Tuple Density: A New Metric for Combinatorial Test Suites (NIER Track) [*]

Baiqiang Chen
State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences
chenbq@ios.ac.cn
Graduate University of Chinese Academy of Sciences

Jian Zhang
State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, P.R.China, 100190
zj@ios.ac.cn

ABSTRACT

We propose tuple density to be a new metric for combinatorial test suites. It can be used to distinguish one test suite from another even if they have the same size and strength. Moreover, it is also illustrated how a given test suite can be optimized based on this metric. The initial experimental results are encouraging.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics

General Terms

Reliability, Measurement, Algorithms

Keywords

Tuple density, combinatorial testing, test suites, metrics

1. INTRODUCTION

Combinatorial testing is a widely used black-box testing approach to detecting defects triggered by parameter interactions. In combinatorial testing, the test suite is often represented by a Mixed Covering Array (MCA).

Definition 1. *$MCA(N; t, k, v_1 v_2 \cdots v_k)$ is an $N \times k$ matrix having the following properties:*

1. *symbols in the i-th column ($1 \le i \le k$) compose a set of size v_i;*
2. *any $N \times t$ sub-matrix covers all value combinations of the corresponding t columns.*

[*]This work is supported in part by the National Science Foundation of China (Grant No. 60903049, 61070039) and the High-Tech (863) program of China (Grant No. 2009AA01Z148).

Here, t is called the strength of the MCA. Each position of the matrix is commonly called an "entry". In the MCA, each column represents a parameter of the System Under Test (SUT), and each row represents a test case. For example, Table 1 shows an instance of $MCA(6; 2, 7, 2^7)$. There are six test cases in the test suite, and they cover all possible value combinations for any two parameters (Property 2).

Table 1: MCA$(6; 2, 7, 2^7)$

p_0	p_1	p_2	p_3	p_4	p_5	p_6
0	0	0	0	0	0	0
0	1	1	0	1	1	0
1	0	1	1	0	1	0
1	1	0	1	1	0	0
0	0	0	1	1	1	1
1	1	1	0	0	0	1

In previous work, if two test suites of strength t have the same size, because both of them can expose potential defects triggered by *no more than* t parameters [1], they are considered to be equally good. This is not the case. In fact, for any MCA of strength t, it not only covers all of the t^*-tuples ($t \ge t^* > 0$), but also covers a number of t'-tuples ($k \ge t' > t$). For instance, let us illustrate the coverage of tuples of different dimensions (from 1 to 7) for the above MCA in Figure 1.

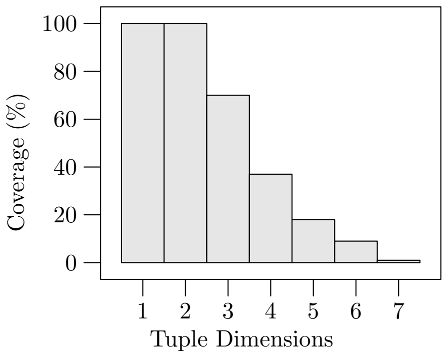

Figure 1: Coverage of Different Tuples

As we can see from Figure 1, the MCA also covers many t'-tuples ($7 \geq t' > 2$) (*e.g.*, 70% 3-tuples). Therefore, it has the ability to detect many defects triggered by more than two parameters. But this ability can not be described by the strength $t = 2$. In order to evaluate the complete detection ability of a test suite more accurately, we should also take the covered t'-tuples ($k \geq t' > t$) into account. Motivated by this fact, we propose the concept of *tuple density*, which synthesizes strength t and coverage of tuples with dimensions higher than t. The idea is supported by our initial experimental results (refer to Section 4), which convince us that coverage of tuples of higher dimensions can be quite different, even if the test suites have the same strength and size. Although the definition is quite simple, tuple density sets up a new and more accurate evaluation criterion for test suites in combinatorial testing. Given a group of candidates with the same size and the same strength, we can choose the specific test suite with the highest tuple density in order to expose more potential defects.

Another contribution is, we propose the optimization problem for test suites based on tuple density. The optimization is on the premise that we do not increase the size, and do not decrease the original strength, either. We also present an optimization algorithm in this paper. It is achieved by identifying a don't care entry set from the original test suite, and then instantiating the don't care entries in a greedy way. The experimental results are encouraging.

The remainder of this paper is organized as follows. In the next section, we introduce the definition of tuple density. In Section 3, we discuss how to optimize a given MCA by identifying and instantiating a don't care entry set. In Section 4, we give some experimental results, and Section 5 gives some future research directions driven by tuple density.

