A Survey of Combinatorial Testing

CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University
HARETON LEUNG, Hong Kong Polytechnic University

Combinatorial Testing (CT) can detect failures triggered by interactions of parameters in the Software Under Test (SUT) with a covering array test suite generated by some sampling mechanisms. It has been an active field of research in the last twenty years. This article aims to review previous work on CT, highlights the evolution of CT, and identifies important issues, methods, and applications of CT, with the goal of supporting and directing future practice and research in this area. First, we present the basic concepts and notations of CT. Second, we classify the research on CT into the following categories: modeling for CT, test suite generation, constraints, failure diagnosis, prioritization, metric, evaluation, testing procedure and the application of CT. For each of the categories, we survey the motivation, key issues, solutions, and the current state of research. Then, we review the contribution from different research groups, and present the growing trend of CT research. Finally, we recommend directions for future CT research, including: (1) modeling for CT, (2) improving the existing test suite generation algorithm, (3) improving analysis of testing result, (4) exploring the application of CT to different levels of testing and additional types of systems, (5) conducting more empirical studies to fully understand limitations and strengths of CT, and (6) combining CT with other testing techniques.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Design, Experimentation, Reliability, Verification

Additional Key Words and Phrases: Software testing, combinatorial testing (CT), test case generation, covering array

ACM Reference Format:
Nie, C. and Leung, H. 2011. A survey of combinatorial testing. ACM Comput. Surv. 43, 2, Article 11 (January 2011), 29 pages.
DOI = 10.1145/1883612.1883618 http://doi.acm.org/10.1145/1883612.1883618

1. INTRODUCTION

The use of computers is entering into every corner of social life and is becoming the important engine of economical and social progress. As the kernel of information technology, software has become a key issue affecting social and economic development. To improve software quality, many software testing methods have been proposed and studied. Different testing methods target different types of fault, and can be effective

This work was supported by the National Natural Science Foundation of China (60773104, 60721002, 90818027), 863 high technical plan of China (2008AA01Z143, 2009AA01Z147).
This article was partially written during C. Nie's sabbatical, sponsored by the Chinese Scholarship Council in CREST (Center for Research in Evolution, Search, and Testing), King's College London.
Authors' addresses: C. Nie, State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, P. R. China 210093; email: changhainie@nju.edu.cn; H. Leung, Department of Computing, the Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong; email: cshleung@inet.polyu.edu.hk.

for certain testing scenarios. For example, mutation testing, which involves modifying the source-code in a small way, can help the tester develop effective tests, identify weaknesses in the test suite, or locate sections of the code that are seldom or never accessed during execution [Offutt 1994]; metamorphic testing can solve the oracle problem in testing based on the property of Software Under Test (SUT) [Dong et al. 2007]; structural testing aims to find faults related to the internal structure of SUT [Marre and Bertolino 2003]. In this article we focus on Combinatorial Testing (CT), also called Combinatorial Interaction Testing (CIT). CT tests SUT with a covering array test suite which tests all the required parameter value combinations. The advantage of CT is that it can detect failures triggered by the interactions among parameters in SUT.

Suppose we want to test a network game software running in the Internet environment. The operation of this game may be influenced by many parameters, such as browser, operating system, the type of network access, graphics, audio, the number of players, and so on. Each of these parameters may take on many possible values. The interactions of these parameters may cause some failures. Due to the large combination space, exhaustive testing by testing all the parameter value combinations is generally impractical. Even if we have the resources to try all value combinations, this is not effective because most of the value combinations do not cause any failure. CT provides a practical way to detect failures caused by parameter interactions with a good trade-off between cost and efficiency. It samples the large combination space using a smaller test suite to cover certain key parameter value combinations.

As the software functions become more complex, and the running environments become distributed, networked, and more complicated, modern software systems need to be designed to be highly configurable so that they can run on and be optimized for a wide variety of platforms and support various usage scenarios. With the adoption of new software development technology, such as component-based software and the service-oriented software, more and more software systems tend to have many parameters. Interactions of these parameters may cause failures. As a result, software testing often faces the problem of a large test combination space. These trends create a heavy demand for more intelligent test data sampling mechanisms to detect interaction triggered failures. CT offers such an effective testing technique which has been well studied and widely used in the last 20 years.

Actually no single software testing technique can provide complete testing. Each type of testing technique selects test cases using its own unique sampling approach, relying on some knowledge of the system to select a subset of tests to execute [Salem et al. 2004]. For example, structural testing selects test cases guided by some structural coverage of source-code. CT selects test cases with a sampling mechanism to systematically cover parameter value combinations using a small test set which is relatively easy to manage and execute. It can avoid the combination explosion problem of exhaustive testing.

CT has the following characteristics: (1) CT creates test cases by selecting values for parameters and by combining these values to form a covering array. The covering array specifies test data where each row of the array can be regarded as a set of parameter values for a specific test. The collection of tests (represented by the rows of the array) covers all τ-way combinations of parameter values, where τ specifies the number of parameters in combination. An example is given in Figure 3, which shows a 2-way covering array for 4 parameters with three values each. The interesting property of this array is that any two columns contain all nine possible value combinations for two specific parameters corresponding to the columns. For example, taking columns c_1 and c_2, we can see that all nine possible 2-way combinations occur somewhere in the rows of the two columns. In fact, this is true for any two columns. Collectively, this set of tests will exercise all 2-way combinations of input values in only 9 tests, as compared with 81 for exhaustive coverage. (2) CT uses a covering array as the test suite. The covering array aims to test as many parameter value combinations as possible in order to detect failures triggered by parameter interactions. (3) Not every parameter of SUT can trigger a fault, and some faults can be exposed by testing interactions among a small number of parameters. Kuhn et al. studied the faults in several software projects, and found that all the known faults are caused by interactions among 6 or fewer parameters [Kuhn and Reilly 2002; Kuhn and Wallace 2004]. Based on this result, CT can be very effective for certain types of applications, with performance approaching that of exhaustive testing while using fewer test cases. (4) Being a specification-based testing technique, CT requires no knowledge about the implementation of SUT. Also, the specification required by CT is "lightweight," since we only need to know the basic system configuration to identify the input parameters and their possible values. (5) Test generation for CT can be automated, which is a key to gaining a wide industrial acceptance.

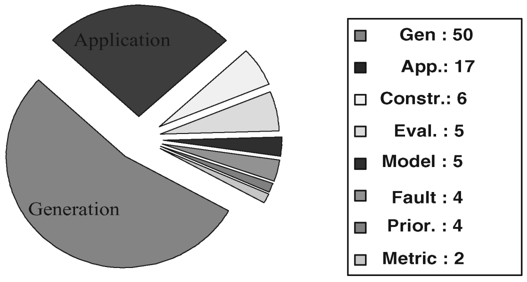
In software testing, there is no guideline that dictates the best course of testing, and there are no "best practices" that can always guarantee success [Bach and Schroeder 2004]. Although CT is useful in detecting certain faults, it can create a false confidence because: (1) CT may be viewed as providing a kind of shortcut of software testing. It has its own pitfalls, such as not testing all possible parameter combinations. (2) If the parameters and their values are not selected properly, this will lower the defect detection capability of CT. (3) If we fail to identify all the interactions between parameters in SUT, CT will not test those "missed" interactions. (4) If we do not have a "good enough" oracle, the verification of testing results will be difficult. To ensure successful testing, we should apply CT wisely. This requires professional skill and good judgement in its application. The full strengths and weaknesses of CT need to be better understood.

In this article, we survey the state of the research of CT. In our study, we have collected over 90 key papers related to CT. We classify these papers into eight categories.

(1) Modeling (Model): Studies on identifying the parameters, values, and the interrelations of parameters of SUT.
(2) Test case generation (Gen): Studies on generating a small test suite effectively.
(3) Constraints (Constr.): Studies on avoiding invalid test cases in the test suite generation.
(4) Failure characterization and diagnosis (Fault): Studies on fixing the detected faults.
(5) Improvement of testing procedures and the application of CT (App.): Studies on practical testing procedure for CT and reporting the results of the CT application.
(6) Prioritization of test cases (Prior.): Studies on the order of test execution to detect faults as early as possible in the most economical way.
(7) Metric (Metric): Studies on measuring the combination coverage of CT and the effectiveness of fault detection.
(8) Evaluation (Eval.): Studies on the degree to which CT contributes to the improvement of software quality.

Note that most of categories are nonexclusive; that is, a paper may focus on modeling of CT and also on evaluation. We assign each paper to one category based on the main objective of the paper. Thus, our classification is subjective and some papers may be classified into another category by other researchers. Nevertheless, we believe that on the whole, the distribution shown in Figure 1 represents a fairly good picture of the current state of research in CT.

The remaining part of this article is organized as follows: Section 2 presents the commonly used terms of CT. Section 3 discusses the eight categories of research, and reviews the contributions from various research groups and the growing trend of CT. In the last section, we make recommendations on the future direction of research in CT.



Total number: 93 papers in the references

Fig. 1. Distribution of CT research areas.

c_1:Browser	c_2:Os	c_3:Access	c_4:Audio
Netscape	Windows	ISDL	Creative
IE	Linux	Modem	Digital
Firefox	Macintosh	VPN	Maya

Fig. 2. Configuration parameters for NGS.

2. BACKGROUND

In this section, we give an overview of CT, including some formal notations and definitions commonly used in the literature. From the successive definitions of test suites for CT, we can understand the evolution of CT. We also review the test case generation methods for CT.

2.1. Notations

We assume the Software Under Test (SUT) has n parameters $c_i (i = 1, 2 \cdots n)$, which may represent configuration parameters, internal or external events, user inputs, etc., and these parameters or their interactions may influence the SUT. Let c_i have a_i discrete values from the finite set V_i, $a_i = |V_i|$. Assume all the parameters are independent, that is, none of the parameter values is determined by the others. Without loss of generality, we assume $a_1 \geq a_2 \geq \cdots \geq a_n$.

Let R be the set of interaction relations which records all the interactions existing among parameters. This information can be obtained from various development documents of the SUT or from interviews with related domain experts. $R \subseteq 2^C$, where $C = \{c_1, c_2, \ldots, c_n\}$. For example, given $R = \{\{c_1\}, \{c_1, c_2\}, \{c_2, c_3\}, \{c_3, c_4, c_5\}, \{c_4, c_5\}\}$. $\{c_1\} \in R$ shows that all the values of parameter c_1 in V_1 can affect the SUT and may trigger a software failure. $\{c_1, c_2\} \in R$ shows that there exist interactions between c_1 and c_2, that is, all the pairs in $V_1 \times V_2$ may trigger a software failure. $\{c_3, c_4, c_5\} \in R$ shows that there exist interactions among c_3, c_4, and c_5, and all the combinations in $V_3 \times V_4 \times V_5$ may affect the SUT. R can be viewed as the covering requirements for CT, specifying which combinations should be covered in testing.

To present the key concepts of CT, we will make use of the following running example.

Example. Consider a Network Game Software (NGS) which may be influenced by the configuration parameters as shown in Figure 2. In our example, $n = 4$, $a_1 = a_2 = a_3 = a_4 = 3$, $V_1 = \{Netscape, IE, Fire\,fox\}$, $V_2 = \{Windows, Linux, Macintosh\}$, $V_3 = \{ISDL, Modem, VPN\}$, and $V_4 = \{Creative, Digital, Maya\}$. $R = 2^C/\emptyset = \{\{c_1, c_2, c_3, c_4\}\}$, where $C = \{c_1, c_2, c_3, c_4\}$, that is, here all the parameters and their combinations may affect the NGS.

	c_1:Browser	c_2:Os	c_3:Access	c_4:Audio
t_1	Netscape	Windows	ISDL	Creative
t_2	Netscape	Linux	Modem	Digital
t_3	Netscape	Macintosh	VPN	Maya
t_4	IE	Windows	Modem	Maya
t_5	IE	Linux	VPN	Creative
t_6	IE	Macintosh	ISDL	Digital
t_7	Firefox	Windows	VPN	Digital
t_8	Firefox	Linux	ISDL	Maya
t_9	Firefox	Macintosh	Modem	Creative

Fig. 3. Test suite generated for pairwise testing (2-way testing).

