Measurement 46 (2013) 3745–3752

Contents lists available at SciVerse ScienceDirect

Measurement

journal homepage: www.elsevier.com/locate/measurement

Combinatorial testing for software: An adaptation of design of experiments

Raghu N. Kacker [a,*], D. Richard Kuhn [a], Yu Lei [b], James F. Lawrence [a,c]

[a] National Institute of Standards and Technology, Gaithersburg, MD 20899, USA
[b] University of Texas, Arlington, TX 76019, USA
[c] George Mason University, Fairfax, VA 22030, USA

ARTICLE INFO

Article history:
Available online 14 March 2013

Keywords:
Computer security
Covering arrays
Design of experiments
Measurement
Metrology
Orthogonal arrays
Software engineering
Software testing

ABSTRACT

Software has become increasingly ubiquitous in tools and methods used for science, engineering, medicine, commerce, and human interactions. Extensive testing is required to assure that software works correctly. Combinatorial testing is a versatile methodology which is useful in a broad range of situations to detect faults in software. It is based on the insight that while the behavior of a software system may be affected by a large number of factors, only a few factors are involved in a failure-inducing fault. We discuss the development of combinatorial testing for software as adaptation of design of experiment methods. Combinatorial testing began as pairwise testing in which first orthogonal arrays and then covering arrays were used to make sure that all pairs of the test settings were tested. Subsequent investigations of actual software failures showed that pairwise (2-way) testing may not always be sufficient and combinatorial t-way testing for t greater than 2 may be needed. Until recently efficient tools for generating test suites for combinatorial t-way testing were not widely available. Combinatorial testing has become practical because efficient and free downloadable tools with support of constraints have become available.

Published by Elsevier Ltd.

1. Introduction

A software fault is a mistake in the code which when encountered may cause the software to fail. Failure means that the software behaves in unexpected (incorrect) ways. A variety of testing methods are used to avoid, detect, and correct faults during and after development of software. A 2002 report estimated that the cost to the US of inadequate infrastructure for software testing was between $22.2 and $59.5 billion [1]. In a decade since then software has become more complex and the detection of faults has become more challenging. An often used approach for detecting software faults is dynamic testing in which the software system under test (SUT) is exercised (run) for a set of test cases, the expected (correct) behavior of the system is predetermined for each test case, and the actual behavior is compared against the expected. The SUT passes a test case when the behavior is as expected and fails when the behavior is different from the expected behavior. When the SUT fails for one or more test cases the underlying faults in the software which induce the failure are searched and then corrected. The SUT could be any part of a software system however small or large for which test cases can be constructed, the expected (correct) behavior can be determined for each test case, tests executed, and the actual behavior can be observed and assessed. Dynamic testing is often used for independent verification and validation of software systems.

Combinatorial testing is a type of dynamic testing in which distinct (but possibly related) test factors are specified from the requirements, knowledge of system implementation and internal operations, and other information available about the SUT. The possible values of test factors may reside on continuous or discrete scales. In either case, for each test factor relatively few discrete test settings are

* Corresponding author. Tel.: +1 301 975 2109.
E-mail address: raghu.kacker@nist.gov (R.N. Kacker).

0263-2241/$ - see front matter Published by Elsevier Ltd.
http://dx.doi.org/10.1016/j.measurement.2013.02.021

specified by equivalence partitioning, boundary value analysis, and expert judgment. Then each test case is expressed as a combination of one test setting for every test factor [2–5]. Suppose the first test factor has v_1 test settings, the second test factor has v_2 test settings, etc., and the k-th test factor has v_k test settings, where $v_1, v_2, \ldots, v_k$ could be all different. Then a test case is a combination of k test settings, one for each test factor (a k-tuple). The number of different test cases possible is the product of all k numbers $v_1, v_2, \ldots, v_k$ of test settings. For example, suppose out of nine test factors, if 3 have three test settings each, 4 have four test settings each, and 2 have five test settings each then the number of possible test cases is $3^3 4^4 5^2 = 1{,}72{,}800$. The exponential expression $3^3 4^4 5^2$ represents the combinatorial test structure and its expanded form 1,72,800 is the number of possible test cases. In many practical applications the number of possible test cases is too large to test them all. In combinatorial testing, combinatorial mathematics and computational methods are used to determine a small set (called test suite) of test cases which covers all test settings of each factor and all t-way combinations (t-tuples) of test settings for some $t \geqslant 2$. The value of t (called strength of the test suite) is chosen with the objective that the test suite will exercise the combinations corresponding to the faults for which the SUT could fail. Methods for specification of test factors, test settings, and the strength t are largely application domain specific and a subject of continuing research [4]. In this paper we address the problem of constructing (generating) the test suite after the test factors and the test settings have been specified.