2. TUPLE DENSITY

2.1 Definition

Definition 2. *For an MCA of strength t, the tuple density is the sum of t and the percentage of the covered $(t+1)$-tuples out of all possible $(t+1)$-tuples.*

For example, since 70% 3-tuples are covered by the MCA shown in Table 1, according to the definition, its tuple density is $2 + 0.7 = 2.7$.

Tuple density is an extension of the strength t. Note in the definition of tuple density, we do not include the coverage of tuples whose dimensions are $t+2$, $t+3$, $\cdots$, k. The reasons are as follows.

1. For an MCA with strength t, the coverage of t'-tuples ($k \geq t' > t+1$) is much lower compared with the coverage of t-tuples and $(t+1)$-tuples. As we can see from Figure 1, as the dimension increases, the coverage decreases quickly.
2. Coverage of $(t+1)$-tuples can partly reflect the coverage of t'-tuples ($k \geq t' > t+1$) in a sense. Note $(t+1)$-tuples are subsumed by t'-tuples ($k \geq t' > t+1$).
3. According to empirical results, the number of defects triggered by i-tuples decreases rapidly as i increases [5]. It is reasonable to focus on the leading target.

2.2 Significance

High-quality test suites are quite crucial for software testing. In particular, for specific test scenarios like railway control systems and missile systems, since the cost of each test case may be extraordinary, to obtain an optimized test suite in order to improve the effectiveness is rather desirable. Tuple density can be used to meet this requirement in at least two aspects.

1. Selection of test suites

   Given a group of test suites generated by different tools with the same strength and the same size, one can evaluate them by comparing their tuple densities, and select the one with the highest tuple density since it covers more tuples of higher dimensions than others.

2. Optimization of test suites

   In certain circumstances, we may just have one available generation tool, or are just given an existing test suite. Tuple density can be used as a measure of optimization for the test suite.

Besides, we can also combine selection and optimization in real-world applications. Since computation of tuple density is straightforward, we will mainly discuss how to optimize a given test suite in the following section.

3. TEST OPTIMIZATION

Given a test suite, the optimization is achieved by modifying some entries of the suite.

Definition 3. *Given an MCA, a Don't Care Entry Set (DCES) is a non-empty entry set such that whatever values the entries of the set take, the strength of the MCA will not be changed. Each entry of the DCES is called a Don't Care Entry (DCE).*

Basically, we should ensure that all the t-tuples are still completely covered after optimization. Therefore, only if there is at least one DCES, can the MCA be optimized. Although for some MCA's, they may have no DCES (*e.g.*, when $t = k$), according to our experiments in the following sections, we can find at least one DCES from most MCA's.

3.1 Identifying a DCES

Intuitively, if we can find a DCES which has the most don't care entries, it is more likely for us to obtain better optimization. But finding such an optimal DCES is challenging, so instead, we try to find a DCES with as many don't care entries as we can.

In this paper, without loss of generality, the test suite TS is represented by an MCA of strength t, and there are k parameters p_i of domain size v_i. Let R_i denote the set of t-tuples which involve p_i. All t-tuples of each R_i can be divided into v_i subcategories according to p_i's assignments. For example, for a Boolean parameter p_0, t-tuples of R_0 are divided into two categories: where $p_0 = 0$ and where $p_0 = 1$ (see Figure 2). The algorithm traverses test cases one after another, and collects the coverage information for the t-tuples in R_i for each parameter p_i. Whenever all of the t-tuples in R_i where p_i is assigned v_i are covered by the traversed test cases, all the remaining entries where $p_i = v_i$ are identified as don't care entries.

We give an example in Figure 2. In the current state, since all 2-tuples in R_0 where $p_0 = 1$ are already covered, we can identify the first entry of $\langle 1, 0, 1\rangle$ where $p_0 = 1$ as a don't care entry.

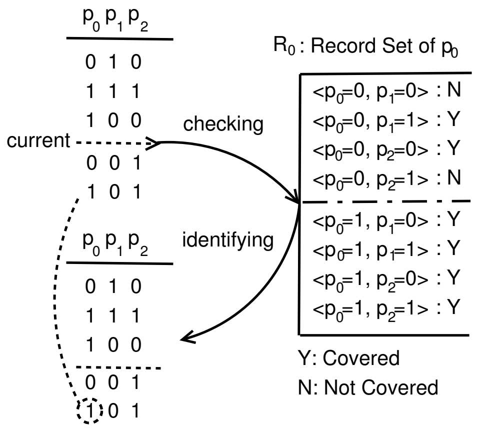

Figure 2: DCE Identification

In order to identify as many as possible don't care entries, two extra strategies can be employed.