Definition 2.1.1 (*Test Case*). $n-$tuple $(v_1, v_2, \ldots, v_n)$ where $(v_1 \in V_1, v_2 \in V_2, \ldots, v_n \in V_n)$ is a test case t or a test configuration of the SUT. Let T_{all} denote the set of all possible test cases for the SUT, that is, $T_{all} \subseteq \{(v_1, v_2, \ldots, v_n)|(v_1 \in V_1, v_2 \in V_2, \ldots, v_n \in V_n)\} = V_1 \times V_2 \times \ldots \times V_n$.

For any given SUT, there exists a T_{all} which is determined by the system parameters, their values and interactions, and some other constraints. For example, a test case t for the NGS is the 4−tuple (Netscape, Windows, ISDL, Creative). NGS has a maximum of $3^4 = 81$ test cases based on all possible parameter value combinations.

Obviously, T_{all} is generally too large to be completely executed in testing, and in fact it is not necessary to run all tests in T_{all}. In this article T_c will denote the test suite generated for CT, and obviously $T_c \subseteq T_{all}$.

The interaction between parameters is in fact the effect caused by the combination of the parameter values. When specific values of these parameters are used together, they may trigger a software failure. A combination of parameter values can be viewed as a value schema.

Definition 2.1.2 (*Value Schema*). For the SUT, the $n-$tuple $(-, v_{n_1}, \ldots, v_{n_k}, \ldots)$ is called a $k - value$ schema $(k > 0)$ when some k parameters have fixed values and the others can take on their respective allowable values, represented as " $-$ ". When $k = n$, the $n-$tuple becomes a test case for the SUT as it takes on specific values for each of its parameters [Shi et al. 2005].

CT is good at detecting failures triggered by some interaction among parameters, which can be represented by a combination of some k parameter values or a $k-$value schema. We can say that it is some $k-$value schema of the SUT that causes the software failure.

For example, if a defect is revealed when we test the NGS with the test case $t = (Netscape, Windows, ISDL, Creative)$, it may be triggered by one of the parameter values, that is, any one of 1-value schemas in $\{(Netscape, -, -, -), (-, Windows, -, -), (-, -, ISDL, -), (-, -, -, Creative)\}$; or by the interaction of some two parameters, that is, any one of 2−value schemas in $\{(Netscape, Windows, -, -), (-, Windows, ISDL, -), (-, -, ISDL, Creative), (Netscape, -, ISDL, -), (-, Windows, -, Creative), (Netscape, -, -, Creative)\}$; or by the interaction of some three parameters, that is, any one of 3−value schemas in $\{(-, Windows, ISDL, Creative), (Netscape, -, ISDL, Creative), (Netscape, Windows, -, Creative), (Netscape, Windows, ISDL, -)\}$; or by the four specific parameter values of t: $(Netscape, Windows, ISDL, Creative)$. In this case, there is a total of $2^4 - 1 = 15$ possible causes.

Let t be a failed test case of SUT, all the $k - value$ schema form a set M_t, $M_t = \{s_k = (-, v_{n_1}, \ldots, v_{n_k}, \ldots)|s_k$ is a $k - value$ schema in t $(k > 0)\}$. We have previously shown that there is a total of $2^n - 1$ schema in this set [Shi et al. 2005].

2.2. Definitions of Test Suite for CT

Next we give the definitions of orthogonal array, covering array, and variable strength covering array, and also present the evolution of test suites for CT at the same time.

Definition 2.2.1 (*Orthogonal Array*). For SUT, a strength τ Orthogonal Array(OA) (or a Mixed strength τ Orthogonal Array (MOA)), denoted as $OA(N; \tau, n, (a_1, a_2, \ldots, a_n))$ is an $N \times n$ array with the following properties.

(1) Each column $i (1 \leq i \leq n)$ contains only elements from the set V_i with $a_i = |V_i|$.
(2) The rows of each $N \times \tau$ subarray cover all $\tau - tuples$ of values from the τ columns exactly $\lambda = \frac{N}{a_{i_1} \cdots a_{i_\tau}}$ times.

The Orthogonal Array Number, $OAN(N; \tau, n, (a_1, a_2, \ldots, a_n))$, is the minimum N required to satisfy the parameter $\tau, n, (a_1, a_2, \ldots, a_n)$. When $a_1 = a_2 = \cdots = a_n = v$, the orthogonal array is written as $OA(N; \tau, n, v)$, and its orthogonal array number as $OAN(N; \tau, n, v)$ [Hartman 2002].

The orthogonal array requires each combination to be covered the same number of times [Mandl 1985; Brownlie et al. 1992]. The most widely used orthogonal array is the one with strength $\tau = 2$. Figure 3 shows a strength 2 orthogonal array for the NGS. There are nine rows in Figure 3, and each row represents a test case. Any two columns of Figure 3 form 9 different pairs. Note that Figure 2 can also be viewed as a strength 1 orthogonal array for the NGS, with 3 test cases covering all the parameter values.

The earliest proposed CT method was OATS (Orthogonal Array Testing System) which used an orthogonal array of strength 2 as the test suite. In general, the orthogonal array was difficult to generate and its test suite was often quite large. But OA has its advantages, such as making it relatively easy to identify the particular combination that caused a failure. Soon τ-way testing became more popular, which uses a $\tau - way$ covering array as the test suite. We next define the $\tau - way$ covering array.

Definition 2.2.2 (*Covering Array*). If an $N \times n$ array has the following properties:

(1) each column $i (1 \leq i \leq n)$ contains only elements from the set V_i with $a_i = |V_i|$;
(2) the rows of each $N \times \tau$ subarray cover all $\tau - tuples$ of values from the τ columns at least once, then

it is called a $\tau - way$ covering array (or a Mixed strength τ Covering Array (MCA)), denoted as $CA(N; \tau, n, (a_1, a_2, \ldots, a_n))$. The covering array number, $CAN(N; \tau, n, (a_1, a_2, \ldots, a_n))$, is the minimum N required to satisfy the parameter τ, n, and $(a_1, a_2, \ldots, a_n)$. When $a_1 = a_2 = \ldots = a_n = v$, the covering array is written as $CA(N; \tau, n, v)$, and its covering array number as $CAN(N; \tau, n, v)$ [Cohen et al. 2003a; 2007b].

Testing with a test suite of $\tau-$way covering array is called $\tau-$way testing. τ-way testing is a kind of CT which requires that every combination of any τ parameter values in the software must be tested at least once. When $\tau = 1$, it is called the Each Choice (EC) combination strategy. It becomes Pairwise Testing (PW) when $\tau = 2$. When $\tau = n$, it is called the All Combination (AC) strategy. From Definitions 2.2.1 and 2.2.2, we can see that an orthogonal array with strength τ is a τ-way covering array, but the reverse is not necessarily true. Figure 3 can also be viewed as a 2-way covering array for the NGS, and Figure 2 can be viewed as an 1-way covering array for the NGS.

A more practical and in some cases more flexible approach to examining interaction coverage than τ-way testing is the variable strength interaction testing [Cohen 2004]. Its test suite is called a variable strength covering array. This method uses a covering array that offers different covering strengths to different parameter groups and aims to

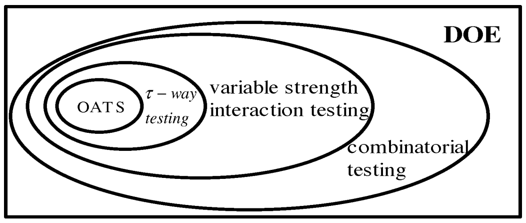


Fig. 4. Relation of several forms of CT and DOE.

cover only combinations of parameters having mutual interactions. This is important because interactions do not often exist uniformly between parameters. Some parameters will have strong interactions with each other while others may have few or no interactions. For this reason, we may from time to time wish to focus testing on a specific set of parameters and apply a higher strength testing on them without ignoring the rest. In this case, the variable strength covering testing may be more effective and efficient.

Definition 2.2.3 (*Variable Strength Covering Array*). Let R denote the set of the interaction relation set. R is the covering requirements for the SUT. Let CA be an $m \times n$ matrix, with the elements of column i of CA from V_i, which is the value set of parameter c_i. If CA covers all the combinations required by R, we call CA a covering array for the SUT. Every row of CA is a test case. When R has elements of different sizes, that is, $R = \{\{c_{i_1}, c_{i_2}, \ldots, c_{i_\tau}\} \mid 1 \le i_1 < i_2 < \cdots < i_\tau \le n, 1 \le \tau \le n\}$, CA is called a Variable strength Covering Array (VCA).

When $R = \{\{c_{i_1}, c_{i_2}, \ldots, c_{i_\tau}\} \mid 1 \le i_1 < i_2 < \cdots < i_\tau \le n, \tau \textit{ is fixed}\}$ and $|R| = C_n^\tau$, the covering array CA for the SUT is a τ-way covering array as defined earlier.

Testing with a variable strength covering array is called variable strength interaction testing. τ-way testing can be viewed as a special case of variable strength interaction testing, and both of them as special cases of CT. Actually CT can be expanded to include other important factors, such as constraints, and assigned test suite (seeds). CT can be viewed as a kind of Design Of Experiment (DOE), which is a structured method that is used to determine the relationship between the different factors affecting a process and the output of that process [Salem et al. 2004]. The relationships between various forms of CT are illustrated in Figure 4.

For prioritized combinatorial testing, [Turban 2006] defines a new type of covering array, called ℓ-biased covering array. An ℓ-biased covering array is a covering array $CA(N; 2, k, v)$ in which the first ℓ rows form tests whose total benefit is as large as possible. That is, no $CA(N'; 2, k, v)$ has ℓ rows that provide a larger total benefit. For instance, certain factors or levels for an input may have an associated benefit or priority that indicates a higher preference that the interaction be covered earlier in testing.

2.3. Test Case Generation Methods

To detect failures triggered by interactions among parameters, CT uses different sampling mechanisms to generate a test suite. There are several methods to generate a test suite for CT. In this section, we first present One Factor One Time (OFOT), a simple and common test case generation method, and its simplification. Then, we compare several common test generation strategies.

Definition 2.3.1 (*One Factor One Time Method*). Let $t = (v_1, v_2, \ldots, v_n)$ be an assigned test case of the SUT. Replace each parameter value v_i with the other values of the parameter c_i one by one, keeping the other parameter values of test case t the same. We can obtain $\sum_{i=1}^{n} a_i - n$ new test cases, and totally need $\sum_{i=1}^{n} a_i - n + 1$ test

t =(Netscape, Windows, ISDL, Creative)
t_{11}=(<u>IE</u>, Windows, ISDL, Creative)
t_{12}=(<u>Firefox</u>, Windows, ISDL, Creative)
t_{21}=(Netscape, <u>Linux</u>, ISDL, Creative)
t_{22}=(Netscape, <u>Macintosh</u>, ISDL, Creative)
t_{31}=(Netscape, Windows, <u>Modem</u>, Creative)
t_{32}=(Netscape, Windows, <u>VPN</u>, Creative)
t_{41}=(Netscape, Windows, ISDL, <u>Digital</u>)
t_{42}=(Netscape, Windows, ISDL, <u>Maya</u>)

generate 8 test cases by one factor one time from (Netscape, Windows, ISDL, Creative)

Fig. 5. Test cases generated from OFOT.

