Combinatorial Testing: Theory and Practice

D. Richard Kuhn[1], Renee Bryce[2], Feng Duan[3], Laleh Sh. Ghandehari[3], Yu Lei[3], Raghu N. Kacker[1]

National Institute of Standards and Technology, University of North Texas, University of Texas at Arlington

kuhn@nist.gov, ylei@uta.edu, renee.bryce@uta.edu, feng.duan@mavs.uta.edu, raghu.kacker@nist.gov

Abstract. Combinatorial testing has rapidly gained favor among software testers in the past decade as improved algorithms have become available, and practical success has been demonstrated. This article reviews the theory and application of this method, focusing particularly on research since 2010, with a brief background providing the rationale and development of combinatorial methods for software testing. Significant advances have occurred in algorithm performance, and the critical area of constraint representation and processing. In addition to these foundational topics, we take a look at advances in specialized areas including test suite prioritization, sequence testing, fault localization, the relationship between combinatorial testing and structural coverage, and approaches to very large testing problems.

Keywords: algorithms, combinatorial testing, constraints, covering array, fault localization, interaction testing, sequence testing, software faults, software testing, test suite prioritization

1 Introduction

An interesting phenomenon often occurs with large software systems. After successful use for a long period of time, the software is installed in a new location, with a different user base, and a new set of bugs appear. Typically the reason is not surprising – the change in usage has resulted in a different set of inputs, and some of these input combinations trigger failures that have escaped previous testing and extensive use. Such failures are known as *interaction failures*, because they are only exposed when two or more input values interact to cause the program to reach an incorrect result.

For example, a pump may be observed to fail only when pressure is below a particular level and volume exceeds a certain amount, a 2-way interaction between pressure and volume. 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. Either one of the conditions, without the other, will not be a problem.

< Figure 1. 2-way interaction failure triggered only when two conditions are true>

1.1 Empirical Data on Failures

The example above focuses on a 2-way interaction failure. *Pairwise testing*, using tests that cover all 2-way combinations of parameter values, has long been accepted as a way of detecting such interaction failures [1][2][3][4]. However, higher order *t*-way interactions may also cause failures. For instance, consider a failure that is triggered by unusual combinations of three or four sensor values. For thorough checking, it may be necessary to test 3-way and 4-way combinations of values. The question arises as to whether testing all 4-way combinations is enough to detect all errors. What is the distribution of interaction failures beyond 2-way in real systems? Surprisingly, this question had not been studied when NIST began investigating interaction failures in 1999 [5]. Results of this and subsequent studies showed that across a variety of domains, all failures were triggered by a maximum of 4-way to 6-way interactions [6, 7, 8]. As shown in Figure 2, the detection rate increases 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 [8, 9, 10] have been consistent with these results.

< Figure 2. Cumulative fault distribution >

The empirical data show that most failures are triggered by a single parameter value, or interactions between a small number of parameters, generally two to six, a relationship known as the *interaction rule*. An example of a single-value fault might be a buffer overflow that occurs when the length of an input string exceeds a particular limit. Only a single condition must be true to trigger the fault: *input length > buffer size*. A 2-way fault is more complex, because two particular input values are needed to trigger the fault, as in the example above. More generally, a *t*-way fault involves *t* such conditions. We refer to the distribution of *t*-way faults as the *fault profile.*

A question naturally arises as to why the fault profiles of different applications are somewhat similar. While there is no definitive answer as yet, one clue can be seen in Figure 3. Fig. 3.1 (left) shows the distribution of conditions in branching statements (e.g., *if, while*) in four large avionics software modules [11]. Nearly all are single conditions or 2-way, with a rapidly declining proportion involving 3-way or more complex sets of conditions. This curve is superimposed on the fault profiles presented above in Fig. 3.2. Note that it closely matches the profile for the NASA database application. The data for this application were from initial testing results, while the other curves are for fielded products. Thus the distribution of faults in this initial testing is quite close to the distribution of conditions documented in the FAA report. (It is not clear why the distribution of faults for the medical device software is as shown, as no information was available on the level of testing or usage for these products.) The fault profiles may

reflect the profile of *t*-way conditions in their application software, but as faults are discovered and removed, the more complex 3-way, 4-way and beyond, faults comprise a larger proportion of the total. Testing and extensive usage thus tend to push the curves down and to the right.

< Fig 3.1 and 3.2 side by side; about ¼ page >
<Figure 3.1. Distribution of conditions in branching statements>

<Figure 3.2. Fault distribution of different application domains>

1.2 Implications for Testing

The fault profiles reported above suggests that testing which covers a high proportion of 4-way to 6-way combinations can provide strong assurance. If we know that *t* or fewer variables are involved in failures, and we can test all *t*-way combinations, then we can have reasonably high confidence that the application will function correctly. As shown above, the distribution of faults varies among applications, but two important facts are apparent: a consistently high level of fault detection has been observed for 4-way and higher strength combinations; and no interaction fault discovered so far, in thousands of failure reports, has involved more than six variables. Thus, the impossibility of exhaustive testing of all possible inputs is not a barrier to high assurance testing. That is, even though we cannot test all possible combinations of input values, failures involving more than six variables are extremely unlikely because they have not been seen in practice, so testing all possible combinations would provide little or no benefit beyond testing 4 to 6-way combinations.

As with all testing, it is necessary to select a subset of values for variables with a large number of values, and test effectiveness is also dependent on the values selected, but testing *t*-way combinations has been shown to be highly effective in practice. This approach is known as *combinatorial testing*, an extension of the established field of statistical Design of Experiments (DoE). Matrices known as *covering arrays* cover all *t*-way combinations of variable values, up to a specified level of *t* (typically $t \leq 6$), making it possible to efficiently test all such *t*-way interactions.

Consider the example in Table 1 that shows four configurations to consider when testing a web application. The tester wants to test their web app on three types of devices (called parameters), three web browsers, three PHP versions, and three network connections. Each parameter has 3 options (called values). To exhaustively test every combination requires 3*3*3*3 = 81 possible combinations.

Device	**Web Browser**	**PHP version**	**Network Connection**
PC	Safari	5.6.6	WiFi
Tablet	Firefox	5.5.22	3G
Smart Phone	Chrome	5.6.5	4G

Table 1 - Sample input for a combinatorial test suite that has 4 parameters that have 3 possible values each

We use the ACTS tool described in Section 3.3 to generate a 2-way Combinatorial Test Suite. This requires only 9 test cases (configurations) in order to test all pairs of parameter-value combinations as shown in Table 2. A pair is a combination of values for two different parameters. For instance, Test Case 1 covers six pairs: (PC, Safari), (PC, PHP version 5.5.22), (PC, 3G), (Safari, PHP version 5.5.22), (Safari, 3G), (PHP version 5.5.22, 3G).

Test No.	Device	Web Browser	PHP Version	Network Connection
1	PC	Safari	5.5.22	3G
2	PC	Firefox	5.6.5	4G
3	PC	Chrome	5.6.6	WiFi
4	Tablet	Safari	5.6.5	WiFi
5	Tablet	Firefox	5.6.6	3G
6	Tablet	Chrome	5.5.22	4G
7	Smart Phone	Safari	5.6.6	4G
8	Smart Phone	Firefox	5.5.22	WiFi
9	Smart Phone	Chrome	5.6.5	3G

Table 2 - Sample Combinatorial Test Suite for the input 3^4 from Table 1

The effectiveness of any software testing technique depends on whether test settings corresponding to the actual faults are included in the test sets. When test sets do not include settings corresponding to actual faults, the faults may not be detected. Conversely, we can be confident that the software works correctly for *t*-way combinations contained in passing tests. When the tests are derived from *t*-way covering arrays, we know that 100% of the *t*-way combinations have been tested.

2 Covering Arrays

Combinatorial testing (CT) is an adaptation of the 'design of experiment (DoE)' methods to test software and systems. CT and DoE are dynamic testing and learning methods in the sense that a system of interest is exercised (run) for a set of different test cases and the behavior or response of the system for those test cases is investigated. Historically, CT evolved from attempts to improve performance of software based systems starting in the 1980s [12]. DoE refers to a methodology for conducting controlled experiments in which a system is exercised (worked in action) in a purposeful (designed) manner for chosen test settings of various input variables called *factors*. In DoE, many factors each having multiple test settings are investigated at the same time and the DoE plans satisfy relevant combinatorial properties. The corresponding values of one or more output variables (called responses) are measured. A statistical model (at least one) for the system response is associated with each DoE test plan. The DoE test plan and the responses values are

used to estimate the unknown parameters of the model. The estimated model so obtained represents statistical information for improving the performance of a class of similar systems [13], [14], [15], [16], and [17].

2.1 History of DoE

Conventional DoE methods were developed starting in the 1920s by British geneticist Ronald Fisher and his contemporaries and their followers, to improve agricultural production [18], [19]. Later DoE were adapted for experiments with animals, medical research, and then to improve manufacturing processes, all subject to unavoidable variation. DoE continues to be a gold standard for research in life sciences, medical technologies, and drug discovery. Recently the US Office of the Secretary of Defense promulgated more effective use of DoE in Defense Operational Test and Evaluation (DOTE) [20]. The objective in conventional DoE is to improve the mean response over replications. A Japanese engineer, Genichi Taguchi, promulgated (starting in the late 1960s Japan and 1980s USA) a variation of DoE methods for industrial experiments whose objective is to determine test settings at which the variation due to uncontrolled factors was least [21], [22], [23], [24], and [25]. Taguchi promoted use of mathematical objects called orthogonal arrays (OAs) as templates for industrial experiments. Orthogonal arrays (OAs) were largely mathematical curiosities before Taguchi stated using them for industrial experiments to develop robust products and processes.

The concept of OAs was formally defined by C. R. Rao [26] as generalization of Latin squares [27]. The matrix shown in table 3 is an orthogonal array (OA) referred to as OA(8, $2^4 \times 4^1$, 2). The first parameter (which is 8) indicates the number of rows and the second parameter (which is $2^4 \times 4^1$) indicates that there are five columns of which four have 2 distinct elements each, denoted here by {0, 1}, and one column has 4 distinct elements, denoted here by {0, 1, 2, 3}. The third parameter (which is 2) indicates that this OA has strength 2, which means that every set of two columns contains all possible pairs of elements exactly the same number of times. Thus every pair of the first four columns contains the four possible pairs of elements {00, 01, 10, 11} exactly twice, and every pair of columns involving the fifth column contains the eight possible pairs of elements {00, 01, 02, 03, 10, 11, 12, 13} exactly once. In an OA of strength *t*, every set of *t* columns contains all possible *t*-tuples of elements exactly the same number of times.

<Insert table 3 about here>

A fixed-value orthogonal array denoted by OA(N, v^k, t) is an N x k matrix of elements from a set of v symbols {0, 1, …, (v – 1)} such that every set of t-columns contains each possible t-tuple of elements the same number of times. The positive integer t is the strength of the orthogonal array. In the context of an OA, elements such as 0, 1, 2, …, (v – 1) used in table 3 are symbols rather than numbers. The combinatorial property of an orthogonal array is not affected by the symbols that are used for the elements. Every set of three columns of a fixed value orthogonal array of strength 2 represents a Latin square (one column representing the rows, one column representing the columns and the third column representing the symbols). A mixed-value orthogonal array is an extension of

fixed-value OA where $k = k_1 + k_2 + \ldots + k_n$; k_1 columns have v_1 distinct elements, k_2 columns have v_2 distinct elements, …, and k_n columns have v_n distinct elements, where $v_1, v_2, \ldots, v_k$ are different. Mathematics of OAs and extensive references can be found in [28]. Neil Sloane maintains an electronic library of known OAs [29].

Consider an industrial DoE which has five factors A, B, C, D, and E and one response Y. Suppose A, B, C, and D have two test values each, denoted by $\{A_0, A_1\}$, $\{B_0, B_1\}$, $\{C_0, C_1\}$ and $\{D_0, D_1\}$, respectively, and the factor E has four test values, denoted by $\{E_0, E_1, E_2, E_3\}$. The combinatorial test structure of this DoE is the exponential expression $2^4 \times 4^1$ which indicates that there are five factors of which four have two test settings each and one has four test settings. The number of possible test cases is $2^4 \times 4^1 = 64$. The OA(8, $2^4 \times 4^1$, 2) can be used to set up an experiment to evaluate the change in response when the test value of each factor is changed. The factors A, B, C, D, and E are associated with the columns of OA(8, $2^4 \times 4^1$, 2) and the test values are associated with the entries of the columns. Then the rows of OA(8, $2^4 \times 4^1$, 2) specify 8 of the 64 possible test cases shown in table 4.

<Insert table 4 about here>

The last column of table 4 displays the values $y_1, y_2, \ldots, y_8$, of the response Y for the eight test cases. The combinatorial properties of an OA enable estimation of the parameters of a statistical model associated with a DoE plan based on the OA. The estimated parameters and the estimated statistical model identify test settings of the five factors at which the system may have improved performance.

2.2 From DoE to Covering Arrays

Along with the advent of computers and telecommunication systems in the 1980s, independent verification and validation of software and hardware-software systems became important. Genichi Taguchi inspired the use of OAs for testing software systems. Software engineers in various companies (especially Fujitsu in Japan and the descendent organizations of the AT&T Bell System in the US) started to investigate use of DoE methods for testing software and hardware-software systems. The earliest papers include the following: [30], [31], [1], [2], [3]. The limitations of OAs for independent verification and validation of software based systems became clear soon after they were used. (i) Often, an OA matching the required combinatorial test structure does not exist; for example, a non-trivial OA of strength 2 matching the test structure $2^4 \times 3^1$ (four factors with two distinct settings and one with three settings) is mathematically impossible. (ii) Frequently, OA based test suites included invalid test cases which are impossible (or meaningless) to execute; for example, in testing jointly various operating systems and browsers Linux cannot be combined with Microsoft Internet Explorer. (iii) Available OA tables were limited to at most strength three, while for testing software systems, test suites of strength larger than three may be required. (iv) In testing software systems, hundreds of factors may be involved, but available OA tables were much smaller. Keizo Tatsumi [2] [3] and Dalal and Mallows [4] provided the insight that in testing software, combinatorial balancing property of OAs (that each *t*-tuple should appear the same

number of times) was not required (because parameters of statistical model were not being estimated). In testing software systems, space filling was needed; that is, each *t*-tuple of interest of the test settings must be covered at least once. Therefore mathematical objects called covering arrays (CAs) are better suited than OAs as templates for generating test suites for software testing.

