

# Two strategies for generating incremental covering arrays

Xintao Niu and Changhai nie  
State Key Laboratory for Novel  
Software Technology  
Nanjing University, China, 210023  
Email: changhainie@nju.edu.com

Hareton Leung  
Department of computing  
Hong Kong Polytechnic University  
Kowloon, Hong Kong  
Email: cshleung@comp.polyu.edu.hk

Jeff Y. Lei  
Department of Computer Science  
and Engineering  
The University of Texas at Arlington  
Email: ylel@cse.uta.edu

**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 interactions 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, such  $t$  cannot be properly determined, especially for systems with complex interaction space. Hence, incremental 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 showed that both strategies have their own advantages with respect to covering array size

## 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. 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 software system. It is, however, not impractical due to the combinatorial explosion. Therefore, to select a group of representative test cases from the whole testing space is required.

Combinatorial testing has been proven to be effective in identifying a good sample [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 are proposed to generate covering arrays. Despite many differences between them, those works all assume that the degree  $t$  is given a 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, such a  $t$  is difficult to determine in practice. There are two reasons for this. 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 too large than required, many redundant test cases will be generated, which is a waste of computing resource. And if  $t$  is estimated to be too small, 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 [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 the faults can be covered by 6-way covering arrays. As incremental covering arrays first cover those lower-degree interactions, then the faults caused by them will be detected sooner.

In consideration of the size of the overall test cases, we argue that this approach of generating incremental covering array may produce too many test cases. This can be easily understood, 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 ? Before answering this question, one obvious conclusion to note is that any high way covering array must cover all the lower way interactions. So the answer for the previous question is *yes*, as we just need 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.

Our contributions include:

- 1) We propose two strategies for generating incremental covering arrays.
- 2) We conduct a series of experiments to compare and evaluate these two strategies.
- 3) We offer a recommendation 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.,  $a_i = |V_i|$  ( $i = 1, 2, \dots, k$ ). Then a *test case* of the SUT is a group of values that are assigned to each parameter, which can be denoted as  $(v_1, v_2, \dots, v_k)$ . An  $t$ -degree interaction can be formally denoted as  $(-, \dots, v_{n_1}, -, \dots, v_{n_2}, -, \dots, v_{n_t}, -, \dots)$ , 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  $MCA(N; t, k, (a_1, a_2, \dots, a_k))$  is 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  $a_1 = a_2 = \dots = a_k = v$ ,  $t$ -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 binary 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  $ICA([N_{t_1}, N_{t_1+1}, \dots, N_{t_2}]; [t_1, t_2], k, (a_1, a_2, \dots, a_k))$  is also a test set in the form of  $N_{t_2} \times k$  table, where  $t_1 < t_2$  and  $N_{t_1} < N_{t_1+1} < \dots < N_{t_2}$ . In this table, the first  $N_{t_1}$  lines ( $t_1 \leq t_i \leq t_2$ ) is a covering array  $MCA(N_{t_1}; t_1 + i, k, (a_1, a_2, \dots, a_k))$ .

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

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  $MCA(N_{t_2}; t_2, k, (a_1, a_2, \dots, a_k))$ , we can find an  $ICA([N_{t_1}, N_{t_1+1}, \dots, N_{t_2}]; [t_1, t_2], k, (a_1, a_2, \dots, a_k))$ , s.t., the test cases of them are the same.

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

First,  $MCA(N_t; t, k, (a_1, a_2, \dots, a_k))$  itself must be an  $MCA(N_t; t-1, k, (a_1, a_2, \dots, a_k))$ , 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  $MCA(N_t; t, k, (a_1, a_2, \dots, a_k))$  can not be reduced. By 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  $(v_1, v_2, \dots, v_k)$  cover the  $(t-1)$ -degree interaction  $(-, -, v_3, -, \dots, v_k)$  which only appears in the test case. Then obviously the  $t$ -degree interaction  $(v'_1, -, v_3, -, \dots, v_k)$  ( $v'_1 \neq v_1$ ) will never be covered by any test case in  $MCA(N_t; t, k, (a_1, a_2, \dots, a_k))$ , and hence  $MCA(N_t; t, k, (a_1, a_2, \dots, a_k))$  is not a  $t$ -way covering array (Note this is based on that the parameter can take more than one value).