Strategy	EC	BC(OFOT)	PW	OA	τ-way($\tau > 2$)	AC
Size	$Max_i(a_i)$	$\sum_{i=1}^{n} a_i - n + 1$	$\sim a_1^2$	$\sim a_1^2$	$\sim a_1 \cdots a_\tau$	$\prod a_i$
Coverage	$EC \leq BC$	$BC \leq PW$	$PW \leq OA$	$OA \leq \tau$-way	τ-way$\leq AC$	all
Test for NGS	3 (Fig 2)	9 (Fig 5)	9 (Fig 3)	9 (Fig 3)	$3^3 = 27(\tau = 3)$	$3^4 = 81$

Fig. 6. Test cost and coverage of different strategies.

cases including the assigned test case. This method of test generation is called One Factor One Time, denoted as OFOT. It is also called the Base Choice(BC) combination strategy [Mats et al. 2006a].

This method requires every value of every parameter be included in at least one test case. For example, let $t = (Netscape, Windows, ISDL, Creative)$ be a test case for the NGS. We first replace the first parameter value "Netscape" by the other two-values and generate two new test cases: $t_{11} = (IE, Windows, ISDL, Creative)$, $t_{12} = (Firefox, Windows, ISDL, Creative)$. Then we replace the second parameter value. All the test cases generated by OFOT are listed in Figure 5.

The one factor one time method can be simplified as follows.

Definition 2.3.2 (*Simplified One Factor One Time Method*). Let $t = (v_1, v_2, \ldots, v_n)$ be an assigned test case of the SUT. We change each value in t one by one, and generate a test suite $\{t_1^t = (*, v_2, \ldots, v_n), t_2^t = (v_1, *, \ldots, v_n), \ldots, t_n^t = (v_1, v_2, \ldots, *)\}$, where * represents an allowable value that is different from the original value in t. This method is called the Simplified One Factor One Time, denoted as SOFOT.

For example, let $t = (Netscape, Windows, ISDL, Creative)$ be a test case for the NGS. The test suite generated by SOFOT from t can be: $T_{st} = \{t_1^t = (\underline{IE}, Windows, ISDL, Creative), t_2^t = (Netscape, \underline{Linux}, ISDL, Creative), t_3^t = (Netscape, Windows, \underline{Modem}, Creative), t_4^t = (Netscape, Windows, ISDL, \underline{Digital})\}$.

Test suites generated from OFOT and SOFOT can detect faults where a single parameter value causes a software failure. They are unlikely to detect faults that are triggered when two or more parameters simultaneously take certain values, because they can not cover τ-value schema (for $\tau \geq 2$). Different test suites for CT have different coverage and different costs.

In Figure 6, we compare the test cost and coverage of various testing strategies. The second row gives the testing cost as measured by the size of the test suite. The size of test suite for EC is $Max_i(a_i)$ because EC just needs to cover each parameter value once. The size of test suite for BC is $\sum_{i=1}^{n} a_i - n + 1$ because only one parameter value in the test case is changed at a time. The size of test suite for PW is at least $a_1 \times a_2$ or $\sim a_1 \times a_2$, because each test case can cover one combination of parameter c_1 and c_2 and there are $a_1 \times a_2$ combinations. The size of test suite for OA is at least $a_1 \times a_2$ because it needs to cover all the $a_1 \times a_2$ combinations of parameters c_1 and c_2 for the same number

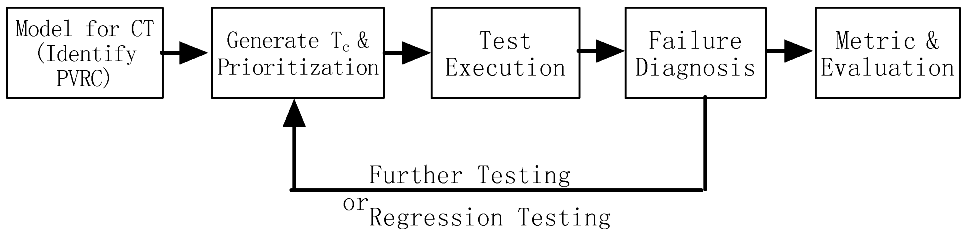


Fig. 7. The testing procedure for CT.

of times. We will approximate its size as $\sim a_1 \times a_2$. The size of test suite for τ-way is $\sim a_1 \cdots a_\tau$ because it needs to cover all the τ parameter value combinations, or the size of τ-way test suite can be given by $\sim a_1^\tau \lg n$ for n parameters. Finally, the size of test suite for AC is $\prod a_i$ since it needs to test all possible combinations.

The third row of Figure 6 gives the coverage of a test suite T, Cov(T), which can be measured by the number of k-value schema covered by T. Let T_{cover}={k-value schema | k-value schema is covered by T}, $Cov(T)$=$|T_{cover}|$. From the definitions provided in Section 2.2, we can see that the following relation holds.

$$Cov(EC) \leq Cov(BC) \leq Cov(PW) \leq Cov(OA) \leq Cov(\tau - way) \leq Cov(AC) \qquad (1)$$

The fourth row of Figure 6 gives the number of test cases required for the different test strategies for our example NGS system. Eq. (1) holds for our running example of NGS.

Manually generating the test suite for PW, OA, τ-way covering array($\tau > 2$), and AC is often difficult. Fortunately, there are many free tools available. For example, www.pairwise.org gives a list of test suite generation tools. Most of these free tools, such as FireEye, are reasonably user friendly and can ease the burden in applying CT.

3. THE STATE OF CT RESEARCH

In this section, we present the research topics of CT one by one. For each topic, we will discuss its motivation, key issues, solutions, the state of research, and some open issues.

By adopting the work of others into the typical testing lifecycle, we propose a generic procedure model of CT, as shown in Figure 7. The first step is to build a model of SUT (Section 3.1), and the second step generates a test suite T_c for CT (Section 3.2) and makes prioritization (Section 3.3). After the third step of test execution, if we find bugs and can fix them, then we can conduct regression testing; else, if we cannot succeed in the failure diagnosis, we can handle some further testing (Section 3.4). Finally, we collect some data and evaluate the testing results (Section 3.5).

Our testing procedure model is very similar to the common process of testing. However, to address the unique characteristics of CT, some special activities and techniques are added. We will discuss the application and the testing procedure of CT together (Section 3.6).

Based on the generic model (Figure 7), we will discuss each step one by one. For the sake of clarity, we will present constraints and test generation together in Section 3.2, and metric and evaluation together in Section 3.5.

3.1. Model of SUT

Modeling of SUT is a very fundamental and important activity for CT, since the effectiveness and efficiency of CT depend on the model used. It is the starting point of CT. Different testers may come up with different models of SUT based on their own skill and experience. A key issue in modeling is to build a precise model at the "right" level of abstraction. We first give the definition of model of SUT, then review some approaches for modeling a SUT.

Definition 3.1.1 (*SUT Model*). For CT, a model of SUT includes four elements: parameters that may affect the SUT; values that should be selected for each parameter; interaction relations that exist between parameters, and constraints that exist between values of the different parameters, which can be used to exclude combinations that are not meaningful from the domain semantics. A model for the SUT can be denoted as a 4-tuple: $Model_{SUT}(P, V, R, C)$.

By Definition 3.1.1, four key issues need to be resolved in modeling of SUT:

(1) how to identify parameters that may affect the SUT;
(2) what values should be selected for each parameter;
(3) what interactions exist between parameters;
(4) what constraints exist between parameters and their values.

Only when these issues are settled can CT be effectively brought into play.

The first issue is to identify parameters for CT. Parameters in the SUT may represent configuration parameters, like those of the NGS in Figure 2, and similar cases found in Cohen et al. [1997], Williams [2000], and Xu et al. [2003b]. They may also represent user input parameters [Schroeder 2002], features of the SUT [Cohen 2004], interfaces or GUI [Williams and Probert 1996; Burr and Young 1998], and components or objectives [Williams 2002]. By reviewing various system documents and conducting some case studies [Mats et al. 2005], we can select an initial set of parameters for the SUT and then refine this set by adding or deleting parameters during the modeling procedure.

The second issue is to determine the applicable values for each parameter. This step is critical to the modeling of SUT, since the behavior of the SUT is governed by the specific combination of parameter values. Note that some parameters may have many values. For example, the parameter "Audio" in the NGS, a configure parameter, can have many possible values, as there are hundreds of audio products in the market. When the parameter is an input of continuous values, it can have an infinite number of values. In this case, we must select some typical values. Equivalence partitioning, boundary value analysis, and primary element selection are the usual methods to be used here. Through selection of typical representative values from the equivalent class, completing with some boundary values, and assigning some important or relevant values for each parameter, we can keep the size of each V_i $(1 \leq i \leq n)$ in the SUT small (with just a few representative values).

The third issue is to identify the actual interactions between parameters. We can study the system documents to identify: (1) parameters which will not interact with any other parameters; (2) parameters which may have strong interactions with each other; and (3) interactions which exist between a small number of parameters.

The fourth issue is to identify the constraints that exist between certain parameters. A common case occurs when some specific values of one parameter conflict with some values of another parameter. For example, the Browser parameter of the NGS cannot take on "Firefox" value when the Access parameter has 'ISDL" value. This represents a kind of mutual exclusion constraint between parameters. The opposite case may also occur when some specific value of one parameter must be combined with some value of another parameters. For example, the Browser parameter of the NGS must take on "IE" value when the Access parameter has "ISDL" value or when the Audio parameter has "Maya" value.

To obtain the information on the interactions and constraints between parameters, we can study the requirement document, design document, codes, and other related

Aspects	Covering Array	Seeds	Constraints	Methods
Options	EC(or BC)	No	No	Greedy
	OA(or PW)	Yes	Yes	Search
	τ−way CA			Math.
	VCA			Rand.
	Comb. of CA			Comb.of Methods

Fig. 8. Combinatorial strategies of test suite generation.

documents. We can also interview designers and programmers to extract the relevant information. Static analysis, such as program slicing, can also be used to identify the interactions between parameters [Schroeder and Korel 2000a, 2000b; Schroeder 2002; Cheng et al. 2003].

With the information of interaction and constraints between parameters, we can make CT more pertinent and effective. The output of modeling of SUT is (P, V, R, C): a set P of parameters, $P = \{c_1, c_2, \ldots, c_n\}$; a set V of value sets, $V = \{V_1, V_2, \ldots, V_n\}$, where each V_i is the value set for parameter $c_i (i = 1, 2, \ldots, n)$; a set R of the interaction relations between parameters, and a set C of constraints which specifies the requirements on what should or should not be covered.

Modeling of SUT provides the foundation for the following steps of CT. There have been many studies on modeling of SUT which helped to advance this field. For example, Dalal et al. [1998a, 1999] learned that iteration and expert knowledge is required to build a proper model. Lott et al. [2005] gave the samples of modeling system which can serve as a tutorial for applying CT. There have been some good guidelines for modeling of SUT. Many applicable scenarios have been given in Williams [2002], and the procedure for determining the parameters and their values, and generating test suite is given in Krishnan et al. [2007]. Mats and Offutt [2007] presented a basic eight-step process for input parameter modeling and gave some initial experience of using this process. Czerwonka [2006] gave some practical ways of modeling that make the pure pairwise testing approach more applicable.

Despite these prior works, there are still several open questions left in this area:

(1) how to effectively and efficiently model SUT for CT?
(2) how to validate the model?
(3) how to evaluate the effectiveness of the different approaches of modeling?

3.2. Combination Strategies and Tools for Test Suite Generation

A combination strategy selects test cases by combining values of the different test object parameters based on some combinatorial strategy [Mats et al. 2005]. It involves four elements: (1) covering array specifying the specific kind of test suite to be used; (2) seeding to assign some specific test cases in advance; (3) constraints to be considered in the test generation; and (4) method to be used to generate test cases. Next we introduce these aspects.