The concept of Covering Arrays (CAs) was formally defined by AT&T mathematician Neil Sloane [32]. Additional developments on CAs can be found in the following recent papers: [33], and [34]. A fixed-value covering array denoted by CA(N, v^k, t) is an N x k matrix of elements from a set of v symbols $\{0, 1, \ldots, (v-1)\}$ such that every set of t-columns contains each possible t-tuple of elements at least once. The positive integer t is the strength of the covering array. A fixed value covering array may also be denoted by CA(N, k, v, t). A mixed-value covering array is an extension of fixed value CA where $k = k_1 + k_2 + \ldots + k_n$; k_1 columns have v_1 distinct elements, k_2 columns have v_2 distinct elements, …, and k_n columns have v_n distinct elements. The six rows of rows of table 5 form a covering array CA(6, $2^4 \times 3^1$, 2). In these six rows each set of two columns contains each possible pair of symbols at least once. The combinatorial property of covering arrays is more relaxed (less stringent) than that of orthogonal arrays: a CA need not be balanced in the sense that not all t-tuples need to appear the same number of times. All OAs are CAs but not all CAs are OAs. (An orthogonal array of index one in which every t-tuple appears exactly once is the best possible covering array.) Thus the concept of covering arrays is a generalization of OAs. Covering arrays have a number of advantages over OAs for testing software systems. (i) CAs can be constructed for any combinatorial test structure of unequal numbers of test settings. (ii) If for a combinatorial test structure an OA exists then a CA of the same or less number of test cases can be obtained. (iii) CAs can be constructed for any required strength (t-way) testing, while OAs are generally limited to strength 2 and 3. (iv) In generating test suites based on CAs invalid combinations can be deliberately excluded. (v) CA for very large number of factors can be constructed.

For a given number of factors k, the size of a combinatorial t-way test suite based on a CA (number of rows of covering array) increases exponentially with the number of test settings v of each factor. Therefore in combinatorial testing it is advisable to limit the number of distinct discrete test settings of each factor to less than ten; preferred values are 2 to 4. The discrete test settings are generally determined by equivalence partitioning and boundary value analysis of the domain of possible values for each factor.

The size of combinatorial t-way test suite also increases rapidly as t increases. For example consider the combinatorial test structure example $3^3 4^4 5^2$ from [35]. The number of possible test cases is $3^3 4^4 5^2 = 172\,800$. Exhaustive testing may not be practical. The sizes (number of test cases) of t-way test suites (determined using ACTS/IPOG) for t = 2, 3, 4, 5, and 6 are respectively 29, 137, 625, 2532, and 9168. This highlights the important question of how the strength t should be set? A reasonable choice of the strength t requires experience with the type of SUT being tested. The available knowledge about the SUT and the nature of possible faults to be detected is used in the specification of test factors, test setting, and the strength t. In one testing experiment involving 128 binary

factors (each having two distinct test settings) CAs of strength t for t = 2, ,…, 10 were needed. The sizes of required covering arrays determined by Jose Torres-Jimenez [36] are respectively, N = 11, 37, 112, 252, 1231, 2462, 17544, 90300, and 316940. When the available knowledge about the SUT is severely limited, the choice of t is difficult. The choice of t requires a tradeoff between the cost of testing (determined by the size of test suite) and the potential benefits of higher strength testing.

Tables

Table 3: Orthogonal array OA(8, $2^4 \times 4^1$, 2)

	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	3
8	1	0	0	1	3

Table 4: DoE plan based on OA(8, $2^4 \times 4^1$, 2)

Test cases	A	B	C	D	E	Response
1	A_0	B_0	C_0	D_0	E_0	y_1
2	A_1	B_1	C_1	D_1	E_0	y_2
3	A_0	B_0	C_1	D_1	E_1	y_3
4	A_1	B_1	C_0	D_0	E_1	y_4
5	A_0	B_1	C_0	D_1	E_2	y_5
6	A_1	B_0	C_1	D_0	E_2	y_6
7	A_0	B_1	C_1	D_0	E_3	y_7
8	A_1	B_0	C_0	D_1	E_3	y_8

Table 5: Covering array CA(6, $2^4 \times 3^1$, 2)

	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

2.3 Combinatorial Coverage

A recent Cambridge University technical report estimates the global cost of debugging software has risen to $312 billion annually. The authors suggest that software developers spend approximately 50% of their programming time on average finding and fixing bugs [117]. While there are many types of defects that contribute to project costs and many ways to test for different types of defects, one type of defect that we examine in this chapter is that of interaction faults. Tests based on covering arrays can be highly effective as they systematically cover *t*-way combinations of values. Covering arrays include these combinations in a very compact form, but as long as all of the combinations are covered, it does not matter whether they come from covering arrays or a less efficient test set, possibly generated randomly. Test quality is obviously of central importance for software assurance, but there are few good measures available. A very basic, minimal foundation is that every requirement has been addressed by at least one test. If source code is available, then coverage measures such as statement or branch coverage may also be useful. Mutation testing is also a popular approach to evaluating test set adequacy. Combinatorial methods offer an additional tool for measuring test set quality.

Any test with *n* variables contains C(*n*,*t*) *t*-way combinations, and any collection of tests will contain a set of combinations, though many are duplicated. If the test set is large enough, it may provide full *t*-way coverage, even if not originally constructed as a covering array. C*ombinatorial coverage*, i.e., the coverage of *t*-way combinations in a test set, is thus a useful measure of test set quality [37][38]. Note that such a coverage measure is independent of other measures of test quality, such as the code coverage induced by a particular set of tests. It is also directly related to fault detection. Combinatorial coverage is a measure of the input space that is tested.

The level of input space coverage also provides some measure of the degree of risk that remains after testing. Combinatorial coverage provides a direct measure of the proportion of input combinations for which the system has been shown to work correctly, which can be used in gauging the residual risk after testing.

2.3.1 Measures of Combinatorial Coverage

Combinatorial coverage measures include the following (definitions and examples from [35]):

Variable-value configuration: For a set of *t* variables, a variable-value configuration is a set of *t* valid values, one for each of the variables, i.e., the variable-value configuration is a particular setting of the variables.

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

Simple t-way combination coverage: For a given test set for *n* variables, simple *t*-way combination coverage is the proportion of *t*-way combinations of *n* variables for which all valid variable-values configurations are fully covered.

Example. Table 6 shows four binary variables, *a, b, c,* and *d,* where each row represents a test. Of the six possible 2-way variable combinations, *ab, ac, ad, bc, bd, cd,* only *bd* and *cd* have all four binary values covered, so simple 2-way coverage for the four tests in Table 6 is 2/6 = 33.3%. There are four 3-way variable combinations, *abc, abd, acd, bcd,* each with eight possible configurations: 000, 001, 010, 011, 100, 101, 110, 111. Of the four combinations, none has all eight configurations covered, so simple 3-way coverage for this test set is 0%. As shown later, test sets may provide strong coverage for some measures even if simple combinatorial coverage is low.

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

Table 6. Test array with four binary components

It is also useful to measure the number of *t*-way combinations covered out of all possible settings of *t* variables.

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

An example helps to clarify these definitions. For the array in Table 6, there are C(4,2) = 6 possible variable combinations and $2^2 \times C(4,2) = 24$ possible variable-value configurations. Of these, 19 variable-value configurations are covered and the only ones missing are *ab*=11, *ac*=11, *ad*=10, *bc*=01, *bc*=10, so the total variable-value configuration coverage is 19/24 = 79%. But only two, *bd* and *cd*, out of six, are covered with all 4 value

pairs. So for simple *t*-way coverage, we have only 33% (2/6) coverage, but 79% (19/24) for total variable-value configuration coverage. Although the example in Table 6 uses variables with the same number of values, this is not essential for the measurement, and the same approach can be used to compute coverage for test sets in which parameters have differing numbers of values.

< Fig 4 about ¼ page >
< Figure 4. Graph of coverage for Table I tests >

Figure 4 shows a graph of the 2-way (red/solid) and 3-way (blue/dashed) coverage data for the tests in Table 6. Coverage is given as the Y axis, with the percentage of combinations reaching a particular coverage level as the X axis. For example, the 2-way line (red) reaches Y = 1.0 at X = .33, reflecting the fact that 2/6 of the six combinations have all 4 binary values of two variables covered. Similarly, Y = .5 at X = .833 because one out of the six combinations has 2 of the 4 binary values covered. The area under the curve for 2-way combinations is approximately 79% of the total area of the graph, reflecting the total variable-value configuration coverage, designated S_t. Two additional quantities are also useful. Φ_t = the proportion of full *t*-way coverage; in the example above, $\Phi_2 = .33$. M_t = minimum coverage for level t; in the example, $M_2 = .50$. It is easy to show that $S_t \geq \Phi_t + M_t - \Phi_t M_t$ [38].

In addition to analyzing the combination coverage of individual test suites, lower bounds for coverage have been established for a number of test criteria, including base choice [39] and MCDC [11]. For example, simple all-values testing provides

$$S_t \geq M_t = \frac{1}{v^{t-1}}.$$

With base-choice testing [39] every parameter value must be covered at least once and in a test where other values are held constant. This process works by specifying one or more values for each parameter as base choices, which may be arbitrary, or "special interest" values, such as values more frequently used. Where parameters p_1 .. p_n have v_i values each, the number of tests required is at least $1 + \Sigma_{i=1,n}(v_i - 1)$, or $1+n(v-1)$ if all n parameters have the same number of values v. An example is shown below in Table 7, with four binary parameters.

TABLE 7. BASE CHOICE TESTS FOR 2^4 CONFIGURATION

	a	*b*	*c*	*d*
base:	0	0	0	0
test 2	1	0	0	0
test 3	0	1	0	0
test 4	0	0	1	0
test 5	0	0	0	1

It can be shown that the minimum combination coverage for base choice is $M_t = \frac{1+t(v-1)}{v^t}$, and consequently also $S_t \geq \frac{1+t(v-1)}{v^t}$. A variety of results for other strategies are given in [38].

2.3.2 Using Combinatorial Coverage

Figure 5 illustrates the application of combinatorial coverage analysis to a set of 7,489 tests developed for spacecraft software [40], using conventional test design methods (not designed as a covering array) to verify normal operation and a variety of fault scenarios. The system includes 82 variables, with the configuration shown in Table 8 of $1^3 2^{75} 4^2 6^2$ (three 1-value, 75 binary, two 4-value, and two 6-value). Figure 5 shows combinatorial coverage for this system (red = 2-way, blue = 3-way, green = 4-way, orange = 5-way). Pairwise coverage is with 82% of the 2-way combinations covering 100% of possible variable-value configurations covered and about 98% of the 2-way combinations have at least 75% of possible variable-value configurations covered (long horizontal portion of red line).

< Figure 5. Configuration coverage for spacecraft example. >

interaction	combinations	settings	coverage
2-way	3321	14761	94.0
3-way	88560	828135	83.1
4-way	1749060	34364130	68.8
5-way	27285336	603068813	53.6

Table 8. Total t-way coverage for Fig. 3 configuration.

3 Algorithms for Combinatorial Testing

As mentioned in previous sections, Combinatorial Testing for the purpose of software testing stems from a rich history of Design of Experiments, including designs such as Orthogonal Arrays [41]. This previous work has had a strong influence on algorithms to generate combinatorial test suites. In this section, we discuss challenges to generating combinatorial test suites, categories of algorithms that have been popular in literature, and automated tools for this purpose.

Numerous algorithms exist to generate covering arrays or mixed-level covering arrays to represent combinatorial test suites. This is an NP-hard problem, meaning no efficient exact method exists. Further complications arise when constraints are required in many practical applications. That is, events may need to run in a particular sequence or that one event may disable the option to run other events. Testers may have test cases that they have already run, called seeds, and want credit for the combinations covered in those test cases rather than generating new tests to cover those interactions again. For some inputs, the best reported test suites or sizes are available [42,43, 44] while others, particularly those of mixed-level covering arrays or inputs with seeds or constraints are not collected and shared. In this section, we briefly review different types of algorithms that generate combinatorial test suites.

3.1 Categories of Algorithms

Four categories of algorithms have been popular for the purpose of generating combinatorial test suites. These include:

- Algebraic techniques,
- Greedy algorithms,
- Heuristic search,
- Constraint Satisfaction Problem algorithms

Software testers need to choose the technique that best applies to their domain. For instance, if a tester has 7 parameters that have 5 options each, an algebraic technique will give the best-known solution [43]. On the other hand, if a system has a varying number of options for each parameter and constraints among parameters, algebraic techniques are often less effective in terms of producing a smaller test suite. Constraint Satisfaction Problem algorithms have also been used, but mainly on small inputs [43]. Table 9 summarizes tradeoffs among these algorithm classes.

	Algebraic	**Greedy**	**Heuristic Search**
Size of test suites	**Accurate on special cases; but not as general as needed**	**Reasonably accurate**	**Most accurate (if given enough time)**
Time to generate tests	**Yes**	**Yes**	**Often time consuming (for good results)**
Seeding/ Constraints	**Difficult to accommodate seeds/constraints**	**Yes**	**Yes**

Table 9. Covering array algorithm characteristics

If a system has parameters with different numbers of options, this requires a mixed-level covering array for the solution. Greedy algorithms and heuristic search are often the best algorithms for the scenario of mixed-level inputs. For instance, consider the input $4^{15}3^{17}2^{29}$. The greedy DDA produces a test suite of size 35, the greedy IPO reports a test suite of size 36, and the AETG algorithm reports a test suite of size 41 [47]. Of course, there is still variation in the results among these three greedy algorithms in which each outperforms the others on different inputs.

On the other hand, the best known result for input $5^{1}3^{8}2^{2}$ is 15 test cases as achieved by a heuristic search technique, simulated annealing [46]. Testers must consider the trade-off in time when selecting a greedy or heuristic search algorithm. Heuristic search algorithms may take much longer to find a "good" solution. This issue of time to generate test suites becomes more exaggerated as the *t*-way strength of coverage for a test suite increases. For instance, Table10 shows five sample inputs and the sizes and time to generate 2-way combinatorial test suites. Please refer to [47] for the exact details of these algorithms. The

deviation in the amount of time to generate the test suites grows dramatically as the size of the input increases.

