Top-down and Bottom-up Strategies for Generating Incremental Covering Arrays

***Abstract*—Combinatorial Testing (CT) is an effective technique for testing the interactions of factors in the Software Under Test (SUT). By designing an efficient set of test cases, i.e., a covering array, CT aims to check every possible valid interaction in SUT. Most existing covering array generation algorithms require a given *degree t* in prior, such that only the interactions with no more than t factors are to be checked. In practice, however, the value of t cannot be properly determined, especially for systems with complex interaction space. Hence, incremental covering arrays are preferred. In this paper, we propose two strategies (*Bottom-up* and *Top-down*) for generating incremental covering arrays which can increase the coverage strength when required. A comparative evaluation of these two strategies is performed on 55 software subjects (including 25 real-world software and 30 synthetic systems). The results shows that strategy *Bottom-up* have a significant advantage over strategy *Top-down*, and it also performs better than the traditional approach.**

***Index Terms*—Combinatorial Testing, Covering Array, Incremental Covering Array, Bottom-up and Top-down strategy**

I. Introduction

With the growing complexity and scale of modern software systems, various factors, such as input values and configure options, can affect the behaviour of a system. Even worse, some interactions between them can trigger unexpected negative effects on the software. To ensure the correctness and quality of the software system, it is desirable to detect and locate these *bad* interactions. The simplest way to solve this problem is to perform exhaustive testing for all the possible valid interactions of the system. It is, however, not practical due to the combinatorial explosion. Therefore, the selection of a group of representative test cases from the whole testing space is required.

Combinatorial testing has been proven to be effective in sampling an effective set of test cases[1]. It works by generating a relatively small set of test cases, i.e., a covering array, to test the interactions involving a given number of factors that may affect the behavior of a system. The number of factors involved in those selected interactions is limited in a moderate range, which is usually from 2 to 6 [2].

Many algorithms have been proposed to generate covering arrays. Despite many differences between them, those works all assume that the degree t is a given priori. The degree t indicates the largest number of factors involved in the interactions to be covered, and the corresponding covering arrays which can satisfy this coverage criteria is called t-way covering arrays. In practice, however, the value of t is difficult to determine due to two reasons as follows. First, many software systems suffer from complex interaction space which make it challenging to estimate t. In such case, even experienced testers can make mistakes and estimate a wrong value for t, which can significantly affect the effectiveness and efficiency of CT. Specifically, if t is estimated to be larger than required, many redundant test cases will be generated, which is a waste of computing resource. And if t is estimated to be smaller than required, then the generated covering array is not sufficient to obtain an effective test set. Second, even though t has been properly determined, there may not be enough time to completely execute all the test cases in the covering array. This is because testing software only makes sense before the next version is released. This time interval between two release versions may sometimes be too short for a complete testing of a high-strength covering array, especially in the scenario of continuous integration [3].

To address these shortcomings of traditional covering arrays, the notion of incremental covering array [3] has been proposed. Such object can be deemed as adaptive covering array, which can increase the degree t when required. As it can generate higher strength covering array based on lower strength covering array, it can reduce the cost when comparing with generating multiple ways of covering arrays. Additionally, it can be better applied on testing the software of which the released time is frequently changed and cannot be predicted. Another advantage for generating incremental covering array is that, testers can detect most faults in the software as soon as possible. This is because according to [2], most faults (about 70% to 80%) are caused by 2-degree interactions, and almost all faults can be covered by 6-way covering arrays. As incremental covering arrays first cover those lower-degree interactions, the faults caused by them will be detected sooner.

In consideration of the size of the total number of test cases, we argue that this approach of generating incremental covering array may produce too many test cases. This is obvious, as generating higher-strength covering array based on the lower-strength covering array (called *bottom-up* strategy later) does not aim to optimize the size of the higher-strength covering array. As a result, it may generate more test cases than those approaches that focus on generating a particular higher-strength covering array.

Then, a natural question, and also the motivation of this paper is, **is it possible to generate an incremental covering array with the same number of the overall test cases as those by the particular high-way covering array generation algorithms ?** Due to the obvious conclusion that any high way covering array must cover all the lower way interactions, the answer for this question is *yes*, as we can just apply a particular covering array generation algorithm to generate

the high-strength covering array, and then generate the lower-strength covering arrays by extracting some subset of the test cases which can cover all the lower degree interactions. We refer to this strategy as the *top-down* strategy.

In this paper, we propose these two strategies and evaluate their performance by comparing them at constructing several incremental covering arrays on 55 wildly used subjects (including 25 real-world software and 30 synthetic systems). The results shows that strategy *Bottom-up* have a significant advantage over strategy *Top-down* at the size of covering arrays (especially for the lower-strength covering array) and the time cost. Additionally, strategy *Bottom-up* also performs better than the traditional approach.

Our contributions include:

1) We propose a formal description of incremental covering array, and give a proof of the existence of it.
2) We propose two strategies for generating incremental covering arrays.
3) We conduct a series of experiments to evaluate the characteristics of these two strategies, and compare the performance of our approach with traditional approach.
4) We offer a guideline for selecting which strategy when generating incremental covering array in practice.

II. Background

This section gives some formal definitions related to CT. Assume that the behaviour of SUT is influenced by *k* parameters, and each parameter p_i has a_i discrete values from the finite set V_i, i.e., $a_i = |V_i|$ (i = 1,2,..k). Then a *test case* of the SUT is a group of values that are assigned to each parameter, which can be denoted as $(v_1, v_2, ..., v_k)$. An t-degree interaction can be formally denoted as $(-, ..., v_{n_1}, -, ..., v_{n_2}, -, ..., v_{n_t}, -, ...)$, where some t parameters have fixed values and other irrelevant parameters are represented as "-". In fact, a test case can be regarded as a k-degree interaction.

