow, small caps, or all caps. The font-processing function within the application that receives these settings as input must process the input and modify the text on the screen correctly. Most options can be combined, such as bold and small caps, but some are incompatible, such as subscript and superscript.

Thorough testing requires that the font-processing function work correctly for all valid combinations of these input settings. But with 10 binary inputs, there are 2^{10} = 1,024 possible combinations. But the empirical analysis reported above shows that failures appear to involve a small number of parameters, and that testing all 3-way combinations may detect 90% or more of bugs. For a word processing application, testing that detects better than 90% of bugs may be a cost-effective choice, but we need to ensure that all 3-way combinations of values are tested. To do this, we create a covering array of strength 3, then map the values to test inputs, as shown in Figure 3.



A	B	C	D	E	F	G	H	I	J
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Figure 3. A 3-way covering array includes all 3-way combinations of values.

In Fig. 3, each column gives values for the 10 parameters, labeled A through J. Each row is a test. Note that for any three columns selected in any order, all eight possible combinations of three binary values can be found, 000, 001, 010, etc. Thus the table in Fig. 3 is a covering array of strength 3 for 10 binary variables.

Input Variables vs. Test Parameters

In the example above, we assumed that the parameters to be included in tests were taken from the set of several inputs to some function in the program, where each parameter had defined values or a range of values, so a covering array can be computed where each column covers values for an input variable. In many testing problems, however, there may be only one or two inputs. This common situation can be illustrated with the example (15) of a "find" command, which takes user input of a string and a file name and locates all lines containing the string. The format of the command is "find <string> <filename>, where <string> is one or more quoted strings of characters such as "john", "john smith", or "john" "smith". Search strings may include the escape character (backslash) for quotes, to select strings with embedded quotes in the file, such as "\"john\"" to report the presence of lines containing john in quotes within the file. The command displays any lines containing one or more of the strings. This command has only two input variables, string and filename, so is combinatorial testing really useful here?

In fact, combinatorial methods can be highly effective for this common testing problem. To check the "find" command, testers will want to ensure that it handles inputs correctly. The input variables in this case are string and filename, but it is common to refer to such variables as parameters. We will distinguish between the two here, but follow conventional practice where the distinction is clear. The test parameters identify characteristics of the command input variables. So the test parameters are in this case different from the two input variables, string and filename. For example, the string input has characteristics such as length, presence of embedded blanks, etc. Clearly, there are many ways to select test parameters, so judgment must be used to determine what are most important. One selection could be the following, where file_length is the length in characters of the file being searched:

String length: {0, 1, 1..file_length, >file_length}

Quotes: {yes, no, improperly formatted quotes}

Blanks: {0, 1, >1}

Embedded quotes: {0, 1, 1 escaped, 1 not escaped}

Filename: {valid, invalid}

Strings in command line: {0, 1, >1}

String presence in file: {0, 1, >1}

For these seven test parameters, we have 4x3x3x4x2x3x3 = 2,592 possible combinations of test parameter values. If we choose to test all 2-way interactions we need only 19 tests. For 3 and 4-way combinations, we need only 67 and 218 tests respectively.

Constraints

In general, there are two types of constraints, environment constraints and system constraints. Environment constraints are imposed by the runtime environment of the system under test (SUT). In the example introduced earlier, when we test a web application to ensure that it works in different operating systems and browsers, combinations of Linux and IE do not occur in practice and cannot be tested. In general, combinations that violate environment constraints could never occur at runtime and thus must be excluded during test generation.

System constraints are imposed by the semantics of the SUT. For example, in a credit card application system, the income of an applicant must be a positive number. Invalid combinations that do not satisfy system constraints may be rendered to the SUT at runtime. When this occurs, these combinations should be properly rejected by the SUT. Therefore, it is important to test these combinations for the purpose of robustness testing, i.e., making sure that the SUT is robust when invalid combinations are presented. Note that in order to avoid potential mask effects, robustness testing often requires that each test contain only one invalid combination.

