Generating strategies for incremental covering array

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

***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., covering array, CT ensures to check every possible validate interactions in SUT. Most existing covering array generating algorithms require to be given a *degree t* in prior, such that only the interactions with the number of factors no more than t are needed to be checked. In practice, however, such t cannot be properly determined, especially for systems with complicated interactions space. Hence, adaptively generating covering arrays is preferred. In this paper, we proposed two strategies for generating covering arrays which can increase the coverage criteria when required. An preliminary evaluation of the two strategies is given, which showed that, in consideration of the size of the covering array, both the two strategies have their own advantages.**

I. Introduction

With the growing complexity and scale of modern software systems, various factors, e.g., input values and configure options, can affect the behaviour of the system. Worse more, 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 take exhaustive testing for all the possible validate 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 to deal with this sampling work [1]. It works by generating a relative small set of test cases, i.e., covering array, to test a group of particular interactions in the 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. Apart from many differences between them, those works have a common condition, i.e., they all need to be given a the degree t in prior. The degree t indicated the largest number of factors involved in the interactions they need to cover, and the corresponding covering arrays which can satisfy this coverage criteria is called t-way covering arrays. In practice, however, such a t can not be properly determined. There are two reasons for this. First, many software systems suffer from complicated interactions space which make it challenging to estimate the degree 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. Second, even though t has been properly determined, there may be no enough time to completely executing all the test cases in the covering array. This is because testing for the software only makes sense before the next version is released. This time interval between two release versions may sometimes too short for a complete testing of a high way covering array [3].

To make up for these shortcomings of traditional covering arrays, the 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 way covering array based on lower way covering array, it can reduce much more cost when comparing with generating multiple different ways of covering arrays. Additionally, it can better applied on the testing software of which the releasing time is frequently changed and cannot be predicted.

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-way covering array based on the lower-way covering array (called *bottom-up* strategy later) does not aim to optimize the size of the higher-way covering array. As a result, it may generate more test cases than those generated by approaches that focus on generating particular higher-way covering array.

Then a nature question, and also the motivation of this paper is, **is it possible to generate the incremental covering array with the number of the overall test cases the same as those by the particular high-way covering array generating 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 generating algorithm to generate the high-way covering array, and then generate the lower-way covering arrays by extracting some subset of the test cases which can cover all the lower degree interactions. Later we call this strategy *top-down*.

In this paper, we implemented this two strategies and evaluated 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 highest-way covering array in the incremental covering arrays, while for the *bottom-up* strategy, it is better at constructing those lower-way covering arrays with smaller size.

Our contributions includes:

1) We proposed two strategies for the generating incremental covering arrays.
2) We conducted a series experiments to compare and evaluate the two strategies.
3) A recommendation for selecting which strategy when generating incremental covering array in practice was given.

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) $CA(9;3,4,2)$

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

II. Background

This section gives some formal definitions in 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 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 $a_1 = a_2 = ... = a_k = v$, 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 arrays has proven to be effective to detect 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. Incremental covering arrays $ICA((t_{min}, t_{max}), k, v)$ is a sequence of covering arrays$\{CA(N_1;t_{min},k,v)$, $CA(N_2;t_{min}+1,k,v)$, ..., $CA(N_n;t_{max},k,v)$ $\}$. For any covering array ($CA(N_i,t_i,k,v)$), ($0 \le i < n$), it must have $CA(N_i;t_{min+i},k,v) \subseteq CA(N_{i+1};t_{min+i+1},k,v)$.

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

As discussed before, generating the incremental covering arrays is of importance, as it support adaptively increasing the coverage strength. By this, when testing a SUT, testers can firstly execute the lowest-way covering array in the incremental covering arrays, and then execute the additional test cases in those higher-way covering array as required. The reuse of previous executed test cases will reduce a lot amount cost against 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 starts from generating the lowest-way covering array and then the higher-way ones, which is called *bottom-up* strategy. The latter one firstly generated the highest-way covering array, then the lower-way, called *top-down* strategy.

