A Weight-based Approach to Combinatorial Test Generation for Higher Fault-detection Rate

Jing Zhao[1], Gao-Rong Ning [3], K-Y Cai [3], Zhiqiang Zhang [5], Jian Zhang [5]
1 Department of Computer Science and Technology, Harbin Engineering University, China, zhaoj@hrbeu.edu.cn
3 Department of Automatic Control, Beihang University, China, ninggaorong@asee.buaa.edu.cn
4 Department of Electrical and Computer Engineering, Duke University, USA, kst@duke.edu
5 The State Key Laboratory of Computer Science, Institute of Software, Chines Academy of Sciences, zhangzy@ios.ac.cn, China

***Abstract*—*Combinatorial testing* (CT) is very efficient to test parameterized systems. Kuhn et al. investigated the interaction faults of some real programs, and found that the faulty combinations are caused by the combination of no more than 6 parameters. Three or fewer parameters triggered a total of almost 90% of the failures in the application [1]. However, for high quality software, simply testing all 3-way combinations is not sufficient [2], which may increase the risk of residual errors that lead to system failures and security weakness [3]. In addition, the test cases at 100% percentage coverage for high-way are huge, which surpass farthest for the test cost restrictions. *covering array* is typically used as the test suite in CT, which should convey much information for the fault detection, We firstly proposed a weighed combinatorial coverage, focusing the fault detection capability of each test case instead of 100% percent *t*-way combinatorial coverage. Secondly, we give the test case selection algorithm using weighted combinatorial coverage metric. For generating each test case, our method first randomly generates several candidates, and selects the one that has the highest fault-detection possibility. Thirdly, we give the theorem for our algorithm and definitions for the weighted combinatorial coverage. Finally, we develop the simulation model to prove that our algorithm efficiency with varying candidate test case size.**

***Index Terms*—combinatorial testing; weighted combinatorial coverage; fault detection rate ; test case selection; simulation**

I. Introduction

Since the 21st century, computer software has been more and more involved in people's life. As software is created by humans, it may have defects. The harm brought by defective software varies from undetectable minor problems to catastrophic accidents such as economic losses, equipment failures or even life endangerment [4]–[6]. Software testing is a common way of ensuring software quality. Although software testing cannot guarantee the system under test (SUT) is error-free, a good testing practice can help detect many defects in the SUT, and provide higher confidence.

There are many software testing techniques, each of which is suits for some circumstances. *Combinatorial testing* (CT) is one of these techniques, which is very efficient to test parameterized systems. In CT, the SUT is modeled as a black-box, whose behavior is affected by several input parameters (or called parameter/configurations). Each parameter may take several possible values. From a black-box view, the failures are caused by the combination effects of parameters. These faults are called *interaction faults*. These failure-causing parameter combinations are called *faulty combinations*. Without any a priori knowledge, any parameter combination may be faulty. In the worst case, we need to test all possible parameter combinations, in which the number of test cases increases exponentially with the number of parameters. The number of *t*-way tests that will be required is proportional to $v^t \log n$, for n parameters with v values each [3], [7]. Kuhn et al. [1], [2], [8] investigated the interaction faults of some real programs, and found that the faulty combinations they studied are caused by the combination of no more than 6 parameters. Empirical investigations have concluded that from 70% to 90% of the failures could be identified by pairwise combinatorial testing [8]. This point can also be supported by other studies [9], [10]. If we have tested all small parameter combinations (e.g. all 3-way combinations), we can detect many interaction faults, three or fewer parameters triggered a total of almost 90% of the failures in the application [1]. However, for high quality software, simply testing all 3-way combinations is not sufficient [2], which may increase the risk of residual errors that lead to system failures and security weakness [3].

In CT, we typically use *covering arrays* (CA) as the test suite. Looking back the typical way of doing combinatorial testing, we give the covering strength in advance, specifying which combinations need to be covered (we call them *target combinations*), and generate a CA to cover all these combinations. Then the test generation is done. We use *t*-way combinatorial coverage as a metric to select the best coverage test case. Which coverage criterion should be chosen or determined by the cost of test and the quality requirement of the software. Suppose a Software Under Test(SUT) has 20 parameters, and each parameter has 10 values, we can check all pairs of these values with only 180 tests by using pair-wise coverage, if *N*-way coverage used then 10^{20} test cases will be used which is far more than a software tester would be test in a lifetime [11]. For finding the remaining faults, one way is to change the *t*-wise coverage criterion. if we change the coverage criterion from pair-wise to 5-wise, Almost all the faults can be found by using 100% percentage coverage. Furthermore, if we employ 6-wise combinatorial coverage, the testing process will be

forced to stop at a large SUT because the test cases at 100% percentage coverage are huge, which surpass farthest for the test cost restrictions. The problem is that we focus too much on the target combination.

