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

Jeff Y. Lei
Department of Computer Science and Engineering
The University of Texas at Arlington
Email: ylei@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., covering array, CT aims to check every possible validate interactions in SUT. Most existing covering array generating 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 complicated interactions space. Hence, incremental arrays are preferred. In this paper, we proposed two strategies for generating incremental covering arrays which can increase the coverage criteria when required. A preliminary evaluation of the two strategies is presented, which showed that, in consideration of the size of the covering array, both two strategies have their own advantages.**

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 the 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 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 in identifying a good sample [1]. It works by generating a relatively 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 degree t in prior. The degree t indicated 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 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 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 testing targeting fault detection. 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-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 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 two-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-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 approaches that focus on generating a particular higher-way covering array.

Then a nature 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 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 these 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 higher-way covering array, while the *bottom-up* strategy is better at constructing those lower-way covering arrays with smaller size.

Our contributions include:

1) We proposed two strategies for generating incremental covering arrays.
2) We conducted 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,..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 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 $ICA([N_{t_1}, N_{t_1+1}, ..., N_{t_2}]; [t_1, t_2]\ , k, v)$ is also a test set in the form of $N_{t_2} \times k$ table, where $t_1 < t_2$ and $N_{t_1} < N_{t_1+1} < , ..., < N_{t_2}$. In this table, the first N_{t_i} lines is a covering array $CA(N_{t_i}; t_1 + i, k, v)$.

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. Each $ICA([N_{t_1}, N_{t_1+1}, ..., N_{t_2}]; [t_1, t_2]\ , k, v)$ is a $CA(N_{t_2}; t_2, k, v)$ covering array. Correspondingly, for each covering array $CA(N_{t_2}; t_2, k, v)$, we can find an $ICA([N_{t_1}, N_{t_1+1}, ..., N_{t_2}]; [t_1, t_2]\ , k, v)$, s.t., the test cases of them are the same.

Proof: According to the definition of ICA, it can be easily got that $ICA([N_{t_1}, N_{t_1+1}, ..., N_{t_2}]; [t_1, t_2]\ , k, v)$ is a $CA(N_{t_2}; t_2, k, v)$ covering array. For the second statement, we just need to prove that for any covering array, $CA(N_t; t, k, v)$, we can find a $CA(N_{t-1}; t-1, k, v)$, such that, $N_{t-1} < N_t$ and $\forall test\ case \in CA(N_{t-1}; t-1, k, v), test\ case \in CA(N_t; t, k, v)$.

First, $CA(N_t; t, k, v)$ must be an $CA(N_t; t-1, k, v)$, 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 $CA(N_t; t-1, k, v)$ 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 $(v_1, v_2, ..., v_k)$ cover the $t-1$-degree interaction $(-, -, v_3, ..-, ..., v_k)$ which only appears in the test case. Then obviously the t-degree interaction $(v_1', -, v_3, ..-, ..., v_k)$ will never be covered by any test case in $CA(N_t; t, k, v)$, and hence $CA(N_t; t, k, v)$ 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 $CA(N_t; t-1, k, v)$, so that it is still a $t-1$-way covering array. ■

This theorem shows that 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-way covering array in the incremental covering arrays, and then execute additional test cases from those higher-way covering arrays as required.

The reuse of previous executed test cases will reduce for 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; *bottom-up* strategy starts from generating the lowest-way covering array and then the higher-way ones. The second strategy; *top-down* strategy firstly generated the highest-way covering array, then the lower-way covering array.

A. Bottom-up strategy

This strategy is listed as Algorithm 1. The inputs for this

Algorithm 1 Bottom-up strategy

Require: $Param$ ▷ parameter values for the SUT
t_1 ▷ the lowest way
t_2 ▷ the highest way
Ensure: ICA ▷ the incremental covering arrays
1: $ICA \leftarrow EmptySet$
2: **for** $t_i = t_1;\ t_i \leq t_2;\ t_i\ INC$ **do**
3: $CA_i \leftarrow EmptySet$
4: **if** $t_i == t_1$ **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

algorithm consists of the values for each parameter of the SUT –$Param$, the lowest way t_1 of the covering array in the incremental covering array, and the highest way t_2 of the covering array . The output of this algorithm is an incremental covering array – ICA.