Input	our-TCG (size/time in secs)	our-AETG (size/time in secs)	SA (size/time in secs)
$5^1 3^8 2^2$	18/6	20/58	15/214
$7^1 6^1 5^1 4^5 3^8 2^3$	42/57	44/489	42/874
$5^1 4^4 3^{11} 2^5$	25/33	28/368	21/379
$6^1 5^1 4^6 3^8 2^3$	32/42	35/376	30/579
10^{20}	213/1,333	198/6,001	183/10,833

Table 10. Example size and runtime for covering array generation

Constraints between parameters/options are common in software systems. Addressing constraints is a challenging issue. We give a brief overview here, but refer the reader to Section 5 for a more detailed discussion. Consider Figure 6 below that shows four scenarios for constraints using the input $3^1 2^3$:

1. **Original scenario without constraints:** This scenario has no constraints. There are 30 pairs to cover and this may be done in as few as 6 test cases.
2. **Scenario with constraints that results in a smaller test suite:** This scenario has 3 constraints, including that f0:0 may not be combined with f1:3, f2:5, or f3:7. These constraints leave 27 pairs left to cover. In this case, we are able to cover all of the pairs while respecting the constraints in as few as 5 test cases.
3. **Scenario with constraints that results in a larger test suite:** This scenario also has 3 constraints: f0:0 may not be combined with f1:3; f1:3 may not be combined with f2:5; and f2:5 may not be combined with f3:7. Due to these constraints, the fewest number of test cases to cover all pairs with respect to the constraints is 7 test cases.
4. **No feasible solution:** The final scenario shows that the tester specified constraints in which it is not possible to construct a solution. If we must select an option for each parameter in order to generate a test case, you will notice that the constraints prohibit us from assigning a value to each parameter and covering all pairs. For instance, you will notice that f0:0 may not be combined with f1:3, so we would have to select f0:0 and f1:4 for the first two values of this test case. However, we are unable to select a value that respects the constraints for f2 since f2:5 may not be combined with f1:3 and f2:6 may not be combined with f1:4. This same scenario repeats if we include f0:1 and f1:3 in a test case.

< Figure 6. Different scenarios with constraints >

Greedy algorithms and heuristic search algorithms may address constraints. On the other hand, algebraic techniques may have difficulty addressing constraints. Section 5 discusses constraints in more depth.

3.2 Algorithms for Higher Strength Combinatorial Test Suites

Algorithms for higher strength combinatorial tests face the challenge that the number of *t*-tuples to cover increases exponentially as *t* increases. For instance, Table 11 below shows four sample inputs and the number of *t*-tuples to cover for *t*=2..*k*. For instance, the input 3^{13} has 702 2-tuples, 7,722 3-tuples,47,915 4-tuples and goes up to 1,594,323 13-tuples! This poses challenges for algorithms, particularly in terms of time to generate test suites and the amount of memory used for computations.

	$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$	10^4	3^{13}	11^{16}
t=2	1,320	600	702	14,520
t=3	18,150	4,000	7,722	745,360
t=4	157,773	10,000	57,915	26,646,620
t=5	902,055	-	312,741	703,470,768
t=6	3,416,930	-	1,250,954	1,301,758,600
...	...	-	...	...
t=k	3,628,800	10,000	1,594,323	45,949,729,863,572,200

Table 11. Higher strength covering array examples

The categories of algorithms to generate test suites for higher strength combinatorial test suites include the same as those mentioned previously in this section: algebraic techniques, greedy algorithms, heuristic search algorithms, and CSP algorithms. While the trade-offs of these algorithms are the same as those mentioned earlier in this section for *t*=2, the amount of time and memory usage for higher strength coverage often require special consideration. For instance, Table 12 shows the size of the test suites and execution time for *t*=2 through *t*=6 coverage of the input 5^{10} using the IPOG algorithm [48]. The result for *t*=2 strength coverage results in a test suite of size 48 in .11 seconds while the algorithm produces 50,920 test cases for *t*=6 in 791.35 seconds.

t-way	2	3	4	5	6
Size	48	308	1843	10119	50920
Time	0.11	0.56	6.38	63.8	791.35

Table 12. IPOG test size and runtime

The major challenges faced by algorithms to generate combinatorial test suites include the time to generate test suites, the size of the test suites, and ability to address seeding and constraints. Numerous algorithms exist to generate combinatorial test suites, with popular categories of algorithms including algebraic, greedy heuristic, and heuristic search algorithms. Testers must consider their particular domain and testing environment when selecting an algorithm to generate covering arrays. Further, testers may seek guidance by visiting website that maintain the best known reported solutions or sizes for many inputs [42, 43, 44].

3.3 Example Tools

A variety of tools can be found on the web site *pairwise.org*, including both commercial and open source. Two of the most widely used covering array generators are Microsoft PICT [106][48] and NIST ACTS [45]. In this section we review ACTS. Tools may vary in features [50][51][52][53][54], and test environments are beginning to make it possible to integrate tools in ways that testers find most useful. One of the most well-developed such frameworks is CITlab [55], which is integrated with the Eclipse editor and provides a means of defining domain-specific languages and connecting other Eclipse plugins.

ACTS is a freely distributed set of research tools for software testing downloadable from a NIST web site [45]. The IPOG algorithm in ACTS generates combinatorial *t*-way test suites for arbitrary combinatorial test structures and any strength *t* with support of constraints (to exclude invalid combinations). CCM (for Combinatorial Coverage Measurement) is a research tool in ACTS for determining combinatorial coverage of a test suite which may not have been developed from a combinatorial viewpoint.

(1) IPOG excludes those combinations of the test settings which are invalid according to the user specified constraints. (2) IPOG tool supports two test generation modes: scratch and extend. The former builds a test suite from the scratch, whereas the latter allows a test suite to be built by extending a previously constructed test suite which can save earlier effort in the testing process. (3) IPOG tool supports construction of variable-strength test suites. For example, of the 10 test factors all could be covered with strength 2 and a particular subset of 4 out of 10 factors (which are known to be inter-related) could be covered with higher strength 4. (4) IPOG tool verifies whether the test suite supplied by a user covers all *t*-way combinations. (5) IPOG tool allows the user to specify expected output for each test case in terms of the number of output parameters and their values. (6) IPOG tool supports three interfaces: a Graphical User Interface (GUI), a Command Line Interface (CLI), and an Application Programming Interface (API). The GUI interface allows a user to perform most operations through menu selections and button clicks. The CLI interface can be more efficient when the user knows the exact options that are needed for specific tasks. The CLI interface is also very useful for scripting. The API interface is designed to facilitate integration of ACTS with other tools (see Fig. 7 below).

< Fig 7 about ¼ page >
< Figure 7. ACTS/IPOG user interface >

ACTS/CCM

CCM [38] measures the combinatorial coverage of a test set, including the measures discussed in Sect. 2.3. It accepts a comma-separated values (CSV) format file where each row represents a test and each column represents a parameter. CSV files can be exported from any spreadsheet program. Constraints can be specified at the beginning of the file, in the same syntax used for ACTS/IPOG; each line will be considered a separate constraint. The invalid combinations will be shown if there are constraints specified. If

any coverage measurement has been specified the invalid combinations will be generated. For continuous-valued parameters, a user can specify equivalence classes by indicating the number of value classes and boundaries between the classes. Boundaries may include decimal values. Where the boundary between two classes c_1 and c_2 is x, the system places input values $< x$ into c_1 and values $\geq x$ in c_2. CCM outputs a graphic display of coverage, as shown in Sect. 2.3, and complete statistics on tests, number of t-way combinations and combination settings needed for full coverage; number of t-way combination settings covered, and invalid combinations as determined by constraints. CCM was developed by NIST and the Centro Nacional de Metrologia, Mexico. The user interface is shown in Fig. 8.

< Fig 8 about ¼ page >
< Figure 8. ACTS/CCM user interface >

4 Input Partitioning and Representation

Covering array algorithms can produce highly compact matrices of t-way value combinations, but how does a tester decide what values to use? Input space partitioning is a critical step in any software test approach, and traditional input modeling and partitioning methods are equally effective for combinatorial testing, but some aspects of combinatorial methods introduce differences from this step in conventional approaches.

4.1 Combinatorial Issues in Input Model Development

When applied to developing input models for a covering array tool, some issues become particularly important – three general classes of combination anomalies known as *missing combinations*, *infeasible combinations* and *ineffectual combinations* [58]. The efficiency of combinatorial testing stems partly from the fact that an individual test covers C(n,t) combinations, so a covering array can compress a large number of combinations into a reasonably small set of tests. Combination anomalies add complications to the generation of covering arrays.

Missing combinations are those that may be required to trigger a failure, but which are not included in the test array. A t-way covering array will of course include all t-way combinations, and some (t+1)-way combinations, but not all (or it would be a (t+1)-way covering array. If there are some combinations that engineering judgment leads testers to believe are significant, they may be included specifically, to supplement the covering array.

Infeasible combinations are extremely common, and are addressed by constraints. These are combinations which will never occur when the application is in use, perhaps because they are physically impossible. These combinations cannot be handled by simply removing tests that contain them, because many other, potentially necessary, combinations would be removed at the same time. Thus constraints are used to prevent the production of infeasible combinations in the covering array. For example, if

parameter B must have the value ‘100’ whenever parameter A = 100, then a constraint such as “A=100 → B=100” can be included in constraints processed by the covering array generator. Sect.4.3, below, discusses this process in detail.

Ineffectual combinations may occur when the presence of another combination causes them to be ignored by the application [57][58]. A common scenario is when an error value causes the application to stop, so other combinations in the same test are not included in processing. This situation is often handled by separating tests for error processing from other tests [59]. Ineffectual combinations may also result when there are dependencies among combinations, which may be handled with constraints, as with infeasible combinations.

It is important to keep in mind that the anomalies discussed above can occur with any test method. For example, a test that triggers an error may terminate the application, so other possible interactions caused by values in the failing tests will never be discovered. It is simply that most methods do not consider parameter interaction to the same degree of granularity as combinatorial testing. Using combinatorial methods helps to expose anomalies that may reduce the effectiveness of other methods.

4.2 Size Considerations with Covering Arrays

Key cost factors are the number of values per parameter, the interaction strength, and the number of parameters. The number of tests produced increases with $v^t \log n$, so the number of values per parameter is a critical consideration. Guidance for combinatorial methods usually recommends keeping the number of values per parameter to a limit of roughly 10. The number of parameters is much less significant for this approach, as the test set size increases with log n, for n parameters, and current tools make it possible to generate covering arrays for systems with a few hundred parameters, at least for lower strength arrays. For larger systems, random test generation may be used. If there are no constraints among variables, random generation makes it possible to probabilistically cover t-way combinations to any desired level [61]. In the more common case where there are constraints, a large test set may be generated randomly, then its combinatorial coverage measured while ensuring maintenance of the constraints [38].

4.3 Modeling Environment Conditions and State

When we perform input parameter modeling, it is important to consider environment conditions that often affect the behavior of a system. There are two types of environment condition, i.e., static and dynamic conditions. Static conditions are set before a system is put into execution and do not change over time. Examples of static conditions include the hardware or software platform, system configurations, and environment variables. For instance, a web application may behave differently depending on whether it is running on a desktop machine or a mobile device like a smart phone. In this case, the runtime platform can be modeled as a parameter in the input model and the different types of platform the web application is designed to run can be modeled as different values of the parameter. As another example, many software applications allow the user to customize

their behavior using a configuration file. Each configuration option can be modeled as a parameter in the input model and the different choices for a configuration option can be modeled as different values of the corresponding parameter.

Dynamic conditions capture the state of the environment that changes as a system runs. For example, many systems use a database to manage their data. The same operation may behave differently depending on the state of the database. Thus, it is important to model the state of the database as a factor that could affect the system behavior. This can be accomplished by identifying a set of abstract parameters that capture some important characteristics of the database. As another example, in an object-oriented system, the behavior of a method often depends on the state of the object on which the method is invoked. Consider that a class manages a *set* of objects that does not allow duplicates. The class provides a method named *insert* that can be used to add an object into the set. This method would behave differently depending on whether the object already exists in the set. Thus, it is important to model the state of the set, which can be accomplished by identifying an abstract parameter that indicates whether the object to be added already exists in the set.

4.4 Types of Constraints for Input Parameter Model

After parameters and values are identified, another important part of input parameter modeling is to identify potential constraints that may exist between different parameters and values. Constraints are restrictions that must be satisfied by each test; otherwise, a test may be rejected by the system under test and thus would not serve the purpose. Similar to identification of parameters and values, constraints can be identified from various sources of information, e.g., requirement document, and domain knowledge.

Different types of constraint can be classified based on different dimensions. A constraint can be an environment or system constraint, depending on whether it is imposed by the runtime environment a system is designed to run or by the system itself. A constraint can be a first-order or higher-order constraint, depending on whether the constraint needs to be evaluated against individual tests or sets of tests. A constraint can be a temporal or non-temporal constraint, depending on whether the constraint specifies properties related to time. In the following, we discuss these different types of constraints in more detail.

4.4.1 Environment Constraints vs. System Constraints

Environment constraints are imposed by the runtime environment of the system under test (SUT). For example, tester may want to ensure a web application executes correctly in different web browsers running on different operating systems. In this scenario, tests are combinations of web browsers and operation systems. Safari 6.0 or later cannot be executed on Windows. If the web browser is Safari 6.0 or later, the operating system cannot be Windows. Therefore, no test should contain the combination of { Safari6, Windows }. In general, combinations that violate environment constraints could never occur at runtime and must be excluded from a test set.

System constraints are imposed by the semantics of the SUT. For example, a hotel reservation system may impose a constraint that the number of guests in a room must be no more than the number of available beds. Note that invalid combinations which do not satisfy system constraints may still be rendered to the SUT at runtime. If this happens, these invalid 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. In order to avoid potential mask effects, robustness testing often requires that each test contains only one invalid combination.

The key difference between environment constraints and system constraints is that environment constraints must be satisfied by all the tests whereas tests that do not satisfy system constraints may be generated for robustness testing.

4.4.2 First-order Constraints vs. Higher-order Constraints

First-order constraints are constraints that restrict parameter values in an individual test. For example, in a debit account transaction, the amount of money to withdraw from an account should be no more than either the balance of the account or the withdrawal limit of the account. In general, first-order constraints can be expressed using first-order logic expressions. Satisfaction of first-order constraints can be evaluated based on individual tests.

Higher-order constraints are constraints that impose certain restrictions on test sets or even sets of test sets, instead of on individual tests. Higher-order constraints can be more complex to understand and more expensive to evaluate. Constraints encountered in practice are typically no higher than second-order constraints. Many systems impose structural constraints, i.e., restrictions on the structure of test data. Structural constraints are typically higher-order constraints. For example, when we test software applications that access a database, we often need to populate some tables in the database as part of the effort to set up the test environment. These tables typically need to satisfy some structural constraints in order to ensure validity of the data that are stored in these tables. One common example is referential integrity, which requires that every data object referenced in one table must exist in some other table. Referential integrity is a second-order constraint, as it must be evaluated against a set of data objects, instead of individual data objects.

