On Use of Coverage Metrics in Assessing Effectiveness of Combinatorial Test Designs

Jacek Czerwonka
Microsoft Research
Redmond, WA USA
jacekcz@microsoft.com

Abstract

Combinatorial test suite design is a test generation technique, popular in part due to its ability to achieve coverage and defect finding power approximating that of exhaustive testing while keeping test suite sizes constrained. In recent years, there have been numerous advances in combinatorial test design techniques, in terms of efficiency and usability of methods used to create them as well as in understanding of their benefits and limitations when applied to real world software. Numerous case studies have appeared presenting practical applications of the combinatorial test suite design techniques, often comparing them with manually-created, random, or exhaustive suites. These comparisons are done either in terms of defects found or by applying some code coverage metric. Since many different and valid combinatorial test suites of strength *t* can be created for a given test domain, the question whether they all have the same coverage properties is a pertinent one.

In this paper we explore the stability of size and coverage of combinatorial test suites. We find that in general coverage levels increase and coverage variability decreases with increasing order of combinations *t*; however we also find exceptions with implications for practitioners. In addition, we explore cases where coverage achieved by combinatorial test suites of order *t* applied to the same program is not different from test suites of order *t-1*. Lastly, we discuss these findings in context of the ongoing practice of applying code coverage metrics to measure effectiveness of combinatorial test suites.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Design, Metrics

Keywords

Pairwise testing, combinatorial testing, test case generation, test case design, coverage metrics, statement and branch coverage

Introduction

A set of possible inputs for any nontrivial piece of software is often too large to be tested exhaustively. Techniques like equivalence partitioning [28] and boundary-value analysis [17] help convert a large number of test levels into a much smaller set with comparable defect-detection power. Still, if software under test (SUT) can be influenced by a number of such factors, exhaustive testing again becomes impractical.

Over the years, a number of combinatorial strategies have been devised to help engineers choose subsets of input combinations that would maximize the probability of detecting defects: *random testing* [16], *each-choice* and *base choice* [2], *anti-random* [15] and finally combinatorial testing strategies with pairwise testing being the most prominent among these.

Pairwise testing strategy is defined as follows:

Given a set of M test factors: f_1, f_2, ..., f_M, with each factor f_i having L_i discrete levels: f_i = { $L_{i,1}$, ..., $l_{i,\ L_i}$ }, a set of tests *R* (matrix) is produced. Each test (row) in *R* by definition contains *M* test levels, one for each test factor f_i, and collectively all tests in *R* cover all possible pairs of test factor levels. Or more formally: for each pair of factor levels $l_{i,p}$ and $l_{j,q}$, where $1 \leq p \leq L_i$, $1 \leq q \leq L_j$, and $i \neq j$ there exists at least one row in *R* that contains both $l_{i,p}$ and $l_{j,q}$.

This concept can easily be extended from covering all possible pairs to covering all t-way combinations where $1 \leq t \leq M$. When $t = 1$, the strategy is equivalent to *each-choice*; if $t = M$, the resulting test suite is said to be *exhaustive*.

Covering all pairs of tested factor levels has been extensively studied. Mandl described using orthogonal arrays in testing of a compiler [16]. Tatsumi, in his paper on Test Case Design Support System used in Fujitsu Ltd [22], talks about two standards for creating test arrays: (1) with all combinations covered exactly the same number of times (orthogonal arrays) and (2) with all combinations covered at least once. When making that crucial distinction, he references an earlier paper by Shimokawa and Satoh [19].

Over the years, numerous case studies of combinatorial test designs have shown the technique to be an efficient and effective strategy of choosing tests [4, 5, 6, 7, 10, 13, 23]. Often they make use of coverage metrics [4, 6, 7] as a way of comparing combinatorial testing to other methods including the aforementioned *base-choice*, *each-choice* and exhaustive testing. Irrespective of the chosen coverage metric, given the basic property of combinatorial design namely that there are possibly very many possible test suites meeting the criteria for covering all t-way combinations derived from the same SUT description, a question remains: how stable and reliable are measurements of coverage for a given program and combinatorial strength t. The same question pertains to defect finding power as well, which is often used as another measure of effectiveness of the strategy, however we decided to focus on the coverage metrics in this paper.

Problem Description

Combinatorial test case generators [1] attempt to create as small a test suite as possible while still maintaining the property of covering all allowed t-way combinations of inputs. Multiple strategies for achieving this result were attempted and implemented over the years. They range from heuristics [7, 8, 26], through deterministic methods [12, 14] to applying exhaustive search in an attempt to find the globally optimal solution [27]. A survey of available methods can be found in [11] and [29].

The fault model assumed when applying combinatorial test designs states that defects occur when factors of the software-under-test interact. The first non-trivial test design of this kind, the pairwise testing design attempts to test all possible 2-way interactions of factors for that very reason.

Test suites with t=k+1 subsume all combinations tested by t=k. It follows therefore, that if we assume the fault model to be holding for a given SUT, t=k+1 test suite should find at least the same number of (and possibly more) defects stemming from factor-interactions. Since we often use code coverage as a measure of goodness of a test suite (relative to other suites), the key questions is whether the same assumption of increasing value of a metric with increasing t, holds for code coverage as well.