3.2.1. Covering Array. There are many kinds of covering array we can select for CT. Figure 6 shows that different covering arrays have different sizes and offer different coverage. Based on the observation in testing that a larger test suite typically requires more testing cost and a higher coverage typically increases the chance of detecting failures, these covering arrays have different failure detection abilities and involve different testing costs. In practice, some of them can be combined together; for example, BC can be combined with OA or with PW by taking the union of the respective test suites. Several studies investigated the effectiveness of these covering arrays. For example, Mats et al. [2006a] evaluated five combination strategies: AC, EC, BC, OA, and PW. These strategies were evaluated with respect to the number of test cases, the number of faults found, the failure size, and the number of decisions covered. Their results indicated that the Each Choice strategy required the least number of tests and found the smallest number of faults. This research provides some suggestion on selecting an appropriate covering array.

3.2.2. Seeding. Seeding means to assign some specific test cases or some specific schema in testing. Cohen et al. [1997] first used the term seeds to guarantee inclusion of their favorite test cases by specifying them as seed tests or partial seed tests. The seed tests are included in the generated test set without modification. The partial seed tests are seed tests that have some input fields which have unassigned values. The Automatic Efficient Test Generator (AETG) can complete the partial test cases by filling in values for the missing fields.

Seeding has two practical applications [Czerwonka 2006]: (1) It allows explicit specification of important combinations. For example, if a tester is aware of combinations that are likely to be used in the field, he can specify a test suite to contain these combinations. All τ-way combinations in these seeds will be considered covered and only incremental test cases which contain the uncovered τ-wise combinations will be generated and added. (2) It can be used to minimize change in the test suite when the test domain description is modified and a new test suite regenerated. For example, a small modification of the test domain description, like adding parameters or parameter values, might cause big changes in the resulting test suite. Containing these changes can save testing cost.

Recently, Fouché et al. [2007] presented a new use of the seeding mechanism: increment adaptive covering array, by building a series of covering arrays incrementally and adaptively. Their approach begins with a seed of a low strength covering array, and continually increases this as resources allow, and based on the revealed faults and results of failure diagnosis. At each stage, the test cases from the previous stage are reused as seeds. This allows failures owing to only one or two configuration settings to be found and fixed as early as possible, and also reduces duplication of testing when multiple covering arrays must be used. For example, we can first test SUT with EC, then generate a PW test suite using the test suite from EC as seeds, and if it is necessary, we can repeat the test generation using this PW test suite as seeds to form a new higher strength covering array.

Many test generation tools of CT also support a seeding mechanism, such as AETG [Cohen et al. 1997], PICT [Czerwonka 2006], and SST developed by us [Nie et al. 2006a]. These tools can first accept a predefined test suite, then generate additional test cases to achieve the required coverage while minimizing redundant coverage of interactions.

3.2.3. Constraints. Constraints occur naturally in most systems. The typical situation is that some combinations of parameter values are invalid. Existence of constraints increase the difficulty in applying CT: (1) Most existing test generation methods cannot deal with constraints, and ignore them. Ignoring constraints may lead to the generation of test configurations that are invalid. This can lead to ineffective test planning and wasted testing effort. (2) It is difficult to design a general algorithm for test generation with due consideration of constraints. (3) Even a small number of constraints may give rise to an enormous number of invalid configurations. When the generated test suite contains many invalid test cases, this will cause a loss of combination coverage. (4) Complicated constraints may exist in SUT, and multiple constraints can interact to produce additional implicit constraints. It is both time consuming and highly error prone to deal with constraints manually in test suite generation. Thus, proper handling of constraints is a key issue we must address in test suite generation.

Since Bryce and Colbourn first addressed “soft constraints” by weighting factors and levels [Bryce and Colbourn 2006], there has been some work done by Hnich et al. to address “hard constraints,” although they only provided examples of small inputs and did not demonstrate that their method would scale [Hnich et al. 2006]. Although handling constraints is still an open problem, there have been several good attempts in dealing with constraints. For example, Cohen et al. [2007b, 2007a, 2008] studied the magnitude and subtlety of constraints found in configurable systems and presented techniques for handling them. They described the variety and type of constraints that can arise in highly configurable systems and reported on the constraints found in two nontrivial software systems. They presented a technique for compiling constraints into a boolean satisfiability (SAT) problem and integrating constraint checking with both greedy and simulated annealing algorithms for interaction test generation.

Mats et al. [2006b] investigated four different constraint handling methods: the abstract parameter method and the submodels method that can result in conflict-free parameter models, the avoid method which ensures that only conflict-free test cases are selected during the test case selection process, and the replace method which removes conflicts from already selected test cases. They also provided a good guideline on when to use these four constraint handling methods.

3.2.4. Test Case Generation Technique. Test case generation is the most active area of CT research. In our survey, we have identified 50 related papers on test generation. We also included a few studies on covering array published in mathematical journals. As the problem of covering array generation is NP-hard, researchers have tried various methods to solve it. To date, four main groups of methods have been proposed: greedy algorithm, heuristic search algorithm, mathematic method, and random method. The first two groups are computational approaches.

Greedy algorithms have been the most widely used method for test suite generation for CT. They construct a set of tests such that each test covers as many uncovered combinations as possible. There are two classes of greedy algorithms. The first class is the one-row-at-a-time variation based on the automatic test case generator AETG [Cohen et al. 1997]. Bryce et al. [2005] presented a generic framework for this class of algorithms. A single row of the array is constructed at each step until all τ-sets have been covered. Algorithms that fit into this class include: the heuristic algorithm used to generate pairwise test suite of the CATS tool [Sherwood 1994], the density-based greedy algorithm for generating a 2-way and higher strength covering array [Bryce and Colbourn 2007a, 2008], and the greedy algorithm with different heuristics used in PICT [Czerwonka 2006]. Tung and Aldiwan [2000] also gave a greedy algorithm for a parametric test case generation tool that applies a combinatorial design approach to the selection of candidate test cases. A typical greedy algorithm works as follows.

Greedy Algorithm

```
Let US be the set of all the $\tau$-way combinations that should be covered according
to the model of SUT (P,V,R,C).
Let Seeds be the set of test cases assigned by testers. Remove all the $\tau$-way
combinations covered by the test cases in Seeds from US.
While (US$\neq \emptyset$)
    Generate a test case $t$ to cover the most uncovered $\tau$-way combinations.
    Check the constraints and ensure it is valid.
    Remove all the the $\tau$-way combinations covered by the test cases $t$ from $US$.
End while.
```

The second class of greedy algorithm is the In Parameter Order (IPO) algorithm. This class of algorithm begins by generating all τ-sets for the first τ factors and then incrementally expands the solution, both horizontally and vertically using heuristics until the array is complete. Lei and Tai [2001], Tai and Lei [2002], and Lei et al. [2007b, 2008] described an algorithm for generating 2-way and τ-way covering array. We gave a greedy algorithm based on solution space tree to generate 2-way array [Nie et al. 2006a] and extended the existing greedy algorithms for VCA [Nie et al. 2005, 2006b; Wang et al. 2007, 2008].

Heuristic search techniques such as hill climbing, great flood, tabu search, and simulated annealing have been applied to τ-way covering array and VCA generation. Also some AI-based search techniques such as Genetic Algorithm (GA) and Ant Colony Algorithm(ACA) have also been used. Heuristic search techniques start from a preexisting test set and then apply a series of transformations to the test set until it covers all the combinations. Ghazi [2003] used a GA-based technique that identifies a set of test configurations that are expected to maximize pairwise coverage with a predefined number of test cases. Bryce and Colbourn [2007b] augmented the one-test-at-a-time greedy algorithm with a heuristic search, which can generate tests that have a high rate of τ-tuple coverage. This approach combines the speed of greedy methods with the slower but more accurate heuristic search techniques. The hybrid approach seems to achieve a more rapid convergence of τ-tuple coverage than either greedy or heuristic search alone. They compared four heuristic search techniques and found that hill climbing is effective only when time is severely constrained, but tabu search, simulated annealing, and the great flood perform better over a longer time. Cohen et al. [2003a, 2003b, 2003c] reported using the computational method of simulated annealing to generate 3-way covering array and variable strength array. Shiba et al. [2004] also used the genetic algorithm and ant colony algorithm to generate the 3-way covering array. Heuristic search techniques can often produce a smaller test set than those from the greedy algorithm, but typically require a longer computation. In the following we give a generic search algorithm for covering array generation.

Search Algorithm

```
Let US be the set of all the $\tau$-way combinations that should be covered according
to the model of SUT (P,V,R,C).
Let Seeds be the set of test cases assigned by testers. Remove all the $\tau$-way
combinations covered by the test cases in Seeds from US.
For each test case t, fitness(t)=the number of $\tau$-way combinations in US
covered by t, but uncovered by the generated test cases and Seeds
While ($US \neq \emptyset$)
    Generate a set of test cases randomly;
    Evolve the test set with a metaheuristic search method, such as simulated
     annealing, hill climbing, great flood, tabu search, particle swarm optimization,
     ant colony optimization, and genetic algorithm.
    Output the valid t (satisfying the constraints) with the highest fitness score.
    Remove all the $\tau$-way combinations covered by the test cases t from US.
End while.
```

Mathematical methods for computing the covering array have been widely studied in the mathematic community and they are often published in mathematical journals. The study of covering array is an example of the interplay between pure mathematics and the applied problems generated by computer science [Hartman 2002]. Some mathematical methods compute test sets directly based on a mathematical function. These approaches are generally extensions of the mathematical methods for

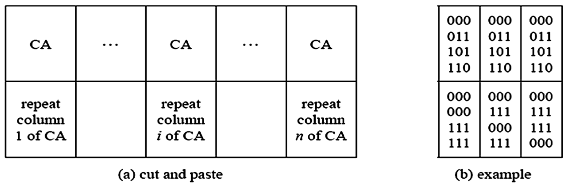


Fig. 9. A recursive method for covering array generation of CT.

constructing orthogonal arrays. Some methods are based on recursive construction, which builds larger test sets from smaller ones [Williams 2000; Kobayashi et al. 2002]. For example, in Figure 9(a), $CA(m; 2, n, v)$ is an $m \times n$ matrix, representing a 2-way covering array. We can construct a larger $CA(2m; 2, n^2, v)$ by first concatenating n copies of CA horizontally, then below each ith copy of CA, generate a new array formed by repeating the i column in CA n times. A concrete example is shown in Figure 9(b), where we use $CA(4; 2, 3, 2)$ to construct a larger $CA(8; 2, 9, 2)$. More detail can be found in Williams [2000]. But mathematical techniques (both direct and recursive) are not general. For example, we cannot use mathematical techniques to built a good covering array for a system with 10 parameters, each having six different values. Several test tools, such as TConfig [Williams 2002], Combinatorial Test Services (CTS) [Hartman 2002], and Test-Cover [Sherwood 1994] all use mathematical constructions to generate covering arrays.

Mathematical methods have two key advantages: (1) The computations involved are typically “lightweight,” and they are immune to any combinatorial effect. (2) They can be extremely fast and can produce optimal test sets in some special cases. But they also have some disadvantages: (1) They often impose restrictions on the system configurations to which they can be applied. For example, many approaches require that the domain size be a prime number or power of a prime number. This significantly limits their applicability. (2) They currently do not effectively deal with test prioritization and constraint.

Compared to the mathematical methods, computational approaches offer the following advantages: (1) They can be applied to an arbitrary system configuration, since there is no restriction on the number of parameters and the number of values each parameter can take. (2) They can be easily adapted for test prioritization and constraint handling. However, the computational approaches also have their own disadvantages: (1) They involve explicitly enumerating all possible combinations to be covered. When the number of combinations is large, explicit enumeration can be prohibitive in terms of both the space for storing these combinations and the time needed to enumerate them. (2) They are typically greedy, in the sense that they construct tests in a locally optimized manner which does not necessarily lead to a globally optimized test set. Thus, the generated test sets from computational approaches are often not minimal.