In software testing, we should consider test effectiveness as well as efficiency for a SUT. And the efficient test case require that the generated test case should contribute much useful information which can be use to find faults in the SUT. we usually want to detect as many defects with as less cost as possible, and also to detect defects as early as possible. For this need, the fault-detection rate of the test suite is important. In CT, we usually use the cover array to represent a test case. Thus, a covering array should convey much information for the fault detection, and we shift our focus from t-way combinatorial coverage to the fault-detection possibility of each test case. We measured the exposed average faults by executing every test case, in which the faults will be 1-way, 2-way,..., 6-way, etc. A t-way combinatorial coverage weight can be calculated based on presumed distribution of faulty combination size. Therefore, the combinatorial coverage can be regarded as the sum of the weighted t-way coverage. Thus, Combinatorial testing focus on generating the optimized test case that has the highest fault detection rate, instead of selecting at 100% percentage combinatorial coverage.

Our approach is based on the one-test-at-a-time strategy. For generating each test case, our method first randomly generates several candidates, and selects the one that has the highest fault-detection possibility. The fault-detection ability is calculated based on the presumed distribution of faulty combination size, which could be given in advance based on one's experience and knowledge about the SUT. The traditional way of CT is a special case of our method where all faulty combinations are of size t, which requires choosing the covering strength first, if the strength is set too large that resources will be wasted or forced to stop, or if the strength is set too small that the covering array is insufficient to find the specific failures [12].

In summary, our main contributions are as follows:

- We propose the metric of weighted combinatorial coverage, which connect the combinatorial coverage and the fault detection capability of each test case.
- We propose an algorithm for the weighted combinatorial test case selection, and it is theoretically proved that our algorithm efficiency with varying candidate test case size.
- We develop a simulation model to prove that our algorithm efficiency with varying candidate test case size.

This paper is organized as follows. In section II, we give the related work for combinatorial testing. In section III, we first give a motivation example for our approach, and next we give the definition for weighted combinatorial coverage, and we prove the theorem that combinatorial coverage rate equals to the fault detection rate. In section IV, we give an algorithm for weighted combinatorial test case selection, and give the theorems for proving algorithm efficiency. In section V, we did the simulation to check the efficiency for our algorithm. In section VI, we give the conclusion for the paper.

II. Related Work

Typically, to apply CT to a system under test requires the following steps:

- Modeling: building a parameterized model for the SUT which is suitable for later test generation. The tester identifies the parameters (factors) that affects the SUT behavior according to the specification and test goal. Then s/he needs to identify the possible values for each parameter, as well as the covering requirement;
- Test generation: generating a covering array according to the SUT model and covering requirement;
- Test prioritization and reduction: postprocessing of the covering array. Test prioritization mainly focuses on reordering the test cases for early fault detection, while test reduction aims at modifying the test suite to reduce number of test cases while still preserving the coverage;
- Test execution: to run test cases on the SUT and observe the results;
- Fault localization: locating the faulty combinations that caused some test cases to fail during the test execution process.

Orthogonal arrays and covering arrays are usually popularly used as combinatorial design [13]. Zhang et al. [14] wrote a book which surveyed the major approaches to this topic. Existing methods can be classified into mathematical construction methods and computational methods.

Mathematical construction methods are usually based on mathematical theories, and they usually know what the CA should be look at in advance. However, the main drawback of these approaches is that their method is not very limited, they can hardly deal with some real testing requirements such as mixed-level covering array (parameters have different number of values), variable strength and constraints.

Different from mathematical construction methods, computational methods does now know what the covering array should be look like. They focus on the coverage requirement, and try to adjust the array to cover more and more target combinations. Computational methods can be categorized into three types of methods [15] based on how they fill entries in the array. The first type of computational methods are based on the one-test-at-a-time strategy [16]. Algorithms of this strategy generates the CA row-by-row, which means that they iteratively generate new test cases and add it into the array, until all target combinations are covered. For generating each test case, these algorithms try to find a test case that covers the greatest number of uncovered test cases. The differences between these algorithms is that they use different strategies to optimize the new test case [17] The AETG algorithm generates each new test case by first generating several candidates and then selecting the one that covers the greatest number of uncovered test cases.