A. Covering array

Definition 1. A t-way *covering array* $MCA(N; t, k, (a_1, a_2, ..., a_k))$ is a test set in the form of $N \times k$ table, where each row represents a *test case* and each column represents a parameter. For any *t* columns, each possible *t*-degree interaction of the t parameters must appear at least once. When $a_1 = a_2 = ... = a_k = v$, a t-way covering array can be denoted as $CA(N; t, k, v)$.

For example, Table I (a) shows a 2-way covering array $CA(5; 2, 4, 2)$ for the SUT with 4 boolean parameters. For any two columns, any 2-degree interaction is covered. Covering array has proven to be effective in detecting the failures caused by interactions of parameters of the SUT. Many existing algorithms focus on constructing covering arrays such that the number of test cases, i.e., N, can be as small as possible.

B. Incremental covering array

Definition 2. An incremental covering array $ICA([N_{t_1}, N_{t_1+1}, ..., N_{t_2}]; [t_1, t_2], k, (a_1, a_2, ..., a_k))$ is a test set in the form of $N_{t_2} \times k$ table, where $t_1 < t_2$ and $N_{t_1} < N_{t_1+1} < , ..., < N_{t_2}$. In this table, the first N_{t_i} lines ($t_1 \leq t_i \leq t_2$) is a covering array $MCA(N_{t_i}; t_1 + i, k, (a_1, a_2, ..., a_k))$.

When $a_1 = a_2 = ... = a_k = v$, an incremental covering array can be denoted as $ICA([N_{t_1}, N_{t_1+1}, ..., N_{t_2}]; [t_1, t_2], k, v)$.

Table I shows an example of incremental covering array, in which the two-way covering array $CA(5; 2, 4, 2)$ is a subset of the three-way covering array $CA(9; 3, 4, 2)$, which is also an incremental covering array $ICA([5, 9]; [2, 3], 4, 2)$.

TABLE I: Experiment of Incremental covering array

(a) *CA(5;2,4,2)*

0	0	1	0
1	0	0	0
1	1	1	0
0	1	0	1
1	0	1	1

(b) *CA(9;3,4,2)* && *ICA([5,9];[2,3],4,2)*

0	0	1	0
1	0	0	0
1	1	1	0
0	1	0	1
1	0	1	1
0	1	0	0
0	0	0	1
0	1	1	1
1	1	0	1

Theorem. For each covering array $MCA(N_{t_2}; t_2, k, (a_1, a_2, ..., a_k))$, we can find an $ICA([N_{t_1}, N_{t_1+1}, ..., N_{t_2}]; [t_1, t_2], k, (a_1, a_2, ..., a_k))$, s.t., their test cases are the same.

Proof. We just need to prove that for any covering array, $MCA(N_t; t, k, (a_1, a_2, ..., a_k))$, we can find a $MCA(N_{t-1}; t-1, k, (a_1, a_2, ..., a_k))$, such that, $N_{t-1} < N_t$ and for any test case $f \in MCA(N_{t-1}; t-1, k, (a_1, a_2, ..., a_k))$, it has $f \in MCA(N_t; t, k, (a_1, a_2, ..., a_k))$.

First, $MCA(N_t; t, k, (a_1, a_2, ..., a_k))$ itself must be an $MCA(N_t; t-1, k, (a_1, a_2, ..., a_k))$, as it must cover all the $(t-1)$-degree interactions.

Then, assume to obtain a $(t-1)$-way covering array, any one test case in $MCA(N_t; t, k, (a_1, a_2, ..., a_k))$ can not be reduced. With this assumption, any test case will cover at least one $(t-1)$-degree interaction that only appears in this test case. Without loss of generality, let test case $(v_1, v_2, ..., v_k)$ cover the $(t-1)$-degree interaction $(-, ..., v_{p_1}, ...-, ...v_{p_2}, ..., -, ..., v_{p_{t-1}}, ... - ..., v_{p_t}, ..., -, ...)$ which only appears in the test case. Then obviously the t-degree interaction $(v'_1, ..., v_{p_1}, ...-, ...v_{p_2}, ..., -, ..., v_{p_{t-1}}, ... - ..., v_{p_t}, ..., -, ...)$ ($v'_1 \neq v_1$) will never be covered by any test case in $MCA(N_t; t, k, (a_1, a_2, ..., a_k))$, and hence $MCA(N_t; t, k, (a_1, a_2, ..., a_k))$ is not a t-way covering array (Note this is based on that the parameter can take on more than one value).

This is a contradiction, and means that we can reduce at least one test case in $MCA(N_t; t, k, (a_1, a_2, ..., a_k))$, so that it is still a $(t-1)$-way covering array. □

This theorem shows the existence of the incremental covering arrays. As discussed before, generating the incremental covering arrays is of importance, as it supports adaptively

increasing the coverage strength. According to this theorem, when testing a SUT, testers can firstly execute the lowest-strength covering array in the incremental covering arrays, and then execute additional test cases from those higher-strength covering arrays as required. The reuse of previously executed test cases will reduce the cost for generating multiple different-ways covering arrays.

III. Generating incremental covering arrays

This section presents two strategies to generate the incremental covering arrays. The first strategy, i.e., the *bottom-up* strategy starts from generating the lowest-strength covering array and then the higher-strength ones. The second strategy, i.e., the *top-down* strategy firstly generated the highest-strength covering array, then the lower-strength covering array.

A. Bottom-up strategy

This strategy is listed as Strategy 1. The inputs for this strategy consists of the values for each parameter of the SUT –$Params$, the lowest strength t_1 of the covering array in the incremental covering array, and the highest strength t_2 of the covering array . The output of this strategy is an incremental covering array – ICA.