The fourth group of test generation method is the random method, an ad hoc approach that randomly selects test cases from the complete set of test cases based on some input distribution. It is often compared with the other methods to study the effectiveness of test suite generation algorithm and the failure detection ability of the proposed methods [Schroeder et al. 2004]. For some special cases, random search and search-based methods can do better than other methods. Techniques from the field of search-based software engineering proposed by Harman can be applied to improve the test suite for CT [Harman 2007]. Test suite generated by search techniques is often smaller in size and better in constraint handling, as it integrates randomness and combination coverage.

Researchers have begun to combine different methods to efficiently generate a small test suite. For example, Cohen et al. [2003b] combine the recursive combinatorial constructions of mathematic methods with the simulated annealing of computational search methods. Their method leverages the computational efficiency and optimality of size obtained through combinatorial constructions while benefiting from the generality of a heuristic search. In theory, we can explore different combinations of the four main classes of test generation methods. For example, we can combine mathematical method with search method to generate a test suite, or combine heuristic greedy, random method and mathematical method together. There are $C_4^2 + C_4^3 + C_4^4 = 11$ combinations that we can study.

There is still a lot of room to develop better methods for test generation for CT, and especially if we can explore some special ways to resolve some specific testing scenarios. Also, it is worthwhile to conduct a thorough study of various test generation methods to fully understand their limitations and strengths under various testing scenarios.

3.2.5. Test Case Generation Tools. More than 20 software tools have been developed for test case generation, for example, CATS (Constrained Array Test System) developed by Sherwood et al. [2005], OATS (Orthogonal Array Test System) developed by Phadke et al. [Brownlie et al. 1992], AETG developed by Cohen et al. [1997], IPO developed by Lei and Tai [2001], Tconfig developed by Williams [2002], CTS (Combinatorial Test Service) developed by IBM, and PICT developed by Microsoft [Czerwonka 2006]. We have also developed the SST tool for test generation [Nie et al. 2006a]. Some new tools are still continually appearing. Many of these tools are freeware. As each tool has its own characteristics and advantages, none is the best for all settings. If possible we may use them together, and then choose the best result. Some integrated tools have been developed to generate a minimal test suite T_c.

Test generation involves mainly four factors and each factor has many cases, as shown in Figure 8. For example, for covering array, we can use EC, BC, OA, PW, τ-way covering array, or the combination of these test suites, and even the increment adaptive covering array [Fouché et al. 2007]. The test generation method can be greedy, heuristic search, mathematic, random, or some combination of these methods. In the process of test generation we may or may not need to deal with constraints, and we may or may not need to assign some seeds. From Figure 8, we can see that there are at least $5 \times 2 \times 2 \times 5 = 100$ scenarios. Therefore it is a big challenge to totally resolve the many issues of test generation.

All existing tools have their own advantages and disadvantages. There is a need to develop some more effective and more convenient tools for test generation. For example, we can build a platform that integrates several tools together. Then, the best test set can be selected depending on the specific testing requirements.

3.3. Test Case Prioritization

Test case prioritization has been well studied. The result of the prioritization is often a schedule of test cases so that those with the highest priority, according to some criterion, are executed earlier in testing. When testing is terminated after running a subset of the prioritized test suite, those test cases deemed most important are executed. Especially when the resource is limited, the important test cases should be tested as early as possible. Well-ordered testing may reveal faults early because it ensures that the important test cases are executed first. The test case prioritization problem is defined as follows.

Definition 3.3.1. Given $(T, \prod, f)$, where T is a test suite, $\prod$ is the set of all test suites obtained by permuting the tests of T, and f is a function from $\prod$ to real numbers, the test case prioritization problem is to find $\pi \in \prod$ such that $\forall \pi' \in \prod$, $f(\pi) \geq f(\pi')$. $\prod$ gives the possible prioritization of T and f is the function to evaluate the prioritization [Bryce and Memon 2007].

Prioritization can be computed in two ways: (1) reorder an existing test suite of CT based on a prioritization criteria; and (2) generate an ordered test suite for CT, taking into account the importance of combinations.

By Definition 3.3.1, the key issues in test prioritization are how to define the function f, and validate the effectiveness of the prioritization. Bryce et al. [2005] and Bryce and Colbourn [2006] adapted a one-test-at-a-time greedy method to take importance of parameter pairs into account. With due consideration of seeding and constraint, they used this method to generate a set of tests in order. Bryce and Memon [2007] explored the effectiveness of the prioritization with interaction coverage in the event-driven software. Compared with other criteria like ordering test by unique event coverage, ordering test by the length, and random ordering, prioritization with interaction coverage was found to provide the fastest rate of fault detection. Qu et al. [2007] tried three code-coverage-based weighting methods and one specification-based method to prioritize the test suite, and found that the prioritized CIT test suites may find faults earlier than unordered CIT test suites, although the prioritized test suites sometimes exhibit decreased fault detection.

Different prioritization methods have different performance. Better methods of test case prioritization for CT may be developed by considering additional factors. For example, we may do prioritization based on required execution time or based on incremental τ-way coverage [Bryce and Memon 2007]. The prioritization done on GUI applications was also applied to Web application [Sampath et al. 2008]. We also need more empirical study to better understand the limitations and strengths of various proposed methods [Li et al. 2007].

3.4. Failure Diagnosis

After detecting a failure, we need to investigate the failure to locate and then remove the fault. Kuhn and Reilly [2002] and Kuhn and Wallace [2004] suggested that software faults are often triggered by only a few interacting variables. Their results have important implications for CT. If all faults in SUT are triggered by a combination of no more than n parameters, then testing all n-way combinations of parameters can provide a high confidence that nearly all faults have been discovered. So CT is an effective approach for detecting failures triggered by the combination of parameter values. When a defect is exposed in CT, we should determine which specific combination of parameter values causes the failure, or which value schema of SUT triggers the failure.

Failure diagnosis by finding failure-triggering schema is also called fault characterization. Fault characterization in CT can help developers quickly pinpoint the causes of failures, hopefully leading to a quicker turn-around time for bug fixes. Automated techniques, which can effectively, efficiently, and accurately perform fault characterization, can save a great deal of time and money. This is especially beneficial when system configuration spaces are large, the software changes frequently, and resources are limited. Yilmaz et al. [2006] applied CT and fed the testing results to a classification tree algorithm to localize the observed faults. Their results suggested that sampling via covering arrays can characterize option-related failures nearly as well as if we had tested exhaustively, but at a much lower cost.

Shi et al. [2005] presented a further testing strategy for fault revealing and failure diagnosis, which first tests SUT with a covering array, then reduces the value schemas contained in the failed test case by eliminating those appearing in the passed test cases. If the failure-causing schema is found in the reduced schema set, failure diagnosis is completed with the identification of the specific input values which caused the failure; otherwise, a further test suite based on SOFOT is developed for each failed test cases, testing is repeated, and the schema set is then further reduced, until no more failure is found or the fault has been located.

For example, when we test NGS (Figure 2) with the test suite in Figure 3, if $t_1 = (Netscape, Windows, ISDL, Creative)$ failed, and other test cases passed, we can generate $M = M_{t_1} - \bigcup_{t\ is\ passed} M_t$, M has many schemas. If we cannot find the failure-causing schema, we then design a further test suite T_{t_1} for t_1 with SOFOT, $T_{t_1} = \{(IE, Windows, ISDL, Creative)\ (Netscape, Linux, ISDL, Creative)\ (Netscape, Windows, VPN, Creative)\ (Netscape, Windows, ISDL, Maya)\}$. After we execute T_{t_1} again, if only the second test failed, we compute $M = M_{t_1} - \bigcup_{t\ is\ passed} M_t$ again. Since M may be greatly reduced, we can find the failure-causing schema in M more easily. More detail can be found in Shi et al. [2005].

Colbourn [2006] and Colbourn and McClary [2008] extended the notion of a covering array to detecting and locating arrays, which forms fault interaction test suites that permit the location of a specified number of faults of specified strength.

Additional work is needed to develop better methods for failure diagnosis. There is also a lack of empirical results on the various failure diagnosis methods. Another fruitful area of study is to make use of the results from CT to localize the fault and fix it, as valuable information about the fault can be gained from the failed test cases.

3.5. Metric and Evaluation

There exist two kinds of evaluation of CT: (1) measure and evaluate CT itself; and (2) measure and evaluate the quality of the SUT after CT.

CT has a natural metric: combination coverage, which measures the percentage of the covered parameter value combinations relative to the total combinations. One way to measure the combination coverage is based on the *k*-value schemas tested relative to the total *k*-value schemas.

Definition 3.5.1. Let T_c is a test suite of CT for SUT, and T_{all} is all the possible test cases of SUT. M_{T_c} is a set of all the schema covered by T_c: $M_{T_c} = \{k\text{-value schema covered by } T_c\}$, $M_{SUT} = \{k\text{-value schema covered by } T_{all}\}$ is all the possible schemas in SUT. The combination coverage of test suite T_c is given by $\frac{M_{T_c}}{M_{SUT}}$.

This kind of CT metric can serve two purposes: (1) to set a test coverage target; and (2) to evaluate different test strategies of CT which use different test suites. Williams and Probert [2001] described the metric for CT and provide a formal definition of CT. Schroeder et al. [2004] compared the failure detection effectiveness of τ-way CT and random testing.

It is not easy to evaluate software quality after CT, as very few works have focused on this topic. We only find one study by Salem [2001] and Salem et al. [2004], in which they developed a logistic regression model of predicting software failure based on the testing result of CT. More studies should be devoted to this area. We can try to enhance existing methods of quality evaluation based on the characteristics of CT. To evolve and obtain a reasonable evaluation method, more empirical evidences are needed.

3.6. The Application of CT and Testing Procedure

The second most popular topic of CT research is the study of applying CT to various types of applications, and the refinement of testing procedures.

The basic concept of CT was first used in other disciplines many years ago. In 1926, Fisher pioneered interaction tests in agricultural experiments, assessing the contributions of different fertilizers to crop yield in the context of soil heterogeneity and environmental factors such as erosion, sun coverage, rainfall, and pollen [Fisher 1926]. Given the limited resources for testing in most cases, exhaustive testing is not feasible. Fisher applied interaction testing so that every pair of factors affecting the yield was included exactly once. Since 1926, CT has been widely used in many other disciplines

[Bryce et al. 2005], with many experiences and empirical studies reported on the testing procedure and the application of CT.

For software testing, Mandl first proposed using pairwise combinatorial coverage to test Ada compiler in 1985 and generated test sets using Orthogonal Latin Squares [Mandl 1985]. Brownlie et al. reported a case study that tested PMX/StarMAIL system using orthogonal array in 1992 and developed the OATS system to generate test cases [Brownlie et al. 1992]. The generated test cases can detect many errors that had never been detected previously.

There are two key questions on the application of CT: (1) Is there a standard procedure we can follow in using CT? (2) What is the effectiveness of the CT procedure? The big issues are how to develop a practical procedure for CT and how to evaluate it in practice.

Many studies have been published in this area. Berling and Runeson [2003] used CT for performance evaluation, and found that a small number of test cases give information of which factors, or interactions between factors, affect the performance measure. This knowledge can be used as a basis for future investigations, where, for example, other environmental factors can be added and where factors which have no effect can be removed. The information gained help to reduce the number of test cases to be run in the real environment.

Krishnan et al. [2007] reported their experience in piloting and applying CT in projects. They designed a supporting process for using CT, and shared details on application of CT in feature testing of a mobile phone application. Their proposed testing process emphasized the first step of modeling, included test prioritization, expected output and failure diagnosis, but neglected the further testing and the evaluation steps.