For generating each candidate test case, it greedily assign values to parameters in a random order. Some other AETG-like algorithms include TCG [18] and mAETG-SAT [19], [20]. PICT [21] generates each new test case by assigning parameters one by one. Each time it greedily selects a parameter-value pair to assign. The deterministic density algorithm (DDA) generates a new test case using a density-based method, where the density is computed based on the uncovered target combinations. The method by Clavagna and Gargantini [22] generates each new test case by greedily selecting a set of consistent uncovered target combinations to assign some of the parameters, and using a model checker to find values for the remaining parameters. Cascade [23], [24] converts the problem of generating a new test case into a pseudo-Boolean algorithm, and calling an optimizations solver to find a solution. There are also some other one-test-at-a-time approach based on meta-heuristic search algorithms, such as simulated annealing, hill climbing, genetic algorithm, ant colony algorithm, tabu search, great flooding and particle swarm optimization [25]–[28].

The second type of computational methods are IPO-based algorithms [29]. Different from the one-test-at-a-time strategy, which expands the array only vertically, IPO-based algorithms expands the array both vertically and horizontally. These algorithms starts from a small covering array, and continuously extends it into a larger array. The third type of computational methods are based on the whole-array-search strategy. These algorithms try to generate a covering array of a given size, which means the number of rows and columns are fixed in advance, and they need to find an assignment to each entry to meet the coverage requirement. Hnich et al. translated the properties of a covering array into SAT constraints, and use a SAT solver to find a feasible solution. CASA [30], [31] used simulated annealing to maximize the number of covered target combinations. If all target combinations are covered, then the array is a covering array.

Like many test case selection methods, combination strategies are based on coverage [13]. Each-used coverage is the simplest coverage criterion. 100% each-used coverage requires that every interesting value of every parameter is included in at least one test case in the test suite. 100% pair-wise coverage requires that possible pair of interesting values of two parameters are included at each teat case in the test suite. A natural extension of pair-wise coverage is *t*-wise coverage, of which, A special case of *t*-wise coverage is *N*-wise coverage. Which cover array should be adopted is determined by the size and quantity expected of the SUT. For example, Grindal [13] evaluated 5 combination strategies: AC, EC, BC, OA and PW. After the cover array is determined, then the test effort is to improve the coverage. For the Heuristic pair wise testing, at each step, the test is to select that covered most pairs. The current CT approach intended to change the coverage criteria by improving the *t*-wise criteria. The size of an test set grows logarithmically in the number of test parameters [7], In practice, test engineers faced with the deadlines and expensive long period testing processes, so high wise combinatorial testing may not satisfy the testing requirements. Fochen et.al proposed an incremental covering array failure characterization in large configuration space [32], which begins at the lower strength and then iteratively increases strength as resources allow. This approach's target is to find the faults as early as possible. However, this approach did not quantitatively reveal the fault detection capability with the used *t*-way covering array,

Another way is the test prioritization process, which improve the fault detection rate at early stage by defining appropriate coverage metric. Many of the current research employ the test prioritization technique, which re-order the test suits to improve the early coverage. Test prioritization focuses on the execution order of test cases by determining the coverage metric that is used to select cover array. Some approaches reorders the test cases for early fault detection [33]–[35]. Bryce and Colbourn [36] adapted the DDA algorithm to prioritize test cases using user-specified weights on parameters, values or combinations. Elbaum et.al give the prioritization technique for the specific modified version, and the correspondingly metric for fault detection rate APFD is proposed to measure the fault detection rate. Some other approaches reorders the test cases to minimize the switching cost (i.e. the cost of switching the current test case to the next one) [37]–[39].

Our method focus on detecting faults at early stage by defining weighted combinatorial coverage metric. The user need to provide the presumed distribution of the faulty combination size. However this distribution highly depends on the SUT itself and the model. Some empirical studies applied CT to different types of systems, and evaluated the effectiveness of CT at different covering strength, as well as the distribution of faulty combination sizes in the subject systems. In the study by Kuhn et al. [2], [8], they found that in their systems under investigation, most faulty combinations are small, and their size is smaller than 6. Some studies on other programs also found similar results [9], [10].

III. The weighted combinatorial coverage paradigm

Consider the problem of testing a object software like Siemens suit(ref) which may be injected faults, such as 1-way faults, 2-way faults, ...,6-way faults. We will give the motivation for the weighted combinatorial testing.

