1 Evolution from Design of Experiments (R. Kacker)

Raghu N. Kacker, D. Richard Kuhn, Yu Lei, James F. Lawrence

Combinatorial testing of software can be viewed as an adaptation of design of experiment (DOE) methods for industrial applications. Although the general outline of DOE and combinatorial software testing are similar, the application of DOE to software is not a simple translation of DOE methods to a different domain. The unique characteristics of software have required significant advances, and understanding how these developments occurred can deepen our understanding of combinatorial testing.

1.1 Background

As discussed in previous chapters combinatorial testing is a type of dynamic testing in which distinct (but related) test factors are specified. Dynamic testing means that the software system is exercised (run) to obtain information about it. The test factors may have all possible values on a continuous scale or on a discrete scale. In either case a set of few discrete possible values is selected by equivalence partitioning, by boundary value analysis, or by expert judgment. Thus for each test factor a set of few discrete test settings is specified. Each test case is then expressed as a combination of one test setting for every test factor [Mandl (1985), Tatsumi (1987), Ammann and Offutt (2008), Mathur (2008)]. Suppose there are k test factors of which k_1 have v_1 test settings each, k_2 have v_2 test settings each, ... , and k_n have v_n test settings each. A test case consists of a combination (k-tuple) of k test settings one for each test factor, where $k_1 + k_2 + \ldots + k_n = k$. The number of different test cases possible is the product of all k test values. For example, if $k_1 = 3$, $v_1 = 3$, $k_2 = 4$, $v_2 = 4$, $k_3 = 2$ and, $v_3 = 5$, then we have $3 + 4 + 2 = 9$ test factors (3 with three test settings each, 4 with four test settings each, and 2 with five test settings each) and the number of possible test cases is $3^3 4^4 5^2 = 172\,800$. The exponential expression $3^3 4^4 5^2$ represents the combinatorial test structure of the k test factors and its expanded form 172 800 is the number of possible test cases. In many practical applications the number of possible test cases is too large to test them all. Combinatorial testing methods are used to determine a small set of test cases (test suite) which exercise the test settings of each test factor and their combinations with the objective that the test suite will cover the combinations for which the SUT could possibly fail.

The test factors and their test settings, and the expected behavior of the SUT are determined from the documents of requirements, descriptions of system implementations and internal operations, and all other available information about the SUT. The choice of test factors and their test settings defines and limits the scope of the combinations that are tested. Clearly, faults involving factors and settings which are not chosen for testing may not be exercised and revealed [Tatsumi (1987)]. Methods for specification of test factors and their test settings are largely application domain specific and a subject of continuing research [Ammann and Offutt (2008)].

Orthogonal arrays (OAs) are mathematical objects which are used as templates for designs (plans) of industrial experiments. Their key property is that every t-way combination appears *exactly the same number of times*. Covering arrays are generalizations of orthogonal arrays which are especially suited for software testing. They require each t-way combination to appear *at least once*. Covering arrays offer significant advantages over orthogonal arrays for testing software. Combinatorial testing began as

pairwise testing in which first orthogonal arrays and then covering arrays were used to make sure that all test settings of each factor and all pairs of the test settings were tested (excluding invalid test cases). In section 2, we discuss pairwise testing. Subsequent investigations of the reports of actual software failures showed that pairwise (2-way) testing is often useful but it may not be sufficient [Kuhn et al (2004)]. In addition, test factors and test settings are always subject to various types of constraints imposed by the runtime environment and the semantics of the SUT. In section 3, we discuss combinatorial (*t*-way) testing (CT) for $t \geq 2$ with support of constraints to exclude invalid combinations. Combinatorial testing for $t \geq 2$ is has became practical because of important advances in algorithms and tools for generating test suites for *t*-way testing with support of constraints. Brief summary and concluding remarks appear in section 4.

1.2 Pairwise (2-way) testing of software systems

The concept of orthogonal arrays (OAs) was formally defined by Rao (1947). OAs are mathematical arrangements of symbols which satisfy certain combinatorial properties. OAs are generalizations of combinatorial arrangements called Latin squares [Raghavarao (1971)]. 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.