Moreover, the community is often focused on methods for finding the smallest t-way test suite. We need to understand the tradeoffs we are making between the test suite size and coverage especially in light of studies suggesting that effectiveness of pairwise testing can be approximated by random testing [3].

We explore this topic with the following questions:

Question 1: For a given SUT, how does the maximum code coverage change with the interaction strength t?

Question 2: What is the range of coverage achieved by a sample of test suites generated for each strength t? Does code coverage variance change with t?

Question 3: Is code coverage achieved by t=k and t=k+1 statistically distinguishable in all (or even most) cases?

Question 4: Are there qualitative and quantitative differences between non-trivial t-way test designs (t=2 and higher) and simpler test strategies like *each-choice* (t=1).

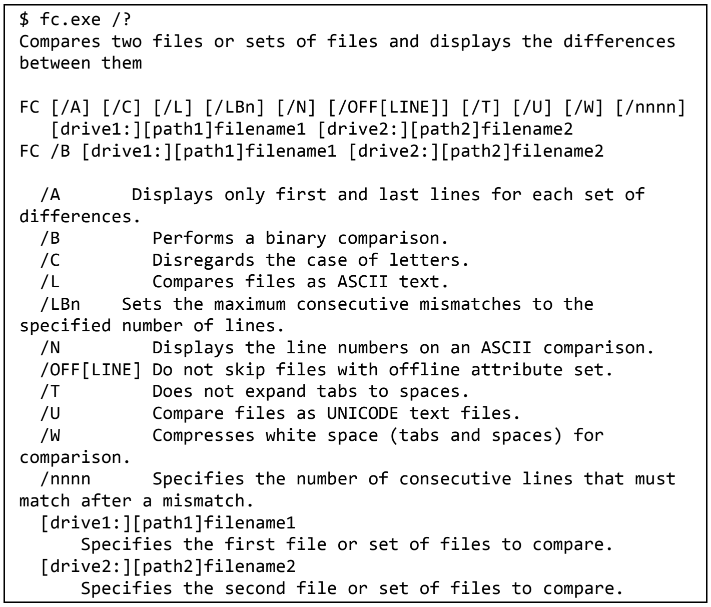
How code coverage behaves with increasing t has implications on the way we measure and report on t-way test designs in our case studies. The issue of using coverage metrics as a way to evaluate quality of *any* test suite notwithstanding, we would like to understand whether typical coverage metrics are even useful as a measurement tool for reliably discriminating between t=k and t=k+1, therefore we ask:

Question 5: Is increase of coverage attributable to increasing t or simply to the fact that more unique test cases get run for test suites with higher t?

Behind all the questions is our desire to understand how to best report on successes and challenges of combinatorial test design methods. We will propose revised practices of generating t-way test suites and measuring their effectiveness using coverage.

Experimental Setup

Four utilities included in the standard Windows 7 operating system installation were chosen for the experiments: **attrib.exe**, which is a tool for setting attributes on files, **fc.exe**, a file comparison tool, and two file content search tools: **find.exe**, **findstr.exe**. For each, a description of inputs was created using PICT's [26] input modeling format. PICT was chosen as it uses a greedy heuristic and can be configured to find multiple t-way covering test suites, each corresponding to some local optimum of the search space. The choice of a tool is dictated by needs of the experimental setup; the results apply to combinatorial designs irrespective of the employed tool.



```
$ fc.exe /?
Compares two files or sets of files and displays the differences
between them

FC [/A] [/C] [/L] [/LBn] [/N] [/OFF[LINE]] [/T] [/U] [/W] [/nnnn]
   [drive1:][path1]filename1 [drive2:][path2]filename2
FC /B [drive1:][path1]filename1 [drive2:][path2]filename2

  /A         Displays only first and last lines for each set of
differences.
  /B           Performs a binary comparison.
  /C           Disregards the case of letters.
  /L           Compares files as ASCII text.
  /LBn      Sets the maximum consecutive mismatches to the
specified number of lines.
  /N           Displays the line numbers on an ASCII comparison.
  /OFF[LINE] Do not skip files with offline attribute set.
  /T           Does not expand tabs to spaces.
  /U           Compare files as UNICODE text files.
  /W           Compresses white space (tabs and spaces) for
comparison.
  /nnnn        Specifies the number of consecutive lines that must
match after a mismatch.
  [drive1:][path1]filename1
      Specifies the first file or set of files to compare.
  [drive2:][path2]filename2
      Specifies the second file or set of files to compare.
```

Figure 1. Arguments and functionality of fc.exe

The input description in PICT-compatible format for **fc.exe** is depicted in Figure 2. Each row of the input model corresponds to a parameter that influences the functionality of **fc.exe**. The model covers all possible arguments that a user of this utility can provide (which are shown in Figure 1) and contains seven factors with two possible values each, one factor with three possible values, one with five and two factors with nine values.

To summarize this model complexity, we will use the following notation: $2^7\ 3^1\ 5^1\ 9^2$.

```
BOUNDARIES: <empty>, /A
COMPARISON: <empty>, /B, /L
CASE_INSENSITIVE: <empty>, /C
MAX_MISMATCH: <empty>, /LB0, /LB1, /LB2, /LB10
LINE_NUMBERS: <empty>, /N
OFFLINE: <empty>, /OFF
TABS_TO_SPACES: <empty>, /T
COMPARE_AS_UNICODE: <empty>, /U
COMPRESS_WHITE: <empty>, /W
FILE1: 1.txt,2.txt,3.txt,4.txt,5.txt,6.txt,7.txt,8.txt,9.txt
FILE2: 1.txt,2.txt,3.txt,4.txt,5.txt,6.txt,7.txt,8.txt,9.txt
```

Figure 2. Input definition (model) for fc.exe

Functionalities of the other three utilities were similarly examined and modelled.