There are two general approaches to handling constraints. The first approach is to transform the input model without changing the test generation algorithm. For example, assume that there exists a constraint, a > b, between two parameters a and b. A new input model can be created by replacing these two parameters with a new parameter c whose domain consists of all the combinations of parameters a and b that satisfy this constraint. A test generation algorithm that does not support constraints can be applied to this new model to create a combinatorial test set.

The second approach is to modify the test generation algorithm such that constraints are handled properly during the actual test generation process. For example, many algorithms are greedy algorithms that typically create a test by choosing, from a pool of candidates, one that covers the most number of uncovered tests. Such algorithms can be modified such that they require each candidate in the pool satisfy all the constraints. Compared to the first approach, this approach often produces a smaller test set, but at the cost of more execution time.

THE TEST ORACLE PROBLEM

Even with efficient algorithms to produce covering arrays, the oracle problem remains – testing requires both test data and results that should be produced for each data input. High interaction strength combinatorial testing may require a large number of tests in some cases, although not always. The oracle problem occurs with all software testing, of course, but combinatorial methods introduce some interesting considerations in dealing with test oracles. Some of the more common ways of handling this problem (for any test methodology) are crash testing, built-in self-test, and model based testing.

Crash testing: The easiest and least expensive approach is to simply run tests against the system under test (SUT) to check whether any unusual combination of input values causes a crash or other easily detectable failure. This is essentially the same procedure used in “fuzz testing” (16), which sends random values against the SUT. Using combinatorial testing in this way could be regarded as a disciplined form of fuzz testing, because although pure random testing will generally cover a high percentage of t-way combinations, 100% coverage of combinations requires a random test set much larger than a covering array. For example, all 3-way combinations of 10 parameters with 4 values each can be covered with 151 tests. Purely random generation requires over 900 tests to provide full 3-way coverage.

Built-in self-test and embedded assertions: An increasingly popular “light-weight formal methods” technique is to embed assertions within code to ensure proper relationships between data, for example as preconditions, postconditions, or input value checks. Tools such as the Java Modeling language (JML) can be used to introduce very complex assertions, effectively embedding a formal specification within the code. The embedded assertions serve as an executable form of the specification, thus providing an oracle for the testing phase. With embedded assertions, exercising the application with all t-way combinations can provide reasonable assurance that the code works correctly across a very wide range of inputs. This approach has been used successfully for testing smart cards, with embedded JML assertions acting as an oracle for combinatorial tests (17). Results showed that 80% - 90% of errors could be found in this way.

Model based test generation uses a mathematical model of the SUT and a simulator or model checker to generate expected results for each input (18). If a simulator can be used, input values for each test are taken from the covering array, and expected results can be generated from the simulation. Model checkers are widely available and can also be used to prove properties such as liveness in parallel processes, in addition to generating tests. Conceptually, a model checker can be viewed as exploring all states of a system model to determine if a property claimed in a specification statement is true. What makes a model checker particularly valuable is that if the claim is false, the model checker not only reports this, but also provides a "counterexample" showing how the claim can be shown false. If the claim is false, the model checker indicates this and provides a trace of parameter input values and states that will prove it is false. To use this method, the model checker is invoked with a mutated specification and input values for each test in the covering array. The counterexample can be post-processed into a complete test case, i.e., a set of parameter values and expected result.

ADVANCED TOPICS

The field of combinatorial testing is developing rapidly. This section reviews some recent advances in combinatorial methods for software testing.

SEQUENCE COVERING ARRAYS

For many types of software, the sequence of events is an important consideration. For example, graphical user interfaces may present the user with a large number of options that include both order-independent (e.g., choosing items) and order-dependent selections (such as final selections of items, quantity and payment information). The software should work correctly, or issue an appropriate error message, regardless of the order of events selected by the user. A number of test approaches have been devised for these problems, including graph-covering, syntax-based, and finite state machine methods.