Orthogonal arrays (OAs) are tabular arrangements of symbols which satisfy certain combinatorial properties. In the1960s and1970s Japan and starting in the 1980s in the US and Europe, Genichi Taguchi promulgated the use of OAs (of strength two) as templates for Design of Experiments (DoEs) [6–8]. In the late 1980s and early 1990s, inspired in part by Taguchi, some software engineers started investigating the use of OAs for pairwise (2-way) combinatorial testing of software and hardware–software systems. In Section 2, we review the use of OAs for DoEs and software testing. Soon the limitations of OAs to generate test suites for software testing became apparent. Covering arrays (CAs) are generalizations of OAs with slightly relaxed combinatorial properties. Covering arrays were found to be better suited than OAs for generating test suites for software testing. In Section 3 we discuss the use of CAs for pairwise (2-way) testing of software. Subsequent investigations of the reports of actual software failures showed that pairwise (2-way) testing is useful but it may not always be sufficient. Also, test factors and test settings are subject to various types of constraints imposed by the semantics of the SUT and the runtime environment. In Section 4, we discuss t-way combinatorial testing (CT) for $t \geqslant 2$ with support of constraints. Combinatorial testing for $t \geqslant 2$ is now practical because efficient tools for generating test suites for t-way testing with support of constraints have become available. A brief summary appears in Section 5.

2. Use of orthogonal arrays for design of experiments and software testing

The concept of Orthogonal arrays (OAs) was formally defined by Rao [9]. OAs are generalization of Latin squares [10]. The matrix shown in Table 1 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 3, every set of three columns contains all possible triplets of elements exactly the same number of times.

A fixed-value orthogonal array denoted by OA(N, v^k, t) is an $N \times k$ 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 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 $0, 1, 2, \ldots, (v-1)$ used in Table 1 are symbols rather than numbers. The combinatorial property 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, etc., 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 [11]. Neil Sloane maintains an electronic library of known OAs [12].

The term design of experiments (DoEs) 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). The corresponding values of one or more output variables (called responses) are measured

Table 1
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

to generate information for improving the performance of a class of similar systems. Conventional DoE methods were developed in the 1920s and 1930s by Ronald Fisher and his contemporaries and their followers to improve agricultural production [13,14]. Later DoEs were adapted for experiments with animals, medical research, and then to improve manufacturing processes subject to uncontrolled variation. Frequently, the effects of many factors each having multiple test settings are investigated at the same time and the DoE plans satisfy relevant combinatorial properties [15–19]. The objective in conventional DoEs is to improve the mean response over replications. Taguchi promulgated 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 [6–8,20,21]. Taguchi promoted use of OAs of strength two as templates for DoE plans. Before Taguchi's use of OAs, they were not very well known outside the world of mathematics and statistics.

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 2.

The last column of Table 2 displays the values $y_1,y_2,\ldots,y_8$, of the response Y for the eight test cases. A salient property of DoE is that they enable evaluation of a statistic called *main effect* of each test factor [18]. The main effect of a factor is its average effect on the response for all test values of the other factors. The average response corresponding to test value A_1 is $(y_2 + y_4 + y_6 + y_8)/4$ and the average response corresponding to test value A_0 is $(y_1 + y_3 + y_5 + y_7)/4$. The main effect of factor A is the difference $(y_2 + y_4 + y_6 + y_8)/4 - (y_1 + y_3 + y_5 + y_7)/4$ between the two averages. The combinatorial property of orthogonal arrays guarantees that the test value of each factor is paired with all test values of every other factors the same number of times. Thus in each of the two averages for A_1 and A_0, each test level of every other factor is represented exactly the same number of times (see Table 2). Thus the main effect of factor A is meaningful for all test values of the other factors. Similarly, the main effects of factors B, C, D, and E are meaningful for all test values of the other factors. Orthogonal arrays (and other DoE plans) enable evaluation of the main effects.

Along with the advent of computers and telecommunication systems in the 1980s, independent verification and validation of software and hardware–software systems became important. 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: [2,3,22–24]. In the US, starting in the late 1980s and early 1990s, AT&T researcher Madhav Phadke and his colleagues developed a tool called OATS (Orthogonal Array Testing System) to generate test suites (based on OAs of strength 2) for Taguchi DoEs and for testing software based systems [25]. OATS was an AT&T proprietary tool for intra-company use only. Use of OAs of strength 2 assured that all test settings for each test factor and all pairs of test settings for every pair of test factors were tested (executed). Thus began pairwise (2-way) testing, a type of dynamic testing in which the SUT is exercised for a test suite which satisfies the property that for every pair of test factors all possible pairs of test settings are tested.