Strategy 1 Bottom-up strategy

Input: $Params$ ▷ Parameters (and their values)
t_1 ▷ the lowest strength
t_2 ▷ the highest strength
Output: ICA ▷ the incremental covering arrays
1: $ICA \leftarrow EmptySet$
2: **for** $t_i = t_1;\ t_i \leq t_2;\ t_i\ ++$ **do**
3: $CA_i \leftarrow EmptySet$
4: **if** $t_i == t_1$ **then**
5: $CA_i \leftarrow CA\_Gen(Params, t_i)$
6: **else**
7: $CA_i \leftarrow extend(Params, t_i, CA_{i-1})$
8: **end if**
9: $ICA.append(CA_i)$
10: **end for**
11: **return** ICA

This strategy generates the covering array from lower-strength to higher-strength (line 2). If the current coverage strength t_i is equal to t_1, it just utilizes a covering array generation algorithm to generate the particular covering array (line 4 - 5). Otherwise, it will first take the previous generated covering array CA_{i-1} as seeds, and then utilize covering array generation algorithm to append additional test cases to satisfy higher coverage criteria (line 6 - 7).

Fig.1 presents an example for constructing $ICA([6, 13, 24]; [2, 4], 5, 2)$ by this strategy. In this example, the covering array generation algorithm used is AETG [4]. The two-way covering array (test cases c_1 to c_6) is directly generated, and the three-way covering array (test cases c_1 to c_{13}) is generated by adding additional test cases (test cases c_7 to c_{13}) based on the previous two-way covering array. The four-way covering array (test cases c_1 to c_{24}) is constructed on the previous three-way covering array. In total, to reach the 4-way covering array, 24 test cases are needed for this strategy.

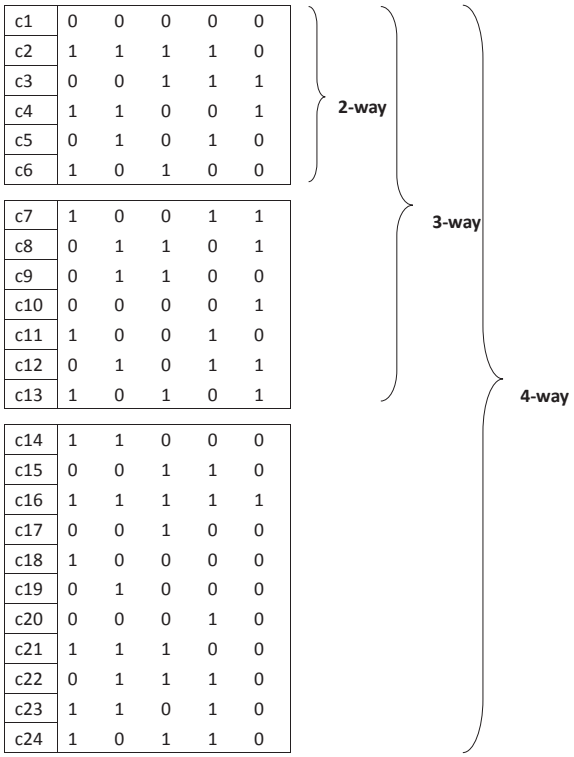

Fig. 1: Bottom-up strategy example

B. Top-down strategy

This strategy is listed as Strategy 2, which generates covering arrays in the opposite order (line 2) of the bottom-up strategy. Similarly, if the current coverage strength t_i is equal to t_2, it directly generates the particular covering array (line 4 - 5). Otherwise, it will extract the covering array from a higher covering array (CA_{i+1}) (line 6 - 7).

Strategy 2 Top-down strategy

Input: $Params$ ▷ Parameters (and their values)
t_1 ▷ the lowest strength
t_2 ▷ the highest strength
Output: ICA ▷ the incremental covering arrays
1: $ICA \leftarrow EmptySet$
2: **for** $t_i = t_2;\ t_i \geq t_1;\ t_i\ --$ **do**
3: $CA_i \leftarrow EmptySet$
4: **if** $t_i == t_2$ **then**
5: $CA_i \leftarrow CA\_Gen(Params, t_i)$
6: **else**
7: $CA_i \leftarrow extract(Params, t_i, CA_{i+1})$
8: **end if**
9: $ICA.append(CA_i)$
10: **end for**
11: **return** ICA

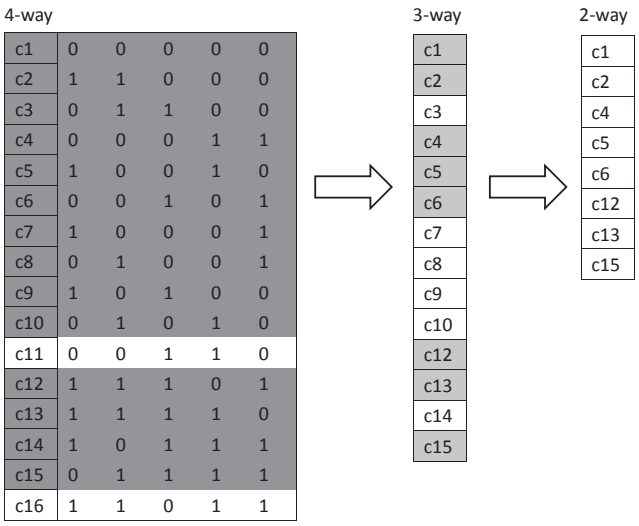

Fig. 2: Top-down strategy example

An example for this strategy is given in Fig.2. In this example, we first generated the 4-way covering array $CA(16; 4, 5, 2)$. Then we selected 14 test cases to form a three-covering array $CA(14; 3, 5, 2)$. Next the two-way covering array $CA(8; 2, 5, 2)$ is extracted from $CA(14; 3, 5, 2)$. The rows with dark background from the higher-strength covering arrays represent those selected test cases.