For each program and its input model 100 possible test suites of combinatorial order *t* were generated, using a PICT feature allowing the user to initialize (seed) the greedy heuristic used for test suite generation. At each iteration, a different seed value was provided. There was no attempt to ensure that chosen seeds resulted in creating 100 unique test suites. This has an effect of mimicking the real-life distribution of test suites resulting from PICT executions.

```
for each software-under-test P
   for t = 1..5       // order of combinations
      for s = 1..100  // generation seed
         Initialize PICT with seed[s]
         Generate t-way covering test suite of P
```

Figure 3. Algorithm for generating test suites for a given SUT

Each of the resulting 500 test suites (per program) was then executed against the software-under-test and statement and branch coverage collected.

Descriptive statistics for each of the programs are included in Tables 1-4. These include counts of test cases generated, branch and statement coverage achieved by test suites for each t=1..5. Each metric is summarized by providing the Minimum, Q1 (first quartile), Median (second quartile), Q3 (third quartile) and the Maximum values. In addition, for branch and statement coverage, these metrics are visualized in box-plots accompanying Tables 1-4. Measures of variance are also provided in the tables, including the spread between Min and Max values and the standard deviation and relative standard deviation of a metric.

Results

Program **attrib.exe** was tested with the configuration of factors and levels: $2^2\ 3^4\ 4^1$ and the results are provided in Table 1. The test suites created for t=1 are all of size 4, which corresponds to the size of the largest factor and a varied number of test cases for each t>1, which is again expected. Both branch and statement coverage appear to be achieving higher maximums with increasing t in line with our intuition. In addition, the minimum coverage achieved by test suites also rises with increasing t at what appears to be a faster rate than the maximum. This results in narrowing of the spread between min and max coverage as t increases; the relative standard deviation remains somewhat stable for all t>2 (0.012-0.016 for branch coverage) and is much lower than the standard deviation of t=1 (0.081).

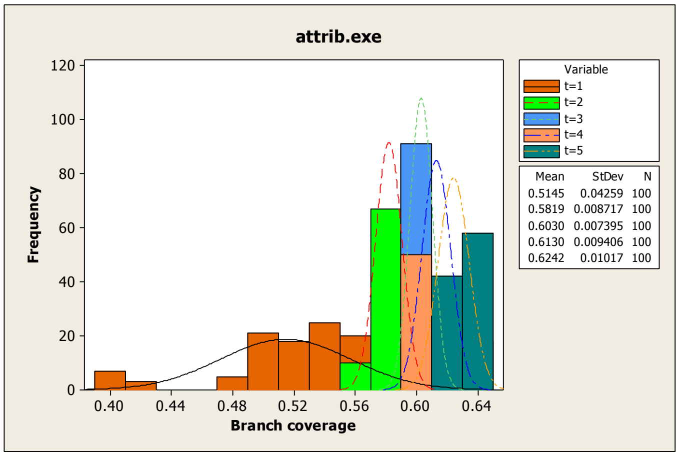


Figure 4. Histogram of branch coverage for attrib.exe

Of note is the fact that even though max coverage for t=4 and t=5 is the same (63.29% and 73.46% for branch and statement coverage respectively), in case of t=5 test suites are much more likely to achieve the maximum coverage.

fc.exe was tested with a $2^7\ 3^1\ 5^1\ 9^2$ configuration of factors. The results are similar to that of **attrib.exe** with t=1 having large coverage variance relative to t>2. Unlike **attrib.exe**, test suites created for **fc.exe** achieve maximum coverage at t=2 (68.09% and 75.60%) and it doesn't change with t increasing beyond 2. The minimum coverage has an overall rising trend with t=4 being an exception with a single test suite achieving 66.61% branch coverage and the next being in line with the lowest for t=3 (66.70%). Interestingly, the statement coverage of this single "outlier" t=4 test suite did not experience the same drop in coverage.

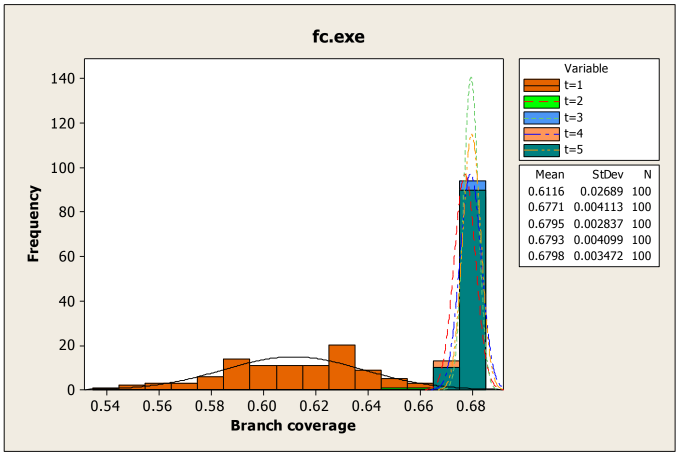


Figure 5. Histogram of branch coverage for fc.exe

find.exe was tested with a 2^4 4^1 18^1 configuration of factors and exhibits the same general properties as other programs however one should view the results for **find.exe** as unusual. We observe that branch coverage generated by 500 test suites for **find.exe** assumes only 6 possible values whereas for **attrib.exe** there are 98 possible unique coverage percentages, 80 for **fc.exe**, and 121 for **findstr.exe**.