3. Pairwise (2-way) testing using covering arrays

When AT&T software tester George Sherwood tried to use OATS to specify "client test configurations for a local area network product", he realized the limitations of OATS and the limitations of using OAs to construct test suites for software testing [26]. Often, an OA matching the required combinatorial test structure does not exist. Also, frequently, OA based test suites included invalid test cases (which cannot be executed). Suppose out of five factors, four have two test settings each and one has three test settings; thus the combinatorial test structure is $2^4 \times 3^1$. An OA of strength 2 matching the test structure $2^4 \times 3^1$ does not exist (it is mathematically impossible). In such cases a suitable OA is modified to fit the need. For example, if the symbols in the rows 7 and 8 of the column 5 of OA$(8, 2^4 \times 4^1, 2)$ shown in Table 1 are changed from 3 to 2, we

Table 2
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 3
Combinatorial arrangement for the test structure $2^4 \times 3^1$.

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

get the combinatorial arrangement shown in Table 3. Table 3 is not an OA but it covers all pairs of test settings and it can be used to construct a pairwise testing suite for the test structure $2^4 \times 3^1$.

Suppose the five test factors of combinatorial test structure $2^4 \times 3^1$ were (1) operating system (OS) with two test settings {XP,Linux}, (2) browser with two test settings {Internet Explorer (IE),Firefox}, (3) protocol with two test settings {IPv4,IPv6}, (4) CPU type with two test settings {Intel,AMD}, and (5) database management system (DBMS) with three test settings {MySQL,Sybase,Oracle}. Then a test suite for pairwise testing based on Table 3 is shown in Table 4. To obtain Table 4 from Table 3, the five test factors OS, Browser, Protocol, CPU type, and the DBMS are associated with the five columns of Table 3 and the symbols in the columns {0,1,2} are replaced with the test settings of the respective test factors. The eight rows of Table 4 form a test suite based on Table 3. Since the browser Internet Explorer (IE) does not run on the operating system Linux, the pair {Linux,IE} appearing in test cases 6 and 8 of Table 4 is invalid. Therefore, these two test cases are disallowed (invalid) and cannot be executed. In that case the other valid pairs of test settings covered by these two test cases will not be tested, for example, the pairs {IPv6,Oracle}, {Intel,Oracle}, {IPv4,Oracle}, and {AMD,Oracle} will not be tested. Thus the test suite shown in Table 4 with test cases 6 and 8 omitted would not test all valid pairs of test settings.

Since for many combinatorial test structures OAs are not mathematically possible and since test suites based on OAs may include invalid pairs of test settings, in the early 1990s, Sherwood developed a tool called CATS (Constrained Array Testing System) to generate test suites which cover all valid combinations of test settings with a small number of test cases [26]. The test suite generation tool CATS (like OATS) was an AT&T proprietary tool for intra-company use only. Test suites generated by CATS were related to tabular arrangements of symbols called covering arrays.

The concept of Covering Arrays (CAs) was formally defined by AT&T mathematician Neil Sloane [27]. Significant earlier contributions leading to the concept of CAs include the following: [28–30]. Additional references on CAs can be found in the following recent papers: [31,32]. A fixed-value covering array denoted by $CA(N, v^k, t)$ is an $N \times k$ 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, etc., and k_n columns have v_n distinct elements. The first six rows of the eight rows of Table 1 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. Thus the concept of covering arrays is a generalization of OAs.

Table 4
Test suite based on Table 3.

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

Methods for constructing CAs can be put in three main categories: (1) algebraic methods (for example [27,30,33,34], and others), (2) meta-heuristic methods such as simulated annealing and tabu search (for example [32,35–38], and others), and (3) greedy search methods (for example [39–44], and others). Algebraic methods apply only to certain mathematically conforming combinatorial test structures; however, when they apply they are extremely fast techniques and may produce CAs of smallest possible size. Meta-heuristic methods are computationally intensive and they have produced CAs of the smallest size known. Greedy methods are faster than meta-heuristic methods; they apply to arbitrary test structures but may or may not produce smallest size CAs. However Michael Forbes's greedy algorithm has produced some CAs of the smallest size known [44]. The three approaches are sometimes combined to yield additional methods for constructing covering arrays, for example [43–46], and others. Charlie Colbourn maintains a web page of smallest known sizes (N) of various covering arrays $CA(N, k, v, t)$ of strength t up to seven [47]. Tables of covering arrays of smallest known sizes are available in the following webpages [48–50].

Dalal and Mallows are among the leading promulgators of the use of covering arrays for software testing [51]. They made the case that evaluation of the main effects of factors is important in DoE and OAs enable evaluations of the main effects; however, in testing software systems there is no need to evaluate the main effects of test factors. Instead interest lies in covering all pairs (in general all t-tuples) of test settings. Therefore covering arrays are better suited than OAs for testing software. A decade earlier, Tatsumi had made the same observation [3]. In addition, he pointed out that in generating test suites invalid combinations must be excluded.