3.1.1 Sorting the Test Cases

In the DCES identifying process, test cases are traversed successively. The ordering of test cases can affect the number of identified don't care entries. Note in general, only if the traversed test cases cover as many distinct t-tuples as possible can we find more don't care entries. Therefore, before the identifying process, we propose the *Hamming distances (HD)* based method to sort the test cases in advance.

HD describes the differences between two test cases. It equals the number of parameters with different assignments, *e.g.*, the HD between $\langle 0, 1, 0\rangle$ and $\langle 1, 1, 1\rangle$ is 2. We illustrate the method in Figure 3 with an example. We start from the first test case - $TS[0]$ (this can be another test case), then $TS[2]$ and $TS[1]$ (or more successors) are selected as successors because they have the farthest two HD's from $TS[0]$. This process is repeated twice. Later on, we just select the farthest test case to be the successor since we do not like to generate too many orderings. In this example, we generate four test orderings. Then, we may randomly pick up one, or try each ordering and select the best result.

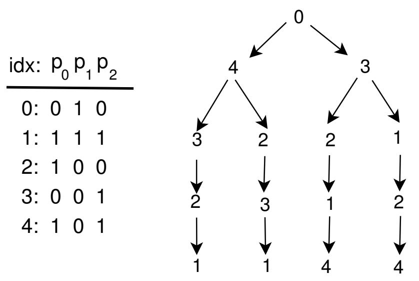

Figure 3: HD-Based Ordering Generation

In Table 2, we evaluate the effectiveness of the HD-based method. As we can see from the results, the sizes of DCES's pretreated by HD-based method (HDB.) has an obvious advantage compared with the random method (Rand.).

Table 2: Sizes of DCES's

Para.	HDB.		Rand.	
	Ave.	Max.	Ave.	Max.
$2^4 3^4 4^4$	50	65	40	51
$2^{10} 3^{10}$	52	66	39	55
$2^{10} 4^{10}$	133	166	94	108
$2^{10} 5^{10}$	219	242	168	201

3.1.2 Repeating the Identifying Process

Since the size of identified DCES is closely related to the ordering of test cases, we can try to find more don't care entries from the test suite by re-sorting the test suites until no extra entries can be identified. We show the process for $MCA(49; 2, 20, 2^{10}5^{10})$ in Figure 4. As we can see, 74 (from 219 to 293) don't care entries are found by repeating the identifying process. Note the increasing rate reaches 34%.

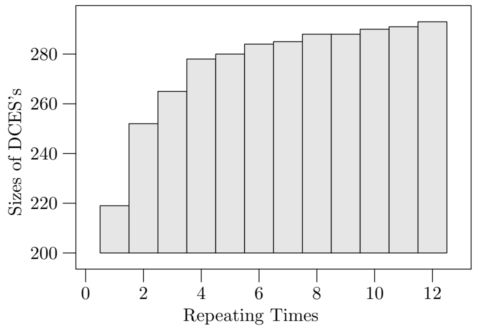

Figure 4: Increment of DCES

3.2 Instantiating the DCES

After identifying the DCES, the don't care entries are instantiated to cover as many $(t + 1)$-tuples as possible. It is quite impractical to try all possible value combinations of the don't care entries due to the exponential explosion, and we simply use a greedy algorithm to instantiate the DCES.

First of all, information on the $(t+1)$-tuples which are not covered by the MCA is collected. After that, test cases are processed one after another. For each test case, whenever encountering a don't care entry, we check how many $(t+1)$-tuples can be newly covered when it is assigned different values from its domain, and then instantiate it with the value which results in most newly covered $(t + 1)$-tuples. In order to alleviate the local maximum problem, we could instantiate more than one don't care entries at a time.

4. EXPERIMENTAL RESULTS

4.1 Setup

We implemented the optimization algorithm in a prototype tool *CTOP* (Combinatorial Testing OPtimization).

The objective of our experiments is twofold. First, we want to show whether the MCA's generated by existing tools (algorithms) can be optimized. Second, we also would like to show how different the tuple densities can be, for MCA's with the same strength and size. Several parameter models are used. Some of them are derived from real-world applications like the system which generates Train Operational Schemes. We mainly try the pairwise testing ($t = 2$), since it is the most popular method used in practice.

In Table 3, the original test suites are generated by PICT [4], which has been in use at Microsoft Corporation since 2000. In fact, we also try other MCA's generated by other tools (see [2]), but due to the page limitation, we only list six instances. Each test suite is optimized in a short time (within 10 seconds) on a common PC.