In testing such software, the critical condition for triggering failures often is whether or not a particular event has occurred prior to a second one, not necessarily if they are back to back. This situation reflects the fact that in many cases, a particular state must be reached before a particular failure can be triggered. For example, a failure might occur when connecting device A only if device B is already connected, or only if devices B and C were both already connected. The methods described in this paper were developed to address testing problems of this nature, using combinatorial methods to provide efficient testing. Sequence covering arrays ensure that every t events from a set of n (n > t) will be tested in every possible t-way order, possibly with interleaving events among each subset of t events (19, 20, 21).

A sequence covering array, SCA(N, S, t) is an N x S matrix where entries are from a finite set S of s symbols, such that every t-way permutation of symbols from S occurs in at least one row and each row is a permutation of the s symbols. The t symbols in the permutation are not required to be adjacent. That is, for every t-way arrangement of symbols x_1, x_2, ..., x_t, the regular expression $.*x_1.*x_2.*x_t.*$ matches at least one row in the array.

Example 1. We may have a component of a factory automation system that uses certain devices interacting with a control program. We want to test the events defined in Table 6. There are 6! = 720 possible sequences for these six events, and the system should respond correctly and safely no matter the order in which they occur. Operators may be instructed to use a particular order, but mistakes are inevitable, and should not result in injury to users or compromise the operation. Because setup, connections and operation of this component are manual, each test can take a considerable amount of time. It is not uncommon for system-level tests such as this to take hours to execute, monitor, and complete. We want to test this system as thoroughly as possible, but time and budget constraints do not allow for testing all possible sequences, so we will test all 3-event sequences.

Event	Description
a	connect air flow meter
b	connect pressure gauge
c	connect satellite link
d	connect pressure readout
e	engage drive motor
f	engage steering control

Table 6. Example system events

With six events, a, b, c, d, e, and f, one subset of three is {b, d, e}, which can be arranged in six permutations: [b d e], [b e d], [d b e], [d e b], [e b d], [e d b]. A test that covers the permutation [d b e] is: [a d c f b e]; another is [a d c b e f]. With only 10 tests, we can test all 3-event sequences, shown in Table 2. In other words, any sequence of three events taken from a..f arranged in any order can be found in at least one test in Table 7 (possibly with interleaved events).

Test	Sequence
1	a b c d e f
2	f e d c b a
3	d e f a b c
4	c b a f e d
5	b f a d c e
6	e c d a f b
7	a e f c b d
8	d b c f e a
9	c e a d b f
10	f b d a e c

Table 7. All 3-event sequences of 6 events.

Returning to the example set of events {b, d, e}, with six permutations: [b d e] is in Test 5, [b e d] is in Test 4, [d b e] is in Test 8, [d e b] is in Test 3, [e b d] is in Test 7, and [e d b] is in Test 2.

A larger example system may have 10 devices to connect, in which case the number of permutations is 10!, or 3,628,800 tests for exhaustive testing. In that case, a 3-way sequence covering array with 14 tests covering all 3-way sequences is a dramatic improvement, as is 72 tests for all 4-way sequences. Construction methods for sequence covering arrays include greedy algorithms (19) and answer-set programming (21). Greedy methods produce good results across a broad range of problem sizes. Answer set programming can generate more compact test sets than greedy methods, but this advantage may not hold for larger problem sizes.

MEASURING COMBINATORIAL COVERAGE

Since it is nearly always impossible to test all possible combinations, combinatorial testing is a reasonable alternative, but determining an appropriate interaction strength t to be tested is critical. Conversely, test suites developed not using combinatorial methods may still cover a relatively large number of combinations. Determining the level of input or configuration state space coverage can help in understanding the degree of risk that remains after testing. If 90% - 100% of the state space has been covered, then presumably the risk is small, but if coverage is much smaller, then the risk may be substantial. A variety of measures of combinatorial coverage can be helpful in estimating this risk and have general application to any combinatorial coverage problem (19, 22, 23).