It turns out that CAs have several advantages over OAs: (1) CAs can be constructed for any combinatorial test structure of unequal numbers of test settings. (2) If for a combinatorial test structure an OA exists (is mathematically possible) then a CA of the same or fewer test cases can be obtained. (3) CAs can be constructed for any required strength (t-way) testing, while OAs are generally limited to strength 2 and 3 [12]. (4) In generating test suites based on CAs invalid combinations can be deliberately excluded.

The first publicly available tool for generating test suites based on CAs for pairwise (and higher strength) testing of

software systems was AETG [52–54]. In 1998, Yu Lei developed a tool called IPO for generating test suites for pairwise testing based on CAs which excluded invalid pairs of test settings [39,40]. Usefulness of CAs led to a great interest among mathematicians and computer scientists to develop tools for generating test suites based on CAs for pairwise testing. A website (pairwise.org) maintained by Jacek Czerwonka lists 37 tools (beginning with OATS, CATS, AETG and IPO) for generating test suites for combinatorial testing [55].

4. Combinatorial (*t*-way) testing with support of constraints

A team of the US National Institute of Standards and Technology (NIST) researchers investigated 15 years' worth of recall data due to failures of software embedded in medical devices and failure reports for a browser, a server, and a database system. The initial purpose of these investigations was to generate insights into the kinds of software testing that could have detected the underlying faults and prevented the failures in use [56–58]. Subsequently the NIST researchers counted the numbers of individual factors that were involved in the faults underlying the actual failures. In the four systems investigated by the NIST team, 29–68% of the faults involved a single factor; 70–97% of the faults involved one or two factors; 89–99% of the faults involved three or fewer factors; 96–100% of the faults involved four or fewer factors; 96–100% of the faults involved five or fewer factors, and no fault involved more than six factors [58]. These investigations suggest the following empirical *interaction rule*. Only a few factors are involved in failure-inducing faults in software. Most failures are induced by single factor faults or by the joint combinatorial effect (interaction) of two factors; progressively fewer failures are induced by interactions between three, four, or more factors. The maximum degree of interaction in actual faults so far observed is six. The interaction rule implies that pairwise (2-way) testing is useful but it may not be sufficient. Combinatorial (*t*-way) testing for t greater than 2 may sometimes be needed. These conclusions are supported by other investigations including the following [59–62]. This insight motivated the authors of this paper to advance methods and tools for combinatorial (*t*-way) testing for $t \geqslant 2$.

Combinatorial (*t*-way) testing (CT) is a type of dynamic testing for software systems in which the SUT is exercised for a suite of test cases which satisfies the property that for every subset of t test factors (out of all k factors where $k \geqslant t$) all t-tuples (of the test settings) are tested at least once (disallowed combinations being excluded). The expected (correct) behavior of the system is predetermined for each test case, and the actual behavior is compared against the expected. The SUT passes a test case when the behavior is as expected and fails when the behavior is different. When the SUT fails for one or more test cases, various approaches are used to isolate the failure-inducing combinations of test settings from the pass/fail data. To isolate failure-inducing combinations additional tests may be required [63]. Once one or more failure-inducing combinations have been identified, follow up investigations are conducted to reveal and correct the underlying faults in the SUT.

Our NIST and UTA (University of Texas at Arlington) team has developed a tool called ACTS for generating combinatorial t-way test suites for arbitrary combinatorial test structures and any strength t with support of constraints (to exclude invalid combinations). ACTS is an extension of the IPO referred to earlier [39–41]. The tool ACTS is a free research tool downloadable from a NIST web site [64]. Special features of the ACTS tool include the following. (1) ACTS excludes those combinations of the test settings which are invalid according to the user specified constraints. (2) ACTS 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) ACTS 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) ACTS verifies whether the test suite supplied by a user covers all t-way combinations. (5) ACTS allows the user to specify expected output for each test case in terms of the number of output parameters and their values. (6) ACTS 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.

Combinatorial testing is a versatile methodology which could be useful in a broad range of testing situations. The most basic use of CT is for testing various configurations of a system or various inputs to a system from a user or from another system [5]. Combinatorial testing can also be used for testing databases and state models [65]. Our team and collaborators have investigated the use of CT in the following situations: (1) testing concurrent systems [66], (2) testing web applications [67], (3) security testing of access control implementations [68], (4) navigation of dynamic web structures [69], (5) optimization of discrete event simulation models [70], (6) analyzing system state-space coverage [71], (7) detecting deadlocks for varying network configurations [72], (8) detecting buffer overflow vulnerabilities [73], (9) conformance testing for standards [74], (10) event sequence testing [75], and (11) prioritizing user session based test suites for web applications [76].