This algorithm generates the covering array from lower-way to higher-way (line 2). If the current coverage strength t_i is equal to t_1, it just utilize a 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 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 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 4-way covering array, there needs 24 test cases for this strategy.

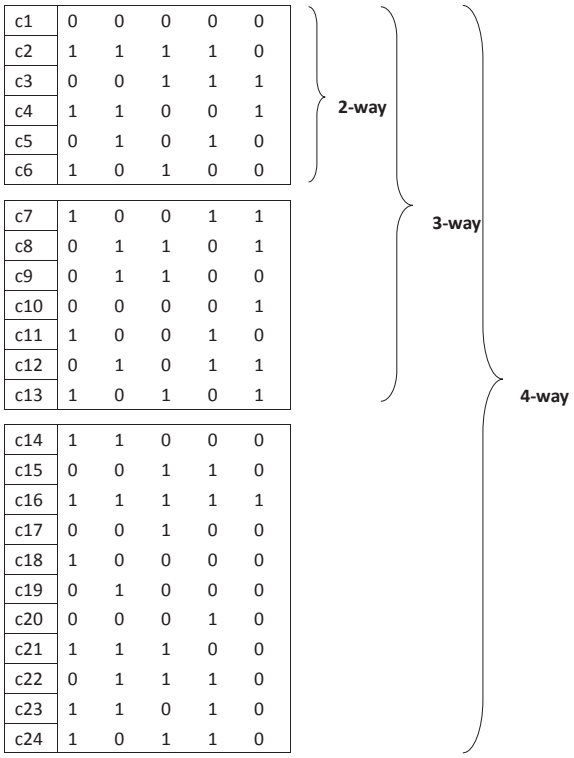

Fig. 1: Bottom-up strategy example

B. Top-down strategy

This strategy is listed as Algorithm 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_{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}. This is impractical if the size of CA_{i+1} is too large.

An example for this strategy is given 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 $CA(8; 2, 5, 2)$ is extracted from $CA(14; 3, 5, 2)$. 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 a significant advantage over the *bottom-up* strategy with respect to the size of the highest-way covering array (16 for *top-down* and 24 for *bottom-up*). But when considering the lower-way covering arrays, the *bottom-up* performed better (13 and 6 for *bottom-up*, while 14 and 8

Algorithm 2 Top-down strategy

Require: $Param$ ▷ parameter values for the SUT
t_1 ▷ the lowest way
t_2 ▷ the highest way
Ensure: ICA ▷ the incremental covering arrays

```
ICA ← EmptySet
for t_i = t_2; t_i ≥ t_1; t_i DEC do
    CA_i ← EmptySet
    if t_i == t_2 then
        CA_i ← CA_Gen(Param, t_i)
    else
        while t_i-way coverage is not satisfied do
            test ← selected_best(CA_{i+1})
            CA_i.append(test)
        end while
    end if
    ICA.append(CA_i)
end for
return ICA
```

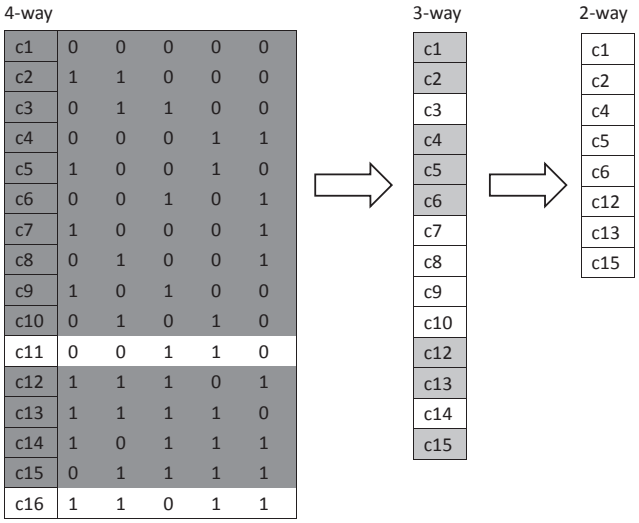

Fig. 2: Top-down strategy example

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 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., $7^3 6^2$ 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.

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 size. The results are shown in Fig.3.