From the two examples, an obvious observation is that for the *top-down* strategy, it has a significant advantage over the *bottom-up* strategy with respect to the size of the 4-way covering array (16 test cases generated by *top-down*, while 24 by *bottom-up*). But when considering the lower-strength covering arrays, the *bottom-up* strategy performs slightly better (13 and 6 by *bottom-up* for degree 3 and 2, respectively; while 14 and 8 by *top-down*).

IV. Implementation of the approach

In this section, we will describe a specific implementation of the two strategies proposed in the previous section.

A. Covering array generation method

Our covering array generation approach is IPOG [5], [6], [7], which firstly builds a *t*-way test set for the first *t* parameters, then extends the test set to build a t-way test set for the first *t+1* parameters, and then continues to extend the test set until it builds a t-way test set for all the parameters. We select this algorithm mainly because it is very efficient, that is, it can quickly generate a t-way covering array even though *t* is a relatively large number (up to 6). This is important, as in our experiments, some subjects contain an extremely large input space, which is unfavorable to apply some time-consuming covering array generation approaches.

B. Constraints handling

Our constraints handling technique is based on Minimum Forbidden Tuples (MFTs) [8], [9], which is well supported by IPOG generation algorithm. A forbidden tuple is a tuple as the value assignment of some parameters that violates constraints. A minimum forbidden tuple is a forbidden tuple of minimum size that covers no other forbidden tuples. Given some constraints for one SUT, we should compute all the MFTs of the SUT, and then limit the test cases to the ones that do not contain any tuple in the MFTs.

C. Seeding test cases

When given a lower-way covering array as seeds, we need to eliminated those higher-degree tuples that have already been covered by this covering array. After this, we need to generate additional test cases to cover the un-covered higher-degree tuples until all the higher-degree tuples are covered.

D. Extract lower-way covering array

This is the key part for the top-down strategy. Considering the test cost, it is desirable to obtain a minimal t-way covering array CA_t from a $t+1$-way covering array CA_{t+1}. However, this is impractical if CA_{t+1} is too large, because to get the minimal CA_t, we need to exhaustively check every possible subset of the CA_{t+1}. Hence, in this paper, we decide to adopt a greedy method. Specifically, at each iteration, we will select the test case from CA_{t+1} which has the longest hamming distance from the test cases that have already been selected. The hamming distance between two test cases, say, c_1 and c_2, is computed in the following formula:

$$\frac{\#the\ num\ of\ different\ elements\ between\ c_1\ and\ c_2}{\#the\ num\ of\ all\ the\ elements\ in\ c_1\ and\ c_2}$$

The selection process repeats until we obtain a complete t-way covering array.

V. Experiments and Results

A. Research Questions

We set up the following three research questions to investigate the effectiveness and the efficiency of the two strategies.

RQ1.What is the size of the incremental covering array generated by the two strategies *bottom-up*, and *top-down*?

To evaluate the two strategies, the size of the covering array is one important metric. In CT, the main goal is to make the size of the covering array as small as possible. With respect to the incremental covering array, instead of focusing on one covering array, we should evaluate all the covering arrays with different strengths in it. How to apply the two strategies in practice depends on the strength of the covering arrays they have advantages at generating them. For example, if one strategy can generate smaller size of lower-way covering array, it is a better choice at generating incremental covering array within a short testing time (current software under testing will be soon expired for the releasing of the next version). On the contrary, if one strategy can generate smaller size of higher-way covering array, it is better to be applied on testing the software within a long testing time.

RQ2.How efficient is the two strategies?

The time that is needed to generate covering array is also a very important metric to evaluate the two strategies. Many approaches in CI can is limited to a given time budgeted for the search optimal covering arrays. But when apply covering array in practice, it is normally not possible to obtain a proper time budgeted in prior. Hence, it is necessary to optimize the covering array generation process to reduce the time cost. Similar to our first research question, we will investigate the time to generate the covering arrays with different test strengths in the the incremental covering array.

RQ3. Compared with the traditional approach, do the two strategies have any advantage at effectiveness and efficiency?

At last, we need to compare with traditional approach. To our best knowledge, the study in [3] is the only research that aims at handling the incremental covering array generation problem. We need to investigate the differences

B. Benchmarks

We used 55 SUT as the subjects of our experiments, which are shown in Table II. These subjects are wildly used to evaluate the performance of the covering array generation approach. Among these subjects, the first 20 subjects (from *Banking1* to *Telecom*), are obtained from a recent benchmark created by Segall et al [10], which have already been used in several works [11], [12]. These 20 subjects cover a wide range of applications, including telecommunications, health care, storage and banking systems. The following five subjects (from *SPIN-S* to *Bugzilla*) were firstly introduced by Cohen et al. [13], [14]. Among them, SPIN-S and SPIN-V are two components for model simulation and verification, GCC is a compiler system, Apache is a web server application, and Bugzilla is a web-based bug tracking system. These five subjects have been wildly used in literatures [15], [13], [14], [16], [17], [18], [12]. The last 30 (from Syn1 to Syn30) are synthetic subject models that are generated by Cohen et al. [14]. These synthetic subject models are designed with similar characterization (input space, constraints, etc.) of those real subjects and have been used in following studies[16], [17], [18], [12].

In Table II, the model is presented in the abbreviated form $\#values^{\#number\ of\ parameters}$, e.g., $7^3 6^2$ indicates the software has 3 parameters that can take 7 values and 2 parameters take 6 values. The constraint is presented in the form $\#degree^{\#number\ of\ constraints}$, e.g., $2^3 3^{18}$ means that the SUT contains 3 constraints with 2-degree and 18 constraints with 3-degree.