Software Test Coverage

Test coverage is one of the most important topics in software assurance. Users would like some quantitative measure to judge the risk in using a product. For a given test set, what can we say about the combinatorial coverage it provides? With physical products, such as light bulbs or motors, reliability engineers can provide a probability of failure within a particular time frame. This is possible because the failures in physical products are typically the result of natural processes, such as metal fatigue. With software the situation is more complex, and many different approaches have been devised for determining software test coverage. Thus a variety of measures have been developed to gauge the degree of test coverage. Some of the better-known coverage metrics include statement coverage (percentage of statements executed), branch coverage (percentage of branches evaluated to both true and false), condition coverage (percentage of conditions within decision expressions evaluated to both true and false), and modified condition decision coverage (MCDC), in which every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision outcome, and that each entry and exit point have been invoked at least once. All of these measures deal with execution sequence and require access to source code.

Combinatorial Coverage

Even in the absence of knowledge about a program's inner structure, we can apply combinatorial methods to produce precise and useful information by measuring the state space of inputs. Suppose we have a program that accepts two inputs, x and y, with 10 values each. Then the input state space consists of the 10^2 = 100 pairs of x and y values, which can be pictured as a checkerboard square of 10 rows by 10 columns. With three inputs, x, y, and z, we would have a cube with 10^3 = 1,000 points in its input state space, and so on.

Looking closely at the nature of combinatorial testing leads to several measures that are useful. For a set of *t* variables, a *variable-value configuration* is a set of *t* valid values, one for each of the variables.

Example. Given four binary variables, *a, b, c,* and *d, a*=0, *c*=1, *d*=0 is a variable-value configuration, and *a*=1, *c*=1, *d*=0 is a different variable-value configuration for the same three variables *a*, *c*, and *d*.

For a given test set for *n* variables, *simple t-way combination coverage* is the proportion of t-way combinations of n variables for which all variable-values configurations are fully covered. If the test set is a covering array, then coverage is 100%, by definition, but many test sets not based on covering arrays may still provide significant t-way coverage.

Example. Table 8 shows an example with four binary variables, a, b, c, and d, where each row represents a test. Of the six 2-way combinations, ab, ac, ad, bc, bd, cd, only bd and cd have all four binary values covered, so simple 2-way coverage for the four tests in 0 is 1/3 = 33.3%. There are four 3-way combinations, abc, abd, acd, bcd, each with eight possible configurations. Of the four combinations, none has all eight configurations covered, so simple 3-way coverage for this test set is 0%.

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

Table 8. An example test array for a system with four binary components

A test set that provides full combinatorial coverage for t-way combinations will also provide some degree of coverage for (t+1)-way combinations, (t+2)-way combinations, etc. For a given test set for n variables, (t+k)-way combination coverage is the proportion of (t+k)-way combinations of n variables for which all variable-values configurations are fully covered. This statistic may be useful for comparing two combinatorial test sets. For example, different algorithms may be used to generate 3-way covering arrays. They both achieve 100% 3-way coverage, but if one provides better 4-way and 5-way coverage, then it can be considered to provide more software testing assurance.

Variable-Value Configuration coverage

So far we have only considered measures of the proportion of combinations for which all configurations of *t* variables are fully covered. But when t variables with v values each are considered, each *t*-tuple has v^t configurations. For example, in pairwise (2-way) coverage of binary variables, every 2-way combination has four configurations: 00, 01, 10, 11. We can define two measures with respect to configurations: *variable-value configuration coverage* is the proportion of variable-value configurations that are covered; for a given set of n variables, *(p, t)-completeness* is the proportion of the C(n, t) combinations that have configuration coverage of at least p.

Example. For Table 8 above, there are C(4, 2) = 6 possible variable combinations and $C(4, 2) \times 2^2 = 24$ possible variable-value configurations. Of these, 19 variable-value configurations are covered and the only ones missing are ab=11, ac=11, ad=10, bc=01, bc=10. But only two, bd and cd, are covered with all 4 value pairs. So for the basic definition of simple t-way coverage, we have only 33% (2/6) coverage, but 79% (19/24) for the configuration coverage metric. For a better understanding of this test set, we can compute the configuration coverage for each of the six variable combinations, as shown in 0. So for this test set, one of the combinations (bc) is covered at the 50% level, three (ab, ac, ad) are covered at the 75% level, and two (bd, cd) are covered at the 100% level. And, as noted above, for the whole set of tests, 79% of variable-value configurations are covered. All 2-way combinations have at least 50% configuration coverage, so (.50, 2)-completeness for this set of tests is 100%.