A. Motivation

Assume that the software testers know in advance that each i_{way} faults existed in the testing suit by their testing experiences. We assume that the presumed distribution of each i_{way} faults are as the follows out of the total faults being n: 1-way faults $n * 20\%$, 2-way faults $n * 35\%$, 3-way faults $n * 25\%$, 4-way faults $n * 10\%$, 5-way faults $n*5\%$, and 6-way faults $n*5\%$. Let the Card(V_t) denote the cardinality of t-way combinations, i.e., the number of t-way combinations for the object system. So that, we can give each

t-way fault detection rate as follows: $e_1 = \frac{n*20\%}{Card(V_1)}$, $e_2 = \frac{n*35\%}{Card(V_2)}$, $e_3 = \frac{n*25\%}{Card(V_3)}$, $e_4 = \frac{n*10\%}{Card(V_4)}$, $e_5 = \frac{n*5\%}{Card(V_5)}$, and $e_6 = \frac{n*5\%}{Card(V_6)}$. The expected detection faults for each test case can be calculated by the weighted combinatorial coverage. Assumed that the m^{th} test case is employed to test the system, and the new added t-way combinations for the m^{th} test case are c_t (t=1,2,3,4,5,6). Then, we can obtain the expected detection faults by calculating $\sum_t e_t \cdot c_t$. Whatever the static combinatorial testing or dynamic combinatorial testing, we can select the test case that the estimated expected faults are highest. Next, we first give the definitions for the i_{way} combinatorial coverage rate, and then we give the definition for the weighted combinatorial coverage rate.

B. Definition for weighted combinatorial coverage rate

Definition 1: i_{way} combinatorial coverage: Assume $m_i(t)$ denote the total i_way combinations used when the n_{th} test case finished testing, and M_i denote the total i_{way} combinations. Then, the i_{way} combinatorial coverage, $\rho_i(t)$, can be defined as: $\rho_i(t) = \frac{m_i(t)}{M_i}$

Definition 2: Weighted combinatorial coverage: Based on the i_{way} combinatorial coverage, $\rho_i(t)$, the weighted combinatorial coverage, $\rho(t)$, can be defined as:

$$\rho(t) = \rho_i(t) \cdot \frac{\hat{b}_i(t)}{\sum_i \hat{b}_i(t)} \tag{1}$$

where $\hat{b}_i(t)$ denote that estimated i_{way} faults in the system when the n^{th} test cases finished testing.

Theorem 3.1: The weighted combinatorial coverage rate, $\rho(t)$, which equals to the fault detection rate, $\frac{\hat{b}_i(t)}{\sum_i \hat{b}_i(t)}$.

PROOF. Consider that $\hat{b}_i(t)$ denote the estimated total i_{way} faults when the n^{th} test case finished testing. We need to estimate the total i_{way} faults since we do not know the number of faults at the beginning testing. Then, the total faults for the system which include each i_{way} is $\sum_i \hat{b}_i(t)$. According to the following formula, in which, $c_i(t)$ denote the finding faults using the number of t test cases when the t^{th} test case finished testing.

$$\frac{\hat{b}_i(t)}{c_i(t)} = \frac{M_i}{m_i^(t)} \tag{2}$$

We can obtain

$$\hat{b}_i(t) = \frac{M_i}{m_i(n)} \cdot c_i(t) \tag{3}$$

Then, we can input into the weighted combinatorial coverage $\rho(t)$, we obtain the following formula:

$$\rho(t) = \frac{c_i(t)}{\hat{b}_i(t)} \tag{4}$$

From (1) and (4), we can deduce that the weighted combinatorial coverage rate $\rho(t)$ equals the fault detection rate. Thus, we connect the weighted combinatorial coverage rate with the fault detection rate. Previous study compute the combinatorial coverage without considering the fault detection rate of individual test case. Given the short time availability during regression testing, a prioritization criterion that is effective at selecting tests that detect faults quickly while at the same time, selects tests that execute quickly would be highly beneficial. With this goal in mind, we modify the combinatorial coverage metric to incorporate weighted fault detection rate of test cases.

IV. Algorithm for weighted combinatorial test case selection

A. algorithm

We incorporate the fault detection rate for i_{way} faults into the i_{way} combinatorial coverage, then we define the weighted combinatorial coverage $\rho(t)$, which is the reasonable criteria in terms of the improved fault detection rate. Given the short time availability during regression testing, the weighted combinatorial coverage is effective at selecting tests that detect faults quickly. Fig.1 shows the weighted combinatorial coverage algorithm. Firstly, we identify the parameters, values, and interactions. Next, we generate test case suit by the requirement, and then we randomly select m test cases from the generated test cases, where $m = 5, 10, 15, 20$. We select the "next test" that it expose the maximum expected number of faults, w_{jx}, from m selected test cases, which equals the total of 1-way faults, 2-way faults, ...etc.