A. Bottom-up strategy

This strategy is listed as Algorithm 1. The inputs for this algorithm consists of the values for each parameter of the SUT –$Param$, the lowest way t_{min} of the covering array in the incremental covering arrays, and the highest way t_{max} of covering array . The output of this algorithm is a incremental covering array – ICA.

Algorithm 1 Bottom-up strategy

Require: $Param$ ▷ parameter values for the SUT
t_{min} ▷ the lowest way
t_{max} ▷ the highest way
Ensure: ICA ▷ the incremental covering arrays
1: $ICA \leftarrow EmptySet$
2: **for** $t_i = t_{min};\ t_i \le t_{max};\ t_i\ INC$ **do**
3: $CA_i \leftarrow EmptySet$
4: **if** $t_i == t_{min}$ **then**
5: $CA_i \leftarrow CA\_Gen(Param, t_i)$
6: **else**
7: $CA_i \leftarrow CA\_Gen\_Seeds(Param, t_i, CA_{i-1})$
8: **end if**
9: $ICA.append(CA_i)$
10: **end for**
11: **return** ICA

This algorithm generate the covering array from lower-way to higher-way (line 2). If the current coverage strength t_i is equal to t_{min}, it just utilize an covering array generating 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

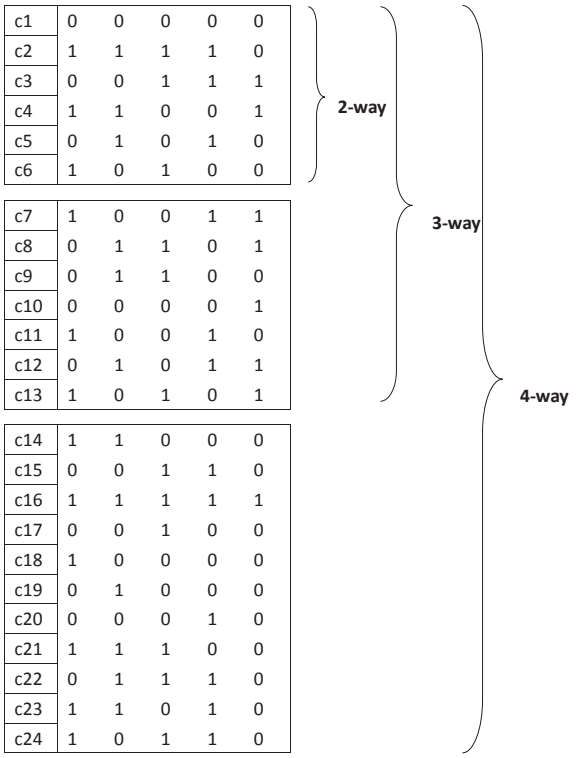

Fig. 1: Bottom-up strategy example

generation algorithm to append additional test cases to satisfy higher coverage criteria (line 6 - 7).

Fig.1 presents an example for constructing $ICA((2,4),5,2)$ by this strategy. In this example, the covering array generating 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 maximal-way covering array, there needs 24 test cases for this strategy.

B. Top-down strategy

This strategy is listed as Algorithm 2, which generate covering arrays in the opposite order (line 2) against the previous strategy. Similarly, if the current coverage strength t_i is equal to t_{max}, it directly generate the particular covering array (line 4 - 5). Otherwise, it will extract the covering array from a higher covering array (CA_{i+1}) (line 6 - 10). The extraction process is a greedy approach. At each iteration, the test case which can cover most number of uncovered t-degree interactions will be selected from the higher-way covering array (line 7 - 10). 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 exhaustive check every possible subset of the higher-covering array CA_{i+1}, which is impractical if the size of CA_{i+1} is too large.