Combinatorial testing can greatly improve the efficiency of dynamic software testing. If faults involving more than t test factors are absent then t-way testing would be as good as the more expensive exhaustive testing requiring a larger number of test cases. A recent NIST study demonstrates the efficiency of CT by showing that 4-way testing could detect all faults using less than five percent of the tests required by exhaustive testing [74].

An alternative to CT is random testing in which test cases are formed from random draws of the discrete test

settings of test factors. Generally combinatorial testing requires fewer test cases than random testing for equivalent coverage [72]. The benefit of CT in terms of the reduced testing effort relative to random testing increases as the strength t increases.

The following example shows that the size of combinatorial t-way test suite increases rapidly as t increases. Android is an open source platform for smart phone applications. A resource configuration file for Android applications has 35 options. These options can be expressed in terms of 9 test factors with the combinatorial test structure $3^3 4^4 5^2$ [77]. The number of possible test cases is $3^3 4^4 5^2 = 1{,}72{,}800$. Exhaustive testing is not practical. The sizes (number of test cases) of t-way test suites determined using ACTS 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 is used in the specification of test factors, test setting, and the strength t. 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. In the four systems investigated by the NIST researchers in 1990s about 96–100% the faults involved four or fewer factors [58]; therefore, t of four or less may be reasonable. In some situations it may be more effective to try t-way test suites for $t = 2$ or 3 with different test settings (and possibly different test factors as well) than testing with higher values of t. Some software testers determine combinatorial coverage of their skillfully developed test suites and then append additional test cases to achieve a desired t-way (or variable strength) coverage.

The challenges in CT include the following. (1) Modeling of the test space including specification of test factors, test settings and their constraints. (2) Efficient generation of t-way test suites, especially involving support of constraints. (3) Determination of the expected behavior of the system for each possible test case and checking whether the actual behavior agrees with the expected behavior. (4) Identification of the failure-inducing test value combinations from pass/fail results of CT. (5) Integration of CT in the existing infrastructures for testing. Our research team is addressing these challenges in collaboration with academia and industry.

The choice of test factors and their test settings defines and limits the scope of the combinatorial testing. Clearly, a failure-inducing fault may not be detected when the test factors and settings associated with that fault are not exercised [3]. When continuous-valued factors are involved, the discrete test settings preclude testing certain values. Therefore combinatorial testing can detect faults but it cannot guarantee their absence. Combinatorial testing complements other approaches used to assure correctness of software.

Combinatorial (t-way) testing is an adaptation of the design of experiment methods for testing software (and hardware–software) systems because in both cases information about a system is gained by exercising it and the test suite (DoE plan) satisfies relevant combinatorial properties. In combinatorial testing (unlike DoE), the expected behavior of the system for each test case must be pre-determined. This requires additional information such as a mathematical model of the SUT or a benchmark implementation. In addition some tool is needed to check whether the actual behavior of the SUT matches the expected behavior and to make a verdict of passing or failing for each test case. Conventional DoE methods (unlike CT) use a statistical model for the relationship between the input factors and the output response to predict response for untested conditions. In CT all conclusions are based on (and limited to) the actual tests conducted. In DoE, experimental error treated as random is an important component of the variation of response values. In CT random error is absent, negligible, or can be avoided by appropriate modeling of the test factors.

5. Summary

Combinatorial (t-way) testing evolved from the use of design of experiments based on orthogonal arrays for generating test suites for software testing. Combinatorial testing requires specification of test factors and their discrete test settings. A test case is a combination of one selected test setting for each test factor. Combinatorial testing began as pairwise (2-way) testing in which the software system under test is exercised for a test suite of test cases which satisfies the property that for every pair of test factors all possible pairs of the test settings are tested at least once. First orthogonal arrays were used as templates for constructing pairwise test suites. However, orthogonal arrays could not support constraints among the test settings and test factors. Therefore covering arrays were found to be better suited than orthogonal arrays for combinatorial testing. Investigations of actual faults indicated that most failures are induced by single factor faults or by the joint combinatorial effect (interaction) of two factors, with progressively fewer failures induced by interactions between three or more factors. Thus pairwise (2-way) testing is useful but not always sufficient and combinatorial (t-way) testing for t greater than 2 is needed. Combinatorial (t-way) testing for $t \geqslant 2$ is now possible because efficient and free downloadable tools for generating test suites for (t-way) testing with support of constraints (to exclude invalid combinations) have become available. Combinatorial testing is a versatile methodology which could be useful in a broad range of testing situations. Combinatorial testing can detect faults but it cannot guarantee their absence. Also combinatorial testing is one of many complementary methods for software assurance.