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

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. The number of times each t-tuple appears is called its index. In the context of an OA, elements such as 0, 1, 2, …, (v – 1) are symbols rather than numbers. The combinatorial property does not depend on 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. A fixed-value orthogonal array may also be denoted by OA(N, k, v, t). 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. Mathematics of orthogonal arrays and extensive references can be found in Hedayat et al (1999). An electronic library of known OAs is maintained by Sloane (webpage).

The term *design of experiments* 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 test factors). The corresponding values of one or more output variables (called system responses) are measured to generate information for improving the performance of a class of similar systems. Conventional DoE methods were developed in the 1920s largely by Fisher (1925, 1935) and his contemporaries and followers to improve agricultural production. Later DoE were adapted for experiments with animals and then to improve manufacturing processes subject to uncontrolled variation. Frequently, the effects of many test factors each having multiple test settings are investigated at the same time and the DoE plans satisfy relevant combinatorial properties [Cochran and Cox (1950), Kempthorne (1952), Snedecor and Cochran (1967), Box, Hunter, and Hunter (1978), and Montgomery (2004)].

An engineer named Genichi Taguchi promulgated a variation of conventional DoE methods in Japan (1960s-1970s), USA (1980s), and elsewhere [Taguchi (1986, 1987, and 1993)]. The objective in conventional DoE is to improve average response; but the objective in Taguchi DoE is to determine test settings at which the variation due to uncontrolled factors was least [Kackar (1985) and Phadke (1989)]. Taguchi promoted use of OAs of strength two as templates for his DoE plans. Before Taguchi's use of OAs, they were not well known outside the world of mathematics. A salient property of DoE based on OAs is that they enable evaluation of a statistic called the main effect of each test factor [Box, Hunter, and Hunter (1978)]. The main effect of a test factor is the average effect over all test settings of every other factor on the objective function (which may be the average response or the average of a measure of variation of response). Evaluation of main effects is important in DoE because for each test factor optimal test setting should apply to the ranges of the values rather than fixed values of the other test factors.

Along with the advent of computers and tele-communication systems based on software, the problem of testing software and software embedded systems became important in the 1980s. Taguchi inspired the use of OAs for testing software systems. Software engineers in various companies (especially Fujitsu in Japan and descendent organizations of the AT&T Bell System in the US) started to investigate use of DoE plans and OAs for testing software systems. The earliest papers include Sato and Shimokawa (1984), Shimokawa (1985), Mandl (1985), Tatsumi (1987), and Tatsumi et al (1987)]. Tatsumi (1987) is a signal paper which included two important insights. (1) In software testing all combinations need not be tested the same number of times, only each combination needs to be tested at least once. (2) In generating test suites for testing software invalid combinations and test cases must be excluded.

In the US, starting in the late 1980s, Phadke (1989) and his colleagues developed a tool called OATS (Orthogonal Array Testing System) to generate test suits (based on OAs of strength 2) for Taguchi DoE and for testing software based systems [Brownlie et al (1992)]. OATS was an AT&T proprietary tool for intra-company use only. Uses 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 of software systems. Pairwise testing is a type of dynamic testing in which the SUT 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 discrete test settings are tested (at least once). Pairwise testing is an economical alternative to exhaustive testing of all combinations of the test factors. For example, if we had 9 test factors with the combinatorial test structure $3^3 4^4 5^2$ then exhaustive testing would require $3^3 4^4 5^2 =$ 172 800 tests. However pairwise testing requires only 29 test cases, a dramatically smaller number than 172 800 test cases required for exhaustive testing Thus pairwise testing could greatly improve the efficiency of dynamic testing of software. Many faults in software involve only one or two factors. Therefore pairwise testing was found to be effective in detecting faults.