Algorithm 2 Top-down strategy

Require: $Param$ ▷ parameter values for the SUT
t_{min} ▷ the lowest way
t_{max} ▷ the highest way
Ensure: ICA ▷ the incremental covering arrays

1: $ICA \leftarrow EmptySet$
2: **for** $t_i = t_{max};\ t_i \geq t_{min};\ t_i\ DEC$ **do**
3: $CA_i \leftarrow EmptySet$
4: **if** $t_i == t_{max}$ **then**
5: $CA_i \leftarrow CA\_Gen(Param, t_i)$
6: **else**
7: **while** t_i coverage is not satisfied **do**
8: $test\_case \leftarrow selected\_best(CA_{i+1})$
9: $CA_i.append(test\_case)$
10: **end while**
11: **end if**
12: $ICA.append(CA_i)$
13: **end for**
14: **return** ICA

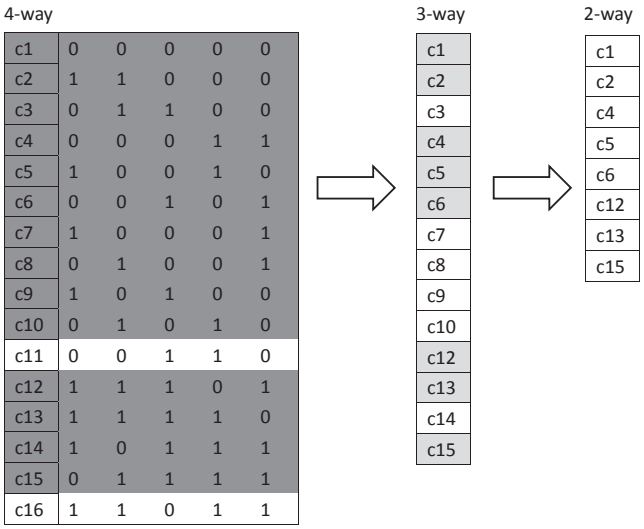

Fig. 2: Top-down strategy example

An example for this strategy is listed in Fig.2. In this example, we first generated the highest-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 $CA(8;2,5,2)$. The rows with dark background from the higher-way covering arrays represent those selected test cases.

From the two examples, an obvious observation is that for the *top-down* strategy, it has an significant advantage over the *bottom-up* strategy at the size of the highest-way covering array (16 for *top-down* and 24 for *bottom-up*). But when in consideration of the lower-way 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 the following experiments.

TABLE II: The parameters of the SUT

$SUT^1(10^7)$	$SUT^2(2^{30})$	$SUT^3(7^3 6^2 5^4))$
$SUT^4(3^{13})$	$SUT^5(15^9)$	$SUT^6(4^{10})$

IV. Preliminary Evaluation

This section describe the experiments. We have prepared 6 SUT with their parameters as shown in II. The parameters are presented in the abbreviated form $\#values^{\#number\ of\ parameters}$..., e.g., $7^3 6^2 5^4$ indicates the software has 3 parameters that can take 7 values, 2 parameters 6 values, and 4 parameter 5 values. We didn't choose SUT with large parameter values because next we will generate covering arrays with the coverage strength reaching to 6, which is time-consuming if the parameter values are too large.

Then for each SUT, we will generate 4 incremental covering arrays, which are $ICA((2,3),k,v)$, $ICA((2,4),k,v)$, $ICA((2,5),k,v)$, $ICA((2,6),k,v)$ respectively. Each incremental covering arrays will be repeatedly generated 30 times by two strategies, and we will compare their average size. The results are shown in Fig.

V. Related work

S.Fouché [3] proposed the incremental covering array, it hence use the classification tree method to classify the failure-inducing interaction, such that it should need to run multiple test cases, which is an augment of the bottom-up.

VI. Conclusions and future works

As a future work, we should apply more test cases generating algorithm, to compare their performance. Another interesting work is to combine the two strategies, that we can first, select, and then . We believe such a generating algorithm can get the two advantages of this two approaches.

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.