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

Xintao Niu*, Changhai Nie*, Hareton Leung†, Jeff Y. Lei‡, Xiaoyin Wang§, Yan Wang ¶, Mengfan Zeng*, and Jiaxi Xu ¶
*State Key Laboratory for Novel Software Technology
Nanjing University, China, 210023
Email: niuxintao@gmail.com, changhainie@nju.edu.cn, zengmengfan@foxmail.com
†Department of Computing Hong Kong Polytechnic University
Kowloon, Hong Kong
Email: cshleung@comp.polyu.edu.hk
‡ Dpartment of Computer Science and Engineering
The University of Texas at Arlington
Email: ylei@cse.uta.edu
§ Department of Computer Science
University of Texas at San Antonio
Email: Xiaoyin.Wang@utsa.edu
¶ School of Information Engineering
Nanjing Xiaozhuang University
Email: wangyan@njxzc.edu.cn, xujiaxi@njxzc.edu.cn

***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 for generating incremental covering arrays which can increase the coverage strength when required. A preliminary evaluation of the two strategies is presented, which shows that both strategies have their own advantages with respect to covering array size.**

***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 ar-

rays, 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. The results of the experiments showed that the *top-down* strategy has an significant advantage at the size of the higher-strength covering array, while the *bottom-up* strategy is better at constructing those lower-strength covering arrays with smaller size. Based on this observation, we applied our two strategies on testing an industrial software (Tomcat). The results of this empirical study showed that our approach generates smaller size of test cases than traditional approach, while keeps a high rate of fault detection.

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 $ICA([5, 9]; [2, 3], 4, 2)$.

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.

TABLE I: Experiment of Incremental covering array

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

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.

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).

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

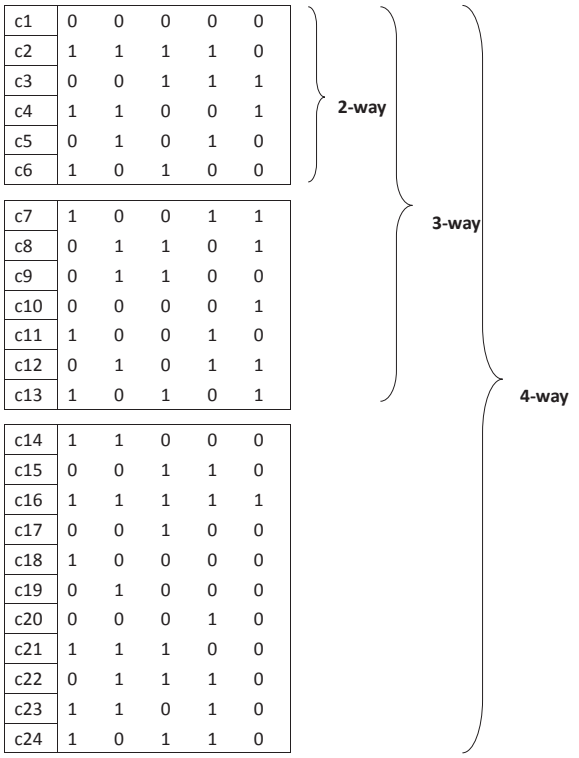

Fig. 1: Bottom-up strategy example

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

the *bottom-up* strategy with respect to the size of the 4-way covering array (16 for *top-down* and 24 for *bottom-up*). But when considering the lower-strength covering arrays, the *bottom-up* strategy performs better (13 and 6 for *bottom-up*, while 14 and 8 for *top-down* respectively). To evaluate the generality of this observation , we conduct experiments in the next section.

IV. Implementation of the approach

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

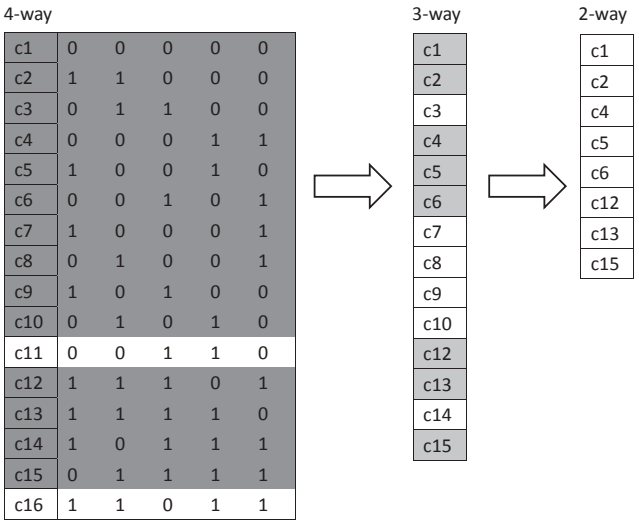

Fig. 2: Top-down strategy example

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 of its efficiency, 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 experiment section, 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 technique

Seeding test cases is the key part for the bottom-up strategy, which regards the lower-way covering array as the seeds of the higher-way covering array. Specifically, we need to eliminated those higher-degree tuples that have already been covered by the lower-way covering array. Then we just need to continue the test case generation approach to cover the remaining uncovered higher-degree tuples.