Disclaimer

NIST does not recommend or endorse any commercial product referenced in this paper or imply that the referenced products are necessarily the best available for the purpose. ACTS is a research tool not commercial product.

Acknowledgments

We thank the NIST management for their support and we thank the Division Reader, the WERB Reader, and those who provided comments on earlier drafts of this paper.

References

[1] Gregory Tassey, The economic impacts of inadequate infrastructure for software testing, in: Final Report prepared for NIST by the Research Triangle Institute, Research Triangle Park, North Carolina USA, 2002.
[2] Robert. Mandl, Orthogonal Latin squares: an application of experiment design to compiler testing, Communications of the ACM 28 (1985) 1054–1058.
[3] Keizo Tatsumi, Test-case design support system, in: Proceedings of International Conference on Quality Control, Tokyo, 1987, pp. 615–620.
[4] Paul. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, Cambridge UK, 2008.
[5] Aditya P. Mathur, Foundations of Software Testing, Addison-Wesley, Boston, 2008.
[6] Genichi. Taguchi, Introduction to Quality Engineering, UNIPUB Kraus International, White Plains New York, 1986.
[7] Genichi Taguchi, System of Experimental Design, vols. 1 and 2, UNIPUB Kraus International, White Plains New York, 1987 (English translations of the 3-rd edition of Jikken Keikakuho (Japanese) published in 1977 and 1978 by Maruzen).
[8] Genichi. Taguchi, Taguchi on Robust Technology Development, ASME Press, New York, 1993.
[9] C.R. Rao, Factorial experiments derivable from combinatorial arrangements of arrays, Journal of Royal Statistical Society (Supplement) 9 (1947) 128–139.
[10] Damaraju. Raghavarao, Constructions and Combinatorial Problems in Design of Experiments, Dover, New York, 1971.
[11] A.S. Hedayat, N.J.A. Sloan, J. Stufken, Orthogonal Arrays: Theory and Applications, Springer, New York, 1999.
[12] Neil J.A. Sloane, <http://www2.research.att.com/~njas/oadir/>.
[13] R.A. Fisher, Statistical Methods for Research Workers, Oliver and Boyd, Edinburgh, 1925.
[14] R.A. Fisher, The Design of Experiments, Oliver and Boyd, Edinburgh, 1935.
[15] William G. Cochran, G.M. Cox, Experimental Designs, Wiley, New York, 1950.
[16] Oscar. Kempthorne, Design and Analysis of Experiments, Wiley, New York, 1952.
[17] George W. Snedecor, W.G. Cochran, Statistical Methods, Iowa State University Press, 1967.
[18] George E.P. Box, W.G. Hunter, J.S. Hunter, Statistics for Experimenters, Wiley, New York, 1978.
[19] Douglas C. Montgomery, Design and Analysis of Experiments, 4th edition., Wiley, New York, 2004.
[20] Raghu N. Kackar, Off-line quality control, parameter design and the Taguchi method, Journal of Quality Technology 17 (1985) 176–209.
[21] M.S. Phadke, Quality Engineering using Robust Design, Prentice Hall, Englewood Cliffs New Jersey, 1989.
[22] Shinobu Sato, H. Shimokawa, Methods for setting software test parameters using the design of experiments method (in Japanese), in: Proceedings of 4th Symposium on Quality Control in Software, Japanese Union of Scientists and Engineers (JUSE), 1984, pp. 1–8.
[23] Hiroki Shimokawa, Method of generating software test cases using the experimental design (in Japanese), in: Report on Software Engineering SIG, Information Processing Society of Japan, No.1984-SE-040, 1985.
[24] Keizo Tatsumi, S. Watanabe, Y. Takeuchi, H. Shimokawa, Conceptual support for test case design, in: Proceedings of 11th IEEE Computer Software and Applications Conference, 1987, pp. 285–290.
[25] R. Brownlie, J. Prowse, M.S. Phadke, Robust testing of AT&T PMX/Starmail using OATS, AT&T Technical Journal 71 (1992) 41–47.
[26] George B. Sherwood, Effective testing of factor combinations, in: Proceedings of 3rd International Conference on Software Testing, Analysis and, Review, 1994, pp. 151–166.
[27] Neil J.A. Sloane, Covering arrays and intersecting codes, Journal of Combinatorial Designs 1 (1993) 51–63.
[28] Alfred. Renyi, Foundations of Probability, Wiley, New York, 1971.
[29] Daniel J. Kleitman, J. Spencer, Families of k-independent sets, Discrete Mathematics 6 (1973) 255–262.
[30] G. Roux, *K*–propriétés dans les tableaux de *n* colonnes: cas particulier de la *k*–surjectivité et de la k–permutivité Unpublished PhD dissertation, University of Paris, 1987.