We note that most existing constraint solvers only handle first-order constraints. In order to handle higher-order constraints, a constraint solver customized for a particular domain is typically required. For example, a customized constraint solver may be developed to handle structural constraints that are commonly encountered in database testing.

4.4.3 Temporal constraints vs. Non-temporal Constraints

Temporal constraints impose restrictions on the temporal behavior exercised by a system. There are two types of temporal constraints, sequencing constraints and real-time constraints. Sequencing constraints specify the possible orders in which a sequence of

actions or events is allowed to take place. For example, a sequencing constraint may specify that a ready signal must be received before any operation is performed on a device.

Real-time constraints specify temporal properties with an explicit reference to time. For example, a real-time constraint may specify that an event must take place 5 milliseconds before another event takes place. This is in contrast with sequencing constraints, which specifies temporal properties using relative timing, i.e., without an explicit reference to time. That is, a sequencing constraint may specify that an event E must happen before another event E', but it does not specify how much time event E should occur before E'.

Non-temporal constraints are properties that are not related to time. Existing work on combinatorial testing has been mainly focused on non-temporal constraints. This is partly because temporal constraints involve the extra dimension of time and are thus more difficult to handle.

5. Constraints Handling in Covering Array Algorithms

In practice, covering array algorithms must be able to process constraints imposed by real-world considerations. The way in which constraints are represented can have significant impacts on algorithm and tool performance.

5. 1 Representations of Constraints

Constraints identified in an input parameter model must be specified in a way that allows them to be automatically processed. One common approach to specifying constraints is representing them as a set of forbidden tuples, i.e., combinations that are not allowed to appear in any test. A test is valid if and only if it does not contain any forbidden tuple. For example, Fig. 9 shows a system consisting of three Boolean parameters A, B, C and two user-specified forbidden tuples {A=0, C=0} and {B=0, C=1}. Test {A=0, B=0, C=0} is invalid since it contains forbidden tuple {A=0, C=0}. Test {A=0, B=0, C=1} is invalid since it contains forbidden tuple {B=0, C=1}. Test {A=0, B=1, C=1} is valid since it doesn't contain any forbidden tuple. When there are a large number of forbidden tuples, it can be difficult for the user to enumerate them.

<Figure 9. Example of Invalid and Valid Tests>

Alternatively, constraints can be specified using logic expressions. A logical expression describes a condition that must be satisfied by all the tests. A test is valid if and only if it satisfies all the logic expressions. Consider the system in Fig. 9, where the forbidden tuples can be represented by two logic expressions, (A=0) => (C!=0) and (B=0) => (C!=1). For complicated systems, logical expressions are more concise than explicit enumeration of forbidden tuples.

5. 2 Major Approaches to Constraint Handling

Due to the existence of constraints, some parameter values cannot be combined in the same test. In this case, a conflict is said to exist between these parameter values. There are four general approaches [70] to constraint handling for constructing covering arrays, including *abstract parameters*, *sub-models*, *avoid*, and *replace*. These approaches employ different strategies to deal with potential conflicts between parameters.

The *abstract parameters* and *sub-models* approaches remove conflicts from the input parameter model by means of model transformation prior to actual test generation. The *avoid* approach makes sure that only conflict-free tests are selected by checking validity of each test during actual test generation. The *replace* approach removes conflicts from a test set that has already been generated by replacing invalid tests with valid ones.

5.2.1 The Abstract Parameters Approach

In the *abstract parameters* approach, the original input parameter model that contains conflicts is transformed to one without conflicts prior to actual test generation. The main idea is to use one or more abstract parameters to represent valid sub-combinations of input parameters. First, conflicting parameters, i.e., parameters that contain one or more conflicts, are identified. Second, abstract parameters are introduced to replace these conflicting parameters. Each abstract parameter is used to represent a group of conflicting parameters. The values of an abstract parameter represent valid combinations of the corresponding conflicting parameters that satisfy the given coverage goal.

For example, assume that there exists a constraint, A > B, between two parameters A and B of the system shown in Fig. 10. For 2-way testing, a new input parameter model can be created by replacing these two parameters with a new abstract parameter AB whose domain consists of all the 2-way valid combinations of parameters A and B, i.e., (A=2, B=1), (A=3, B=1), and (A=3, B=2). A test generation algorithm that does not support constraints can be applied to this new model to create a 2-way test set for this example.

<Figure 10. Example of Abstract Parameters >

The *abstract parameters* approach may lead to over-representation of some sub-combinations. Consider the example in Figure 10, the number of 2-way tests for abstract parameter AB and parameter C would be $3 \times 2 = 6$, where any valid combination between parameters A and B will occur twice. In fact, five tests are enough to satisfy 2-way coverage, in which one of the three sub-combinations between A and B only occurs once while each of the others two sub-combinations occurs twice. As a result, for systems with wide ranged parameters, it may create a test set that is too large unnecessarily.

5.2.2 The Sub-models Approach

Similar to the *abstract parameters* approach, the *sub-models* approach removes conflicts by transforming the original input parameter model. In this approach, an input parameter model containing conflicts is rewritten into two or more smaller conflict-free models. A test set is generated for each smaller model and the final test set is the union of all the test sets for the smaller models.

The key notion in the *sub-models* approach is called split parameter. A split parameter is a parameter that is involved in a conflict and that has the least number of values. After a split parameter is identified, the input parameter model is split into a number of sub-models, one for each value of the split parameter. Next, for each sub-model, two-parameter conflicts involving the value of the split parameter are eliminated by removing values of the other parameters involved in the conflicts. Note that conflicts involving more than two parameters can be reduced to conflicts involving two parameters.

Again, consider the example in Fig. 10. Parameters A and B are involved in the conflicts. Suppose that parameter B is used as the split parameter. The input parameter model is split into three sub-models, one for each value of B. Then conflicts are eliminated in these three sub-models, which are shown in Fig. 11.

<Figure 11. Example of Sub-models>

Note that if conflicts still exist in the sub-models that do not involve the split parameter, the process is applied recursively. When all sub-models are conflict-free, some sub-models can be merged. A merge is possible if two sub-models differ only in one parameter.

When no further merges can be done, tests are generated for each sub-model. The final test set is the union of the tests generated for each sub-model. Consider the example in Fig. 11, and generate 2-way tests. For sub-model 1, there will be $2 \times 2 = 4$ tests; for sub-model 2, there will be 2 tests; for sub-model 3, there will be 0 tests. The union of these 6 tests is the final test set.

Similar to the *abstract parameters* approach, this approach may create test sets that are unnecessarily large. This is because parameter values that are not involved in any conflict will be included in every sub-model, which may create overlapping tests that do not contribute to test coverage. In Fig. 11, suppose that {A=3, B=1, C=2} from sub-model 1 is the last test (of the six possible tests) to be included in the final test set. This test doesn't contribute to coverage, since all the 2-way combinations covered by this test have been covered by other tests. That is, {A=3, B=1} has already been covered by test {A=3, B=1, C=1} from sub-model 1, {B=1, C=2} has already been covered by test {A=2, B=1, C=2} from sub-model 1, and {A=3, C=2}has already been covered by test {A=3, B=2, C=2} from sub-model 2.

5.2.3 The Avoid Approach

The *avoid* approach does not perform any transformation on the input parameter model. Instead, it tries to avoid conflicts when tests are actually generated. The key idea is to avoid generating invalid tests, i.e., tests that do not satisfy all the constraints. This is accomplished by checking the validity of each candidate test and discarding tests that do not pass the check. More discussion on how to perform validity check is provided in Section 5.c.

Compared to the *abstract parameters* and *sub-models* approaches, the *avoid* approach often produces a smaller test set. This approach, however, is mainly applicable to greedy methods, which construct a test set one value at a time. That is, greedy methods consider a test set to be a matrix of values and a test set is built by choosing each value one at a time. The *avoid* approach cannot be directly applied to algebraic methods, which construct a test set based on some mathematic formulas without the notion of selecting a test over multiple candidates.

5.2.4 The Replace Approach

The *replace* approach allows conflicting tests, i.e., tests that contain conflicts, to be created in a test set. Conflicting tests are then removed from the test set by replacing them with conflict-free tests while preserving test coverage of the test set. Note that conflicting tests cannot be simply discarded. This is because some combinations that do not have a conflict may be covered by these conflicting tests only. In this case, new conflict-free tests must be created to cover these combinations in order to preserve test coverage.

One approach to replacing an invalid test is to make multiple clones of the invalid test, each of which changes one or more of the parameter values to remove the conflicts from the test. The number of clones is chosen according to the strength of coverage and the number of conflicting parameters, in order to make sure that test coverage is preserved. If multiple conflicts exist in the same test, conflicts are removed one at a time via cloning, until all conflicts are eliminated.

Table 13 shows how to apply the *replace* approach to build a 2-way test set for the system in Fig. 10. Since it is 2-way coverage and any conflict only involves two parameters, two clones are created for each of the conflicting tests. For each pair of clones, the value of the first parameter involved in the conflict is changed in the first clone and the value of the second parameter involved in the conflict is changed in the second clone. For example, T1 consists of a conflict {A=1, B=1}. This conflict is removed by replacing T1 with T1a in which the value of A is changed from 1 to 2. Finally, after removing the invalid and redundant tests, a test set consisting of T1a, T2a, T4, T5a and T7 is found.

Tests Ignoring Constraints	**Parameters**		
	A	**B**	**C**
T1	**1**	**1**	2
T2	**1**	**2**	1
T3	**1**	**3**	2
T4	2	1	1
T5	**2**	**2**	2
T6	**2**	**3**	1
T7	3	1	2
T8	3	2	1
T9	**3**	**3**	2

Tests Cloned	**Parameters**		
	A	**B**	**C**
T1a	*	1	2
T1b	1	*	2
T2a	*	2	1
T2b	1	*	1
T3a	*	3	2
T3b	1	*	2
T4	2	1	1
T5a	*	2	2
T5b	2	*	2
T6a	*	3	1
T6b	2	*	1
T7	3	1	2
T8	3	2	1
T9a	*	3	2
T9b	3	*	2

Tests Replaced	**Parameters**		
	A	**B**	**C**
T1a	2	1	2
T1b	1	-	2
T2a	3	2	1
T2b	1	-	1
T3a	-	3	2
T3b	1	-	2
T4	2	1	1
T5a	3	2	2
T5b	2	1	2
T6a	-	3	1
T6b	2	1	1
T7	3	1	2
T8	3	2	1
T9a	-	3	2
T9b	3	1	2

Table 13. Application of the replace approach to the system in Fig. 10. Conflicts are highlighted in each test. "*" indicates a value that needs to be changed; "-" indicates that no value can be assigned.

Similar to the *abstract parameters* and *sub-models* approaches, the *replace* method may create test sets that are unnecessarily large. The reason is that clones are often overlapping. Thus, some combinations are covered more than once. This may create redundant tests that can be removed without compromising test coverage.

Note that the three approaches, i.e., *abstract parameters*, *sub-models*, and *replace*, may create unnecessarily large test sets. Test sets generated by these approaches can be reduced by removing redundant tests, i.e., tests that can be removed without compromising test coverage. This can be accomplished by processing the tests one by one and discarding tests that do not cover new combinations.

5.3 Validity Checking for the Avoid Approach

A key step in the *avoid* approach is to perform validity checking, i.e., checking whether all the constraints are satisfied for a given test. In general, there are two ways to check the validity of a test, including constraint solving and forbidden tuples.

The way in which validity checking is performed is independent from the way in which constraints are specified. Constraints specified using forbidden tuples can be converted into a set of logic expressions, which can be handled using a constraint solver. Similarly, a set of forbidden tuples can be derived from constraints specified using logic expressions and can be handled using a forbidden tuple-based approach.

5.3.1 Constraint Solving Based Validity Checking

In this approach a constraint solver is typically employed to perform validity checking. The main idea is to encode the problem of validity checking as a constraint satisfaction problem. Each time when a parameter value is to be assigned in a test, it must pass a check performed by the constraint solver to ensure all the constraints are satisfied.

The main challenge of this approach is dealing with the fact that the constraint solving process can be time-consuming, especially when constraints are complex. In particular, existing constraint solvers are designed to check satisfiability of individual formulae. That is, they typically do not exploit information from the solving history to speed up constraint solving that may be performed in the future.

Several approaches have been reported aiming to optimize the use of constraint solvers in the context of combinatorial testing [66, 67]. For example, an algorithm called IPOG-C [66] is developed that tries to reduce the number of calls to the constraint solver. In particular, algorithm IPOG-C reduces the number of validity checks on target combinations by leveraging the fact that if a test is determined to be valid, then all the combinations covered by this test would be valid, and thus do not have to be explicitly checked. In case that a call to the constraint solver cannot be avoided, algorithm IPOG-C tries to simplify the solving process as much as possible. It divides constraints into non-intersecting groups to reduce the number of constraints that have to be checked during a

validity check.

5.3.2 Forbidden Tuples Based Validity Checking

An alternative approach to performing validity checking is to ensure that no forbidden tuple is contained in the test. As discussed above, forbidden tuples can be used to verify if a (complete) test is valid or not. However, a partial test that contains no forbidden tuples may be invalid. Consider the example shown in Fig. 12. A partial test {A=0, B=0} is invalid even when it includes no forbidden tuples, because we cannot later assign a valid value for parameter C to make a complete test.

<Figure 12. Example of Invalid and Valid Partial Tests>

Generally speaking, we cannot directly use forbidden tuples to check a partial test's validity. This is because user-specified forbidden tuples may imply more other forbidden tuples that are not explicitly specified. A partial test that covers no explicit forbidden tuple may cover some implicit forbidden tuples. In Fig. 12, {A=0, B=0} is an implicit forbidden tuple, making the partial test {A=0, B=0} invalid.

It is not practical for the user to specify all implicit forbidden tuples in a system. Thus, it is desired to automatically derive all implicit forbidden tuples from a set of forbidden tuples given by the user. This would allow the validity of a partial test to be determined in the same way as for a complete test, i.e., by ensuring the partial test does not contain any implicit or explicit forbidden tuple. However, the number of forbidden tuples can be large, making this approach very inefficient.

The concept of minimum forbidden tuple [68] is proposed to address this challenge. Intuitively, a minimum forbidden tuple (MFT) is a forbidden tuple of minimum size. It is shown that if a tuple is invalid, it must cover at least one MFT. Thus, a partial test is valid if and only if it covers no MFT. This makes it possible to use MFTs to perform validity checks on both complete and partial tests. The number of MFTs is typically much smaller than the number of all possible forbidden tuples. Thus, the cost of managing forbidden tuples, in terms of both storage and lookup cost, can be significantly reduced.