D. Extract lower-way covering array

This is the key part for the top-down strategy. The extraction process is a greedy approach in this paper. For any t_{i+1}

covering array, at each iteration, we only select the test case which can cover most uncovered t_i-degree interactions from the t_{i+1}-way covering array. The selection process repeats until we obtain a complete t_i-way covering array. Note that this greedy selection does not promise to obtain the CA_i covering array with the minimal size. But to get the minimal size, we need to exhaustively check every possible subset of the higher-covering array CA_{i+1}. This is impractical if the size of CA_{i+1} is too large.

V. Experiments and Results

A. Research Questions

We set up the following four research questions to investigate the effectiveness and the efficiency of our approach.

RQ1. Compared with the traditional t-way test generation, can DICOT deliver higher t0-way coverage with higher interaction strength t0(¿ t)? If so, how big are the improvement and computational overhead?

RQ2. Compared with the Enumerating Choice (EC) approach, how eective and ecient is DICOT w. r. t. t0(¿ t)-way coverage and computational overhead?

RQ3. Compared with the DT approach, how eective and ecient is DICOT w. r. t. the t-way test suite size and t0(¿ t)-way coverage?

RQ4. How different is the performance when using Hamming distance and chi-square distance as a distance metric in DICOT?DICOT employs another appr This section describes the experiments.

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*), come from a recent benchmark created by Segall et al [10], which have already been used in several works [11], [12]. There 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.) with 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. The last Column *ICA* shows the specific incremental covering array that needs to be generated for each model. For most subjects, we will generate an incremental covering array from 2-way to 6-way (2-6), but for some particular subjects with extremely large input space, we decide to limit the degree of the covering array up to 5 or 4 (2-5, or 2-4).

TABLE II: The models of the SUT

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

C. Experiment set-up

Then for each SUT, we generate a incremental covering arrays, shown in Table **??**. which are $ICA([N_2, N_3]; [2, 3], k, v)$, ICA $([N_2, N_3, N_4]; [2, 3, 4], k, v)$, and $ICA([N_2, N_3, N_4, N_5]$ $; [2, 3, 4, 5], k, v)$, respectively. Each incremental covering array will be repeatedly generated 30 times by two strategies. Finally, we will compare their average sizes. The results are shown in Table III.

We run them for two weeks for all these subjects. Our machine is a .

D. Results and Discussion

We consider three metrics for the experiments.

The coverage and fault rfd is .

From this point of view.

For each subject in this table, we list the results of the two strategies for the three incremental covering arrays. In detail, for each incremental covering array, we list the average size of the covering array and the corresponding standard deviation of the 30 repeated experiments. For example, consider the result of $ICA([N_2, N_3]; [2, 3], k, v)$ for SUT_1. There are two columns, representing the results of 2-way covering array and 3-way covering array, respectively. For each cell in this table, the result is shown in the form *' average size / standard deviation'*.

One observation is that the result is relatively stable according to the small standard deviation against the average value. In fact, the deviation is about 1 to 2 for the 2-way covering array, 1 to 3 for the 3-way covering array, 1 to 7 for 4-way, and 6 to 15 for the 5-way. The stability of the results is owe to the covering array generation algorithm AETG, by which the size of the covering array is similar for different runs.

With respect to comparison of the size of incremental covering array for the two strategies, one observation is that *bottom-up* strategy obtained smaller size of the lower-strength covering array in the incremental covering array, while *top-down* strategy achieved smaller size of the higher-strength covering array. To make the observation more clear, we depict the average sizes of the covering arrays by the two strategies in Fig3.