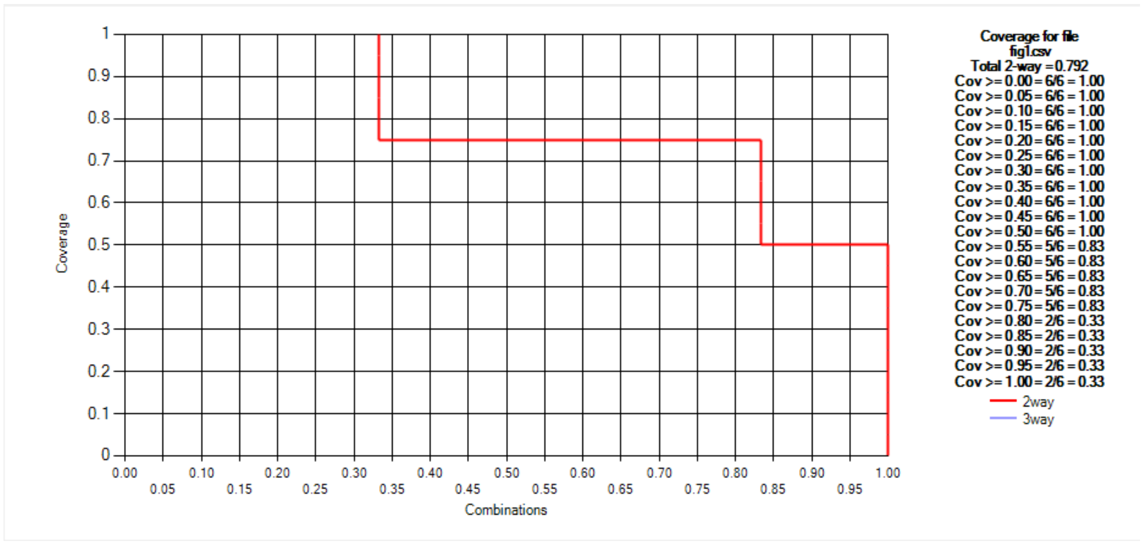
Although the example in 0 uses variables with the same number of values, this is not essential for the measurement. Coverage measurement tools that we have developed compute coverage for test sets in which parameters have differing numbers of values, as shown in **Error! Reference source not found.**.

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

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

The graph in Fig. 5 shows a graphical display of the coverage data for the tests in Figure 4. Coverage is given as the Y axis, with the percentage of combinations reaching a particular coverage level as the X axis. Note from Fig. 1 that 100% of the combinations are covered to at least the .50 level, 83% are covered to the .75 level or higher, and a third covered 100%. Thus the rightmost horizontal line on the graph corresponds to the smallest coverage value from the test set, in this case 50%.



```
Coverage for file
fig1.csv
Total 2-way =0.792
Cov >= 0.00 = 6/6 = 1.00
Cov >= 0.05 = 6/6 = 1.00
Cov >= 0.10 = 6/6 = 1.00
Cov >= 0.15 = 6/6 = 1.00
Cov >= 0.20 = 6/6 = 1.00
Cov >= 0.25 = 6/6 = 1.00
Cov >= 0.30 = 6/6 = 1.00
Cov >= 0.35 = 6/6 = 1.00
Cov >= 0.40 = 6/6 = 1.00
Cov >= 0.45 = 6/6 = 1.00
Cov >= 0.50 = 6/6 = 1.00
Cov >= 0.55 = 5/6 = 0.83
Cov >= 0.60 = 5/6 = 0.83
Cov >= 0.65 = 5/6 = 0.83
Cov >= 0.70 = 5/6 = 0.83
Cov >= 0.75 = 5/6 = 0.83
Cov >= 0.80 = 2/6 = 0.33
Cov >= 0.85 = 2/6 = 0.33
Cov >= 0.90 = 2/6 = 0.33
Cov >= 0.95 = 2/6 = 0.33
Cov >= 1.00 = 2/6 = 0.33
```

Figure 5. Graph of coverage from example test data.