When a software tester [Sherwood (1994)] 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 suits for software testing. Often, an OA matching the required combinatorial test structure does not exist. Also, frequently, OA based test suites included invalid test cases. Suppose four factors have two test settings each and one has three test settings. Thus the 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 elements in the row 7 and the row 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 get the combinatorial arrangement shown in table 2. Table 2 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 2 is shown in table 3. To obtain table 3 from table 2, the five test factors OS, Browser, Protocol, CPU type and the DBMS are associated with the five columns of table 2 and the elements in the columns {0, 1, 2} are replaced with the test settings of the respective test factors. The 8 rows of table 3 form a test suite for pairwise testing.

Table 2: 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

Table 3: Pairwise testing suite based on Table 2

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

Since the browser Internet Explorer (IE) does not run on the OS Linux, the pair {Linux, IE} appearing in test cases 6 and 8 is invalid. Therefore, these two test cases are not legitimate and cannot be executed. In that case other valid pairs of test settings covered by these two test cases are not tested, for example, the pairs {IPv6, Oracle}, {Intel, Oracle}, {IPv4, Oracle}, and {AMD, Oracle} are not tested. Thus the test suite shown in table 3 with test cases 6 and 8 omitted would not test all valid pairs of test settings.

Since OAs are not available for many combinatorial test structures and since test suites based on OAs may include invalid test cases, in the early 1990s, Sherwood (1994) developed a tool called CATS (Constrained Array Test System) to generate test suites which covered all valid combinations of test settings with a small number of test cases. 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 mathematical arrangements of symbols called covering arrays.

The concept of Covering Arrays (CAs) was formally defined by Sloane (1993). Significant earlier contributions leading to the concept of CAs include: Renyi (1971), Kleitman and Spencer (1973), and Roux (1987). Additional references on CAs can be found in Lawrence et al (2011) and Torres-Jimenez et al (2012).

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 strength of the covering array. A fixed value orthogonal array may also be denoted by CA(N, k, v, t). A mixed-value covering array is an generalization of the concept 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. Unlike an OA, 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. (A fixed-value orthogonal array of index one is the smallest covering array.)

Methods for constructing CAs can be put in three categories. (1) *Algebraic methods* [for example, Bush (1952), Roux (1987), Sloane (1993), Hartman (2005), and others]. (2) *Meta-heuristic methods* such as simulated annealing and tabu search [for example, Cohen et al (2003), Colbourn (2004), Nurmela (2004), Cohen et al (2008), Torres-Jimenez et al (2012), and others]. (3) *Greedy search methods*, [Lei and Tai (1998, 2001), Tai and Lei (2002), Lei et al (2007b, 2008), Forbes (2008), and others]. Algebraic methods apply only to certain special combinatorial test structures; however, when they apply they are extremely fast mathematical techniques and may produce CAs of smallest possible size. Meta-heuristic methods are computationally intensive and they have produced some 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. A particular greedy algorithm has produced some CAs of the smallest size known [Forbes (2008)]. The three approaches sometimes combined to yield additional methods for constructing covering arrays [Lei et al (2008), Sherwood et al (2006), Walker II et al (2009), and others].

Colbourn (webpage) maintains a web page of smallest known sizes (N) of various covering arrays CA(N, k, v, t) of strength t up to seven. The site gives the current best known upper bound on the least number of rows (the least number of rows is the covering array number CAN(t, k, v)). Web pages showing CAs tables of smallest known sizes include the following Forbes (webpage), Nurmela (webpage), and Torres-Jimenez (webpage).

The use of covering array for software testing was promulgated by Dalal and Mallows (1998) among others. They noted that evaluation of main effects is important in DoE and OAs enable evaluations of the main effects; however, in testing software systems there is no need to evaluate 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. It turns out that CAs have several advantages over OAs. (1) CAs can be constructed for test factors with arbitrary combinatorial test structures of unequal numbers of test settings. (2) For a combinatorial test structure if an OA exists then a CA of the same or fewer test runs 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 [Sloane (webpage)]. (4) In generating test suites for combinatorial testing certain combinations may be illegitimate (invalid) and they need to be excluded; in constructing test suites based on CAs illegitimate combinations can be deliberately excluded (in that case the combinatorial property of the test suite applies only to those combinations which are included.)