Table 3: Tuple Densities Comparison

Para	Size	d_0	d_1	d_2	Δ_1	Δ_2
$2^4 3^4$	12	2.63	2.67	2.62	0.04	0.05
$2^4 3^4 4^4$	22	2.61	2.64	2.57	0.03	0.07
$2^{10} 8^2$	64	2.83	2.87	2.71	0.04	0.16
$2^{10} 4^{10}$	31	2.69	2.71	2.64	0.02	0.07
$2^{10} 3^{10}$	19	2.73	2.76	2.70	0.03	0.06
2^{15}	10	2.83	2.86	2.79	0.03	0.07

In Table 3, d_0 is the tuple density of the original MCA, and d_1 is the tuple density of the MCA after optimization. In order to obtain a "worse" MCA, we instantiate the identified DCES by assigning values which result in fewer newly covered $(t+1)$-tuples to the don't care entries. In the table, d_2 is the tuple density of the MCA after being processed in such a reverse way.

4.2 Analysis

We mainly analyze two differences:

(1) $\Delta_1 = d_1 - d_0$

As we can see from Table 3, all of the original test suites can be improved by the optimization algorithm since $\Delta_1 > 0$. The tuple densities are improved by $0.02 - 0.04$. At first sight, one may suspect the effectiveness since $0.02 - 0.04$ seems to be not so significant.

Let us clarify this by selecting $2^{10}4^{10}$ from Table 3 for example. Note it even has the least Δ_1 (0.02) among the presented results. It is not difficult to verify that there are in total 30240 possible 3-tuples, and it requires 158 test cases to cover these tuples by PICT, so each test case covers about 192 (30240/158) 3-tuples on average. After the optimization, the number of covered 3-tuples is increased by 823 (from 20740 to 21563). Therefore, for defects triggered by interactions of three parameters, it is equivalent to testing the SUT with more than four ($823/192 = 4.3$) extra test cases compared with the original test suite. Note these extra benefits are obtained on the premise that we do not need to increase the real size of the test suite, so even the least value 0.02 is still quite encouraging.

(2) $\Delta_2 = d_1 - d_2$

Compared with Δ_1, Δ_2 is more notable. The largest difference of Δ_2 reaches 0.16. It convinces us that for a group of MCA's, even if they have the same size and the same strength, their tuple densities can be quite different. This is also the reason why it is necessary for us to select an optimal one carefully, and why we need to do the optimization.

5. FUTURE DIRECTIONS

In this paper, we propose tuple density as a new metric for combinatorial test suites, which draws attention to coverage of tuples with higher dimensions. We also illustrate how tuple density can be used to select and optimize test suites in combinatorial testing. Our work is still preliminary. We would like to give some future reseach directions driven by tuple density.

First, in some real-world applications, there may exist certain constraints between parameters. Under such circumstances, test suites are represented by Constrained Mixed-level Covering Arrays (CMCA) [3]. Other extensions of MCA like variable strength covering arrays and prioritized covering arrays were also proposed . Even so, we believe the intrinsic idea of tuple density is also applicable. Future work should be focused on extending this metric and optimization to such circumstances.

Second, the algorithm we proposed may be improved, *e.g.*, to introduce some meta-heuristic techniques in the process of instantiating the DCES. We may also extend the DCES identifying algorithm to reduce the number of rows for a given MCA [6].

Last but not the least, based on tuple density, combinatorial test suite can be generated in a new view. In previous work, combinatorial test suites are generated according to a given strength, and the main objective is to construct a test suite that has as few as possible test cases. Actually, in most real-world applications, the budgets of time and costs are often decided as a constant in advance, and thus the size of test suites is fixed as a constant accordingly. In the future, it is important and interesting for us to solve the problem: given an integer N, how can we construct a test suite of size N with the highest tuple density?

Acknowledgement

The authors would like to thank Jeff Lei and the anonymous referees for their helpful comments and suggestions.

6. REFERENCES

[1] B. Chen, J. Yan, and J. Zhang. Combinatorial testing with shielding parameters. In *17th Asia-Pacific Software Engineering Conference*, pages 280–289, Sydney, Australia, 2010.

[2] B. Chen and J. Zhang. Evaluation and optimization of test suites in combinatorial testing. *Technical Report ISCAS-LCS-10-18*, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, September,2010.

[3] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. Software Eng.*, 34(5):633–650, 2008.

[4] J. Czerwonka. Pairwise testing in real world: Practical extensions to test case generator. In *PNSQC '06: Proc. of 24th Pacific Northwest Software Quality Conference*, pages 419–430, Portland, Oregon, USA, 2006.

[5] R. Kuhn, Y. Lei, and R. Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10:19–23, 2008.

[6] P. Nayeri, C. J. Colbourn, and G. Konjevod. Randomized postoptimization of covering arrays. *LNCS 5874*, pages 408–419, 2009.