C. Experiment set-up

For each SUT listed in Table II, we adopted the two strategies, i.e., *Top-down* and *Bottom-up*, to generate an incremental covering array from 2-way to 6-way. The size of each covering array contained in this incremental covering array is recorded. We also record the time (seconds) that is consumed to generate these covering arrays.

TABLE II: The models of the SUT

Name	Model	Constraint
Banking1	$3^4 4^1$	5^{112}
Banking2	$2^{14} 4^1$	2^3
CommonProtocal	$2^{10} 7^1$	$2^{10} 3^{10} 4^{12} 5^{96}$
Concurrency	2^5	$2^4 3^1 5^2$
Healthcare1	$2^6 3^2 5^1 6^1$	$2^3 3^{18}$
Healthcare2	$2^5 3^6 4^1$	$2^1 3^6 5^{18}$
Helathcare3	$2^{16} 3^6 4^5 5^1 6^1$	2^{31}
Helathcare4	$2^{13} 3^{12} 4^6 5^2 6^1 7^1$	2^{22}
Insurance	$2^6 3^1 5^1 6^2 11^1 13^1 17^1 31^1$	-
NetworkMgmt	$2^2 4^1 5^3 10^2 11^1$	2^{20}
ProcessorComm1	$2^3 3^6 4^6$	2^{13}
ProcessorComm2	$2^3 3^{12} 4^8 5^2$	$1^4 2^{121}$
Services	$2^3 3^4 5^2 8^2 10^2$	$3^{386} 4^2$
Storage1	$2^1 3^1 4^1 5^1$	4^{95}
Storage2	$3^4 6^1$	-
Storage3	$2^9 3^1 5^3 6^1 8^1$	$2^{38} 3^{10}$
Storage4	$2^5 3^7 4^1 5^2 6^2 7^1 9^1 13^1$	2^{24}
Storage5	$2^5 3^8 5^3 6^2 8^1 9^1 10^2 11^1$	2^{151}
SystemMgmt	$2^5 3^4 5^1$	$2^{13} 3^4$
Telecom	$2^5 3^1 4^2 5^1 6^1$	$2^{11} 3^1 4^9$
SPIN-S	$2^{13} 4^5$	2^{13}
SPIN-V	$2^{42} 3^2 4^{11}$	$2^{47} 3^2$
GCC	$2^{189} 3^{10}$	$2^{37} 3^3$
Apache	$2^{158} 3^8 4^4 5^1 6^1$	$2^3 3^1 4^2 5^1$
Bugzilla	$2^{49} 3^1 4^2$	$2^4 3^1$
Syn1	$2^{86} 3^3 4^1 5^5 6^2$	$2^{20} 3^3 4^1$
Syn2	$2^{86} 3^3 4^3 5^1 6^1$	$2^{19} 3^3$
Syn3	$2^{27} 4^2$	$2^9 3^1$
Syn4	$2^{51} 3^4 4^2 5^1$	$2^{15} 3^2$
Syn5	$2^{155} 3^7 4^3 5^5 6^4$	$2^{32} 3^6 4^1$
Syn6	$2^{73} 4^3 6^1$	$2^{26} 3^4$
Syn7	$2^{29} 3^1$	$2^{13} 3^2$
Syn8	$2^{109} 3^2 4^2 5^3 6^3$	$2^{32} 3^4 4^1$
Syn9	$2^{57} 3^1 4^1 5^1 6^1$	$2^{30} 3^7$
Syn10	$2^{130} 3^6 4^5 5^2 6^4$	$2^{40} 3^7$
Syn11	$2^{84} 3^4 4^2 5^2 6^4$	$2^{28} 3^7$
Syn12	$2^{136} 3^4 4^3 5^1 6^3$	$2^{23} 3^4$
Syn13	$2^{124} 3^4 4^1 5^2 6^2$	$2^{22} 3^4$
Syn14	$2^{81} 3^5 4^3 6^3$	$2^{13} 3^2$
Syn15	$2^{50} 3^4 4^1 5^2 6^1$	$2^{20} 3^2$
Syn16	$2^{81} 3^3 4^2 6^1$	$2^{30} 3^4$
Syn17	$2^{128} 3^3 4^2 5^1 6^3$	$2^{25} 3^4$
Syn18	$2^{127} 3^2 4^4 5^6 6^2$	$2^{23} 3^4 4^1$
Syn19	$2^{172} 3^9 4^9 5^3 6^4$	$2^{38} 3^5$
Syn20	$2^{138} 3^4 4^5 5^4 6^7$	$2^{42} 3^6$
Syn21	$2^{76} 3^3 4^2 5^1 6^3$	$2^{40} 3^6$
Syn22	$2^{72} 3^4 4^1 6^2$	$2^{31} 3^4$
Syn23	$2^{25} 3^1 6^1$	$2^{13} 3^2$
Syn24	$2^{110} 3^2 5^1 6^4$	$2^{25} 3^4$
Syn25	$2^{118} 3^6 4^2 5^1 6^4$	$2^{23} 3^3 4^1$
Syn26	$2^{87} 3^1 4^3 5^4$	$2^{28} 3^4$
Syn27	$2^{55} 3^2 4^2 5^1 6^4$	$2^{17} 3^3$
Syn28	$2^{167} 3^{16} 4^2 5^3 6^6$	$2^{31} 3^6$
Syn29	$2^{134} 3^7 5^3$	$2^{19} 3^3$
Syn30	$2^{73} 3^3 4^3$	$2^{20} 3^2$

Besides these two strategies, we also adopted the strategy proposed in the study [3] to generate these incremental covering arrays. This strategy can be regarded as one special bottom-up strategy, but with one difference: it should generate a higher-way covering array at first, and then take some part in it to fill in the lower-covering array.