To our knowledge, the first publicly available tool for generating test suites based on CAs for pairwise (and higher strength) testing of software systems was AETG [Cohen et al (1994, 1996, and 1997)]. In 1998, a graduate student Yu Lei developed an algorithm called IPO for generating test suites for pairwise testing based on CAs which excluded invalid pairs of test settings [Lei and Tai (1998, 2001)]. Usefulness of CAs for pairwise testing led to a great interest among mathematicians and computer scientists to develop tools for generating test suites based on CAs.

1.3 Combinatorial t-way testing of software systems

NOTE: REFERENCE LIST MAY BE MODIFIED WITH DIFFERENT CITATION NUMBERS

A team of NIST researchers investigated fifteen years of recall data due to failures of software embedded in medical devices [190] and did a series of followup studies, including failure reports for a browser, a server, and a NASA database system [91, 92, 93]. The primary purpose of the initial investigation was to generate insights into the kinds of software testing that could have detected the underlying faults and prevented the failures in use. In the subsequent investigations the researchers were able to determine the numbers of individual factors that were involved in the faults underlying actual failures of software, leading to the empirically derived of the Interaction Rule introduced in Chapter 1: *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.* The maximum degree of interaction in actual faults so far observed is six.

The interaction rule suggests that pairwise (2-way) testing is useful but it is may not always be sufficient. Combinatorial *t*-way testing for *t* greater than 2 is often needed. This conclusion is also supported by investigations by other researchers, including a study of network security data [14, 15], audio file processing software [199], and an early study of NASA software failures [61]. These insights from investigation of actual failures motivated the authors of this chapter to advance methods and tools for combinatorial *t*-way testing of software for $t \geq 2$.

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 $k \geq t$) all *t*-tuples of the test settings are tested at least once (disallowed combinations being excluded). When the SUT fails for one or more test cases, the pass/fail data and various approaches are used to isolate the failure-inducing combinations of test settings. To isolate failure-inducing combinations additional tests may be required [Laleh et al (2012)].

CT methods can significantly improve the efficiency of software testing. Suppose faults involving more than *t* test factors are unlikely then *t*-way testing would be as good as exhaustive testing requiring a larger number of test cases. A conformance testing for standards conducted by Montanez-Rivera et al (2012) demonstrated the efficiency of combinatorial (*t*-way) testing relative to exhaustive testing by showing that 4-way testing could detect all faults found by significantly more expensive exhaustive testing.

An alternative to CT is random testing in which test cases are formed from random draws of the discrete test settings of the test factors. As discussed in Chapter 9, combinatorial testing generally requires fewer test cases than random testing for equivalent coverage [Kuhn et al (2009)]. The benefits of CT in terms of reduced testing effort over random testing increase with *t*.

Combinatorial (*t*-way) testing may be regarded as an adaptation of the DoE methods for testing 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. Unlike DoE, in CT for each possible test case the expected behavior of the system is pre-determined from the available information such as a mathematical model of the SUT. A software tool is often used to check whether the actual behavior matches the expected behavior and to make a verdict of passing or failing for each test case. Generally, CT does not require access to the source code; however, once one or more failure-inducing combinations have been identified, the source code may be needed in the follow up investigations to reveal the underlying faults in the SUT.

Combinatorial testing is a versatile methodology which could be useful in a broad range of testing situations. As discussed earlier, CT can be used for (i) testing various configurations of a system (configuration testing) and for (ii) testing various possible inputs to the system (testing input space) [Mathur (2008)]. A software system may receive inputs from a user or from another system. Suppose a system accepts ten input test factors with binary test settings: {off, on}. For example the inputs factors could be choice of font options in word processing such as subscript, superscript, underline, bold, italic, strikethrough, emboss, shadow, small cap, and all cap. The total number of test cases is $2^{10} = 1024$. Suppose no fault may involve more than 3 factors (which is a reasonable assumption), then it would be sufficient to test all 3-way combination; their number being 960. It turns out that a test suite of only 13 test cases (determined from a CA) can exercise all 3-way combinations [Kuhn et al (2008)]. This is a very small fraction of the 1024 total number of test cases.