TEST PRIORITIZATION

One the the advantages of combinatorial testing is efficiency – by including all t-way combinations, a tests based on a covering array can encompass a large portion of the input state space in a relatively small number of tests. But efficiency can be improved still further by prioritizing tests according to some metric. The basic idea is to run tests in the order most likely to find faults as rapidly as possible. Prioritization of this sort can be particularly valuable if each test requires significant setup or execution time. Test prioritization is also used with other (non combinatorial) test methodologies, and is an important criterion in test design. There are generally two approaches to prioritization with combinatorial testing: (1) combinatorial coverage may be used as the ordering criterion, i.e., selecting the next test on the basis of which covers the most previously uncovered combinations, and (2) for a test set based on a complete covering array, use some metric to order the tests within the array.

Prioritization by Combinatorial Coverage

In this approach to test ordering, an existing test suite, which may not have been developed as a covering array, is re-ordered to cover combinations as rapidly as possible. The test suite will probably not cover all combinations (if it was not specifically developed as a covering array), but if it is sufficiently large, a high percentage of combinations will be covered (see Combinatorial Coverage section above). These tests are then ordered according to which ones cover the most combinations that have not been included in previously executed tests. This method has been shown to be very effective, with a higher rate of fault detection than other prioritization criteria (24).

Prioritization of Covering Arrays

Test prioritization is still possible when the test suite is a covering arry and thus includes all t-way combinations (25). A variety of ordering heuristics can be used, including code coverage, specification-specific weightings, Hamming distance or other measures of difference between tests, and combination coverage (25, 26). If a greedy algorithm is used to generate the covering array, the tests will automatically be ordered by combination coverage, since these algorithms iteratively select tests according to which covers the most previously uncovered combinations. Code coverage metrics order tests according to which produces the highest level of source code coverage; branch coverage has been used for this metric (25). In some cases it may be practical use particular aspects of the system under test, such as assigning higher weights to tests that invoke system features as compared with those that turn off features. Similarly, selecting tests by which are most different from previously run tests may be helpful in exercising as many different system functions rapidly. All of the prioritization schemes discussed in this section have been found to be effective, but their efficacy is likely to vary with characteristics of the system under test. Determining where to use the various schemes is an area of active research.

CONCLUSIONS

Combinatorial testing practice has grown rapidly from early efforts using orthogonal arrays taken from the statistical field of Design of Experiments, through simple pairwise testing of 2-way interactions, to t-way testing of interactions of 3, 4, or more variables. The method’s effectiveness is based on the empirical Interaction Rule, which states that most failures are caused by one or two parameters interacting, with progressively fewer by 3, 4, or more parameter interactions. Advances in algorithms have made it possible to generate t-way covering arrays, for at least t=6, that are practical for testing large programs. Covering arrays can be applied to either configuration testing or input testing. In configuration testing, the arrays are used to determine t-way combinations of configurable parameter values for software that is designed to run on a variety of platforms, or has variable settings that are established at run time. With input testing, a covering array may be used to set discretized values for a number of input variables, or to choose combinations of characteristics for a single input variable.

Combinatorial methods are also being applied to software testing to produce efficient tests for sequences of events or inputs. This work is recent and additional study is needed to determine its applicability in various testing problems. Measuring combinatorial coverage of a test set can be valuable in understanding the test set's potential for revealing interaction faults, even if the tests were not designed using combinatorial methods. A t-way covering array by definition has 100% combination coverage for combinations of t or fewer variables, but it may also provide some coverage at higher values of t. Coverage measurement helps testers understand the higher strength coverage of a t-way covering array. Tools to implement the combinatorial methods described in this article are widely available and are being used by testers to reduce cost and increase test effectiveness.

REFERENCES

1. Wallace, D.R.; Kuhn, D.R. Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data, International Journal of Reliability, Quality, and Safety Engineering, Vol. 8, No. 4, 2001.