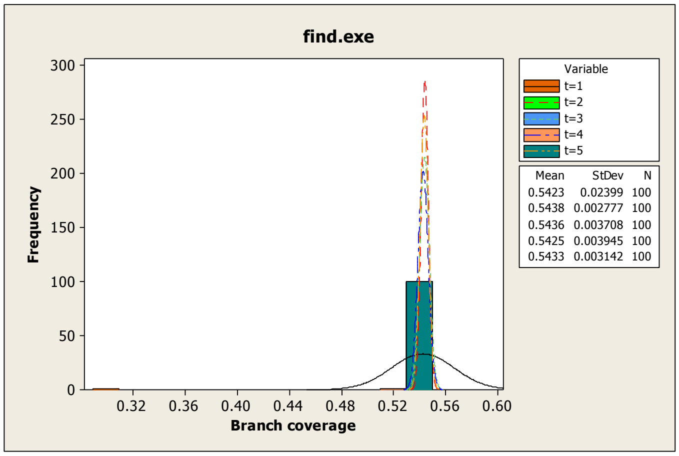


Figure 6. Histogram of branch coverage for find.exe

In case of **find.exe**, it is t=2 that exhibits higher minimum coverage than both t=3 and t=4. Considering the "difficulty" in which lower than maximum coverage can be achieved for this program (Q1 is equal to maximum branch and statement coverage for all values of t), it is likely this is a result of a specific random draw of the 100 test suites for t=2.

Lastly, **findstr.exe** was tested with 2^1 3^4 4^1 5^2 25^1 configuration of factors. General trends remain the same however in case of t=3, the minimum branch and statement coverage are significantly lower than the same metrics for t=2. Moreover, the number of t=3 test suites with lower coverage than the minimum for t=2 is also significant at 24 out 100, contributing to large variability of coverage for t=3. This result is counter-intuitive.

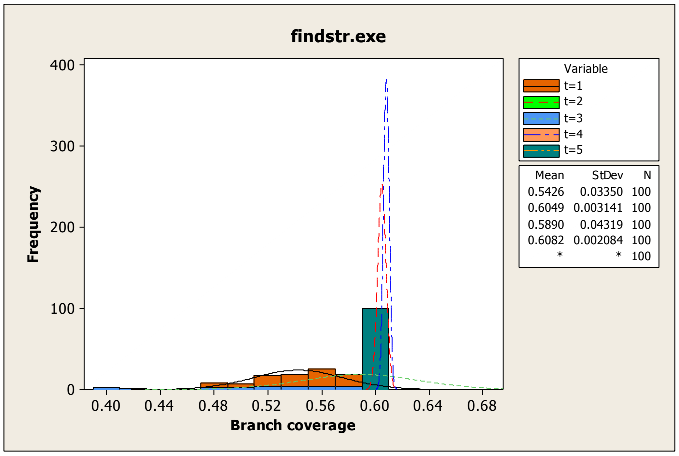


Figure 7. Histogram of branch coverage for findstr.exe

Discussion

Question 1: For a given SUT, how does code coverage change with t?

Question 2: What is the range of coverage achieved by test suites generated for each strength t? Does code coverage variance change with t?

Our intuition is that test suite sizes, even though growing significantly with t in absolute terms will also exhibit larger *spreads* between minimum and maximum test suite size with growing t. However they might exhibit at least stable, if not lower, *relative variance* of test suites sizes within each set of 100. In addition, for coverage, both the minimum and the maximum coverage achieved by test suites created for each t should increase with t. The maximum coverage increases should become smaller (in absolute as well as relative terms) with growing t, which is consistent with diminishing returns of testing with test suites created for larger values of t. Lastly, our intuition is that both branch and statement coverage achieve at least the same or, in significantly many cases, higher maximums with increasing t for all tested programs.

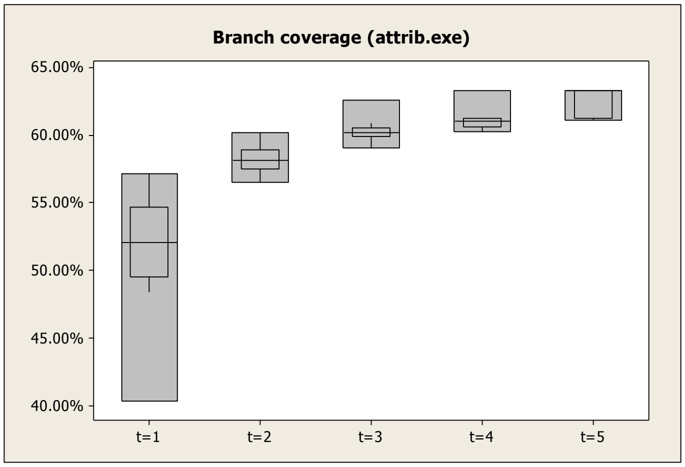
attrib.exe and **fc.exe** seem to conform best to our intuition and exhibit the theorized behavior in all aspects. The absolute size difference between the smallest and the largest test suite for a given t, increases with t but that is a side-effect of having much larger test suites as t increases. Compare for example test suites sizes for t=2 and t=5 for **attrib.exe**. In t=2 case, the range of sizes is 15 to 18 (spread of 3) and for t=5 it is 351 to 373 test cases (spread of 22). However, the *relative standard deviations* exhibit a downward trend with the increasing t suggesting relative tightening of the range of possible test suites sizes as one attempts to increase t.