In Fig.3, there are three main rows, representing the results of three incremental covering arrays , $ICA([N_2, N_3]; [2, 3], k, v)$, ICA $([N_2, N_3, N_4]; [2, 3, 4], k, v)$, and $ICA([N_2, N_3, N_4, N_5]$ $; [2, 3, 4, 5], k, v)$, respectively. The nine columns represents the nine SUTs in Table II. For each sub-figure in Fig.3, the horizontal axis depicts the results of covering arrays of different strengths, and the vertical axis represents the size of the covering array. Note that we do not directly show the value of the size; instead, we normalize them so that they can be put into one figure. In detail, for each covering array of a subject, the point in the figure for one strategy indicates the proportion of the average size obtained by this strategy and the larger one of the two strategies.

From Fig.3, we can observe that, for most cases, the *top-down* strategy obtained smaller higher-strength covering arrays (about 90 % of that of *bottom-up*), while the *bottom-up* strategy performed better at the lower-strength covering arrays. This conclusion coincides with the case study presented in Section 3. There are also some exceptions; for example in the second row in Fig. 3 ($ICA([N_2, N_3, N_4]; [2, 4], k, v)$), the *top-down* strategy generated smaller 2-way covering arrays for SUT_2, SUT_4, and SUT_6 than that of *bottom-up*. One possible explanation for the exception is that the covering array generated by greedy approach AETG sometimes may produce more test cases than needed.

Above all, the preliminary results suggested that when the coverage strength of the final covering array is low, the *bottom-up* strategy is preferred; otherwise, the *top-down* strategy is a better choice.

Besides this, it is noted that the *top down* strategy may generate larger lower strength arrays which are definitely to be executed while the *bottom up* strategy may generate larger higher strength arrays which may not be executed. This usually happens when the current software under testing is expired for the releasing of the next version. In that case, the *bottom-up* strategy is recommended.

VI. Threats to Validity

In this section, we will co

VII. Related work

Combinatorial testing has been proven to a effective testing technique in practice [19], especially on domains like configuration testing [20], [21], [22] and software inputs testing [4], [23], [24]. 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. [25], [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.[26] 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 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. [27] proposed another work

TABLE III: Experiment results

		$ICA([2,3])$ (avg/stdev)		ICA $([2,3,4])$ (avg/stdev)			$ICA([2,3,4,5])$ (avg/stdev)			
SUT_1	Bottom-up	26.2/0.91	126.07/2.16	26.73/1.12	127.13/2.38	519.23/3.88	26.8/0.95	125.9/1.62	519.9/0.95	1906.17/8.01
	Top-down	30.87/1.06	123.53/2.46	30.77/0.8	137.93/1.95	509.23/0.8	30.17/1.04	139.13/1.93	553.07/1.04	1865.5/7.75
SUT_1	Bottom-up	85.53/1.98	226.23/3.95	85.87/2.03	227.1/4.78	703/7.51	86.1/1.76	227.8/4.28	704.07/1.76	1466.1/13.88
	Top-down	90.03/0.75	167.87/3.3	84.93/1.88	244.47/2.26	661.93/1.88	84.1/1.58	246.63/2.39	610.27/1.58	1340.83/12.63
SUT_3	Bottom-up	17.27/0.96	54.13/1.71	17.03/0.98	54.97/1.97	138.37/3.02	16.97/0.87	55.13/1.89	138.93/0.87	302.23/5.04
	Top-down	19.03/0.98	53.5/2.47	19.23/0.8	55.83/1.75	135.67/0.8	18.53/0.92	56.7/2.12	136.2/0.92	291.33/3.47
SUT_4	Bottom-up	12.23/1.05	32.97/1.2	12.8/1.22	33.27/1.44	87.1/4.37	12.67/1.11	33.13/1.65	87.5/1.11	212.13/7.91
	Top-down	12.47/0.72	31.43/1.58	12.53/0.76	33.2/1.22	83.33/0.76	12.87/1.02	34.4/1.11	83.4/1.02	200.13/8.2
SUT_5	Bottom-up	21.9/1.47	85.4/2.12	21.67/1.11	85.73/1.93	307.9/3.46	21.73/1.34	85.63/2.17	306.7/1.34	984.1/6.39
	Top-down	23.43/0.96	84.97/1.82	23.93/0.85	91.1/1.83	300.87/0.85	23.73/0.77	92.9/1.87	318.87/0.77	962.37/6.87
SUT_6	Bottom-up	85.9/1.72	228.27/3.63	86.5/2.01	226.97/3.7	701.57/7.93	86.1/1.49	226.23/3.78	700.6/1.49	1457.73/14.43
	Top-down	90.17/0.58	167.13/2.2	85.5/2.14	244.33/2.05	663.53/2.14	83.8/1.7	245.87/2.95	612.07/1.7	1341.23/15.25
SUT_7	Bottom-up	29.53/2.12	100.33/2.57	29.33/2.33	100.67/2.71	336.8/7.07	29.27/2.06	100.43/3.19	334.53/2.06	953.97/15.52
	Top-down	29/1.51	96.13/3.26	29.07/1.59	103/1.67	324.87/1.59	28.6/1.45	103.93/2.24	312.97/1.45	896.07/13.89
SUT_8	Bottom-up	21.13/1.12	76.83/4.45	21.2/0.95	75.23/3.4	218.27/4.84	21.17/1.29	75.7/3.56	216.23/1.29	557.53/9.34
	Top-down	21.43/1.05	75.17/3.66	21.57/1.12	76.23/2.26	203.67/1.12	21.43/0.96	76.2/2.41	211.37/0.96	516.93/7.78
SUT_9	Bottom-up	17/0.89	55.03/1.72	17/0.89	55.13/1.02	167.1/4.22	16.6/0.71	54.6/1.76	164.83/0.71	444.1/9.64
	Top-down	17.6/0.92	53.13/1.82	18.1/0.87	56.17/1.63	160.67/0.87	17.8/0.79	57.33/1.51	160.03/0.79	418.1/12.46