The MFTs generation algorithm iteratively applies two processes, i.e., derive and simplify, on the set of forbidden tuples until it converges.
(Derive) Given a parameter *P* having *n* values as its domain, and *n* forbidden tuples each of which contains a different value of parameter *P*, a new forbidden tuple can be constructed by combining all values in these *n* forbidden tuples, excluding the values of parameter *P*.
(Simplify) A tuple within the set of forbidden tuples can be removed if it covers any other forbidden tuple in the set.

The MFTs generation algorithm starts from the set of explicit forbidden tuples. It iteratively derives new forbidden tuples and simplifies the set of forbidden tuples, until no new forbidden tuples can be derived. The final set of forbidden tuples consists of all

MFTs which explicitly indicate all the constraints implied by user-specified forbidden tuples and parameter domains. We use an example shown in Fig. 13 to describe how it works. In steps 1 and 2, three new forbidden tuples are derived using parameter A and B. There are no new forbidden tuples can be derived using parameter C, so we move to the simplify process, as in step 3, but no tuples can be removed at this time. The next iteration then starts with the three new forbidden tuples which are marked with "*". In step 4 we derive a new tuple {C=0} using parameter A. There are no new forbidden tuples can be derived using parameter B and C, so we move to the simplify process, as in step 5, six forbidden tuples covering {C=0} are removed. Now there are only three forbidden tuples remaining in the set and no new tuples can be derived from them. They are MFTs and can be used to perform validity checking.

<Figure 13. Example of MFTs Generation Process>

6 Case Studies

Combinatorial testing has found extensive use in software development, and a variety of examples for diverse industries can be found in the proceedings of the International Workshop on Combinatorial Testing [69]. The two highlighted in this section illustrate quite different aspects of this method in practice. The first, on the Document Object Model, is an interesting validation of the interaction rule and its implications. Tests covering 4-way combinations detected all faults found in complex real-world software that had previously been detected with exhaustive testing of discretized values. The second example below is, to our knowledge, the largest published study on industrial use of combinatorial testing, a 2.5 year investigation of the method to aerospace software that demonstrated significant cost savings and improved test coverage.

6.1 Document Object Model

The Document Object Model (DOM) [71][72] is a World Wide Web Consortium (W3C) standard for representing and interacting with documents through web browsers. DOM makes it easier for developers to incorporate non-sequential access in web sites by providing conventions for updating the content, structure, and style of documents dynamically. Implemented in browsers, DOM components typically include tens of thousands of lines of source code. Because of its importance to internet applications worldwide, developed the DOM Conformance Test Suites, to assist developers in ensuring interoperability and predictable behavior of web site components. The conformance tests are comprehensive, providing exhaustive testing (all possible combinations) of discretized values for 35 DOM events, a total of more than 36,000 tests. Multiple commercially produced DOM implementations were tested.

Since the DOM test suite was designed for exhaustive testing, it provided a unique opportunity to evaluate one of the major advantages of combinatorial testing – the empirical interaction rule that faults involve a small number of factors interacting, so covering all *t*-way faults, for small value of *t*, can be nearly as effective as exhaustive testing. Five new DOM test suites were created, covering 2-way through 6-way

combinations, to compare the effectiveness of *t*-way combinatorial testing with the original exhaustive test suite [73]. According to the interaction rule, testing all *t*-way combinations, for a suitable value of *t*, should be as effective as exhaustive testing of discretized values. Results, shown in Table 14, were consistent with the rule. At $t = 4$, the combinatorial test suite detected all DOM faults discovered in exhaustive testing.

***t*-way**	**Tests**	**Pct Original**	**Test Results**	
			Pass	**Fail**
2 Way	702	1.92%	202	27
3 Way	1342	3.67%	786	27
4 Way	1818	4.96%	437	72
5 Way	2742	7.49%	908	72
6 Way	4227	11.54%	1803	72

Table 14. Comparison of *t*-way with exhaustive test set size.

Several interesting observations can be made about these results. Notice that 2-way tests detected only 37.5% of the faults, pairwise testing is clearly inadequate for this application, and even 3-way tests detected no additional faults. However, with 4-way covering arrays, all faults found in exhaustive testing were discovered, with less than 5% of the original test set size. This is an enormous savings, particularly for a user-interface related application such as DOM, where human involvement is required to verify test results involving images on a screen. We can also observe another aspect of these results consistent with the observations made in the introduction to this chapter. While the distribution of 1-way and 2-way faults was broad (e.g., under 20% to more than 90% for 1-way), a very narrow distribution was observed for 4-way to 6-way faults. In other words, empirical data suggest that results could be quite varied for 1-way, 2-way, and even 3-way covering arrays. On the other hand, when we reach *t*-way strengths of 4-way and beyond, fault detection should be both stronger and more consistent across applications. The DOM testing results are an example of such a situation.

6.2 Lockheed Martin

Lockheed Martin is one of the world's largest aerospace firms. In 2005, the company began investigating application of pairwise testing to improve test effectiveness and reduce costs [74, 75]. This work led to discussions with NIST, and subsequently a Co-operative Research and Development Agreement (CRADA) to evaluate the cost/benefit tradeoffs and areas of suitable application for combinatorial testing of complex industrial software [76]. (One of the ways in which NIST conducts joint research with US industry is through CRADAs, which allow federal laboratories to work with US industry and provide flexibility in structuring projects, intellectual property rights, and in protecting industry proprietary information and research results.)

The pilot project objectives included: investigating CT across multiple application areas, including system, software, and hardware testing; estimating possible improvements in fault detection with combinatorial methods; and quantifying possible reductions in test

cost and overall lifecycle cost through earlier fault detection. The ACTS tool was used, supplemented with other tools that provided complementary capabilities, including: Air Academy Associates: SPC XL, DOE KISS, DOE PRO XL, DFSS MASTER; Phadke & Associates: rdExpert; and Hexawise's web-based Hexawise tool.

A diverse set of eight pilot projects were included in the evaluation, spanning a cross-section of the company's mission areas:

- F-16 Ventral Fin Redesign Flight Test Program – system-level problem analysis, comparing with historical results [75]
- Electronic Warfare (EW) system testing – evaluating and extending existing tests
- Navigation Accuracy, EW performance, Sensor information, and Radar detection – generating test cases for subsystems
- Electromagnetic Effects (EMI) Engineering - CT tests were compared with tests developed using conventional methods
- Digital System Command testing –file function testing with multiple parameters
- Flight Vehicle Mission Effectiveness (ME) – comparing CT with tests generated from a statistical analysis tool
- Flight Vehicle engine failure modes – CT tests were compared with tests developed using conventional methods
- Flight Vehicle engine upgrade –combinations of flight mode factors were compared with existing tests

Pilot projects found CT effective for reducing the number of tests, and for improving test coverage [76]. While there was some variation among projects, the company estimated that CT would reduce testing cost by roughly 20%, while providing 20% - 50% better test coverage. In some cases, significant but previously undiscovered bugs were found. As a result of this experience, Lockheed Martin established a process to encourage adoption of combinatorial methods in company projects, documented lessons learned and developed recommendations for the testing community at large.

7 Advanced Topics in Combinatorial Testing

As CT has evolved in practice, new opportunities and challenges have been identified. This section reviews research in a number of specialized topics that are increasingly finding use for solving test and assurance problems.

7.1 Test Suite Prioritization

Test suite prioritization by combinatorial-based coverage has been studied from two perspectives. The first generates combinatorial test suites using inputs that contain weights on the parameter-values. The second takes an existing test suite and reorders the test cases by combinatorial-based interaction coverage.

7.1.1 Generation of Prioritized Test Suites by Combinatorial-based Coverage

Test suites that are generated by combinatorial-based coverage use an ℓ-biased covering, defined as:

> A ℓ-*biased covering array* is a covering array *CA*(*N*; 2, *k*, *v*) in which the first rows form tests whose utility is as large as possible according to some criterion. That is, no *CA*(*N*; 2, *k*, *v*) has rows that provide larger utility according to the chosen criterion.

We refer to an ℓ-*biased covering array* as a prioritized combinatorial test suite. To generate a prioritized combinatorial test suite, a tester must carefully assign weights to parameters and their values. The weights are assigned a value between 0 (low priority) to 1 (high priority). A test then computes the weights of pairs by multiplying their weights. For instance, assume we have a pair with weights .2 and .1. The total weight is then .2*.1 = .02. The goal of the algorithm is then to cover as much "weight" among pairs as soon as possible rather than simply covering pairs. As discussed in the section on Algorithms, there are many possible categories of algorithms that are able to generate covering arrays and they may certainly be modified to cover weight as needed for ℓ-*biased covering arrays.* Bryce et al. give one example that uses a greedy algorithm [77].

7.1.2 Prioritization of Existing Test Suites by Combinatorial-based Coverage

Test suite prioritization by combinatorial-based coverage has been applied to Event Driven Systems, focusing on combinations of parameter-values on or across windows. In this section, we briefly discuss this test suite prioritization problem and then give an example.

The Test Suite Prioritization problem is defined by Rothermel et. al. [80]:

> Given T, a test suite, Π, the set of all test suites obtained by permuting the tests of T, and f, a function from Π to the set of real numbers, the problem is to find $\pi \in \Pi$ such that $\forall \pi' \in \Pi, f(\pi) \geq f(\pi')$. In this definition, Π refers to the possible prioritizations of T and f is a function applied to evaluate the orderings.

Example: Consider the case of test suite prioritization for a web application in which the source of the test cases is a set of user-sessions. Figure 14 shows that users connect a website where their actions (POST/GET requests) are recorded by a webserver. A tool converts of these user visits to a test case. Given that there are a large number of test cases, we then prioritize these test cases according to a criterion. In the case of combinatorial-based coverage criteria for GUI and web applications, intra-window and inter-window interactions have been proposed and empirical studied.

< Figure 14. Test suite prioritization example >

For instance, consider the example input shown below where we have three webpages that have the parameters and values as shown in Table 15. We will prioritize by inter-

window combinatorial-based coverage. That is, combinations of parameter-values between pages.

Page	Values for parameter 1	Values for parameter 2	Values for parameter 3
Page 1	0,1,2,3	4,5	
Page 2	6	7,8	9
Page 3	10, 11	12	

Table 15. Example web interface parameter values

Next consider that we have the following test cases that visit some of these pages and specify values for parameters.

Test	Test Case
1	0,4,6,8,11
2	0,6,10
3	4,6,8,11

Table 16. Test cases for web pages

Give the input and test cases, Table 17 shows the inter-window pairs that are covered in these test cases. In this scenario, we select Test Case 1 as the first test case since it covers 8 pairs while the other test cases cover fewer pairs. We mark these pairs in Test Case 1 as covered and then select the next case such that it covers the most remaining “uncovered pairs”. In this case, we select Test Case 2 since it covers 2 new pairs, but Test Case 3 does not cover any new pairs.

Test	Covered pairs	No. of pairs covered

1	(0,6)(0,8)(0,11)(4,6)(4,8)(4,11)(6,11)(8,11)	8
2	(0,6)(0,10)(6,10)	3
3	(4,6)(4,8)(4,11)(6,11)(8,11)	5

Table 17. Pairwise coverage of tests

Empirical studies have shown that prioritization by combinatorial-based coverage has been valuable for improving the rate of fault reduction in several studies. For instance, Bryce et al. studied seven systems and observed a trend that test suite prioritization by combinatorial-based coverage often improved the rate of fault detection for GUI and web applications [77]. A tool, CPUT, is freely available for testers to repeat this process with their own web logs. CPUT converts Apache web logs to user-session-based test suites and prioritizes those test suites by combinatorial-based coverage [81].

Test suite prioritization by combinatorial-based criteria has been investigated from two viewpoints: (1) generate test suites from scratch by incorporating the weights of *t*-tuples into the test generation process and (2) reorder existing test suites by a combinatorial-based coverage criterion. Existing work in this area is quite promising in regard to the ability to improve fault detection effectiveness. It is simple to incorporate weights into algorithms that generate combinatorial test suites, but testers must take care in assigning weights. If a tester has existing test suites, they may also prioritize by combinatorial-based coverage. Testers may use and extend the CPUT tool to apply test suite prioritization for user-session-based testing in their own domains [81].

7.2 Sequence covering arrays

Event sequences are important in many aspects of software testing [86, 87, 88, 89, 90, 94]. For example, a typical e-commerce web system presents a variety of controls to the user, such as buttons, text entry fields, selection lists, including many with sub-options such as pull-down menus. It should be possible for the user to engage these controls in any order with the system working correctly irrespective of the order used. Another example (in fact the application for which the methods described here were developed) is the process of plugging in various peripherals. If the developer has made assumptions about the order in which peripherals are connected and controls engaged, then a user who violates this expected sequence may encounter errors. Applications should work correctly regardless of the sequence of events selected by the user, or else indicate that a different order is required.

In many cases, the key factor in triggering a failure is whether a particular event has occurred prior to a second event, regardless of whether other events have occurred between these two. For example, the system may fail if a pump has been started before a

particular valve has been opened at some point, even though other events may have occurred in between. Sequence covering arrays were developed to locate faults of this type, using combinatorial methods to increase efficiency [91]. Tests based on these 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.

Definition. 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 [79]. 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.

Sequence covering arrays were introduced in [79] for software testing but were later shown to be equivalent to t-scrambling sets [92][93]. Margalit [95] provides closer bounds, and additional results and algorithms were presented in [96] and [97].

7.2.1 Example

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 18. 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 18. 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 19. 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 19 (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 19. 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.

With 10 events, the number of permutations is 10!, or 3,628,800 sequences for exhaustive testing. In that case, a 3-way sequence covering array requires only 14 tests to cover all 3-way sequences, and 72 tests are all that is needed for 4-way sequences.

7.2.2 Generating Sequence Covering Arrays

Any 2-way sequence covering problem requires only two tests. A 2-way sequence covering array can always be constructed by listing the events in some order for one test and in reverse order for the second test. See Table 20 for an example.

Test	**Sequence**
1	*a b c d e*
2	*e d c b a*

Table 20. 2-way sequence covering array.

Sequence covering arrays are related to covering arrays in covering *t*-way combinations, but there are significant limitations in producing SCAs from covering arrays [97]. Consequently specialized algorithms have been developed for SCAs, and are a continuing subject of research. For *t*-way sequence covering, t > 2, greedy methods are efficient and produce arrays with number of tests proportional to log *n*, for *n* events [91]. An improved greedy algorithm was developed by Erdem [96], producing fewer tests, and further results by Chee et al developed algorithms producing significantly smaller arrays than either [79] or [96], and results are provided up to strength 5.