[31] James F. Lawrence, R.N. Kacker, Yu Lei, D.R. Kuhn, M. Forbes, A survey of binary covering arrays, The Electronic Journal of Combinatorics 18 (2011) P84.
[32] Jose. Torres-Jimenez, E. Rodriguez-Tello, New bounds for binary covering arrays using simulated annealing, Information Sciences 185 (2012) 137–152.
[33] K.A. Bush, Orthogonal arrays of index unity, Annals of Mathematical Statistics 23 (1952) 426–434.
[34] Alan Hartman, Software and hardware testing using combinatorial covering suites, in: M.C. Golumbic, I.B. Hartman (Eds.), Graph Theory, Combinatorics, and Algorithms: Interdisciplinary Applications, Springer, 2005, pp. 237–266.
[35] Myra B. Cohen, P.B. Gibbons, W.B. Mugridge, C.J. Colbourn, Constructing test suites for interaction testing, in: Proceedings of 25th IEEE International Conference on, Software Engineering, 2003, pp. 38–49.
[36] Charles J. Colbourn, Combinatorial aspects of covering arrays, Le Matematiche Catania 59 (2004) 125–172.
[37] K. Nurmela, Upper bounds for covering arrays by tabu search, Discrete Applied Mathematics 138 (2004) 143–152.
[38] Myra B. Cohen, C.J. Colbourn, A.C.H. Ling, Constructing strength three covering arrays with augmented annealing, Discrete Mathematics 308 (2008) 2709–2722.
[39] Yu Lei, K.C. Tai, In-parameter order: a test generation strategy for pairwise testing, in: Proceedings of 3rd IEEE High Assurance Systems Engineering Symposium, 1998, pp. 254–261.
[40] Yu Lei, K.C. Tai, A test generation strategy for pairwise testing, in: Technical Report TR-2001003, Department of Computer Science, North Carolina State University, 2001.
[41] K.C. Tai, Yu. Lei, A test generation strategy for pairwise testing, IEEE Transactions on Software Engineering 28 (2002) 109–111.
[42] Yu Lei, R.N. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG: a general strategy for t-way software testing, in: Proceedings of 14th Annual IEEE International Conference on the Engineering of Computer-based Systems, 2007, pp. 549–556.
[43] Yu Lei, R.N. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing, Software Testing Verification and Reliability 18 (2008) 125–148.
[44] Michael Forbes, J. Lawrence, Yu Lei, R.N. Kacker, D.R. Kuhn, Refining the in-parameter-order strategy for constructing covering arrays, Journal of Research of NIST 113 (2008) 287–297.
[45] George B. Sherwood, S.S. Martirosyan, C.J. Colbourn, Covering arrays of higher strength from permutation vectors, Journal of Combinatorial Designs 14 (2006) 202–213.
[46] Robert A. Walker II, C.J. Colbourn, Tabu search for covering arrays using permutation vectors, Journal of Statistical Planning and Inference 139 (2009) 69–80.
[47] Charles J. Colbourn, <http://www.public.asu.edu/~ccolbou/src/tabby/catable.html>.
[48] Michael Forbes, <http://math.nist.gov/coveringarrays/>.
[49] K. Nurmela, <http://www.tcs.hut.fi/~kjnu/covarr.html>.
[50] Jose Torres-Jimenez, <http://www.tamps.cinvestav.mx/~jtj/CA.php>.
[51] Siddhartha R. Dalal, C.L. Mallows, Factor-covering designs for testing software, Technometrics 40 (1998) 234–243.
[52] David M. Cohen, S.R. Dalal, A. Kajla, G.C. Patton, The automatic efficient test generator (AETG) system, in: Proceedings of 5th IEEE International Symposium on Software, Reliability Engineering, 1994, pp. 303–309.
[53] David M. Cohen, S.R. Dalal, J. Parelius, G.C. Patton, The combinatorial design approach to automatic test generation, IEEE Software 13 (1996) 83–89.
[54] David M. Cohen, S.R. Dalal, M.L. Fredman, G.C. Patton, The AETG system: an approach to testing based on combinatorial design, IEEE Transactions on Software Engineering 23 (1997) 437–444.
[55] Jacek Czerwonka, <http://www.pairwise.org/>.
[56] Dolores 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 8 (2001) 351–371.
[57] D. Richard Kuhn, M.J. Reilly, An investigation of the applicability of design of experiments to software testing, in: Proceedings of 27th NASA/IEEE Software Engineering Workshop, Goddard Space Flight Center, 2002, pp. 91–95.
[58] D. Richard Kuhn, D.R. Wallace, A.J. Gallo Jr., Software fault interactions and implications for software testing, IEEE Transactions on Software Engineering 30 (2004) 418–421.