B. Theorem for Algorithm1

Theorem 4.1: Let B_i represent total i_ways faults, and M_i represent the number of i_ways combinations. Let $b_i(t)$ denote the number of faults for i_ways combination when the t test cases finished testing, and $m_i(t)$ denote the used number of i_ways combinations when the t test cases finished testing. The counting process $b_i(t), m_i(t)$ satisfy the following : $\forall \varepsilon > 0$,

$$\mathbb{P}\{|\frac{b_i(t)}{m_i(t)} - \frac{B_i}{M_i}| < \varepsilon\} \geq \ 1 - \frac{1}{m_i(t) \cdot \varepsilon^2} \tag{5}$$

We consider $m_i(n)$ independent Bernoulli trials with the probability of finding failure equal to $\frac{B_i}{M_i}$ on each trial. Thus, $b_i(n)$ follow $m_i(n)$ binomial distribution. Apply Chebyshev's inequality:

$$\mathbb{P}(|X - E(X)| < \varepsilon) > 1 - \frac{D(X)}{\varepsilon^2} \tag{6}$$

Thus, the variance of $b_i(t)$ is $m_i(t) \cdot \frac{B_i}{M_i} \cdot (1 - \frac{B_i}{M_i})$, and the mean value of $b_i(t)$ is $m_i(t) \cdot \frac{B_i}{M_i}$. Accordingly, we can obtain

$$\mathbb{P}\{|b_i(t) - \frac{B_i}{M_i} \cdot m_i(t) \leq \varepsilon \cdot m_i(t)\} \geq 1 - \frac{M_i(t) \cdot \frac{B_i}{M_i}(1 - \frac{B_i}{M_i})}{m_i(t)^2 \cdot \varepsilon^2} \tag{7}$$

And also, from the equation 8,

$$\frac{m_i(t) \cdot \frac{B_i}{M_i}(1 - \frac{B_i}{M_i})}{m_i(t)^2 \cdot \varepsilon^2} \leq \frac{1}{m_i(t) \cdot \varepsilon^2} \tag{8}$$

Algorithm 1 weighted combinatorial coverage algorithm

$identify\ parameters,\ values,\ and\ interactions$;
$Design\ test\ cases\ suit\ by\ requirements$;
$1 \leq\ i \leq N,\ 1 \leq\ k\ \leq m,\ j = 1,\ S_{0i} = \emptyset,\ F_i = 0$
while $j \leq\ maximum\ number\ of\ cases\ or F_i \neq\ 0$ **do**
    $Random\ select\ a\ test\ case\ from\ set\ t_{j1}, t_{j2}, ..., t_{ji}, ..., t_{jm}$;
    $the\ weight\ of\ t_{jk}\ is\ w_{jk}$;
    $w_{jk} = |u_{j1k}| \cdot e_1 + |u_{j2k}| \cdot e_2 + ...|u_{jik}| \cdot e_i + ... + |u_{jNk}| \cdot e_N$;
    $where\ u_{jik} = \{i - value\ slices\ in\ t_{jk}\ but\ not\ in\ S_{(j-1)i}\}$
    $w_{jx} = max(w_{jk})$;
    $Let\ t_{jx}\ be\ the\ test\ case\ with\ largest\ w_{jx}$;
    $\{u_{jix}, 1 \leq i \leq N\}\ as\ the\ correspondingly\ new\ added\ combinations$; ;
    $S_{ji} = S_{(j-1)i} \cup \{u_{jix,1 \leq i \leq N}\}$;
    if find the failure **then**
        $F_i = F_i - 1$
    else
        $Continue\ test$;
    end if
    $j = j + 1$;
end while

Therefore, we obtain the 9

$$1 - \frac{m_i(t) \cdot \frac{B_i}{M_i}(1 - \frac{B_i}{M_i})}{m_i(t)^2 \cdot \varepsilon^2} \geq 1 - \frac{1}{m_i(t) \cdot \varepsilon^2} \tag{9}$$

Finally, we can prove the 4.1.

Theorem 4.2: Assume that $b_i^{m_1}(t)$ and $b_i^{m_2}(t)$ are the faults detected as the number of t test case finished testing, when employing the optimized weight for Algorithm.1. With the parameter $m = m_1,\ m_2$. Let the expected detected faults with parameter $m = m_1,\ m_2$, are $E(b_i^{m_1}(t))$, $E(b_i^{m_2}(t))$, respectively. Thus, we have the following equation 10

$$E(b_i^{m_1}(t)) \leq E(b_i^{m_2}(t)) \tag{10}$$

when $t = 1$, we select the best test case with the parameter $m = m_2$, and the expected detected bugs are $E(b_i^{m_2})$. This process can be regarded as first randomly selecting the m_1 test cases, in which we select the best case to detect the number of $E(b_i^{m_1})\ i - way$ faults. And then we continue randomly select the $m_2 - m_1$ test cases, the expected number of detected bugs is $E(b_i^{(m_2-m_1)}(1))$. While $E(b_i^{m_2}(1)) = E(max\{b_i^{m_1}(1), b_i^{(m_2-m_1)}(1)\})$. Thus, it is clearly that $E(b_i^{m_1}(1)) \leq\ E(b_i^{m_2}(1))$.