It is contradiction, and means that we can reduce at least one test case in  $MCA(N_t; t, k, (a_1, a_2, \dots, a_k))$ , 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. By this, 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 previous executed test cases will reduce cost 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 1 Bottom-up strategy

---

**Input:** *Params*  $\triangleright$  Parameters (and their values)  
 $t_1$   $\triangleright$  the lowest strength  
 $t_2$   $\triangleright$  the highest strength  
**Output:** *ICA*  $\triangleright$  the incremental covering arrays

```

1: ICA  $\leftarrow$  EmptySet
2: for  $t_i = t_1$ ;  $t_i \leq t_2$ ;  $t_i++$  do
3:   CAi  $\leftarrow$  EmptySet
4:   if  $t_i == t_1$  then
5:     CAi  $\leftarrow$  CA_Gen(Params,  $t_i$ )
6:   else
7:     CAi  $\leftarrow$  extend(Params,  $t_i$ , CAi-1)
8:   end if
9:   ICA.append(CAi)
10: end for
11: return ICA

```

---

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*.

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 utilize 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<sub>i-1</sub>* 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

c1	0	0	0	0	0
c2	1	1	1	1	0
c3	0	0	1	1	1
c4	1	1	0	0	1
c5	0	1	0	1	0
c6	1	0	1	0	0
c7	1	0	0	1	1
c8	0	1	1	0	1
c9	0	1	1	0	0
c10	0	0	0	0	1
c11	1	0	0	1	0
c12	0	1	0	1	1
c13	1	0	1	0	1
c14	1	1	0	0	0
c15	0	0	1	1	0
c16	1	1	1	1	1
c17	0	0	1	0	0
c18	1	0	0	0	0
c19	0	1	0	0	0
c20	0	0	0	1	0
c21	1	1	1	0	0
c22	0	1	1	1	0
c23	1	1	0	1	0
c24	1	0	1	1	0

Fig. 1: Bottom-up strategy example

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) against the previous 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<sub>i+1</sub>*) (line 6 - 7). The extraction process is a greedy approach in this paper. At each iteration, the test case which can cover most number of uncovered  $t$ -degree interactions will be selected from the higher-strength covering array. Note that this greedy selection does not promise to obtain the *CA<sub>i</sub>* covering array with the minimal size. But to get the minimal size, we need to exhaustive check every possible subset of the higher-covering array *CA<sub>i+1</sub>*. This is impractical if the size of *CA<sub>i+1</sub>* is too large.

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.