Note that the study [3] did not give a specific implementation of the strategy. Hence, in this paper, we implemented this

strategy with the covering array generation approach–IPOG, which is same approach used in our approaches. By doing this, we can eliminated the influence from the covering array generation approach, and only focus on the performance among different strategies. Another point that needs to note is that the approach listed in [3] generates three distinct incremental covering arrays for one SUT. Generating multiple covering arrays can increase their abilities to handle the random or non-determined failures. To make a fair comparison, we only record one covering array in it (with the minimal size among the three covering arrays), and the corresponding time that is needed to generate it.

We ran all the experiments on a server which has a 6-core CPU of 2.0GHz (Intel Xeon E5-2640 v2), and a 16 GB memory. It took us two weeks in total to obtain all the results. Note that not all the covering arrays can run up to 6 way because of the memory limitation and computing time.

D. Results and Discussion

The results are listed in Fig. 3 and Fig. 4 for covering array size and consuming time, respectively. There are 55 sub figures in these two main figures, one for each subject. In each subject, the x-, listed the detail covering array strength, ranged from 2-way to 6-way. The y shows the , which are nomralized to.

These data are normalized to rend a clear view.

One important observation in this figure is that there is significant advantage over the lower covering array. Specifically, among the 55 subjects, there are XXX subjects (including , , etc) that peforms good than in 4-way or equal, XXX subjects that ,2-way or equal . 2 way . On the other hand, there are only XXX subjects that (Note that the adavantge is trival such that it can not be), and only XXX subjects.

This result implies that . One reason is that our extracting covering array is not optimal. Due to the greedy , we use ,. this may. Another reason is our algorithm IPOG. This is beaccause that even though for higher-covering array, the is not as week as . It indicates that IPOG did not optiaml the higher-coverin array (mainly because it is enssitall build on lower covering array,).

Above all, the answer to the first question is :

With respect to the effiecy, we can also find that bottom-up can performs well than top-down. In detail, among the 55 subjects, there are . and there, while in 55, there is .

Another important observation that can support the conclusion is that Bottom-up can run more higher-way covering array. Specifically, there are 6 subjects that bottom up can run more strength than top down.

Hence, the answer to the second question is :

For the last reaseach question, we can first observe that .

When compared to bottom up. With repsect to the size, the is almost the same as bot up strategy, only with slightl disadavantage (See subject). We believe the reason is that they are all eseensal bottom up strategy. With respect to the time cost. We found that is much more than bot up. In fact, among 55 subjects, there are. We believe the reason is that it should geneates a higher-way covering array first, which does no need in our bot-up startegy.

When compared to top down. The approach has also signficant adavantge over it . In fact, among 55 subjects, there are, while for top down , it only has XXXX over traditinoal. Even for the time cost, traditional apprapoch also has adavange on it (there are XXX subjects, that.

Above all, the answer to the 3rd question is:

traditional is better than top-down, but not as good as bot up.

VI. Threats to Validity

There are several threats to validity in our studies.

First, our incremental covering array generation approach is implemented by using IPOG. Although IPOG can efficiently generate higher-way covering array for the SUT with large input space, it may generate too many test cases (due to the greedy strategy). It is necessary to try more covering array generation approaches to make the conclusion more general.

Second, the method to extract lower-way covering arrays from higher coverings is a greedy algorithm in this paper. This may result in that the size of the lower-way covering arrays generated by *Top-down* strategy may be larger than needed. If adopted some other methods, e.g., the post-optimal approach [19], [20], the number of test cases generated by *Top-down* strategy may be significantly decreased.

At last, our results are based on the 55 wildly used input models. That is, with other input models, the results may be different from listed in this paper. It is appealing to further study the influence of the characteristics of the input models, e.g., the number of parameters, the number of values, and the constraints, on the results of incremental covering array generation approach.

VII. Related work

Combinatorial testing has been proven to a effective testing technique in practice [21], especially on domains like configuration testing [22], [23], [24] and software inputs testing [4], [25], [26]. The most important task in CT is to generate covering arrays, which support the checking of various valid interactions of factors that influence the system under testing.

Nie et al. [1] gave a survey for combinatorial testing, in which the methods for generating covering arrays are classified. However, traditional static covering arrays can be hardly directly applied in practice, hence many adaptive methods are proposed.

Cohen et al. [27], [14] studied the constraints that can render some test cases in the covering array invalid. They proposed a SAT-based approach to avoid the impact. Nie et al.[28] proposed a model for adaptive CT, in which both the failure-inducing interactions and constraints can be dynamically detected and removed in the early test cases generation iteration. Further, the coverage strength of covering array can be adaptively changed as required.

S.Fouché et al. [3] proposed the incremental covering array, and gave a method to generate it. The method can be deemed