Many papers reported the experience of using CT in practice. Dunietz et al. [1997] examined the correlation between τ-way coverage and achieved code coverage. Huller [2000] used CT to test the ground system for satellite communications. Burroughs et al. [1994] reported how both the quality and efficiency of protocol testing were improved by CT, and Williams and Probert [1996] described a practical strategy for the pairwise testing of network interface. These researches provided good guidelines for using CT.

CT has also been applied to other applications. White and Almezen [2000] proposed a method which focused on user sequences of GUI objects and selections which collaborate, called Complete Interaction Sequences (CIS), that produce a desired response for the user. An empirical investigation of this method shows that a substantially reduced test set can still detect the defects in the GUI. Future research will prioritize testing related to the CIS testing for maximum benefit if testing time is limited. Burr and Young [1998] generated test cases to test an email system with AETG. We also explored the configuration testing [Xu et al. 2003a] and the browser compatibility testing [Xu et al. 2003b] with CT. Lei et al. [2007a] described a CT strategy for concurrent programs.

From the many prior studies, it appears that CT can be applied to many types of systems. However, each research team followed its own testing procedure. We believe better testing results can be obtained by following a more effective testing procedure. Some research effort should be devoted to this topic.

3.7. Summary

Since Mandl [1985] and Brownlie et al. [1992] first used orthogonal arrays to test software, pairwise testing and then combinatorial testing were proposed and adopted. There have been more than ten research groups working in the CT field. We summarize the contribution of various research groups in Figure 10, with more detail given in the Appendix.

Group	Model	Gen.	App.	Fault	Prior.	Metric	Constr.	Eval.
Bryce		√		√	√		√	
Cohen DM	√	√	√				√	
Cohen MB		√	√	√	√		√	√
Grindal	√	√	√				√	√
Hartman		√	√					
Kabayashi		√	√					
Kuhn			√	√				
Lei		√	√					
Nie	√	√	√	√		√		√
Salem			√					√
Schroeder	√	√	√				√	√
William	√	√	√					

Fig. 10. Research group and their contribution to CT.

Group	≤ 92	94	96	97	98	99	00	01	02	03	04	05	06	07	08	$\sum$
App.	2	1	2	1	3		1		1	3	2			1		17
Constr.							1					1	2	2		6
Eval.								1			2		2			5
Gen.	1	2		1	1		5	1	8	6	4	7	5	5	4	50
Fault									1		1	1	1			4
Metric								2								2
Model					1	1						1		2		5
Prior.												1	1	2		4
$\sum$	3	3	2	2	5	1	7	4	10	9	9	11	11	12	4	93

Fig. 11. Distribution of published work from 1985 to 2008.

Combinatorial testing is a subject that crosses both computer science and mathematics. CT has attracted interest because we can combine mathematical methods and computational methods from computer science to find solutions to the testing problem. CT also presents many new and challenging problems.

Looking at Figure 10, we can conclude that test suite generation is the most active area, and the application of CT is the second most studied area in CT. Most of the research groups contributed in these two areas. As for the evolution of this research field, from Figure 11 we can see that the research of CT begins to expand to include modeling, metric, evaluation, failure diagnosis, constraint, and prioritization. But test generation and the application continue to attract interest in recent years. We observe a steep rise in the number of publications commencing from 2002, more than tripling the average annual publications from the previous six years. Thus, the research of CT is increasing, both in breadth and in depth, and CT has become a mature field where the large-scale deployment of this technology is possible.

4. CONCLUSION AND FUTURE WORK

In the last 20 years, combinatorial testing has been widely studied and applied. Now, it has become a well-accepted testing method, given its proven ability of detecting interaction failures. Many topics have been studied over the years, such as modeling of SUT, test suite generation, the application of CT, test prioritization, failure diagnosis, metric, and evaluation.

In this article, we make four major contributions: (1) To our best knowledge, this is the first complete and systematic survey for CT, as an independent software testing technique, although Mats et al. [2005] have given a survey on combinatorial testing strategy, focusing mainly on test cases generation. We have made an careful survey on over 90 key papers. (2) We classified all the work of CT into eight categories according to the testing procedure. For each category, we surveyed the motivation, key issues, solutions, and the current state of research. (3) We reviewed the contributions of the different research groups, and presented the growing trend of CT research. (4) Based on our study and other knowledge, we recommend the following areas of focus in CT research in the future.

(1) A good model of the test parameters is critical to CT. We need effective ways to identify the parameters of SUT, determine the values of each parameter, and explore the interactions and constraints existing among the parameters. To be effective in detecting defects, not only do we need a good understanding of the limitations and strengths of CT, but also a good model of SUT that includes all relevant parameters, their values, interactions, and constraints.
(2) Although many methods have been proposed to generate test suite for CT, as the problem of test suite generation is NP-hard, there is room for further improvement of these test generation methods. In particular, a good method should support the use of seeding and make full use of constraints in generating a set of feasible test cases. Various mathematical methods, computational methods, and their crossover combinations can play an important role in improving test generation.
(3) As there exist failures which are caused by the interaction of more than 2 factors in SUT, and for certain critical SUT, we may want to test more combinations to ensure its quality. Thus there is a need for the test suite generation algorithms for τ-way($\tau > 2$). Although the IPOG, density algorithm, simulated annealing, and some mathematical techniques have been used to generate τ-way covering array(usually $\tau = 3$), there has been relative little work on τ-way compared to 2-way, and more work is needed in this area.
(4) To realize the full potential of CT, better strategies and methods need to be developed to make use of the testing result from CT to support further testing, failure diagnosis, and evaluation.
(5) As CT generally requires a large number of tests, it is impractical to conduct manual testing and results analysis. Tools are needed to support the many testing steps. More work is needed in integrating the test generation tools with other tools to automate the entire testing process of CT.
(6) Another fruitful area of research in CT is to expand it to different levels of testing. For example, we can explore the application of CT in unit testing, integration testing, and system testing. We can also investigate how to apply CT to different types of software, such as object-oriented software, service-oriented software, etc.
(7) There is a lack of empirical results in CT. We need to conduct more studies and collect more evidence to fully understand the limitations and strengths of CT. For example, there is still no empirical study on the effect of the prioritization by interaction coverage.
(8) One approach to overcome the weakness of CT is to combine this technique with other testing techniques. We can try to incorporate the concept of CT into other testing techniques or to expand CT by including other testing techniques. For example, we may combine CT with test case prioritization to ensure the most important test cases are executed early; we can also combine CT with metamorphic testing to solve the oracle problem of CT by automating the process to determine whether a test passes or fails.

APPENDIX: THE CONTRIBUTIONS FROM DIFFERENT RESEARCH GROUPS

A.1. Bryce et al.

Bryce et al. made many contributions on test generation, failure diagnosis, and prioritization. Sherwood first introduced the CATS tool, which implemented a heuristic algorithm for pairwise coverage [Sherwood 1994]. This group discussed two algebraic approaches to generate covering array, which could be used to build mixed covering array of strength 2 and covering array of higher strength [Colbourn et al. 2005; Bryce and Colbourn 2005; Sherwood et al. 2005; Bryce et al. 2005], and introduced several greedy algorithms to construct covering arrays, mixed-level covering arrays, and ℓ-biased covering arrays [Turban 2006].

Colbourn later proposed the failure locate and detect array [Colbourn 2006; Colbourn and McClary 2008]. Bryce and Colbourn proposed a deterministic density algorithm to generate a test set for pairwise testing and a higher strength covering array [Bryce and Colbourn 2007a, 2008]. They then defined the problem of test suite prioritization, proposed solutions both generating prioritized test suites from scratch and prioritizing existing test suites by combinatorial coverage for GUI and Web applications, and conducted an empirical study on prioritized CT with seeding and constraints [Bryce 2005; Bryce and Colbourn 2006].

A.2. Cohen D. M., Dalal et al. and AETG from Bellcore

The group from Bellcore has worked in modeling, test generation, and the application of CT. They presented the first use of test generation system called AETG to generate a test suite of CT for various application scenarios in 1990's [Cohen et al. 1994, 1996, 1997; Cohen and Fredman 1998] and concluded that AETG was an effective and efficient way to create test case [Burroughs et al. 1994].

Cohen et al. showed that the number of tests required to cover n-way parameter combinations grows logarithmically with the number of parameters, and showed that AETG was able to reduce the number of test cases compared to OA. They used AETG to generate both a high-level test plan and detailed test cases by incorporating some constraints and seeds. Other studies also found that the AETG pairwise test sets gave good coverage in a variety of settings [Burr and Young 1998].

Dalal et al. later presented the method of factor covering design for testing, and offered some testing guidelines based on their experience. They also proposed some models and some measures of effectiveness [Dalal and Mallows 1998], and reported four case studies in which their model-based test architecture has been used. The main lessons learned were that the model of the test data was fundamentally important, and considerable domain expertise and iteration were often required to find a suitable model [Dalal et al. 1998a, 1999]. After providing a publicly available Web interface to AETG [Dalal et al. 1998b], to serve as a tutorial for CT, they recently presented example system requirements and the corresponding methods for applying the combinatorial approach to those requirements [Lott et al. 2005].

A.3. Cohen M. B. et al.

Cohen et al. worked on many areas of CT, including test generation, application, test prioritization, failure diagnosis, constraints, and evaluation. They first examined the need of variable strength covering array and proposed this new subject of research, then they presented some computational methods, such as simulated annealing, to find variable strength array [Cohen et al. 2003c; 2003a]. They also explored a method for building covering array of strength three that combined algebraic constructions with computational search. This method leverages the computational efficiency and optimality of size obtained through algebraic constructions while benefiting from the generality of a heuristic search [Cohen et al. 2003b; Cohen 2004].

They later tried to quantify the effects of different configurations on a test suite's operation and effectiveness [Cohen et al. 2006]. They used covering array to detect option-related defects and gave fault characterization in complex configuration spaces using the classification tree analysis [Yilmaz et al. 2006]. They also examined the effectiveness of CT on regression testing in evolving programs with multiple versions as well as studied several prioritization techniques [Qu et al. 2007], presented a general constraint representation and a technique based on the existing constraint handling techniques [Cohen et al. 2007b; Fouché et al. 2007; Cohen et al. 2008]. Recently they presented an incremental and adaptive approach to building covering array schedules [Cohen et al. 2007a].

A.4. Grindal et al.

Grindal et al. studied modeling of SUT, test generation, application, constraints solving, and evaluation. They surveyed 15 different combination strategies extracted from more than 30 related papers, reviewed almost all of the published papers in this field before 2004, and gave a subsumption hierarchy to relate the various coverage criteria associated with the identified combination strategies [Mats et al. 2005]. They presented four different methods for handling constraints in parameter models when using combination strategies to select test cases, and compared them to three existing methods [Mats et al. 2006b]. They also presented results from a comparative evaluation of five combination strategies, and evaluated them with respect to the number of test cases, the number of faults found, failure size, and the number of decisions covered [Mats et al. 2006a]. Recently, they presented an eight-step structured method for the creation of input parameter models custom-designed for combination strategies [Mats and Offutt 2007].

A.5. Hartman et al.

Hartman defined the general problems of covering array and gave a series of construction methods for covering array, and later defined several problems motivated by the application of CT and discussed their solutions [Hartman 2002; Hartman and Raskin 2004]. Hartman et al. later implemented their algorithm and offered their IBM Witch tool free of charge. It used to be called CTS in an earlier version.

A.6. Kobayashi et al.

Kobayashi et al. studied design and evaluation of automatic test generation strategies for functional testing with CT [Kobayashi 2002]. They proposed an algebraic method to generate pairwise test set [Kobayashi et al. 2002], and they later [Shiba et al. 2004] used artificial life techniques such as generic algorithm and ant colony algorithm to generate test cases for CT.

A.7. Kuhn et al.