CT can also be used for testing (1) data-bases and (2) state models [Sherwood (2011)]. We have investigated use of CT for the following particular applications. (1) Testing concurrent systems [Lei et al (2007a)], (2) Testing web applications [Wang et al (2008)], (3) Security testing of access control implementations [Hu et al (2008)], (4) Navigation of dynamic web structures [Wang et al (2009)], (5) Optimization of discrete event simulation models [Johansson et al (2009)], (6) Analyzing system state-space coverage [Maximoff et al (2010)], (7) Detecting deadlocks for varying network configurations [Kuhn et al (2009)], (8) Detecting buffer overflow vulnerabilities [Wang et al (2011)], (9) Conformance testing for standards [Montanez-Rivera et al (2012)], (10) Event sequence testing [Kuhn et al (2012)], (11) Prioritizing user session based test suits for web applications [Sampath et al (2010)].

We have developed a tool called ACTS for generating *t*-way combinatorial test suites for arbitrary combinatorial test structures and any strength *t* with support of constraints (to exclude invalid combinations). The tool ACTS is a freely distributed research tool downloadable from a NIST web site to anyone who emails request [Kuhn et al (webpage)]. 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 mixed-strength test suites. For example, of the 10 test factors all could be covered with strength 2 and a particular subset of 4 (which are known to be inter-related) could be covered with higher strength 4. (4) ACTS verifies whether a test suite supplied by user covers all *t*-way combinations. (5) ACTS allows 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. More information about ACTS is given in the Appendix.

The following example shows that the sizes of *t*-way test suites increase 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 of which 3 have 3 test settings each, 4 have 4 test settings each and 2 have 5 test settings each [Kuhn et al (2010)]. Thus the combinatorial test structure is $3^3 4^4 5^2$ and the total number of possible test cases is $3^3 4^4 5^2 = 172\,800$. Exhaustive testing is not practical. The sizes (number of test cases) in *t*-way test suites determined using ACTS [Kuhn et al (webpage)] for $t = 2, 3, 4, 5,$ and 6 are respectively 29, 137, 625, 2532, and 9168. Therefore many software testers hesitate to do *t*-way testing for *t* more than 2 or 3. In some situations it may be more effective to execute *t*-way test suites for $t = 2$ or 3 with different test factors and settings than execute higher strength test suites. On the other hand suppose an output of a SUT is known to depend on the values of *t* input variables where $t \geq 2$. The value of *t* may be more than 4, or 5. So faults involving four or more factors may be possible. In that case it may be necessary to do combinatorial *t*-way testing for $t \geq 4$.

The challenges in use of CT include at least 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. These challenges are the subject matter of on-going research involving industry, academia, and government research laboratories.

A failure inducing fault may not be detected when the test factors and settings associated with that fault are not included in the test plan and exercised. When continuous-valued factors are involved, their chosen discrete test settings preclude testing certain values. Therefore testing can detect faults but it cannot guarantee their absence.

It is difficult to categorize the kinds of faults a software system can have. Multiple testing methods during and after development are generally needed to assure correctness of software. Combinatorial testing complements other software testing methods.

1.4 Chapter Summary

Combinatorial (*t*-way) testing may be regarded as an adaptation of design of experiment methods for testing software systems because in both cases information about the SUT is gained by exercising it and the test plan satisfies relevant combinatorial properties. Indeed, CT evolved from the use of DoE plans based on OAs for generating test suits for software testing. CT requires specification of test factors and discrete test settings for each. A test case is a combination of one selected test setting for each test factor. CT began as pairwise (2-way) testing in which the SUT 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, OAs could not support constraints among the test settings of test factors. Therefore covering arrays were found to be better suited than OAs for combinatorial *t*-way testing for $t \geq 2$. Investigations of actual faults indicated that while pairwise ($t = 2$) testing is useful it usually is not adequate. Therefore combinatorial *t*-way testing for *t* greater than 2 may be needed. Combinatorial *t*-way testing for $t > 2$ 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. As with any test method, CT can detect faults but it cannot guarantee their absence. CT is one of many complementary methods for software assurance.