## Strategy 2 Top-down strategy

**Input:**  $Params$   $\triangleright$  Parameters (and their values)  
 $t_1$   $\triangleright$  the lowest strength  
 $t_2$   $\triangleright$  the highest strength  
**Output:**  $ICA$   $\triangleright$  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  $ICA$ 

```

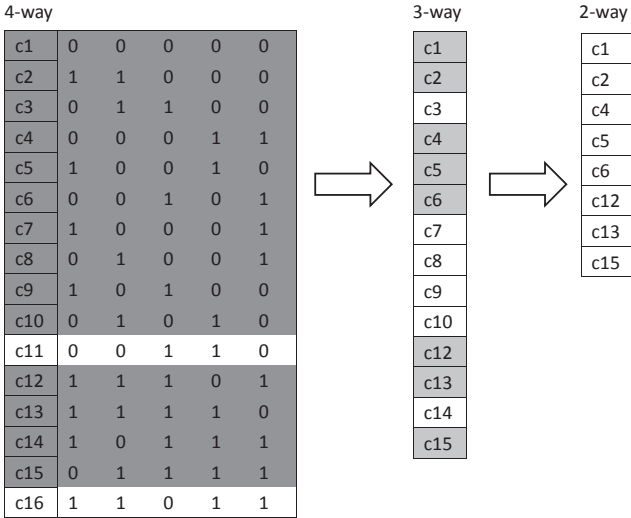


Fig. 2: Top-down strategy example

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 for *top-down* and 24 for *bottom-up*). But when considering the lower-strength covering arrays, the *bottom-up* performed 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. PRELIMINARY EVALUATION

This section describes the experiments. We have randomly prepared 9 SUT with their parameters as shown in Table II. The parameters are presented in the abbreviated form  $\#values\#number\ of\ parameters$  ..., e.g.,  $7^36^2$  indicates the software has 3 parameters that can take 7 values and 2 parameters take 6 values. We didn't choose SUT with many parameter values because we will generate covering arrays with the coverage strength reaching to 5, which is quite time-consuming for AETG algorithm.

TABLE II: The parameters of the SUT

$SUT_1(4^7)$	$SUT_2(2^{15})$	$SUT_3(2^53^25^1)$
$SUT_4(2^{10}3^2)$	$SUT_5(3^74^2)$	$SUT_6(2^89^1)$
$SUT_7(2^93^25^2)$	$SUT_8(2^84^3)$	$SUT_9(2^83^34^1)$

Then for each SUT, we generate 4 incremental covering arrays, 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, and we will compare their average sizes. The results are shown in Table III.

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, see 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 vertical axis represents size of the covering array. Note that we did 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 maximal one between the two strategies.

From Fig.3, we can observe that for most cases, *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

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
	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
$SUT_1$	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
	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
$SUT_3$	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
	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
$SUT_4$	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
	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
$SUT_5$	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
	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
$SUT_6$	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
	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
$SUT_7$	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
	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
$SUT_8$	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
	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
$SUT_9$	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
	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

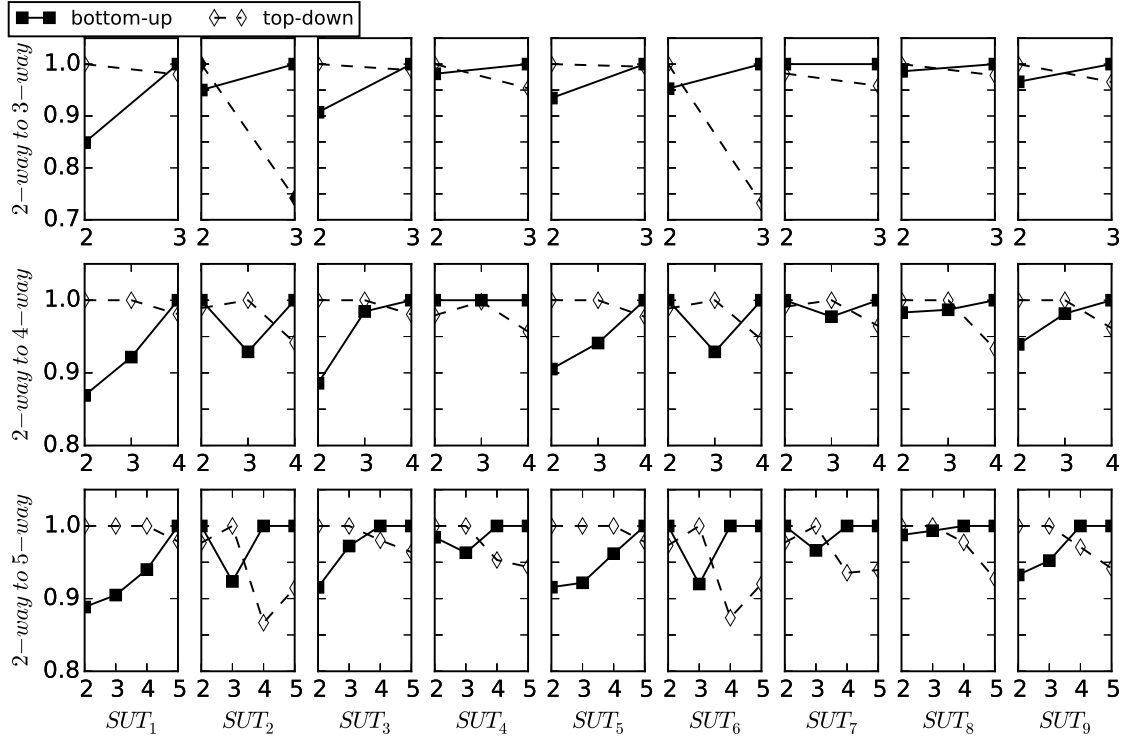


Fig. 3: The comparison of the two strategies

the second row in Fig. 3 ( $ICA([N_2, N_3, N_4]; [2, 4], k, v)$ ), *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, *bottom-up* strategy is preferred, otherwise, *top-down* 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 is usually happened when the currently software under testing is expired for the releasing of the next version. In that case, *bottom-up* strategy is recommended.

## V. RELATED WORK

Nie et al. [1] gave a survey for combinatorial testing, in which the methods for generating covering arrays are classified. Further Nie et al.[5] proposed a model for adaptive CT, in which the coverage strength of covering array needs to 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 one special case of *bottom-up* strategy, the only difference is 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 needed to characterize the failure-inducing interactions in the covering array, in which ,multiple covering arrays can support a better diagnosis information.

## VI. CONCLUSIONS AND FUTURE WORKS

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