In addition to the increases in maximum coverage, the minimum coverage achieved by test suites for these programs also rises with increasing t and, at what appears to be a faster rate than the maximum. This results in narrowing of the spread between minimum and maximum coverage as t increases; the relative standard deviation remains somewhat stable for all t>2 (0.012-0.016 for **attrib.exe**'s branch coverage) and much lower than the standard deviation for t=1 (0.081).

Of note is the fact that even though the maximum coverage for t=4 and t=5 is the same (63.29% and 73.46% for branch and statement coverage respectively), in case of t=5 test suites are much more likely to achieve the maximum coverage. In other words, for t=4 achieving the maximum coverage of 73.46% is still unusual while it is the norm for t=5 suites.

Table 1. Descriptive statistics for attrib.exe

	Test Suite Size					Branch Coverage					Statement Coverage				
	t=1	t=2	t=3	t=4	t=5	t=1	t=2	t=3	t=4	t=5	t=1	t=2	t=3	t=4	t=5
Min	4	15	51	149	351	40.34%	56.52%	59.06%	60.27%	61.11%	50.55%	67.77%	69.35%	70.30%	71.09%
Q1	4	16	53	152	359	49.61%	57.49%	59.90%	60.63%	61.23%	61.93%	68.68%	70.30%	70.62%	71.09%
Median	4	16	54	154	363	52.05%	58.15%	60.14%	61.05%	63.29%	63.74%	69.35%	70.46%	71.09%	73.46%
Q3	4	16	55	156	365	54.71%	58.94%	60.51%	61.23%	63.29%	66.39%	69.83%	71.09%	71.09%	73.46%
Max	4	18	58	159	373	57.13%	60.14%	62.56%	63.29%	63.29%	68.88%	71.72%	73.46%	73.46%	73.46%
Spread	0	3	7	10	22	0.16787	0.03623	0.03502	0.03019	0.02174	0.18325	0.03949	0.04107	0.03160	0.02370
StdDev	0.0000	0.7410	1.6792	2.5976	4.6027	0.04238	0.00867	0.00736	0.00936	0.01012	0.04548	0.00827	0.00876	0.01044	0.01170
RelStdDev	0.0000	0.0463	0.0311	0.0169	0.0127	0.08142	0.01491	0.01223	0.01533	0.01599	0.07135	0.01192	0.01244	0.01469	0.01592



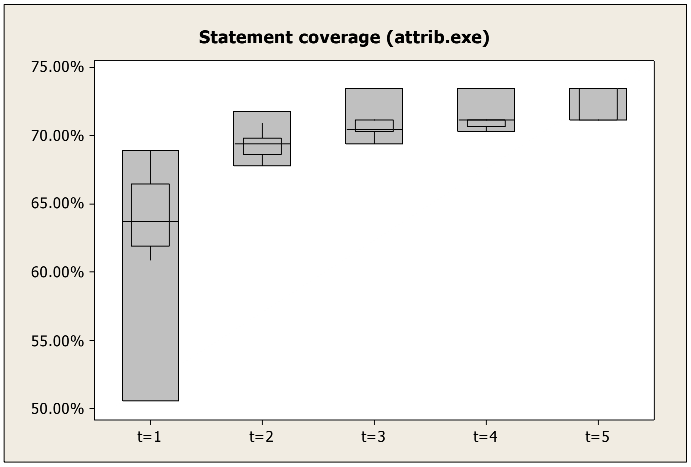
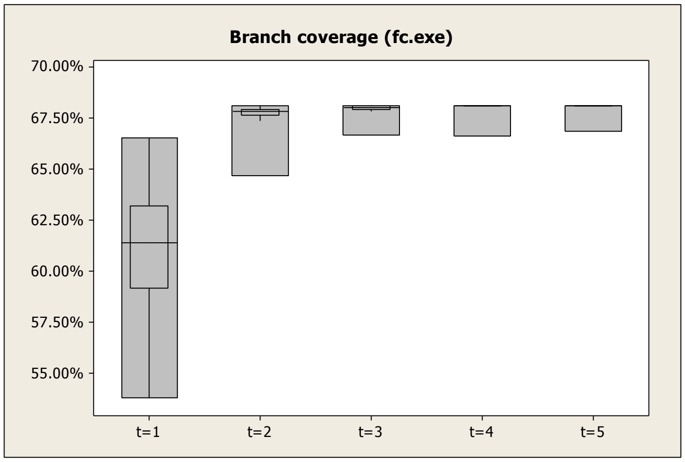