Event sequences are encountered frequently in testing, and combinatorial methods are effective in reducing the testing burden, especially for applications that require human involvement for test setup or system configuration. Since the test array size grows only logarithmically with the number of events, *t*-way sequence coverage is practical in many applications. Areas for future research in sequence covering arrays include algorithms to

provide smaller test arrays, or in shorter time; measures of fault detection in practical application; and handling of constraints. Constraints are a particularly challenging issue with SCAs [91, 95] since even a small limitation on *t*-way sequences can severely limit the possible arrangements in the sequence covering array. Variations such as multiple occurrences of an event and missing events are also possible, so an additional question is how sequence covering arrays compare with other methods of event sequence testing, such as those based on finite automata or other approaches that are frequently used in protocol testing.

7.3 Fault localization

After executing a combinatorial test set, the execution status, i.e., pass or fail, of each test is obtained. When one or more tests fail, the next task is fault localization, i.e. identifying faults that cause the failure. The problem of fault localization can be divided into two sub-problems: 1) Identifying failure-inducing combinations. A combination is failure-inducing if its existence in a test causes the test to fail. 2) Identifying actual faults in the source code. A fault is a code defect that can be an incorrect, extra, or missing statement.

7.3.1 Identifying failure-inducing combinations

One naïve approach to identifying failure-inducing combinations is to execute all possible tests and then identify combinations that only appear in failed tests. This approach is, however, not practical as it requires exhaustive testing. In the literature, several approaches have been reported that try to identify failure-inducing combinations by executing only a small set of tests. These approaches are essentially approximate solutions. That is, failure-inducing combinations identified by these approaches are suspects, but not guaranteed, to be failure-inducing.

Existing approaches on identifying failure-inducing combinations can be largely classified into two groups. The first group of approaches takes as input a single test as well as its execution status and tries to identify failure-inducing combinations in the test. A simple solution involves checking every possible combination, one at a time, contained in the failed test. This solution is expensive due to the fact that the number of combinations contained in a test is an exponential function of the size of the test. Two efficient algorithms called FIC and FIC_BS are reported to quickly locate a failure-inducing combination by checking only a small number of possible combinations [10]. These two algorithms, however, make certain assumptions that may not be satisfied in practice. In particular, they assume that no new inducing combination is introduced when a value is changed to create a new test.

The second group of approaches takes as input a set of tests as well as their execution statuses and try to identify failure-inducing combinations that may appear in any of these tests. This group could further divided into two sub-groups. The approaches in the first sub-group identify failure-inducing combinations without adding any new test to the initial test set. For example, a machine learning-based approach was reported that uses a technique called classification tree to identify failure-inducing combinations [99]. Based on the execution result of a test set, this approach builds a classification tree that encodes information needed to predict status of a test execution. A

score is assigned to each combination that is likely to cause an execution to fail. If the combination's score is greater than a predefined threshold, the combination is marked as inducing.

The approaches in the second sub-group generate and execute additional tests to identify failure-inducing combinations. These approaches first identify suspicious combinations with respect to the initial test set. Suspicious combinations are combinations that appear in failed tests but not in passed tests. These combinations are candidates that may be failure-inducing. Then a small set of new tests is generated to refine the set of suspicious combinations.

One approach called AIFL [100] first identifies all the suspicious combinations in a test set. Next it uses a strategy called OFOT (One Factor One Time) to systematically change one value of the failed test at a time. Therefore, k new tests are generated for each failed test of size k. These new tests are executed to refine the suspicious combinations set. In particular, if a suspicious combination appears in any new test that passes, then this combination is removed from the suspicious set. This process can be repeated until a stable point is reached where the suspicious set does not change in two consecutive iterations [101].

Another approach implemented in a tool called BEN [102] ranks suspicious combinations, after they are identified, based on three notions of suspiciousness, including suspiciousness of component, combination, and environment. A component represents a parameter value. Suspiciousness of combination is computed based on suspiciousness of components that appear in the combination. Suspiciousness of environment with respect to a combination is computed based on suspiciousness of components that appear in the same test but not in the combination. The higher the suspiciousness of a combination and the lower the suspiciousness of its environment, the higher this combination is ranked.

The ranking of suspicious combinations allows the next step to focus on the most suspicious combinations. New tests are generated for a given number of top-ranked suspicious combinations to refine the set of suspicious combinations. A new test is generated for a top-ranked suspicious combination in a way such that it includes this suspicious combination while minimizing the suspiciousness of environment for this combination. If the new test fails, it is likely that this suspicious combination is a failure-inducing combination. Otherwise, this suspicious combination is not suspicious any more and is removed from the suspicious set. The process of ranking and refinement is repeated until a stable point is reached, e.g., the set of suspicious combination does not change in two consecutive iterations.

7.3.2 Identifying faults in the source code

The problem of how to identify faults in the source code is one of the most studied problems in software engineering. Many approaches have been reported and can be applied after combinatorial testing [104][105]. For example, spectrum-based approaches try to identify faults by analyzing the spectrums of passed and failed test executions. The key idea behind spectrum-based approaches is that faults are more likely to be exercised in failed test executions than in

passed executions, which is independent from the way tests are generated. Thus, it is possible to apply spectrum-based approaches after combinatorial testing.

The BEN approach introduced in the previous section is later extended to locate faults in the source code [103]. The main idea of the BEN approach consists of leveraging the notion of failure-inducing combination to generate a group of tests that are very similar but produce different outcomes. Similar tests are likely to exercise similar execution traces. Different outcomes produced by similar tests are more likely due to existence of faults. In contrast, different outcomes produced by tests that are significantly different are more likely due to program logic.

Given a failure-inducing combination, BEN generates a group of tests that includes one failed test and several passed tests. The failed test is referred to as a core member and contains the failure-inducing combination while the suspiciousness of environment with respect to this combination is minimized. The passed tests are referred to as derived members and are derived from the core member by changing only one value of the core member. In other words, derived members differ from the core member in only one value but produce a different outcome.

For each statement, a suspiciousness value is computed by comparing the execution trace of the core member and each of the derived members. A statement is highly suspicious if it is only exercised in failed tests but not in passed tests. Statements are ranked based on a non-ascending order of their suspiciousness values. The higher a statement is ranked, the more likely it is considered to be faulty.

7.4 Relationship between combinatorial testing and structural coverage

Before an application is purchased or accepted, and especially when a system fails, one of the first questions that will be asked is "How well was it tested?" A variety of measures have been developed to answer this question, based on the extent and manner in which components of the system have been exercised. Code coverage is one component to the answer for this question, so it is natural to consider how combinatorial testing relates to code coverage. Do higher strength covering arrays produce greater code coverage? If so, at what rate does code coverage increase with increasing values of t? Additionally, what impact does the input model have on the relationship between covering array strength and coverage? We briefly review some of the more widely used measures, then consider results relating t-way testing to these measures.

- **Statement coverage is** the proportion of source statements exercised by the test set. Statement coverage is a relatively weak criterion, but provides a level of confidence that some basic testing has been done.

- **Decision or branch coverage is a measure** of branches that have been evaluated to both *true* and *false* in testing. When branches contain multiple conditions, branch coverage can be 100% without instantiating all conditions to true/false.

- **Condition coverage measures the proportion** of conditions within decision expressions that have been evaluated to both true and false. Note that 100% condition coverage does not

guarantee 100% decision coverage. For example, "`if (A || B) {do something} else {do something else}`" is tested with [0 1], [1 0], then A and B will both have been evaluated to 0 and 1, but the *else* branch will not be taken because neither test leaves both A and B false.

- **Modified condition decision coverage (MCDC)** requires that every condition in a decision in the program has taken on all possible outcomes at least once, each condition has been shown to independently affect the decision outcome, and that each entry and exit point have been traversed at least once [11].

Since *t*-way testing has been shown effective in detecting faults, we might expect it to generate a high level of code coverage as well. Although there are only a few studies regarding this question, results indicate that tests based on covering arrays can produce good code coverage, but the degree of coverage is heavily dependent on the input model used.

7.4.1 Basic Structural Coverage

Czerwonka [106] studied branch and statement coverage generated by covering arrays of tests for t=1 to t=5, including questions of how the minimum, maximum, and range of coverage varied with increasing strength. Also considered was whether *t*-way tests produced statistically significant differences in coverage as compared with basic test criteria such as all-values, and if any improvements in coverage with increasing t were the result of combinatorial effects or simply larger test suites. Four relatively small command line utilities were used in this study, with 100 different test suites for each level of *t*.

Consistent with early work on combinatorial testing, results in [106] showed that code coverage does increase as covering array strength increases, as intuition would predict. Additional interesting findings included:

- Statement and branch coverage generated by the test suites at *t*=2 and beyond were not extremely high, ranging from 64% to 76% for statement and 54% to 68% for branch.

- As covering array strength increased, the difference between minimum and maximum code coverage became narrower; thus higher strength test arrays produced better coverage and were also more stable in the level of coverage produced.

- Both statement and branch coverage increased significantly at *t*=2 as compared with all-values (*t*=1), but increases diminished rapidly with additional increases in *t*.

- The relationship between test suite size and covering array strength varied among the programs tested. For some, it appeared that improved coverage was not simply the result of additional tests at higher t levels, but in some other cases, test suite size, coupled with greater input combination diversity, was responsible for the improvement.

- The low levels of coverage may have been the result of factor and levels chosen for the covering arrays not sufficiently modeling the possible inputs for each program.

7.4.2 Effects of Input Model

The last point noted above may also explain the significant difference in coverage success shown in a different study that investigated the effectiveness of combinatorial testing for achieving MCDC coverage. Bartholomew [107][108] applied combinatorial methods in producing MCDC-adequate test suites for a component of software defined radio system, showing that tests based on covering arrays could produce 100% MCDC coverage. Recall that MCDC subsumes branch coverage, which in turn subsumes statement coverage, so full MCDC coverage means that statement and branch coverage were 100% as well. A key feature in the application of MCDC is that tests are constructed based on requirements. Achieving structural coverage is viewed as a check that the test set is adequate, i.e., the MCDC source coverage is not the goal in itself, only a metric for evaluating the adequacy of the test set.

In this study, a module of 579 lines was instrumented for branch and condition coverage, then tested with the objective of achieving MCDC requirements specified by the Federal Aviation Administration. Initial tests obtained results similar to those in [106], with approximately 75% statement coverage, 71% branch coverage, and 68% MCDC coverage. However, full branch coverage, and therefore statement coverage also, was obtained after "a brief period of iterative test case generation" [107], which required about four hours. MCDC, a substantially more complex criterion, was more difficult. In a few cases, obtaining complete MCDC coverage required construction of code stubs to force a particular sequence of tests, with specific combinations, to be executed. This process required two additional iterations, and a total of 16 additional hours. Complete test cases, based on covering arrays, were generated with a model checker, using the process described in [35]. This iterative process is consistent with the traditional use of the MCDC criterion as a check on test adequacy, as described previously. The integrated use of covering array based tests, with a model checker to determine expected results for each test, was found to be extremely successful in reducing testing costs for MCDC. A NASA report [116] indicates that achieving MCDC coverage often requires seven times the initial cost of code development, so the results reported in [107] suggest the potential for significant cost savings if replicated on larger systems.

7.5 Testing Very Large Systems

Thus far in this chapter we have discussed primarily combinations of input values, but the same combinatorial ideas can be used with configurations and software product lines. Such uses are increasingly common, as mobile applications and other types of software with extensive configuration options have proliferated. These systems are often referred to as *highly configurable* software [109]. Software product lines [110,111, 112, 113, 114] are a particularly interesting type of configurable system, where components may be assembled according to a user's feature specification, resulting in potentially millions of possible instantiations. Since a product line with 50 features that can be included or excluded will have 2^{50}, or roughly 10^{15}, possible instantiations, only a small proportion of these possible configurations will ever be built. Since it is naturally impossible to test even a fraction of this number of configurations, combinatorial methods have been used to make testing more tractable. Instead of testing all configurations, it may be practical to test all 2-way, 3-way, or higher strength interactions among features. One of the most significant differences with SPL testing is simply the number of variables that must be considered. For example, an SPL may have hundreds of features that can be selected, with many

more constraints than in other testing problems. One SPL is reported with 6,000 features [110]. Several techniques have been devised to deal with this scale of test design.

One of the key problems with applying combinatorial testing for a large number of variables is that covering array algorithms are often limited in the number of input variables that can be handled, and may be severely impacted by the presence of constraints. To process constraints, covering array generators often use Satisfiability Modulo Theory (SMT) constraint solvers. An alternative is to "flatten" the model to boolean values, then use boolean satisfiability (SAT) solvers.

A model can be "flattened" by systematically replacing variable values with boolean variables that represent a variable-value combination, with constraints to ensure that only one of the values (per variable) in the original model is selected. The process is straightforward: for each variable p_i with values $v_1, v_2, \ldots, v_k$, create k boolean variables that represent the selection of one of the k values for p_i. Then establish constraints as follows. We represent p_i set to value v_j as p_{ij}; thus boolean variables are $p_{i1} \ldots p_{ik}$.

- One constraint, $p_{i1} + p_{i2} + \ldots + p_{ik}$ ensures that at least one of the k values is selected
- One constraint for each pair of values to ensure that at most one of the k values is selected (where $\overline{x}$ represents x negated): $(\overline{p}_{i1} + \overline{p}_{i2}), (\overline{p}_{i1} + \overline{p}_{i3}), \ldots , (\overline{p}_{ik-1} + \overline{p}_{ik})$

For example, constraints can be flattened for the configuration in Table 2. In this example, if we have a Linux system to test, there should be no tests containing IE as the browser, since this combination will not be seen in practice. Thus there must be a constraint such as "Linux → !IE".

Parameter	Values
Operating system	XP, OS X, RHEL
Browser	IE, Firefox, Opera
Protocol	IPv4, IPv6
CPU	Intel, AMD
DBMS	MySQL, Sybase, Oracle

Table 21. Application configurations

Using the process described previously, we arrive at the following constraint set to prevent more than one operating system to be selected in the flattened model.