[59] George B. Finelli, NASA software failure characterization experiments, Reliability Engineering & System Safety 32 (1991) 155–169.
[60] Kera Z. Bell, M.A. Vouk, On effectiveness of pairwise methodology for testing network-centric software, in: Proceedings of 3rd ITI International Conference on Information & Communications Technology, 2005, pp. 221–235.
[61] Kera Z. Bell, Optimizing effectiveness and efficiency of Software testing: a hybrid approach, Unpublished PhD dissertation, North Carolina State University, 2006.
[62] Zhiqiang Zhang, Xiaojian Liu, Jian Zhang, Combinatorial Testing on ID3v2 Tags of MP3 Files, in: Proceedings of 5th IEEE International Conference on Software Testing, Verification and Validation, Workshop on Combinatorial Testing, 2012, pp. 587–590.
[63] Laleh S.G. Ghandehari, Yu lei, Tao Xie, D.R. Kuhn, R.N. 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.
[64] D. Richard Kuhn, <http://csrc.nist.gov/groups/SNS/acts/index.html>.
[65] George B. Sherwood, Getting the Most from Pairwise Testing, A Guide for Practicing Software Engineers, Testcover.com, Colts Neck New Jersey, 2011.
[66] Yu Lei, R. Carver, R.N. Kacker, D. Kung, "A combinatorial strategy for testing concurrent programs" Journal of Software Testing, Verification and Reliability 17 (2007) 207–225.
[67] Wenhua Wang, S. Sampath, Yu Lei, R.N. Kacker, An interaction-based test sequence generation approach for testing web applications, in: Proceedings of 11th IEEE International Conference on High Assurance, Systems Engineering, 2008, pp. 209–218.
[68] Vincent C. Hu, D.R. Kuhn, Tao Xie, Property verification for generic access control models, in: Proceedings of IEEE/IFIP International Symposium on Trust, Security and Privacy for Pervasive Applications, 2008, pp. 243–250.
[69] Wenhua Wang, Yu Lei, S. Sampath, R.N. Kacker, D.R. Kuhn, J. Lawrence, A combinatorial approach to building navigation graphs for dynamic web applications, in: Proceedings of 25th IEEE International Conference on Software, Maintenance, 2009, pp. 211–220.
[70] Björn Johnasson, R.N. Kacker, R. Kessel, C. McLean, R. Sriram, Utilizing combinatorial testing to detect interactions and optimize a discrete event simulation model for sustainable manufacturing, in: Proceedings of ASME International Design Engineering Technical Conferences & Computers and Information in, Engineering Conference, DETC2009-86522, 2009.
[71] Joshua R. Maximoff, M.D. Trela, D.R. Kuhn, R.N. Kacker, A method for analyzing system state-space coverage within a t-wise testing framework, in: Proceedings of 4th Annual IEEE Systems Conference, 2010, pp. 598–603.
[72] D. Richard Kuhn, R.N. Kacker, Yu Lei, Random vs. combinatorial methods for discrete event simulation of a grid computer network, in: Proceedings of Modeling and Simulation, World, 2009, pp. 83–88.
[73] Wenhua Wang, Yu Lei, D. Liu, D. Kung, C. Csallner, D. Zhang, R.N. Kacker, D.R. Kuhn, A combinatorial approach to detecting buffer overflow vulnerabilities, in: Proceedings of 41st Annual IEEE/IFIP International Conference on Dependable Systems and, Networks, 2011, pp. 269–278.
[74] Carmelo Montanez-Rivera, D.R. Kuhn, M. Brady, R.M. Rivello, J. Reyes, M.K. Powers, Evaluation of fault detection effectiveness for combinatorial and exhaustive selection of discretized test inputs, ASQ Software Quality Professional 14 (2012) 32–38.
[75] D. Richard Kuhn, J.M. Higdon, J. Lawrence, R.N. Kacker, Yu Lei, Combinatorial methods for event sequence testing, in: Proceedings of 5th IEEE International Conference on Software Testing, Verification and Validation, Workshop on Combinatorial Testing, 2012, pp. 601–609.
[76] Sreedevi Sampath, R.C. Bryce, G. Viswanath, V. Kandimalla, A.G. Koru, Prioritizing user-session-based test cases for web applications testing, in: Proceedings of 1st IEEE International Conference on Software Testing, Verification and Validation, 2008, pp. 141–150.
[77] D. Richard Kuhn, R.N. Kacker, Yu Lei, Practical Combinatorial Testing, NIST Special Publication 800-142, 2010 <http://csrc.nist.gov/groups/SNS/acts/documents/SP800-142-101006.pdf>.