Assume $E(b_i^{m_1}(k)) \leq E(b_i^{m_2}(k))$ when $t \leq k$. We prove that when $t = k + 1$ theorem4.2 establish. We prove it by contradiction.

First we let $E(b_i^{m_1}(k+1))>E(b_i^{m_2}(k+1))$, algorithm find the number of faults X_k when the number of k test cases finished testing with $m = m_1$. And, Y_k with $m = m_2$. Then, we obtain the formula as follows:

$$b_i^{m_1}(k) = \sum_{n=1}^{k} X_n,$$

and

$$b_i^{m_2}(k) = \sum_{n=1}^{k} Y_n.$$

The expected detected bugs for the $k + 1^{th}$ using the best selected case is $E(X(k + 1)), E(Y(k + 1))$ with parameter $m = m^1$ and $m = m^2$, respectively.

We have already assumed that

$$E(b_i^{m_1}(k + 1))>E(b_i^{m_2}(k + 1)),$$

i.e.,

$$E(b_i^{m_1}(k)) + E(X(k + 1)) > E(b_i^{m_2}(k)) + E(Y(k + 1)).$$

As for $E(X(k+1))$, we can regard m_1 as first select m_0 and then select $m_1 m_0$. $E(X^{m_0}(k+1))$ denote the expected number of detected bugs with m^0, and $E(X^{(m_1-m_0)}(k + 1))$ denote the expected number of detected bugs with $m_1 - m_0$. In which,

$$E(X(k+1)) = E(max\{X^{m_0}(k+1)), X^{(m_1-m_0)}(k+1))\}).$$

For $m = m_2$, We employ the same approach as $m = m_1$. Thus,

$$E(Y(k + 1)) = E(max\{Y^{m_2}(k + 1), Y^{(m_2-m_0)}(k + 1))\})$$

There exists a m_0, which make

$$E(b_i^{m_0}(k)) + X(k + 1) = E(b_i^{m_0}(k)) + Y(k + 1)$$

At the second phase of selecting, we have $m_1 - m_0$,$m_2 - m_0$. Because $m^1 - m^0 < m^2 - m^0$ so that we have $X^{(m_1-m_0)}(k+1)) \leq Y^{(m_2-m_0)}(k+1))$, And also because

$$E(X(k+1)) = E(max\{X^{m_0}(k+1)), X^{(m_1-m_0^1)}(k+1))\}),$$

$$E(Y(k+1)) = E(max\{Y^{m_0}(k+1)), Y^{(m_2-m_0^1)}(k+1))\}),$$

Moreover,

$$E(b_i^{m_1}(k)) + X^{m_0}(k + 1)) = E(b_i^{m_2}(k)) + Y^{m_0}(k + 1)),$$

i.e.,

$$E(b_i^{m_1}(k)) = E(b_i^{m_2}(k)) + (Y^{m_0}(k+1)) - X^{m_0}(k+1))).$$

Because,

$$(Y(k+1)) - X(k+1))) \geq (Y^{m_0}(k+1)) - X^{m_0}(k+1)))$$

Thus,

$$E(b_i^{m_1}(k)) \leq E(b_i^{m_2}(k)) + (Y(k+1) - X(k+1))$$

i.e.,

$$E(b_i^{m_1}(k)) + X(k+1)) \leq E(b_i^{m_2}(k)) + Y(k+1))$$

i.e.,

$$E(b_i^{m_1}(k+1)) \leq E(b_i^{m_2}(k+1))$$

This contract with the assumption $E(b_i^{m_1}(k+1)) \geq E(b_i^{m_2}(k+1))$ Therefore, $\forall n \leq 1, E(b_i^{m_1}(k)) \leq E(b_i^{m_2}(k))$

Proof finished.

V. Experiment

We did the simulation to check the effectiveness of the static algorithm. The input for the simulation model had 13 parameters, and the parameter values is $\{4, 2, 6, 7, 3, 6, 2, 3, 10, 4, 5, 5, 3\}$. That is, the first parameter had 4 values, the second parameter had 2 values, ..., and the last parameter had 3 values. This gives a total of 108864000 combinations. We injected faults for $1-way$, 2_{way}, ..., 6_{way} into the simulation environment. And, the faults size for i_{way} are denoted as $\{8, 65, 78, 68, 35, 1\}$. For the testers experience, we employ the good weight that is used to denote the testers good experience for guessing the software faults for i_{way}. The employed good_weight is $\{9, 67, 76, 66, 37, 2\}$, and the bad_weight is $\{10, 170, 70, 8, 0, 0\}$.