Table 2. Descriptive statistics for fc.exe

	Test Suite Size					Branch Coverage					Statement Coverage				
	t=1	t=2	t=3	t=4	t=5	t=1	t=2	t=3	t=4	t=5	t=1	t=2	t=3	t=4	t=5
Min	9	81	405	1240	3579	53.79%	64.69%	66.70%	66.61%	66.87%	63.41%	73.19%	74.45%	74.45%	74.45%
Q1	9	81	405	1247	3597	59.31%	67.65%	67.92%	68.09%	68.09%	69.19%	75.60%	75.60%	75.60%	75.60%
Median	9	81	405	1250	3607	61.42%	67.83%	68.00%	68.09%	68.09%	70.83%	75.60%	75.60%	75.60%	75.60%
Q3	9	81	405	1254	3613	63.21%	67.92%	68.09%	68.09%	68.09%	72.15%	75.60%	75.60%	75.60%	75.60%
Max	9	81	407	1263	3641	66.52%	68.09%	68.09%	68.09%	68.09%	74.91%	75.60%	75.60%	75.60%	75.60%
Spread	0	0	2	23	62	0.12729	0.03400	0.01395	0.01482	0.01221	0.11507	0.02417	0.01151	0.01151	0.01151
StdDev	0.0000	0.0000	0.4440	4.8384	11.5704	0.02676	0.00409	0.00282	0.00408	0.00345	0.02422	0.00308	0.00273	0.00387	0.00345
RelStdDev	0.0000	0.0000	0.0011	0.0039	0.0032	0.04356	0.00603	0.00415	0.00599	0.00507	0.03420	0.00408	0.00361	0.00512	0.00457



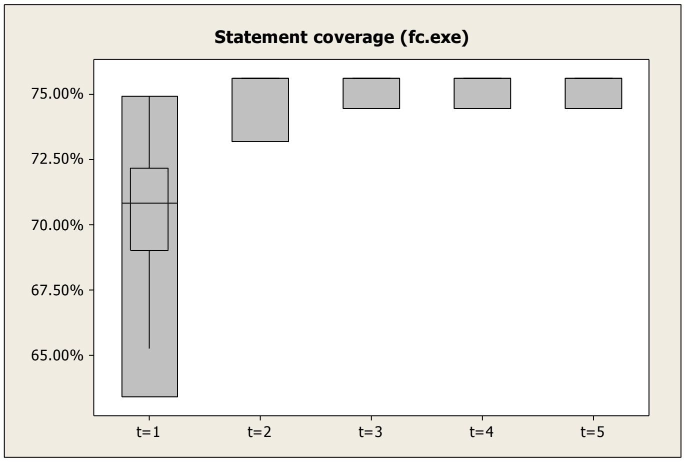
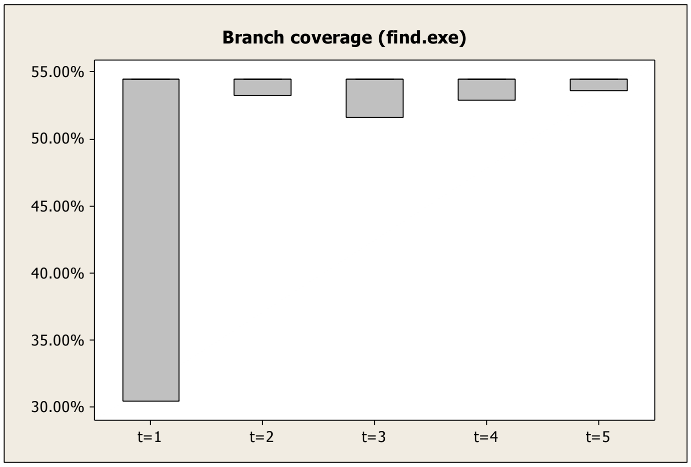


Table 3. Descriptive statistics for find.exe

	Test Suite Size					Branch Coverage					Statement Coverage				
	t=1	t=2	t=3	t=4	t=5	t=1	t=2	t=3	t=4	t=5	t=1	t=2	t=3	t=4	t=5
Min	18	72	161	402	632	30.47%	53.24%	51.66%	52.89%	53.59%	39.02%	62.75%	61.86%	62.53%	62.97%
Q1	18	72	164	408	647	54.47%	54.47%	54.47%	54.47%	54.47%	64.30%	64.30%	64.30%	64.30%	64.30%
Median	18	72	166	411	656	54.47%	54.47%	54.47%	54.47%	54.47%	64.30%	64.30%	64.30%	64.30%	64.30%
Q3	18	72	169	414	663	54.47%	54.47%	54.47%	54.47%	54.47%	64.30%	64.30%	64.30%	64.30%	64.30%
Max	18	72	176	423	682	54.47%	54.47%	54.47%	54.47%	54.47%	64.30%	64.30%	64.30%	64.30%	64.30%
Spread	0	0	15	21	50	0.23993	0.01226	0.02802	0.01576	0.00876	0.25277	0.01552	0.02439	0.01774	0.01330
StdDev	0.0000	0.0000	2.8023	3.8059	11.3067	0.02387	0.00276	0.00369	0.00392	0.00313	0.02515	0.00396	0.00445	0.00578	0.00475
RelStdDev	0.0000	0.0000	0.0169	0.0093	0.0172	0.04383	0.00507	0.00677	0.00721	0.00574	0.03911	0.00616	0.00692	0.00898	0.00739