Kuhn et al. studied the effectiveness of CT in a variety of application domains. Their research showed that about 20%–70% of software faults were triggered by single parameters, about 50%–95% of faults were triggered by two or fewer parameters, and about 15% were triggered by three or more parameters. Thus, CT, especially pairwise testing, is very effective in practice. Later they studied the fault interactions of large distributed systems, and discovered that the failure-triggering interactions of this kind of systems are mostly 4 to 6 [Kuhn and Reilly 2002; Kuhn and Wallace 2004]. Their work shows that CT can be as effective as exhaustive testing in some cases, if all failures can be triggered by an interaction of 6 or fewer parameter values.

A.8. Lei et al.

Lei et al. proved that the problem of generating a minimum test set for pairwise testing is NP-complete, proposed the *IPO* test generation strategy for pairwise testing by extending parameters [Lei and Tai 2001; Tai and Lei 2002], then generalized the strategy to τ-way testing and developed the FireEye tool to generate test set [Lei et al. 2007b, 2008]. They also presented an application that implements τ-way reachability testing for concurrent programs, and evaluated its effectiveness with several case studies [Lei et al. 2007a].

A.9. Nie et al.

In recent years, we have studied CT test case generation, failure diagnosis, and applications of CT. We proposed algorithms to generate test suite for pairwise testing [Nie et al. 2006a], for τ-way testing [Nie et al. 2005], for neighbor factor combinatorial testing [Nie et al. 2006b], for real interaction testing [Wang et al. 2007], and for variable strength combinatorial testing [Wang et al. 2008]. We also developed a method of failure diagnosis for CT [Shi et al. 2005] and applied CT to browser compatibility testing [Xu et al. 2003b] and configuration testing [Xu et al. 2003a].

A.10. Salem et al.

Salem et al. proposed testing with DOE and predicted software quality based on logistic regression [Salem 2001; Salem et al. 2004]. They utilized DOE to efficiently minimize the number of test cases.

A.11. Schroeder et al.

Schroeder et al. worked on modeling, test generation, application, constraint, and evaluation. They proposed a CT test suite reducing approach using additional information [Schroeder and Korel 2000a; Schroeder 2002; Cheng et al. 2003], and a new approach to deal with the constraints in test generation [Schroeder and Korel 2000b]. They also proposed a method to generate the expected output for CT [Schroeder et al. 2002], and compared the fault detection effectiveness of N-way CT to random testing [Schroeder et al. 2004]. Recently, they discussed the weaknesses of the pairwise testing and warned testers against blindly accepting best practices [Bach and Schroeder 2004].

A.12. Williams et al.

Williams et al. contributed to the test suite generation, metric, and application of CT. They presented a guide of CT application to test network elements of a telephony system [Williams and Probert 1996], then explored the applicability of pairwise coverage to configuration testing. They also studied how to apply CT to software component testing, and identified eight contexts that CT can be applied, such as network component interaction testing, Commercial Off-The-Shelf (COTS) components testing, and Real-Time Object-Oriented Methodology (ROOM) substitutable components testing [Williams 2000].

They later proposed a metric for CT and provided a formal definition of the system test interaction problems [Williams and Probert 2001], and showed how the interaction test coverage problem may be formulated as a 0-1 integer programming problem and gave a minimal solution [Williams and Probert 2002]. Williams then proposed an algebraic approach to generate pairwise test set [Williams 2002].

A.13. Others

Other researchers have also studied test suite generation and application of CT. Some of them focused on test generation, for example, Ghazi [2003] used a GA-based technique to generate test suite for pairwise testing. Maity and Nayak [2005] presented a pairwise test set generation strategy for parameters with different number of values. Tung and Aldiwan [2000] proposed an algorithm to generate test suite using the combinatorial design approach. Dumitrescu [2003] described an efficient algorithm for generating tests that cover a prescribed set of combinations of parameters based on repeatedly coloring the vertices of a graph. Denny and Gibbons [2000] reimplemented the block design enumeration algorithm with a number of enhancements and demonstrated its use with several case studies involving different kinds of incidence structures. Nurmela [2004] used a tabu search heuristic to construct covering arrays and improved on the previously known upper bounds of the sizes of optimal covering arrays. Recently Yan and Zhang [2006] presented a SAT-based approach and a backtracking search algorithm to generate a covering array.

Other researchers also reported their experience in the application of CT. For example, Blass and Gurevich [2002] discussed the pairwise testing and its application. Czerwonka [2006] focused on ways to make the pure pairwise testing approach more practical and on features of tools that support pairwise testing in practise. Kahng and Reda [2004] adapted and applied a number of CT algorithms to resolve the diagnosis problem. Seroussi and Bshouty [1988] gave methods to generate the τ-way covering array for logic circuits where each parameter has two values, and proved that the design of an optimal test suite for an arbitrary circuit is an NP-complete problem. Hnich et al. [2005, 2006] developed constraint programming models of the problem of finding an optimal covering array and exploited global constraints, multiple viewpoints, and symmetry-breaking constraints. Huller [2000] also presented his experience with applying CT. White and Almezen [2000] generated a complete interaction sequence for GUI testing.

ACKNOWLEDGMENTS

We are grateful to Professor Mark Harman for his comments. Especially we appreciate the valuable comments from the anonymous referees, which helped us to revise this article.

REFERENCES

Bach, J. and Schroeder, P. J. 2004. Pairwise testing - A best practice that isn't. In *Proceedings of the 22nd Pacific Northwest Software Quality Conference*. 180–196.

Berling, T. and Runeson, P. 2003. Efficient evaluation of multifactor dependent system performance using fractional factorial design. *IEEE Trans. Softw. Engin. 29,* 9, 769–781.

Blass, A. and Gurevich, Y. 2002. Pairwise testing. *Bull. Euro. Assoc. Theor. Comput. Sci. 78,* 100–132.

Brownlie, R., Prowse, J., and Phadke, M. S. 1992. Robust testing of at and t pmx/starmail using oats. *AT and T Techn. 3,* 71, 41–47.

Bryce, R. C. and Colbourn, C. J. 2005. Constructing interaction test suites with greedy algorithms. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. ACM, New York, 440–443.

Bryce, R. C. and Colbourn, C. J. 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Inform. Softw. Technol. 48,* 10, 960–970.

Bryce, R. C. and Colbourn, C. J. 2007a. The density algorithm for pairwise interaction testing. *Softw. Test. Verif. Reliab. 17,* 3, 159–182.

Bryce, R. C. and Colbourn, C. J. 2007b. One-Test-at-a-Time heuristic search for interaction test suites. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07)*. ACM, New York, 1082–1089.

Bryce, R. C. and Colbourn, C. J. 2008. A density-based greedy algorithm for higher strength covering arrays. *Softw. Test. Verif. Reliab. 17,* 3, 1–17.

Bryce, R. C., Colbourn, C. J., and Cohen, M. B. 2005. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM, New York, 146–155.

Bryce, R. C. 2005. Test prioritization for pairwise interaction coverage. In *Proceedings of the 1st International Workshop on Advances in Model-Based Testing (A-MOST'05)*. ACM, New York, 1–7.

Bryce, R. C. and Memon, A. M. 2007. Test suite prioritization by interaction coverage. In *Workshop on Domain Specific Approaches to Software Test Automation (DOSTA'07)*. ACM, New York, 1–7.

Burr, K. and Young, W. 1998. Combinatorial test techniques: Table-Based automation, test generation, and code coverage. In *Proceedings of the International Conference on Software Testing Analysis and Review*. 503–513.

Burroughs, K., Jain, A., and Erickson, R. 1994. Improved quality of protocol testing through techniques of experimental design. In *Proceedings of the IEEE International Conference on Record, 'Serving Humanity Through Communications.'* Vol. 2. 745–752.

Cheng, C., Dumitrescu, A., and Schroeder, P. 2003. Generating small combinatorial test suites to cover input-output relationships. In *Proceedings of the 3rd International Conference on Quality Software (QSIC'03)*. IEEE Computer Society, Los Alamitos, CA, 76–82.

Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C. 1997. The aetg system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Engin. 23,* 7, 437–444.

Cohen, D. M., Dalal, S. R., Kajla, A., and Patton, G. C. 1994. The automatic efficient tests generator. In *Proceedings of the 5th IEEE International Symposium Software Reliability Engineering*. IEEE Press, 303–309.

Cohen, D. M., Dalal, S. R., Parelius, J., and Patton, G. C. 1996. The combinatorial design approach to automatic test generation. *IEEE Softw. 13,* 5, 83–88.

Cohen, D. M. and Fredman, M. L. 1998. New techniques for designing qualitatively independent systems. *J. Combin. Des. 6,* 6, 411–416.

Cohen, M., Gibbons, P., Mugridge, W., Colbourn, C., and Collofello, J. 2003a. Variable strength interaction testing of components. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*. 413–418.

Cohen, M. B. 2004. Designing test suites for software interaction testing. Ph.D. thesis, University of Auckland, New Zealand.

Cohen, M. B., Colbourn, C. J., and Ling, A. C. H. 2003b. Augmenting simulated annealing to build interaction test suites. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*. IEEE Computer Society, Los Alamitos, CA, 394.

Cohen, M. B., Dwyer, M. B., and Shi, J. 2007a. Exploiting constraint solving history to construct interaction test suites. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART-MUTATION'07)*. IEEE Computer Society, 121–132.

Cohen, M. B., Dwyer, M. B., and Shi, J. 2007b. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07)*. ACM, New York, 129–139.

Cohen, M. B., Dwyer, M. B., and Shi, J. 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. Softw. Engin. 34,* 5, 633–650.

Cohen, M. B., Gibbons, P. B., Mugridge, W. B., and Colbourn, C. J. 2003c. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society, Los Alamitos, CA, 38–48.

Cohen, M. B., Snyder, J., and Rothermel, G. 2006. Testing across configurations: Implications for combinatorial testing. *SIGSOFT Softw. Engin. Notes 31,* 6, 1–9.

Colbourn, C. J. 2006. Detecting and locating interaction faults. *Electron. Not. Discr. Math. 27,* 14, 17–18.

Colbourn, C. J., Martirosyan, S. S., Mullen, G. L., Shasha, D., Sherwood, G. B., and Yucas, J. L. 2005. Products of mixed covering arrays of strength two. *J. Combin. Des. 2,* 14, 124–138.

Colbourn, C. J. and McClary, D. W. 2008. Locating and detecting arrays for interaction faults. *J. Combin. Optimiz. 15,* 1, 17–48.

Czerwonka, J. 2006. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proceedings of the 24th Pacific Northwest Software Quality Conference*.

Dalal, S., Jain, A., Karunanithi, N., Leaton, J., and Lott, C. 1998a. Model-based testing of a highly programmable system. In *Proceedings of the 9th International Symposium on Software Reliability Engineering (ISSRE'98)*. IEEE Computer Society, Los Alamitos, CA, 174.

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. 1999. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. IEEE Computer Society Press, Los Alamitos, CA, 285–294.

Dalal, S. R., Jain, A., Patton, G., Rathi, M., and Seymour, P. 1998b. Aetgsm web: A web based service for automatic efficient test generation from functional requirements. In *Proceedings of the 2nd Workshop on*

Industrial-Strength Formal Specification Techniques (WIFT'98). IEEE Computer Society, Los Alamitos, CA, 84–85.

DALAL, S. R. AND MALLOWS, C. L. 1998. Factor-Covering designs for testing software. *Technometrics 40,* 3, 234–243.

DENNY, P. C. AND GIBBONS, P. B. 2000. Case studies and new results in combinatorial enumeration. *J. Combin. Des. 8*, 239–260.

DONG, G., NIE, C., XU, B., AND WANG, L. 2007. An effective iterative metamorphic testing algorithm based on program path analysis. In *Proceedings of the 7th International Conference on Quality Software (QSIC'07)*. IEEE Computer Society, Los Alamitos, CA, 292–297.