In one simulation, we use good_weight and bad_weight parameter to test the effectiveness of algorithm and compare between the effectiveness between good_weight and bad_weight. we compare the test effectiveness for different value settings when $m = 5, 10, 20, 40$, and we prove theorem4.2 by using this application.

In the simulation settings, the total injected faults are 255. At each m value, we have 20 runs using different seeds. Thus, at each fault of total 255, we calculate the mean value of 20 runs. We obtain the test cases used for good_weight, bad_weight, respectively. Furthermore, we obtain the combinatorial coverage, i_way coverage at good_weight, bad_weight case. Next we give the metric for comparison.

A. Comparison Metric

Metric_1: $1 \cdot C_1 + 2 \cdot C_2 + ... + i \cdot C_i + ... + N \cdot C_N$
Metric_2: $C_1 + C_2 + ... + C_N$
Metric_1 denote the number of detected faults(x-axis value) multiply the number of tested cases(y-axis value),C_i. Or, the number of tested cases (x-axis value) multiply the number of detected faults(y-axis value). Metric_2 denote the sum of the values of y-axis at fixed x-axis value.

B. Results

Figure 1a shows that at the good_weight case, the x-axis denote the bug number, and the y-axis denote the number of test cases used to detect the correspondingly bugs. As shown in the Figure 1a for example, $m = 5 : 20161026, 1429 and 87226.6429$ means that Metric_1 value is 87226.6429, Metric_2 value is 87226.6429. The same meaning with $m = 10, 20, 40$. The less y-axis value denote that the less test cases used to detect the same amount of faults. Figure 1b also shows when $m = 40$, the test efficiency is best, and then for $m = 10$, $m = 5$, and the last one is $m = 20$. The above figures are all enlarged parts.

Figure.**??** is the case for bad_weight pair-wise, and figure.**??** is the enlarged figure. From these figures, we can see that best test efficiency is $m = 5$, and then $m = 40, 20, 10$ using Metric_1 as well as Metric_2.

Table.I also shows the comparisons between the good_weight and the bad_weight algorithm for bug_testcase. From table I, we see that at $m = 5, 10, 20, 40$, the good_weight perform better than that of bad_weight algorithm.

We also give the weighted combinatorial coverage as shown in figure2a, figure2b. i_way combinatorial coverage as shown in figure3a for good_weight, figure3b for bad_weight case, respectively.

VI. conclusion

References

[1] J. Hagar, T. Wissink, D. Kuhn, and R. Kacker, "Introducing combinatotial testing in a large organization," *IEEE Computer*, vol. 48, no. 4, pp. 64–72, 2015.

[2] D. Kuhn and M. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop (SEW '02)*, 2002, pp. 91–95.

[3] D. Kuhn, R. Kacker, and Y. Lei, "Practical combinatorial testing," *NIST special Publication*, vol. 800, no. 142, 2010.

[4] G.Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 011, pp. 96–17, 2002.

[5] B.Inquiry, "Ariane 5 - flight 501 failure," 1996.

[6] L. N. G and C.S.Turner., "An investigation of the therac-25 accidents," *IEEE Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[7] D. Cohen, D. Dalal, M. Fredman, and G. Patton, "The aetg system: an approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, 1997.

[8] D. Kuhn, D. Wallace, and G. J. AM, "Software fault interactions and implications for software testing," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 418–421, 2004.

[9] Z. Zhang and J. Zhang, "Characterizing failure-causing parameter interactions by adaptive testing," in *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 2011, pp. 331–341.

[10] S. Vilkomir, O. Starov, and R. Bhambroo, "Evaluation of t-wise approach for testing logical expressions in software," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW '13)*. IEEE, 2013, pp. 249–256.

[11] R. Kuhn, R. kacker, Y. Lei, and J. Hunter, "Combinatorial software testing," *IEEE Computer*, vol. 42, no. 8, pp. 94–96, 2009.

[12] C. Nie, H. Leung, and K.-Y. Cai, "Adaptive combinatorial testing," in *2013 13th international conference on quality software*, 2013, pp. 284–287.

[13] M. Grindal, J. Offutt, and S. Andler, "Combination testing strategies: A survey," *Software Testing,Verification and Reliability*, vol. 15, no. 3, pp. 167–199, 2005.