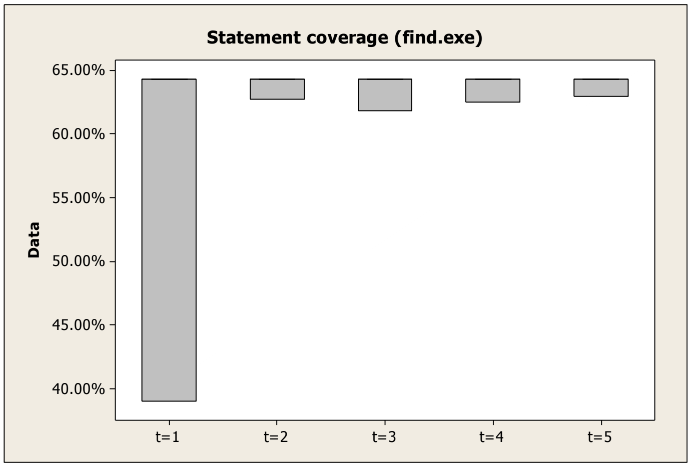
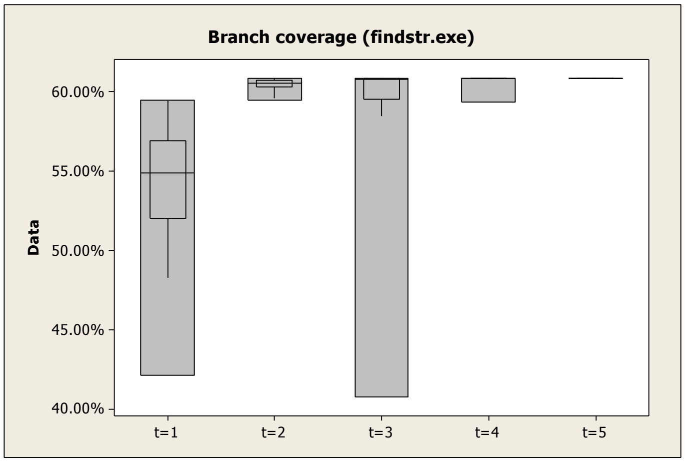
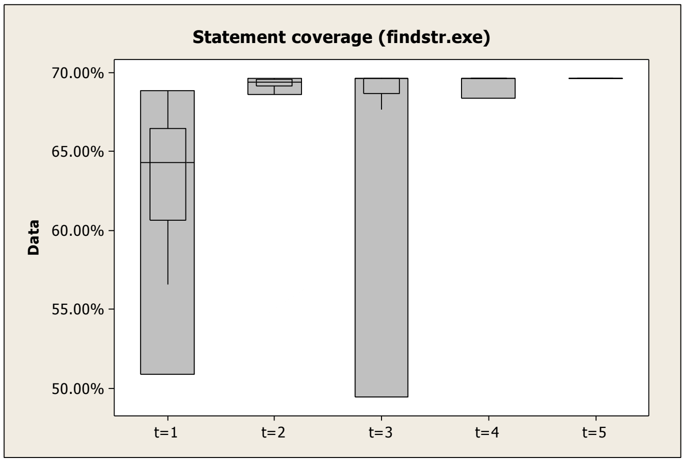


Table 4. Descriptive statistics for findstr.exe

	Test Suite Size					Branch Coverage					Statement Coverage				
	t=1	t=2	t=3	t=4	t=5	t=1	t=2	t=3	t=4	t=5	t=1	t=2	t=3	t=4	t=5
Min	25	125	643	2851	10630	42.10%	59.52%	40.76%	59.38%	60.86%	50.87%	68.63%	49.46%	68.36%	69.64%
Q1	25	125	652	2868	10660	52.06%	60.30%	59.55%	60.86%	60.86%	60.71%	69.22%	68.73%	69.64%	69.64%
Median	25	126	655	2873	10669	54.88%	60.56%	60.84%	60.86%	60.86%	64.31%	69.44%	69.64%	69.64%	69.64%
Q3	25	126	657	2880	10683	56.93%	60.77%	60.86%	60.86%	60.86%	66.44%	69.57%	69.64%	69.64%	69.64%
Max	25	128	664	2902	10730	59.52%	60.86%	60.86%	60.86%	60.86%	68.90%	69.64%	69.64%	69.64%	69.64%
Spread	0	3	21	51	100	0.17421	0.01340	0.20102	0.01479	0.00000	0.18029	0.01005	0.20174	0.01273	0.00000
StdDev	0.0000	0.7102	3.9354	10.4760	21.0278	0.03334	0.00313	0.04297	0.00207	0.00000	0.03549	0.00245	0.04250	0.00159	0.00000
RelStdDev	0.0000	0.0056	0.0060	0.0036	0.0020	0.06075	0.00516	0.07063	0.00341	0.00000	0.05518	0.00353	0.06102	0.00228	0.00000





In case of **fc.exe**, there is one exception to the overall increasing trend and we observe that both the minimum and median coverage of t=4 is lower than that of t=3. However, running a two-tailed t-test on the samples indicates that these metrics are not statistically different at 95% confidence. The same way, the difference between medians for t=4 and t=5 are not statistically significant. Nevertheless, these may not preclude an observer of a much smaller sample (a single test suite per t for a given program in the worst case) to conclude otherwise.