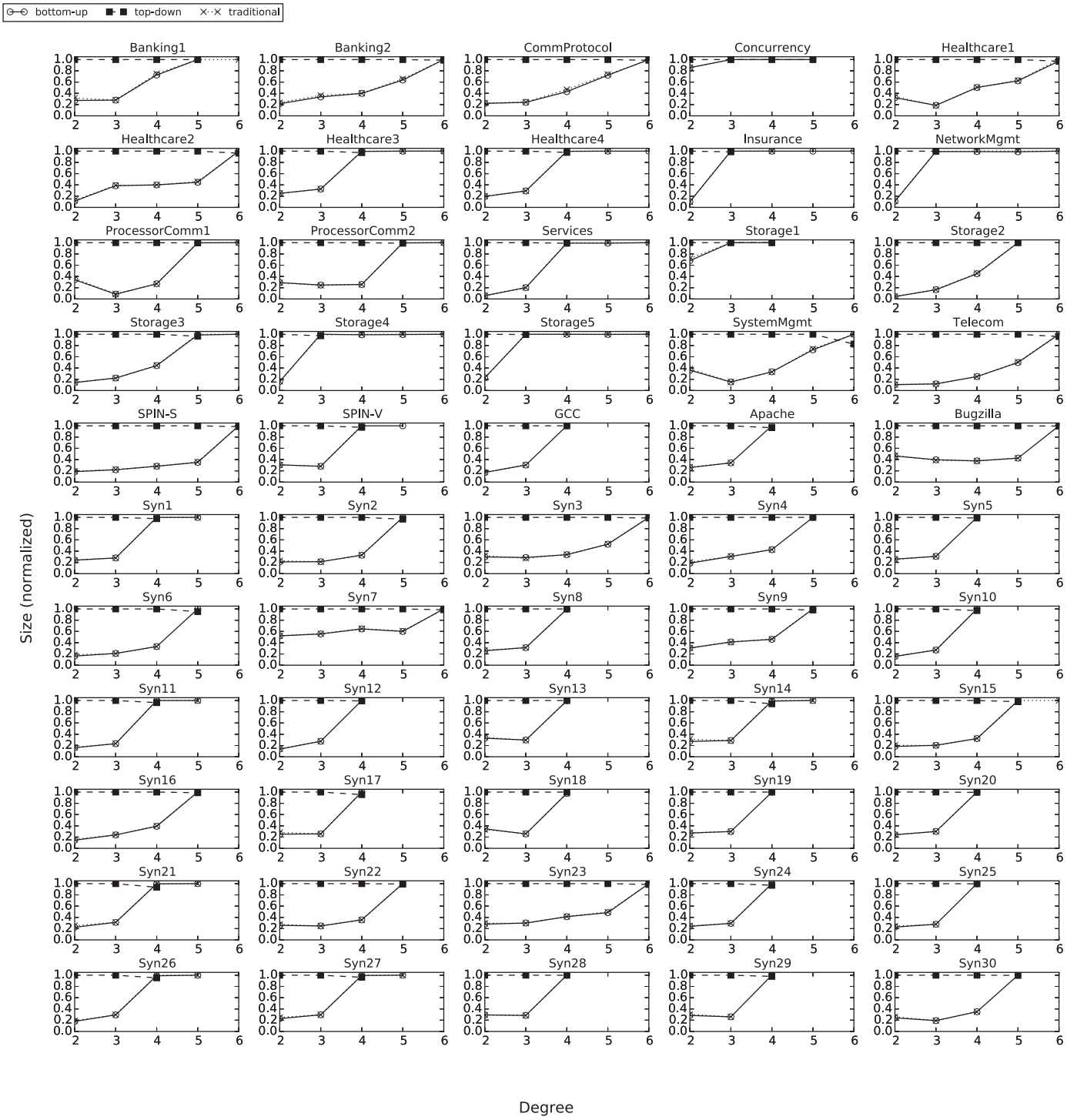

Fig. 3: The comparison of the three strategies with respect to the size of covering arrays