(a) Algorithm 1:Good Distribution(enlarge)

(b) Algorithm 1:Bad Distribution(enlarge)

Fig. 1: Algorithm 1:good weight and bad weight(enlarge)

TABLE I: Algorithm 1:Good and Bad(enlarge),m=5,10,20,40

	m=5	m=10	m=20	m=40
Good	$M_1 = 23443232.6$ $M_2 = 100419.45$	$M_1 = 21194024.4$ $M_2 = 91356.55$	$M_1 = 19752087.8$ $M_2 = 95169.8$	$M_1 = 19217087.3$ $M_2 = 83223.5$
Bad	$M_1 = 21909925.5$ $M_2 = 94109$	$M_1 = 21799671.85$ $M_2 = 93675.85$	$M_1 = 22519801.2$ $M_2 = 96537.25$	$M_1 = 20914560.3$ $M_2 = 89803.85$

(a) Algorithm 1:Combination coverage good(enlarge)

(b) Algorithm 1:Combination coverage bad(enlarge)

Fig. 2: Algorithm 1:Combinatorial coverage good and bad enlarge

(a) Algorithm 1:i ways combinatio coverage good(enlarge)

(b) Algorithm 1:i ways combinatio coverage bad(enlarge)

Fig. 3: Algorithm 1:i ways combinatorial coverage good and bad(enlarge)

[14] J. Zhang, Z. Zhang, and F. Ma, *Automatic Generation of Combinatorial Test Data*. Springer Berlin Heidelberg, 2014.

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

[16] D. Kuhn, R. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*. CRC press, 2013.

[17] G. Sherwood, "Effective testing of factor combinations," in *Proceedings of the 3rd International Conference on Software Testing, Analysis & Review*, 1994, pp. 151–166.

[18] Y. Tung and W. Aldiwan, "Automating test case generation for the new generation mission software system," in *Proceedings of the 2000 IEEE Aerospace Conference*, vol. 1. IEEE, 2000, pp. 431–437.

[19] M. Cohen, M. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the 16th International Symposium on Software Testing and Analysis (ISSTA '07)*. ACM, 2007, pp. 129–139.

[20] M. Cohen, M. Dwyer, and J.Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 633–650, 2008.

[21] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Proceedings of 24th Pacific Northwest Software Quality Conference*, 2006, pp. 419–430.

[22] A. Calvagna and A. Gargantini, "A logic-based approach to combinatorial testing with constraints," in *Proceedings of the 2nd International Conference on Tests and Proofs (TAP '08)*. Springer, 2008, pp. 66–83.

[23] Y. Zhao, Z. Zhang, J. Yan, and J. Zhang, "Cascade: a test generation tool for combinatorial testing," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW '13)*. IEEE, 2013, pp. 267–270.

[24] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, "Generating combinatorial test suite using combinatorial optimization," *Journal of Systems and Software*, vol. 98, pp. 191–207, 2014.

[25] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC '04)*. IEEE, 2004, pp. 72–77.

[26] R. Bryce and C. Colbourn, "One-test-at-a-time heuristic search for interaction test suites," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO '07)*. ACM, 2007, pp. 1082–1089.

[27] B. Ahmed and K. Zamli, "A variable strength interaction test suites generation strategy using particle swarm optimization," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2171–2185, 2011.

[28] B. Ahmed, K. Zamli, and C. Lim, "Application of particle swarm optimization to uniform and variable strength covering array construction," *Applied Soft Computing*, vol. 12, no. 4, pp. 1330–1347, 2012.

[29] Y. Lei, R. Kacker, D. Kuhn, V. Okun, and J. Lawrence, "Ipog/ipog-d: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[30] B. Garvin, M. Cohen, and M. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE '09)*. IEEE, 2009, pp. 13–22.

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

[32] S. Fouche, M. Cohen, and A. Porter, "an incremental covering array failure characterization in large configuration space," in *2009 ISSTA*, 2009, pp. 177–188.

[33] X. Qu, M. Cohen, and K. Woolf, "Combinatorial interaction regression testing: a study of test case generation and prioritization," in *Proceedings of the 23th International Conference on Software Maintenance (ICSM '07)*. IEEE, 2007, pp. 255–264.

[34] X. Qu, M. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, 2008, pp. 75–86.

[35] X. Qu and M. Cohen, "A study in prioritization for higher strength combinatorial testing," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW '13)*. IEEE, 2013, pp. 285–294.

[36] R. Bryce and C. Colbourn, "Prioritized interaction testing for pairwise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[37] S. Kimoto, T. Tsuchiya, and T. Kikuno, "Pairwise testing in the presence of configuration change cost," in *Proceedings of the 2nd International Conference on Secure System Integration and Reliability Improvement (SSIRI '08)*. IEEE, 2008, pp. 32–38.

[38] H. Srikanth, M. Cohen, and X. Qu, "Reducing field failures in system configurable software: Cost-based prioritization," in *Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE '09)*. IEEE, 2009, pp. 61–70.

[39] H. Wu, C. Nie, and F. Kuo, "Test suite prioritization by switching cost," in *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW '14)*. IEEE, 2014, pp. 133–142.