XP ∨ OX ∨ or RHEL	(1)
!XP ∨ !OSX	(2a)
!XP ∨ !RHEL	(2b)
!OSX ∨ !RHEL	(2c)

Constraint (1) ensures that at least one of the options is selected, and constraints 2a, 2b, and 2c prevent more than one from being selected at the same time. Thus this set of constraints preserves the original semantics that these are mutually exclusive options. Note that a large number of these constraints may be generated – for k options, we will need one constraint like (1)

above to ensure at least one option is selected, and C(*k*,2) constraints to ensure at most one is selected. However, modern SAT solvers have become extremely efficient, so this approach may work well. In some cases, it may be preferable to use the original model with an SMT solver, and in others a flattened model with SAT solver may perform better. The tradeoffs between these two approaches are an area of ongoing research [115].

[See below for Section 8: Future Directions]

Conclusions

Combinatorial testing has gained acceptance as a method to reduce cost and increase the effectiveness of software testing in many industries. The key insight underlying this method is that not every parameter contributes to every failure and most failures are caused by interactions between relatively few parameters. Empirical data indicate that software failures are triggered by only a few variables interacting (generally six or fewer). This finding has important implications for testing because it suggests that testing up to *t*-way combinations of parameters for small values of *t* can provide highly effective fault detection.

Industrial use has validated this conclusion. Combinatorial testing has seen tremendous growth in both industrial usage and research in the past 10 years. From an average of less than 10 papers a year prior to 2005, the field has grown to include an annual conference (since 2012) [69] and 100 or more papers a year in conferences and journals. Efficient covering array generation algorithms have been developed, and sophisticated tools have incorporated covering array algorithms with the capacity to process constraints that may be encountered in practical applications. As with any technology, extensions and new applications are being discovered, and the challenges introduced by these new uses are being addressed.

References

1. Robert Mandl (1985) “Orthogonal Latin squares: an application of experiment design to compiler testing” Communications of the ACM, 28, pp 1054-1058
2. Keizo Tatsumi (1987) “Test-case design support system” Proceedings of International Conference on Quality Control, Tokyo, pp 615-620
3. Keizo Tatsumi, S. Watanabe, Y. Takeuchi, and H. Shimokawa (1987) “Conceptual support for test case design” Proceedings of 11th IEEE Computer Software and Applications Conference, pp 285-290
4. Siddhartha R. Dalal, and C. L. Mallows (1998) “Factor-covering designs for testing software” Technometrics, 40, pp 234-243

5. D.R. Wallace, D.R. Kuhn, 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.
6. D.R. Kuhn, M.J. Reilly, 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 .
7. D.R. Kuhn, D.R. Wallace, and A. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Transactions on Software Engineering*, 30(6): 418-421, 2004
8. K. Z. Bell and Mladen A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. *Proceedings of the ITI Third IEEE International Conference on Information & Communications Technology*, pages 221–235, Cairo, Egypt, December 2005.
9. K.Z. Bell, Optimizing Effectiveness and Efficiency of Software Testing: a Hybrid Approach, PhD Dissertation, North Carolina State University, 2006.
10. Z. Zhang, and J. Zhang, "Characterizing failure-causing parameter interactions by adaptive testing," In Proceeding of ACM International Symposium on Software Testing and Analysis (ISSTA), 2011, pp. 331-341.
11. J. J. Chilenski, An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion, Report DOT/FAA/AR-01/18, April 2001, 214 pp.
12. Raghu N. Kacker, D. Richard Kuhn, Yu Lei, and James F. Lawrence (2013) "Combinatorial testing for software: An adaptation of design of experiments" Measurement, 46, pp 3745-3752
13. William G. Cochran and G. M. Cox (1950) Experimental Designs, New York: Wiley
14. Oscar Kempthorne (1952) Design and Analysis of Experiments, New York: Wiley
15. George W. Snedecor and W. G. Cochran (1967) Statistical Methods, Iowa State University Press
16. George E. P. Box, W. G. Hunter, and J. S. Hunter (1978) Statistics for Experimenters, New York: Wiley
17. Douglas C. Montgomery (2004) Design and Analysis of Experiments, 4th edition, New York: Wiley
18. R. A. Fisher (1925) Statistical Methods for Research Workers, Edinburgh: Oliver and Boyd
19. R. A. Fisher (1935) The Design of Experiments, Edinburgh: Oliver and Boyd
20. C. McQueary, "Using Design of Experiments for Operational Test and Evaluation," Memo, Office of the Secretary of Defense, May 2009, www.dote.osd.mil/pub/policies/2009/200905UsingDoEforOTE_MOA.pdf
21. Genichi Taguchi (1986) Introduction to Quality Engineering, White Plains New York: UNIPUB Kraus International
22. Genichi Taguchi (1987) System of Experimental Design, Vol. 1 and Vol. 2, White Plains New York: UNIPUB Kraus International (English translations of the 3-rd edition of Jikken Keikakuho (Japanese) published in 1977 and 1978 by Maruzen)
23. Genichi Taguchi (1993) Taguchi on Robust Technology Development, New York: ASME Press
24. Raghu N. Kackar (1985) "Off-line quality control, parameter design and the Taguchi method" Journal of Quality Technology, 17, pp 176-209

25. M. S. Phadke (1989) Quality Engineering using Robust Design, Englewood Cliffs New Jersey: Prentice Hall

26. C. R. Rao (1947) “Factorial experiments derivable from combinatorial arrangements of arrays” Journal of Royal Statistical Society (Supplement), 9, pp 128-139

27. Damaraju Raghavarao (1971) Constructions and Combinatorial Problems in Design of Experiments, Dover: New York

28. A. S. Hedayat, N. J. A. Sloan, and J. Stufken (1999) Orthogonal Arrays: Theory and Applications, New York: Springer

29. Neil J. A. Sloane (webpage) http://www2.research.att.com/~njas/oadir/

30. Shinobu Sato and H. Shimokawa (1984) “Methods for setting software test parameters using the design of experiments method (in Japanese)” Proceedings of 4th Symposium on Quality Control in Software, Japanese Union of Scientists and Engineers (JUSE), pp 1-8

31. Hiroki Shimokawa (1985) “Method of generating software test cases using the experimental design (in Japanese)” Report on Software Engineering SIG, Information Processing Society of Japan, No.1984-SE-040

32. Neil J. A. Sloane (1993) “Covering arrays and intersecting codes” Journal of Combinatorial Designs, 1, pp 51-63

33. James F. Lawrence, R. N. Kacker, Yu Lei, D. R. Kuhn, and M. Forbes (2011) “A survey of binary covering arrays” The Electronic Journal of Combinatorics, 18, P84

34. Jose Torres-Jimenez and E. Rodriguez-Tello (2012) “New bounds for binary covering arrays using simulated annealing” Information Sciences, 185, pp 137-152

35. D. Richard Kuhn, R. N. Kacker, and Yu Lei (2010) Practical Combinatorial Testing, NIST Special Publication 800-142 (http://csrc.nist.gov/groups/SNS/acts/documents/SP800-142-101006.pdf)

36. Jose Torres-Jimenez (webpage) http://www.tamps.cinvestav.mx/~jtj/CA.php

37. D.R. Kuhn, R. N. Kacker, and Yu Lei. Combinatorial Measurement Tool User Guide, Available online at http://csrc.nist.gov/groups/SNS/acts/documents/ComCoverage110130.pdf, Published on January 30, 2011 and last accessed on May 14, 2012.

38. Kuhn, D. R., Dominguez Mendoza, I., Kacker, R. N., & Lei, Y. (2013, March). Combinatorial coverage measurement concepts and applications. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (pp. 352-361).

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

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

41. Raghu N. Kacker, D. Richard Kuhn, Yu Lei, James F. Lawrence. Combinatorial testing for software: an adaptation of design of experiments, *Measurement*, vol. 46, no. 9, November 2013, pp. 3745-3752.

42. NIST Covering Array Tables, available online at: http://math.nist.gov/coveringarrays/ipof/ipof-results.html, accessed on 3/23/15

43. C. Colbourn. Covering Array Tables for t=2,3,4,5,6, available online at: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html, accessed on 3/23/15
44. J. Torres. Covering Arrays, available at: http://www.tamps.cinvestav.mx/~jtj/#, access on 3/23/15
45. Combinatorial Methods in Software Testing. National Institute of Standards and Technology, http://csrc.nist.gov/acts
46. M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering (ICSE 2003)*, pages 28–48, May 2003.
47. R. Bryce. Algorithms for Covering arrays, Arizona State University Ph.D. Dissertation, 2006.
48. J. Czerwonka, "Pairwise Testing in the Real World", http://msdn.microsoft.com/en-us/library/cc150619.aspx
49. Zhao, Y., Zhang, Z., Yan, J., & Zhang, J. (2013, March). Cascade: a test generation tool for combinatorial testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (pp. 267-270). IEEE.
50. Satish, P., Sheeba, K., & Rangarajan, K. (2013, March). Deriving Combinatorial Test Design Model from UML Activity Diagram. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (pp. 331-337). IEEE.
51. Qu, X., & Cohen, M. B. (2013, March). A study in prioritization for higher strength combinatorial testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (pp. 285-294). IEEE.
52. Arcaini, P., Gargantini, A., & Vavassori, P. (2014, March). Validation of models and tests for constrained combinatorial interaction testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 98-107). IEEE.
53. Wu, H., Nie, C., & Kuo, F. C. (2014, March). Test suite prioritization by switching cost. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 133-142). IEEE.
54. Farchi, E., Segall, I., Tzoref-Brill, R., & Zlotnick, A. (2014, March). Combinatorial Testing with Order Requirements. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 118-127). IEEE.
55. A. Gargantini, & P. Vavassori, (2012, April). Citlab: a laboratory for combinatorial interaction testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (pp. 559-568). IEEE.
56. Y. Lei, R. Kacker, D. Kuhn, V. Okun and J. Lawrence, IPOG/IPOD: Efficient Test Generation for Multi-Way Combinatorial Testing, *Software Testing, Verification, and Reliability*, vol. 18, no. 3, September 2008, pp. 125-148.
57. Segall, I., Tzoref-Brill, R., & Zlotnick, A. (2012, April). Simplified modeling of combinatorial test spaces. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (pp. 573-579). IEEE.
58. E. Miranda, Test Parameter Analysis. Chapter 5 in D. Kuhn, R.N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, CRC Press, 2013.

59. Segall, I., Tzoref-Brill, R., & Zlotnick, A. (2012, April). Common patterns in combinatorial models. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (pp. 624-629). IEEE.

60. N. Changhai, L. Hareton, A Survey of Combinatorial Testing, ACM Computing Surveys, 43(2): , 2014.

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

62. Kuhn, D. Richard, Raghu N. Kacker, and Yu Lei. "Introduction to Combinatorial Testing". CRC press, 2013.

63. Chays, David, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weber. "A framework for testing database applications." ACM SIGSOFT Software Engineering Notes 25, no. 5 (2000): 147-157.

64. Grindal, M. and Offutt, J. , "Input Parameter Modeling For Combination Strategies", Proceedings of the IASTED International Conference on Software Engineering (SE2007), Innsbruck, Austria, 13-15 Feb 2007, pages 255-260.

65. Vilkomir, Sergiy A., W. Thomas Swain, and Jesse H. Poore. "Software input space modeling with constraints among parameters." In Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International, vol. 1, pp. 136-141. IEEE, 2009.

66. Yu, Linbin, Yu Lei, Mehra Nourozborazjany, Raghu N. Kacker, and D. Richard Kuhn. "An efficient algorithm for constraint handling in combinatorial test generation." In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 242-251. IEEE, 2013.

67. Cohen, Myra B., Matthew B. Dwyer, and Jiangfan Shi. "Exploiting constraint solving history to construct interaction test suites." In Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007, pp. 121-132. IEEE, 2007.

68. Yu, Linbin, Feng Duan, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. "Combinatorial Test Generation for Software Product Lines Using Minimum Invalid Tuples." In High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on, pp. 65-72. IEEE, 2014.

69. IEEE, International Workshop on Combinatorial Testing, http://ieeexplore.ieee.org/xpl/conhome.jsp?reload=true&punumber=1001832

70. Grindal, Mats, Jeff Offutt, and Jonas Mellin. "Handling constraints in the input space when using combination strategies for software testing." (2006).

71. World Wide Web Consoritum, "Document Object Model", accessed 28 Mar 2015, http://www.w3.org/DOM/

72. World Wide Web Consoritum, DOM Level 3 Events Specification, 8 Sept 2009. http://www.w3.org/TR/DOM-Level-3-Events/

73. C. Montanez-Rivera, D.R. Kuhn, M. Brady, R.M. Rivello, J. Reyes and M.K. Powers, Evaluation of Fault Detection Effectiveness for Combinatorial and Exhaustive Selection of Discretized Test Inputs, *Software Quality Professional*, vol. 14, no. 3, June 2012.

74. J. Hagar, R. Kuhn, R. Kacker, and T. Wissink, Introducing Combinatorial Testing in a Large Organization: Pilot Project Experience Report [poster], *Third International Workshop on Combinatorial Testing (IWCT 2014), in Proceedings of the Seventh IEEE International Conference on Software, Testing, Verification and Validation (ICST 2014)*, Cleveland, Ohio, March 31-April 4, 2014, p. 153.

75. Cunningham, A. M., Hagar, J., & Holman, R. J. (2012, April). A system analysis study comparing reverse engineered combinatorial testing to expert judgment. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (pp. 630-635). IEEE.

76. J.D. Hagar, T.L. Wissink, D.R. Kuhn, R.N. Kacker, "Introducing Combinatorial Testing in a Large Organization", IEEE Computer, v. 48, n. 4, Apr. 2015.

77. R. Bryce, S. Sampath, A. Memon. Developing a Single Model and Test Prioritization Strategies for Event-Driven Software, Transactions on Software Engineering, (January 2011), 37(1):48-64.

78. R. Bryce, S. Sampath, J. Pedersen, S. Manchester. Test Suite Prioritization by Cost-based Combinatorial Interaction Coverage, International Journal on Systems Assurance Engineering and Management (Springer), (April 2011), 2(2): 126-134.

79. D.R. Kuhn, J.M. Higdon, J.F. Lawrence, R.N. Kacker, Y. Lei, "Combinatorial Methods for Event Sequence Testing", Workshop on Combinatorial Testing, co-located with the International Conference on Software Testing Verification and Validation, April 2012

80. G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. IEEE Trans. on Software Engineering, (October 2001), 27(10):929–948.

81. S. Sampath, R. Bryce, S. Jain, S. Manchester. A Tool for Combinatorial-based Prioritization and Reduction of User-Session-Based Test Suites, Proceedings of the International Conference on Software Maintenance (ICSM) - Tool Demonstration Track, Williamsburg, VA (September 2011), pp. 574-577.

82. S. Sampath, R. Bryce. Improving the effectiveness of test suite reduction for user-session-based testing of web applications, Information and Software Technology Journal (IST, Elsevier), (July 2012), 54(7): 724-738.