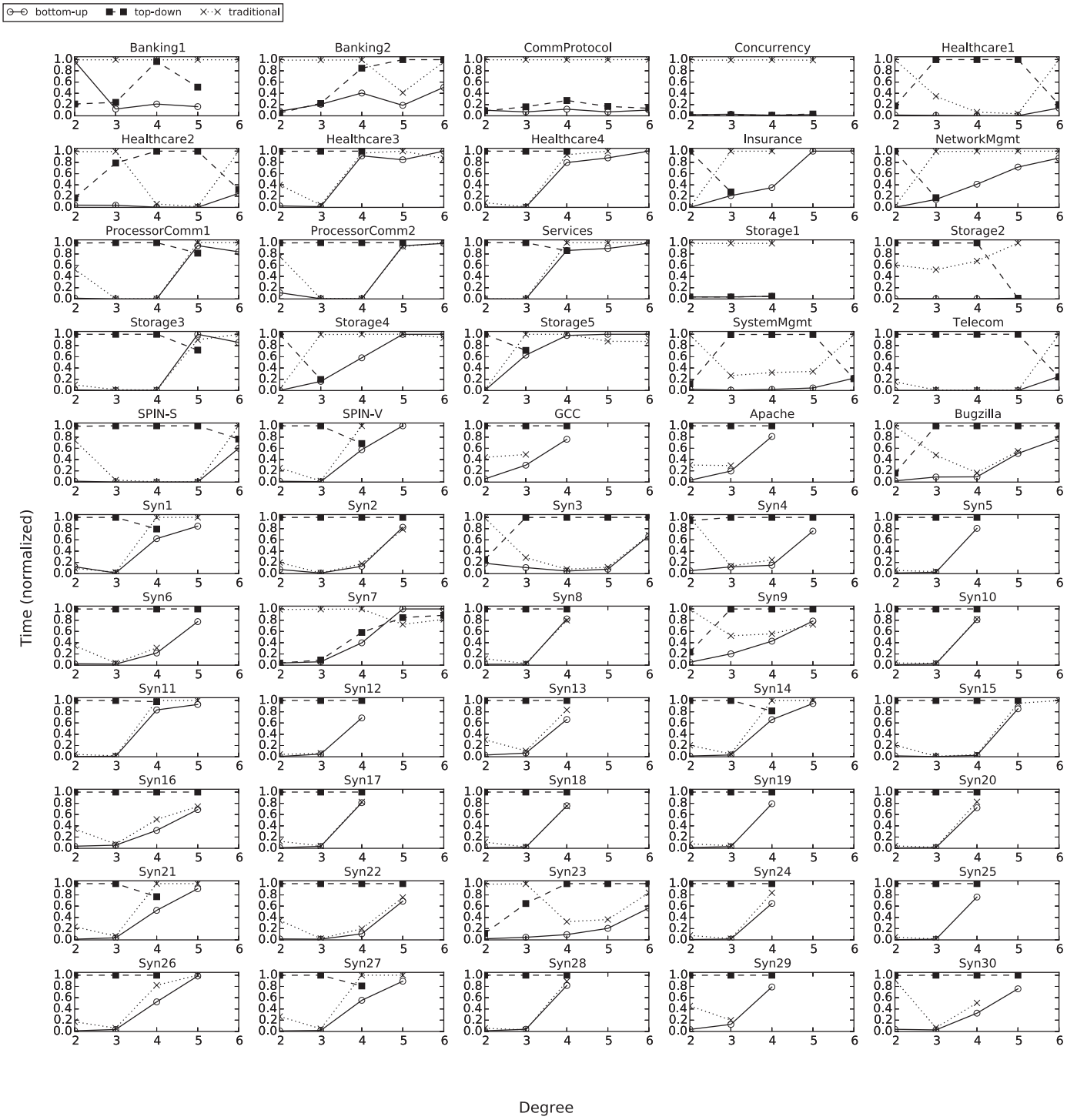

Fig. 4: The comparison of the three strategies with respect to the time cost

as a special case of *bottom-up* strategy, with the only difference being that it used multiple lower-strength covering arrays instead only one in this paper to construct the higher-strength covering array. This is because their work needs to characterize the failure-inducing interactions in the covering array, in which ,multiple covering arrays can support a better diagnosis.

Calvagna et al. [29] proposed another work to generate higher-way in an iterative way. Different than our work, its main task to generate a covering array with a specific given degree. It builds a relationship between higher-way covering array with lower-covering array. Based on this, it can reduce the higher-way covering array generation problem to multiple lower covering array generation problem.

VIII. Conclusions and future works

Incremental covering array is an important object to make CT used in practice, because it makes CT more adaptive, such that we do not need to give the specific test strength in prior. This paper formally defined the incremental covering array and gave a proof to its existence. Furthermore, this paper proposed two strategies for generating incremental covering arrays. Experimental results show that strategy *bottom-up* have a significant advantage over *top-down* at the size of the covering arrays and time cost. It is also performs more efficient than traditional approach with respect to the time cost.

As a future work, we will apply more covering array generation algorithms, to compare their performance at generating incremental covering arrays. Another interesting work is to combine the two strategies, so that we can first select a median coverage strength t and generate incremental covering arrays by *top-down* strategy. Then if the maximal-way covering array is generated, we can use *bottom-up* strategy to generate further higher-strength covering arrays. We believe such a combination strategy may offer a better performance.

References

[1] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[2] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*. IEEE, 2002, pp. 91–95.

[3] S. Fouché, M. B. Cohen, and A. Porter, "Incremental covering array failure characterization in large configuration spaces," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 177–188.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *Software Engineering, IEEE Transactions on*, vol. 23, no. 7, pp. 437–444, 1997.

[5] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog: A general strategy for t-way software testing," in *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*. IEEE, 2007, pp. 549–556.

[6] ——, "Ipog/ipog-d: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[7] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 242–251.

[8] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Combinatorial test generation for software product lines using minimum invalid tuples," in *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*. IEEE, 2014, pp. 65–72.

[9] ——, "Constraint handling in combinatorial test generation using forbidden tuples," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 2015, pp. 1–9.

[10] I. Segall, R. Tzoref-Brill, and E. Farchi, "Using binary decision diagrams for combinatorial test design," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 254–264.

[11] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 540–550.

[12] E.-H. Choi, C. Artho, T. Kitamura, O. Mizuno, and A. Yamada, "Distance-integrated combinatorial testing," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE, 2016, pp. 93–104.

[13] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 129–139.

[14] ——, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *Software Engineering, IEEE Transactions on*, vol. 34, no. 5, pp. 633–650, 2008.

[15] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA*. IEEE, 2006, pp. 153–158.

[16] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *Search Based Software Engineering, 2009 1st International Symposium on*. IEEE, 2009, pp. 13–22.

[17] ——, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.

[18] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang, "Tca: An efficient two-mode meta-heuristic algorithm for combinatorial test generation (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 494–505.

[19] X. Li, Z. Dong, H. Wu, C. Nie, and K.-Y. Cai, "Refining a randomized post-optimization method for covering arrays," in *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 143–152.

[20] P. Nayeri, C. J. Colbourn, and G. Konjevod, "Randomized post-optimization of covering arrays," *European Journal of Combinatorics*, vol. 34, no. 1, pp. 91–103, 2013.

[21] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," *NIST Special Publication*, vol. 800, p. 142, 2010.

[22] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on*, vol. 32, no. 1, pp. 20–34, 2006.

[23] M. B. Cohen, J. Snyder, and G. Rothermel, "Testing across configurations: implications for combinatorial testing," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 6, pp. 1–9, 2006.

[24] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 75–86.

[25] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn, "Combinatorial testing of acts: A case study," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 591–600.

[26] L. S. G. Ghandehari, M. N. Bourazjany, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Applying combinatorial testing to the siemens suite," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 362–371.

[27] M. B. Cohen, M. B. Dwyer, and J. Shi, "Exploiting constraint solving history to construct interaction test suites," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007.* IEEE, 2007, pp. 121–132.

[28] C. Nie, H. Leung, and K.-Y. Cai, "Adaptive combinatorial testing," in *Quality Software (QSIC), 2013 13th International Conference on*. IEEE, 2013, pp. 284–287.

[29] A. Calvagna and E. Tramontana, "Incrementally applicable t-wise combinatorial test suites for high-strength interaction testing," in *Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual*. IEEE, 2013, pp. 77–82.