2. Kuhn, D.R.; Reilly, M.J. An Investigation of the Applicability of Design of Experiments to Software Testing, 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center, 4-6 December, 2002 .

3. Kuhn, D.R.; Wallace, D.R.; and Gallo, A. “Software Fault Interactions and Implications for Software Testing,” IEEE Transactions on Software Engineering, 30(6): 418-421, 2004

4. Kuhn, D.R.; Okun, V. “Pseudo-exhaustive Testing for Software,” Proceedings of 30th NASA/IEEE Software Engineering Workshop, pp. 153-158, 2006

5. Bell, K.Z. Optimizing Effectiveness and Efficiency of Software Testing: a Hybrid Approach, PhD Dissertation, North Carolina State University, 2006.

6. Sloane, N. J. A. Covering arrays and intersecting codes, Journal of Combinatorial Designs, 1 (1993), 51–63.

7. Dalal, S.R.; Mallows, C.L. Factor-covering designs for testing software, Technometrics, 40(3) (1998), 234–242.

8. Colbourn, C. J. Combinatorial aspects of covering arrays. Le Matematiche (Catania), 58, 121–167. (2004).

9. Box, G. E., Hunter,W.G., Hunter, J.S., Hunter,W.G., "Statistics for Experimenters: Design, Innovation, and Discovery", 2nd Edition, Wiley, 2005.

10. Sherwood, G. B. (1994), “Effective testing of factor combinations,” Proceedings of the Third International Conference on Software Testing, Analysis and Review, Washington D.C. May 8–12, 1994.

11. Cohen, M. B., Colbourn, C. J., & Ling, A. C. H. (2008). Constructing Strength 3 Covering Arrays with Augmented Annealing. Discrete Mathematics, 308, 2709–2722.

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

13. Lei, Y.; Kacker, R.; Kuhn, D.R.; Okun, V.; Lawrence, J. "IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing", Software Testing, Verification, and Reliability.

14. Hartman. A. Software and hardware testing using combinatorial covering suites. Haifa Workshop on Interdisciplinary Applications of Graph Theory, Combinatorics, and Algorithms, June 2002.

15. Ostrand, T.J.; Balcer, M.J. The Category-Partition method for specifying and generating functional tests, Communications of ACM 31, 1988.

16. Miller, B.P.; Fredrikson, L., and So, B., An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM 33, 12 (December 1990), 32-44.

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

18. Ammann, P.; Black, P.E. Abstracting Formal Specifications to Generate Software Tests via Model Checking, Proc. 18th Digital Avionics Systems Conference, Oct. 1999, IEEE, vol. 2. pp. 10.A.6.1-10

19. Kuhn, D.R.; Kacker, R.; Lei, Y. Practical Combinatorial Testing, NIST SP800-142, National Institute of Standards and Technology, October 6, 2010.

20. Zamli, K.Z., Othman, R.R., Zabil, M.H.M., On Sequence Based Interaction Testing, IEEE Symp. on Computers and Informatics, IEEE, 20-23 Mar. 2011, pp. 662-667.

21. Erdem, E.; Inoue, K. J.; Oetsch, J. Puhrer, H. Tompits, C. Yilmaz, Answer-Set Programming as a New Approach to Event-Sequence Testing, Tech. Rpt., Sabanci U., Istanbul, Turkey, 2011.

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

23. Chen, B. Zhang, J. Tuple Density: A New Metric for Combinatorial Test Suites. 33rd Intl. Conf. on Software Engineering, Honolulu, Hawaii, May 21 – 28, 2011.

24. Bryce, R. C.; Memon, A. M.. Test suite prioritization by interaction coverage. In *the Workshop on Domain-Specific Approaches to Software Test Automation*, pages 1–7, Sep.2007.

25. Bryce, R. Colbourn, C. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Journal of Information and Software Technology, 48(10):960–970, 2006

26. Qu, X.; Cohen, M. B.; Woolf, K. M.. Combinatorial interaction regression testing: A study of test case generation and prioritization. In Intl. Conference on Software Maintenance, pages 255–264, Oct. 2007.