83. Sreedevi Sampath, Renée Bryce, Gokulanand Viswanath, Vani Kandimalla, A. Günes Koru, "Prioritizing User-Session-Based Test Cases for Web Application Testing", International Conference on Software Testing, Verification, and Validation (ICST) (April 2008), pp..141-150.

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

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

86. D.L. Parnas, "On the Use of Transition Diagrams in the Design of User Interface for an Interactive

Computer System," *Proc. 24th ACM Nat'l Conf.*, pp. 379-385, 1969.

87. W. E. Howden, G. M. Shi: Linear and Structural Event Sequence Analysis. *ISSTA 1996*: pp. 98-106, 1996.
88. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. Softw. Eng.*, vol. 4, no. 3, pp. 178-187, 1978.
89. J. Offutt, L. Shaoying, A. Abdurazik, and P. Ammann, "Generating Test Data From State-Based Specifications," *J. Software Testing, Verification and Reliability*, vol. 13, no. 1, pp. 25-53, March, 2003.
90. B. Sarikaya, "Conformance Testing: Architectures and Test Sequences," *Computer Networks and ISDN Systems*, vol.17, no. 2, North-Holland, pp. 111-126, 1989.
91. Kuhn, D. R., Higdon, J. M., Lawrence, J. F., Kacker, R. N., & Lei, Y. (2012, April). Combinatorial methods for event sequence testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (pp. 601-609). IEEE.http://csrc.nist.gov/groups/SNS/acts/documents/event-seq101008.pdf
92. B. Dushnik, Concerning a certain set of arrangements, Proc. Amer. Math. Soc, 1 (1950), 788-796.
93. J. Spencer, Minimal scrambling sets of simple orders, Acta Math. Acad. Sci. Hungary, 22 (1971)/1972), 349-353.
94. X. Yuan, M.B. Cohen, A. Memon, "Covering Array Sampling of Input Event Sequences for Automated GUI Testing", November 2007 *ASE '07: Proc. 22nd IEEE/ACM Intl. Conf. Automated Software Engineering*, pp. 405-408.
95. Margalit, O. (2013, March). Better Bounds for Event Sequencing Testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (pp. 281-284). IEEE.
96. E. Erdem, K. Inoue, J. Oetsch, J. P¨uhrer, H. Tompits, and C. Yilmaz, Answer-set programmingas a new approach to event-sequence testing, in Proceedings of the 2nd InternationalConference on Advances in System Testing and Validation Lifecycle, XpertPublishing Services, 2011, pp. 25–34.
97. Chee, Y. M., Colbourn, C. J., Horsley, D., & Zhou, J. (2013). Sequence covering arrays. *SIAM Journal on Discrete Mathematics*, *27*(4), 1844-1861.
98. C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," in Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, 2004, pp. 45–54.

99. E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter, “Feedback driven adaptive combinatorial testing,” in Proceedings of the 2011 International Symposium on Software Testing and Analysis, New York, NY, USA, 2011, pp. 243–253.

100. L. Shi, C. Nie, and B. Xu, “A software debugging method based on pairwise testing,” in Proceedings of the 5th international conference on Computational Science, Berlin, Heidelberg, 2005, pp. 1088–1091.

101. Z. Wang, B. Xu, L. Chen, and L. Xu, “Adaptive Interaction Fault Location Based on Combinatorial Testing,” In Proceedings of 10th International Conference on Quality Software (QSIC), 2010, pp. 495–502.

102. L. S. Ghandehari, Y. Lei, T. Xie, D. R. Kuhn, and R. Kacker, “Identifying Failure-Inducing Combinations in a Combinatorial Test Set,” in Proceedings of 5th IEEE International Conference on Software Testing, Verification and Validation, 2012, pp. 370–379.

103. L. S. Ghandehari, Y. Lei, R. Kacker, R. Kuhn, D. Kung, “Fault Localization Based on Failure-Inducing Combinations”, In -IEEE International Symposium on Software Reliability Engineering (ISSRE), Pasadena, CA, 2013, pp. 168-177.

104. J. Jones, M. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” In Proceedings of the 24th International Conference on Software Engineering (ICSE), 2002, pp. 467-477.

105. M. Renieris and S. Reiss, “Fault localization with nearest neighbor queries,” In Proceedings of the International Conference on Automated Software Engineering, 2003, pp. 30-39.

106. Czerwonka, J. (2013, March). On Use of Coverage Metrics in Assessing Effectiveness of Combinatorial Test Designs. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (pp. 257-266). IEEE.

107. Bartholomew, R. (2013, May). An industry proof-of-concept demonstration of automated combinatorial test. In *Automation of Software Test (AST), 2013 8th International Workshop on* (pp. 118-124). IEEE.

108. Bartholomew, R., & Collins, R. (2014). Using Combinatorial Testing to Reduce Software Rework. *CrossTalk*, 23.

109. Cohen, M. B., Dwyer, M. B., & Shi, J. (2007, July). Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis* (pp. 129-139). ACM.

110. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., & Traon, Y. L. (2012). Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines. *arXiv preprint arXiv:1211.5451*.

111. White, J., Dougherty, B., & Schmidt, D. C. (2009). Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, *82*(8), 1268-1284.

112. Lee, J., Kang, S., & Lee, D. (2012, September). A survey on software product line testing. In *Proceedings of the 16th International Software Product Line Conference-Volume 1* (pp. 31-40). ACM.

113. Xu, Z., Cohen, M. B., Motycka, W., & Rothermel, G. (2013, August). Continuous test suite augmentation in software product lines. In *Proceedings of the 17th International Software Product Line Conference* (pp. 52-61). ACM.

114. Garvin, B. J., Cohen, M. B., & Dwyer, M. B. (2013). Failure avoidance in configurable systems through feature locality. In *Assurances for Self-Adaptive Systems* (pp. 266-296). Springer Berlin Heidelberg.

115. Henard, C., Papadakis, M., Traon, Y. L. Flattening or Not the Combinatorial Interaction Testing Models? Intl. Wkshp. on Combinatorial Testing, IEEE, 2015.
116. Moy, Ledinot, Delseny, Wiels, Monate, "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience", IEEE Software, May/June 2013. citing: NASA ARMD Research Opportunities in Aeronautics 2011 (ROA-2011), research program System-Wide Safety and Assurance Technologies Project (SSAT2), subtopic AFCS-1.3 Software Intensive Systems, p. 77.
117. T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013

[See Additional References associated with Section 8]

[Back to end of Section 7]

8. Future Directions

Combinatorial testing has evolved into an accepted practice in software engineering. As it has entered mainstream use, research interest has become more specialized and application-oriented. Progress continues to be made in covering array generation algorithms, often with the aim of applying combinatorial methods to a broader range of testing problems, particularly those with larger inputs and complicated constraints. Concurrently, researchers are improving combinatorial test design methods by focusing on input model analysis and tools to assist in this phase of test design. Combinatorial testing continues to expand into domains such as Software Product Lines and mobile applications. Here we review current and upcoming developments in these areas, and suggest potential impacts for practical testing. Finally, we briefly discuss harder problems in the field for which broadly effective solutions are not fully perfected.

8.1 Algorithms

While conventional algorithms produce very compact arrays for many inputs, improvements are being achieved. One recent trend in covering array algorithms is the use of reduction strategies on existing arrays. That is, a *t*-way covering array with *N* tests is systematically reduced to fewer than *N* tests using a variety of mathematical transformations. The near-term impacts of algorithm improvements in array construction include extending the applicability of combinatorial methods. For applications such as modeling and simulation, where a single test may run for hours, reducing covering array size by even a few tests is of great value.

These methods have recently improved upon the best-known sizes of some covering array configurations [1, 2] and active research continues in this area. Similar transformations can also be done where there are constraints, and if the existing test suite was not designed as a covering array [3], using reductions that preserve the combinatorial coverage of the original test suite. An extension of this strategy [4] includes the option of allowing a subset of parameters to have values freely assigned, i.e., new tests can be generated rather than requiring them to be selected from the original test set. Other work shows that heuristic search can in some cases compete with greedy methods in speed and practicality for covering array construction [6]. Additionally, greedy algorithms can be improved using graph-coloring methods [7], to improve on a covering array generation phase that is optimal for t=2 but does not retain optimal properties at t>2.

A somewhat different aspect of applying combinatorial methods in test suite reduction is the use of interaction coverage as a criterion for reducing a test suite [8]. This may be particularly valuable for regression testing. Various test reduction strategies have been applied in the past, but sometimes result in deteriorating fault-detection effectiveness. Since combination coverage is effective in fault detection, retaining high combinatorial coverage in a reduced test set can preserve effectiveness using fewer tests. Yet another practical consideration is the setup time between tests. Many testing problems, especially for system integration or other large system tests, require changes to the SUT configuration with each test. Minimizing this time, while retaining high combination coverage can thus be an effective strategy [5].

8.2 Input Modeling

A second major research trend involves the integration of combinatorial methods in the development environment, and addressing practical problems particular to various domains. The first step in any testing effort is to understand and define the input model, that is, the set of parameters and values that will be included in tests, along with any constraints on values or sequencing. This phase is an issue for any testing approach, not just combinatorial, but the unique aspects of CT have led researchers to tailor conventional methods. Test environments tailored to CT are being developed [9, 10] to work with

popular frameworks such as Eclipse. These environments will allow for validating the consistency and other meta-properties of constraint sets [11].

Software product lines are increasingly used and their enormous range of possible configurations provides a natural domain for combinatorial testing. An extensive survey [16] shows the variety of ways in which *t*-way testing is now being applied in SPL testing and evaluation. Because of the large number of parameters in many SPLs, methods are being devised to extend the range of practical application for covering array generators. Software product lines often have hundreds, or even thousands, of variables. Conventional covering array algorithms are resource-limited in both time and storage to a few hundred. One approach is flattening of the input models, as described in Sect.7.5 [13]. Such methods are an active area of research.

Two current lines of research for improving definition of the input model are classification trees and UML models. UML sequence diagrams can be used as inputs to rule-based tools that extract an input model that can be used with a covering array generator [12]. Input variables and values are extracted from UML message specifications and guard conditions, providing partial automation of the process to reduce effort for test designers. Classification trees fit well with *t*-way testing, because they allow easy analysis and definition of test parameters in a tree structure [14]. Leaf nodes of the tree can be treated as category partitions and used directly in generating covering arrays. Robust tools based on classification trees, UML diagrams, and related concepts can help make combinatorial methods easier to use for test developers.

8.3 Harder problems

Combinatorial testing will continue to find new domains of application, but some research problems remain to be solved. Two broad areas in particular are likely to receive attention from researchers, because of their practical significance in industrial applications.

Very large systems: As with many areas of software engineering, *scalability* is essential. Fortunately, current combinatorial methods and covering array generators can address the vast majority of testing requirements. As noted earlier in the chapter, however, development approaches such as software product lines may involve thousands of parameters, with large numbers of constraints. Current covering array algorithms do not scale to such large problems, and existing constraint solvers are also insufficient for an extremely large number of constraints and variables.

Test development time: Case studies and experience reports show that combinatorial methods can provide better testing at lower cost, but these methods can require significant expertise and do not necessarily speed up the testing process. As such, if time-to-market is the primary concern, conventional test methods are likely to be preferred by developers. Application domains where CT has seen the most rapid acceptance so far are those with very high assurance requirements, such as aerospace/defense, finance, and manufacturing. Reducing the time required for using combinatorial methods is a significant challenge.

Research and practice have shown that combinatorial testing is highly effective across a range of testing problems, and this range of applicability continues to expand for new domains and technologies. The current high level of research interest in the field suggests that it may continue to advance, providing stronger testing at reduced cost for developers.

[Next Section: Conclusions]

[Back to main References]

Additional references

1. Avila-George, H., Torres-Jimenez, J., Gonzalez-Hernandez, L., & Hernández, V. (2013). Metaheuristic approach for constructing functional test-suites. *IET software*, *7*(2), 104-117.

2. Li, X., Dong, Z., Wu, H., Nie, C., & Cai, K. Y. (2014, March). Refining a Randomized Post-optimization Method for Covering Arrays. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 143-152). IEEE.

3. Farchi, E., Segall, I., Tzoref-Brill, R., & Zlotnick, A. (2014, March). Combinatorial Testing with Order Requirements. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 118-127). IEEE.

4. Itai Segall, Rachel Tzoref-Brill and Aviad Zlotnick. Combining Minimization and Generation for Combinatorial Testing In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Sixth International Conference on* IEEE.

5. Wu, H., Nie, C., & Kuo, F. C. (2014, March). Test suite prioritization by switching cost. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 133-142). IEEE.

6. Petke, J., Cohen, M., Harman, M., & Yoo, S. Practical Combinatorial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detection. IEEE TSE, preprint, 2015.

7. Linbin Yu, Feng Duan, Yu Lei, Raghu N. Kacker and D. Richard Kuhn. Constraint Handling In Combinatorial Test Generation Using Forbidden Tuples

8. Mayo, Q., Michaels, R., & Bryce, R. (2014, March). Test Suite Reduction by Combinatorial-Based Coverage of Event Sequences. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 128-132). IEEE.

9. Gargantini, A., & Vavassori, P. (2012, April). Citlab: a laboratory for combinatorial interaction testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (pp. 559-568). IEEE.

10. Garn, B., & Simos, D. E. (2014, March). Eris: A tool for combinatorial testing of the Linux system call interface. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 58-67). IEEE.

11. Arcaini, P., Gargantini, A., & Vavassori, P. (2014, March). Validation of models and tests for constrained combinatorial interaction testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 98-107). IEEE.

12. Satish, P., Paul, A., & Rangarajan, K. (2014, March). Extracting the combinatorial test parameters and values from UML sequence diagrams. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on* (pp. 88-97). IEEE.

13. Christopher Henard, Mike Papadakis and Yves Le Traon. Flattening or Not the Combinatorial Interaction Testing Models?, *Software Testing, Verification and Validation (ICST), 2015 IEEE Fifth International Conference*

14. Zeppetzauer, U., & Kruse, P. M. (2014, September). Automating test case design within the classification tree editor. In *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on* (pp. 1585-1590). IEEE.

15. Gargantini, A., & Vavassori, P. (2014). Efficient Combinatorial Test Generation Based on Multivalued Decision Diagrams. In *Hardware and Software: Verification and Testing* (pp. 220-235). Springer International Publishing.

16. Roberto Erick Lopez-Herrejon, Stefan Fischer, Rudolf Ramler and Alexander Egyed. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines, *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Seventh International Conference on*