TABLE II: The parameters of the SUT

$SUT_1(4^7)$	$SUT_2(2^{15})$	$SUT_3(2^5 3^2 5^1)$
$SUT_4(2^{10}3^2)$	$SUT_5(3^7 4^2)$	$SUT_6(2^8 9^1)$
$SUT_7(2^9 3^2 5^2)$	$SUT_8(2^8 4^3)$	$SUT_9(2^8 3^3 4^1)$

TABLE III: The results of the experiments

$SUT_1(4^7)$	$SUT_2(2^{15})$	$SUT_3(2^5 3^2 5^1)$
$SUT_4(2^{10}3^2)$	$SUT_5(3^7 4^2)$	$SUT_6(2^8 9^1)$
$SUT_7(2^9 3^2 5^2)$	$SUT_8(2^8 4^3)$	$SUT_9(2^8 3^3 4^1)$

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 different-way covering array, and vertical axis represents size of the covering array. Note that we did not directly shows, instead, we normalize. This is because, the gap between the size of different-way covering array is too big to put into one figure.

From Fig.3, we can observe that for most cases, *top-down* strategy obtained smaller higher-way covering arrays (about 90 % of that of *bottom-up*), while *bottom-up* strategy performed better at the lower-way 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 ($ICA([N_2, N_3, N_4]; [2, 4], k, v)$), *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.

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-way covering arrays instead only one in this paper to construct the higher-way covering array. This is because their work needed to characterize the failure-inducing interactions in the covering array, in which ,multiple

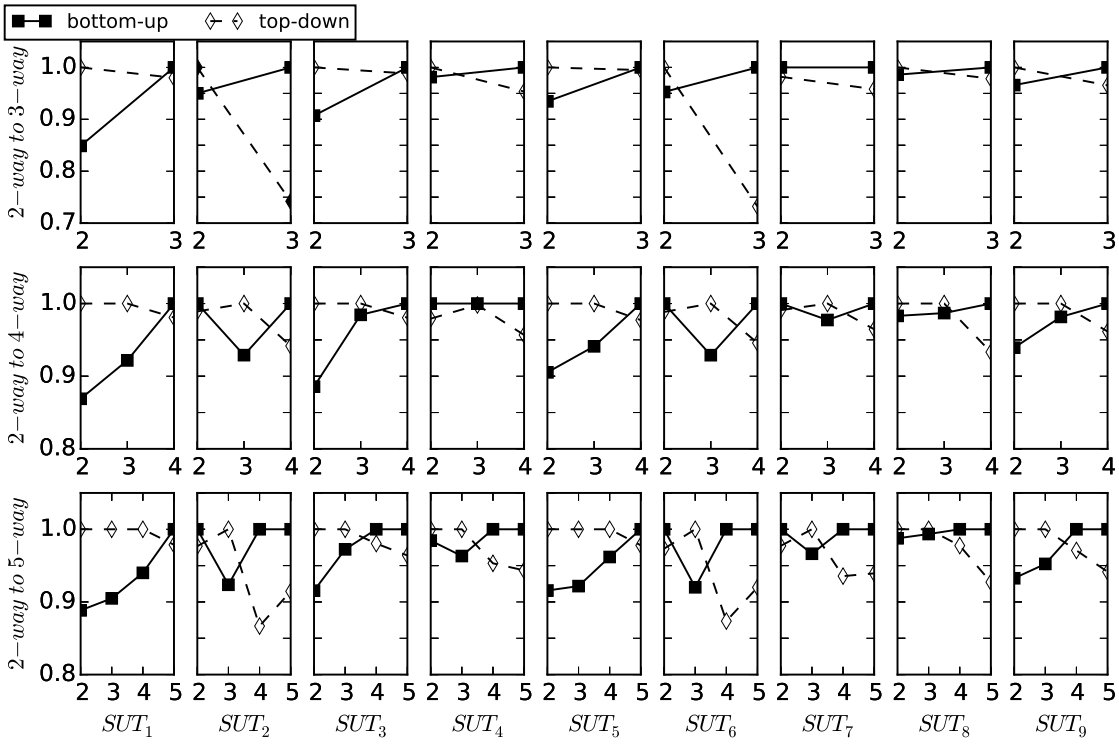

Fig. 3: Experiment results

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-way covering arrays, while *bottom up* performed better at lower-way ones .

As a future work, we will apply more covering array generating 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-way covering arrays. We believe such a combination strategies 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.