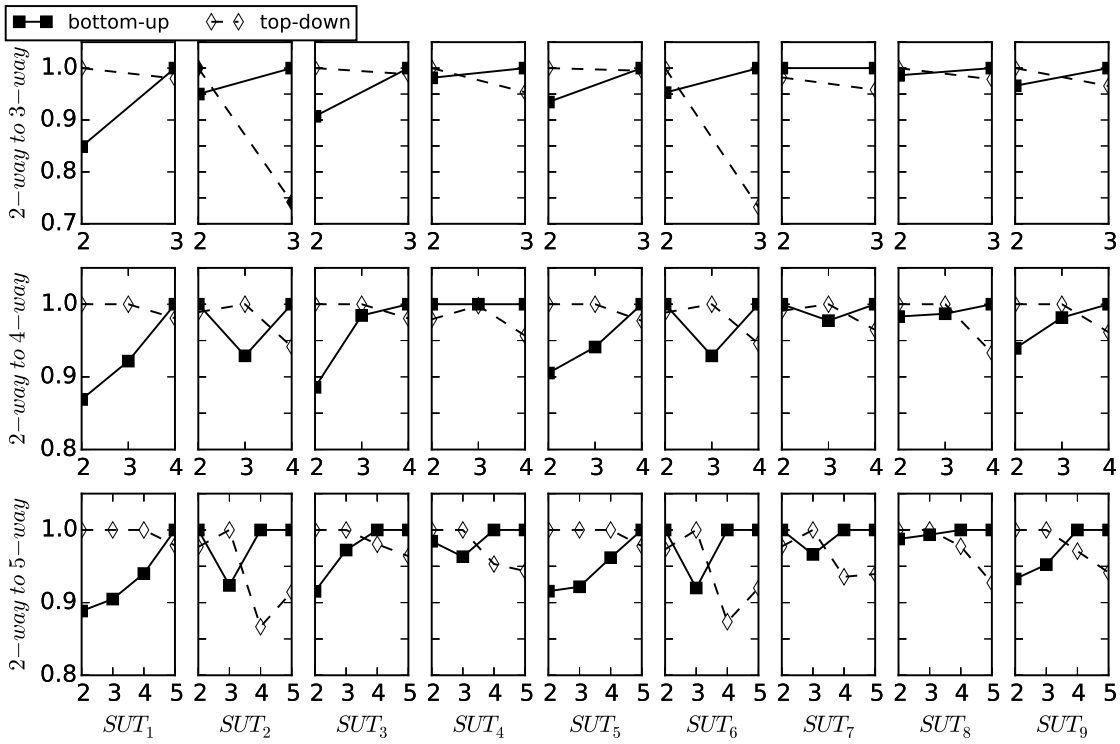

Fig. 3: The comparison of the two strategies

VIII. Conclusions and future works

This paper proposed two strategies for generating incremental covering arrays. Experimental results show that both strategies have their own advantages; *top-down* strategy is better at generating higher-strength covering arrays, while *bottom up* strategy performs better at lower-strength ones .

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.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61272079), the Research Fund for the Doctoral Program of Higher Education of China (No.20130091110032), the Science Fund for Creative Research Groups of the National Natural Science Foundation

of China(No. 61321491), and the Major Program of National Natural Science Foundation of China (No. 91318301)

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.