DUMITRESCU, A. 2003. Efficient algorithms for generation of combinatorial covering suites. In *Proceedings of the 14th Annual International Symposium Algorithms and Computation (ISAAC'03)*. Springer, 300–308.

DUNIETZ, I. S., EHRLICH, W. K., SZABLAK, B. D., MALLOWS, C. L., AND IANNINO, A. 1997. Applying design of experiments to software testing: experience report. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*. ACM, New York, 205–215.

FISHER, R. 1926. The arrangement of field experiments. *J. Ministry Agricult. Great Britain 33*, 503–513.

FOUCHÉ, S., COHEN, M. B., AND PORTER, A. 2007. Towards incremental adaptive covering arrays. In *Proceedings of the the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE Companion'07)*. ACM, New York, 557–560.

GHAZI, S.A. AHMED, M. 2003. Pair-wise test coverage using genetic algorithms. In *Proceedings of the Congress on Evolutionary Computation (CEC'03)*. IEEE Computer Society, Los Alamitos, CA, 1420–1424.

HARMAN, M. 2007. The current state and future of search based software engineering. In *Proceedings of the Conference on Future of Software Engineering FOSE'07*, 342–357.

HARTMAN, A. 2002. Software and hardware testing using combinatorial covering suitess. *Graph Theory, Combin. Algor. 6,* 3, 237–266.

HARTMAN, A. AND RASKIN, L. 2004. Problems and algorithms for covering arrays. *Discr. Math. 284,* 1-3, 149–156.

HNICH, B., PRESTWICH, S., AND SELENSKY, E. 2005. Constraint-Based approaches to the covering test problem. In *Proceedings of the Joint ERCIM/Co-LogNET International Workship on Constraint Solving and Constraint Logic Programming*. Lecture Notes in Computer Science. Springer Berlin, 172–186.

HNICH, B., PRESTWICH, S. D., SELENSKY, E., AND SMITH, B. M. 2006. Constraint models for the covering test problem. *Constraints 11,* 2-3, 199–219.

HULLER, J. 2000. Reducing time to market with combinatorial design method testing. In *Proceedings of the the International Council on Systems Engineering (INCOSE) Conference*.

KAHNG, A. B. AND REDA, S. 2004. Combinatorial group testing methods for the bist diagnosis problem. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'04)*. IEEE Press, 113–116.

KOBAYASHI, N. 2002. Design and evaluation of automatic test generation strategies for functional testing of software. Ph.D. thesis, Osaka University, Osaka, Japan.

KOBAYASHI, N., TSUCHIYA, T., AND KIKUNO, T. 2002. A new method for constructing pair-wise covering designs for software testing. *Inform. Process. Lett. 81,* 2, 85–91.

KRISHNAN, R., KRISHNA, S. M., AND NANDHAN, P. S. 2007. Combinatorial testing: Learnings from our experience. *SIGSOFT Softw. Engin. Notes 32,* 3, 1–8.

KUHN, D. R. AND REILLY, M. J. 2002. An investigation of the applicability of design of experiments to software testing. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW'02)*. IEEE Computer Society, Los Alamitos, CA, 91.

KUHN, D. R. AND WALLACE, D. R. 2004. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Engin. 30,* 6, 418–421.

LEI, Y., CARVER, R. H., KACKER, R., AND KUNG, D. C. 2007a. A combinatorial testing strategy for concurrent programs. *Softw. Test., Verif. Reliab. 17,* 4, 207–225.

LEI, Y., KACKER, R., KUHN, D. R., OKUN, V., AND LAWRENCE, J. 2007b. Ipog: A general strategy for t-way software testing. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE Computer Society, Los Alamitos, CA, 549–556.

LEI, Y., KACKER, R., KUHN, D. R., OKUN, V., AND LAWRENCE, J. 2008. Ipog/ipog-d: Efficient test generation for multi-way combinatorial testing. *Softw. Test., Verif. Reliab. 18,* 3, 125–148.

LEI, Y. AND TAI, K. C. 2001. In-parameter-oder: A test generation strategy for pairwise testing. *Tech. rep.* TR 2001-03 *5*, 109–111.

LI, Z., HARMAN, M., AND HIERONS, R. M. 2007. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Engin. 33,* 4, 225–237.

LOTT, C., JAIN, A., AND DALAL, S. 2005. Modeling requirements for combinatorial software testing. *SIGSOFT Softw. Engin. Notes 30,* 4, 1–7.

MAITY, S. AND NAYAK, A. 2005. Improved test generation algorithms for pair-wise testing. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*. IEEE Computer Society, Los Alamitos, CA, 235–244.

MANDL, R. 1985. Orthogonal latin squares: An application of experiment design to compiler testing. *Commu. ACM 28,* 10, 1054–1058.

MARRE, M. AND BERTOLINO, A. 2003. Using spanning sets for coverage testing. *IEEE Trans. Softw. Engin. 29,* 11, 974–984.

MATS, G., LINDSTR, B., OFFUTT, J., AND ANDLER, S. F. 2006a. An evaluation of combination strategies for test case selection. *Empir. Softw. Engin. 11,* 4, 583–611.

MATS, G. AND OFFUTT, J. 2007. Input parameter modeling for combination strategies. In *Proceedings of the 25th Conference on IASTED International Multi-Conference (SE'07)*. ACTA Press, 255–260.

MATS, G., OFFUTT, J., AND ANDLER, S. F. 2005. Combination testing strategies: A survey. *Softw. Test. Verif. Reliab. 15,* 3, 167–199.

MATS, G., OFFUTT, J., AND MELLIN, J. 2006b. Handling constraints in the input space when using combination strategies for software testing. Tech. rep. HS-IKI-TR-06-001 *1*, 1–39.

NIE, C., XU, B., SHI, L., AND DONG, G. 2005. Automatic test generation for n-way combinatorial testing. In *Proceedings of the 2nd International Workshop on Software Quality (SOQUA)*. Springer Berlin, 203–211.

NIE, C., XU, B., SHI, L., AND WANG, Z. 2006a. A new heuristic for test suite generation for pair-wise testing. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, (SEKE'06)*, 517–521.

NIE, C., XU, B., WANG, Z., AND SHI, L. 2006b. Generating optimal test set for neighbor factors combinatorial testing. In *Proceedings of the 6th International Conference on Quality Software (QSIC'06)*. IEEE Computer Society, Los Alamitos, CA, 259–265.

NURMELA, K. J. 2004. Upper bounds for covering arrays by tabu search. *Discr. Appl. Math. 138,* 1-2, 143–152.

OFFUTT, A. J. 1994. A practical system for mutation testing: Help for the common programmer. In *Proceedings of the IEEE International Test Conference on TEST: The Next 25 Years*. IEEE Computer Society, Los Alamitos, CA, 824–830.

QU, X., COHEN, M., AND WOOLF, K. 2007. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *Proceedings of the IEEE International Conferance on Software Maintenance (ICSM)*. IEEE Computer Society, 413–418.

SALEM, A. M. 2001. A software testing model: Using design of experiments (doe) and logistic regression. Ph.D. thesis, Florida Institute of Technology, Melbourne, Florida.

SALEM, A. M., REKAB, K., AND WHITTAKER, J. A. 2004. Prediction of software failures through logistic regression. *Inform. Softw. Technol. 46,* 12, 781–789.

SAMPATH, S., BRYCE, R., VISWANATH, G., KANDIMALLA, V., AND KORU, A. 2008. Prioritizing user-session-based test cases for web applications testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, 141–150.

SCHROEDER, P. J. 2002. Black-Box test reduction using input-output analysis. Ph.D. thesis, Illinois Institute of Technology, Chicago, IL.

SCHROEDER, P. J., BOLAKI, P., AND GOPU, V. 2004. Comparing the fault detection effectiveness of n-way and random test suites. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE)*. IEEE Computer Society, 49–59.

SCHROEDER, P. J., FAHERTY, P., AND KOREL, B. 2002. Generating expected results for automated black-box testing. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 139–148.

SCHROEDER, P. J. AND KOREL, B. 2000a. Black-box test reduction using input-output analysis. *SIGSOFT Softw. Engin. Notes 25,* 5, 173–177.

SCHROEDER, P. J. AND KOREL, B. 2000b. Constraint of model-based test generation using input-output analysis. In *Proceedings of the Automated Software Engineering Doctoral Symposium*. ACM, 35–43.

SEROUSSI, G. AND BSHOUTY, N. 1988. Vector sets for exhaustive testing of logic circuits. *IEEE Trans. Inf. Theory 34,* 3, 513–522.

SHERWOOD, G. 1994. Effective testing of factor combinations. In *Proceedings of the 3rd International Conference on Software Testing, Analysis, and Review(STAR94).*

SHERWOOD, G. B., MARTIROSYAN, S. S., AND COLBOURN, C. J. 2005. Covering arrays of higher strength from permutation vectors. *J. Combin. Des. 3,* 14, 202–213.

SHI, L., NIE, C., AND XU, B. 2005. A software debugging method based on pairwise testing. In *Proceedings of the International Conferences on Computational Science (ICCS'05)*. Springer, 1088–1091.

SHIBA, T., TSUCHIYA, T., AND KIKUNO, T. 2004. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*. IEEE Computer Society, Los Alamitos, CA, 72–77.

TAI, K. C. AND LEI, Y. 2002. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Engin. 28,* 1, 109–111.

TUNG, Y.-W. AND ALDIWAN, W. 2000. Automating test case generation for the new generation mission software system. In *Proceedings of the IEEE Aerospace Conference.* 431–437.

TURBAN, R. C. 2006. Algorithms for covering arrays. Ph.D. thesis, University of Ahzam Tempe, AZ.

WANG, Z., NIE, C., AND XU, B. 2007. Generating combinatorial test suite for interaction relationship. In *Proceedings of the 4th International Workshop on Software Quality Assurance (SOQUA'07)*. ACM, New York, 55–61.

WANG, Z., XU, B., AND NIE, C. 2008. Greedy heuristic algorithms to generate variable strength combinatorial test suite. In *Proceedings of the International Conference on Quality Software (QSIC)*. 155–160.

WHITE, L. AND ALMEZEN, H. 2000. Generating test cases for gui responsibilities using complete interaction sequences. In *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*. IEEE Computer Society, Los Alamtos, CA, 110.

WILLIAMS, A. W. 2000. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom'00)*. Kluwer, 59–74.

WILLIAMS, A. W. 2002. Software component interaction testing: coverage measurement and generation of configurations. Ph.D. thesis, University of Ottawa, Canada.

WILLIAMS, A. W. AND PROBERT, R. L. 2002. Formulation of the interaction test coverage problem as an integer program. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV (TestCom'02)*. Kluwer, 283.

WILLIAMS, A. W. AND PROBERT, R. L. 1996. A practical strategy for testing pair-wise coverage of network interfaces. In *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE'96)*. IEEE Computer Society, Los Alamtos, CA, 246.

WILLIAMS, A. W. AND PROBERT, R. L. 2001. A measure for component interaction test coverage. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'01)*. IEEE Computer Society, Los Alamitos, CA, 304.

XU, B., NIE, C., SHI, L., CHU, W. C., YANG, H., AND CHEN, H. 2003a. Test plan design for software configuration testing. In *Software Engineering Research and Practice*, CSREA Press, 686–692.

XU, L., XU, B., NIE, C., CHEN, H., AND YANG, H. 2003b. A browser compatibility testing method based on combinatorial testing. In *Proceedings of the International Conference on Web Engineering (ICWE*. Springer, Berlin, 310–313.

YAN, J. AND ZHANG, J. 2006. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In *Proceedings of the 30th Annual International Conference on Computer Software and Applications (COMPSAC'06)*. 385–394.

YILMAZ, C., COHEN, M. B., AND PORTER, A. A. 2006. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Softw. Engin. 32,* 1, 20–34.

Received December 2008; revised April 2009; accepted August 2009