As we mentioned before, the results for **find.exe** are unusual in general with only a small number of coverage percentages around which the program execution coalesces. This possibly indicates that the configuration of factors and levels is not penetrating the program's functionality deep enough, stopping execution too early for any meaningful testing to occur. Such indications would be very easy to miss when testing with only one test suite per t as the statement coverage of **find.exe** could be considered "good enough" at 64%. This aspect requires further study.

Low cardinality of a set of achieved total coverage rates may indicate that the description of factors and levels is not adequately reflecting the complexity of the program.

With the qualification that we might be performing only shallow testing on **find.exe**, the results still conform to majority of the intuitions regarding changes in test suites sizes and coverage. Also here, as it is the case for **fc.exe**, the *maximum* coverage achieved by t=3..5 is not different than t=2. However, increasing frequency of achieving higher-order combinations with increasing t mean that the coverage minima and coverage averages are generally higher for higher t. It is likely a side-effect of increasing test suite size with increasing t.

findstr.exe exhibits an aberration, already mentioned earlier, of t=3 test suites having a large portion of the resulting test suites having a significantly lower minimum than any t, including t=1 (!). The case is made stronger when perusing the detailed coverage data for **findstr.exe**. For t=3, even with 654 tests on average, we managed to produce 2 test suites with 40.76% branch and 49.46% statement coverage which is substantially lower than the maximum achieved by a t=3 suite of 60.86% and 69.64% branch and statement respectively. Moreover, such low coverage is well below the minimums achieved by t=2 and somewhat lower than any of the t=1 suites.

This indicates that there might be cases where a relatively high strength t of a combinatorial design does not guarantee high penetration of the functionality of the program. One has to be aware of this phenomenon and take preventing steps; practitioners would be well-served by trying several different test suites of nominally the same strength t as to avoid such degenerate cases.

If possible, many different test suites should be created from the same domain description and for the same t to avoid accidentally stumbling into an artificially low coverage situation.

Question 3: Is code coverage achieved by t=k and t=k+1 statistically distinguishable in all (or even most) cases?

attrib.exe - branch coverage

t=1	different	different	different	different
	t=2	different	different	different
		t=3	different	different
			t=4	different
				t=5

attrib.exe - statement coverage

t=1	different	different	different	different
	t=2	different	different	different
		t=3	different	different
			t=4	different
				t=5

fc.exe - branch coverage

t=1	different	different	different	different
	t=2	different	different	different
		t=3	same	same
			t=4	same
				t=5

fc.exe - statement coverage

t=1	different	different	different	different
	t=2	same	same	same
		t=3	same	same
			t=4	same
				t=5

findstr.exe - branch coverage

t=1	different	different	different	different
	t=2	different	different	different
		t=3	different	different
			t=4	same
				t=5

findstr.exe - statement coverage

t=1	different	different	different	different
	t=2	different	different	different
		t=3	different	different
			t=4	same
				t=5

Table 5. Results of comparing medians of coverage achieved by t-way test suites

Comparing medians of coverage for each program and t-way test suites reveals that in many cases, the median coverage for t=k is not statistically different than t=k+1 and often even t=k+2. Since the results for **find.exe** indicate it is an abnormal case with likely low quality of testing provided by the chosen test suite, we are excluding **find.exe** from this analysis.

The results, which are summarized in Table 5, suggest that when generating t-way test suites, there's a clear benefit to moving from t=1 to t=2 as far as coverage improvement. In some cases, the same benefit persist as t increases beyond 2 however it seems to have diminishing returns. The exact cut off point is program specific.

The benefit of extra coverage must be evaluated against the much increased size of test suites as one attempts higher values of t. A randomly generated test suite at t=5 may be indistinguishable in terms of coverage from a suite with t=4 despite significantly higher cost to execute.

Question 4: Are there qualitative and quantitative differences between non-trivial t-way test designs (t=2 and higher) and simpler test strategies like *each-choice* (t=1).

The specific configuration of resulting test cases, even though nominally covering the same levels of all factors heavily influences the branch and statement coverage numbers for t=1. In all studied cases however the "low coverage outliers" are a smaller proportion of all generated test suites and often the coverage difference between Min and Q1 is higher than Q1 – Max indicating a skew: a large concentration of low-coverage outliers below the 25th percentile of test suites.

The min-max spread of achieved coverage percentages for each t, exhibits a downward trend. Also, there is a pronounced difference in variance between t=1 (each-choice) test suites and t>=2 (pairwise and higher orders) with t>=2 having substantially lower variance and maintaining relative stability as t increases above 2.

When test suites with t=1 (each-choice) are used, they *must* be varied across executions to prevent low coverage test suites to be used accidentally.

With one exception (**findstr.exe,** t=3), starting at t=2 the variance of coverage from resulting test suites narrows significantly. The relative standard deviations of coverage for t>1 is an order of magnitude lower than that of t=1. However, suites generated for t=2, 3, 4, and 5 have comparable relative standard deviations.

Test suites with t>1 provide significantly more stable levels of coverage than t=1. t=2 is the minimum that should be attempted for all programs.

Question 5: Is increase of coverage attributable to increasing t or simply to the fact that more test cases get run for test suites with higher t?

Figure 8 depicts the relationships between test suites sizes and achieved branch coverage. Omitted here for sake of brevity are charts depicting the relationship between size and statement coverage; branch and statement coverage are very highly correlated and convey the same results.

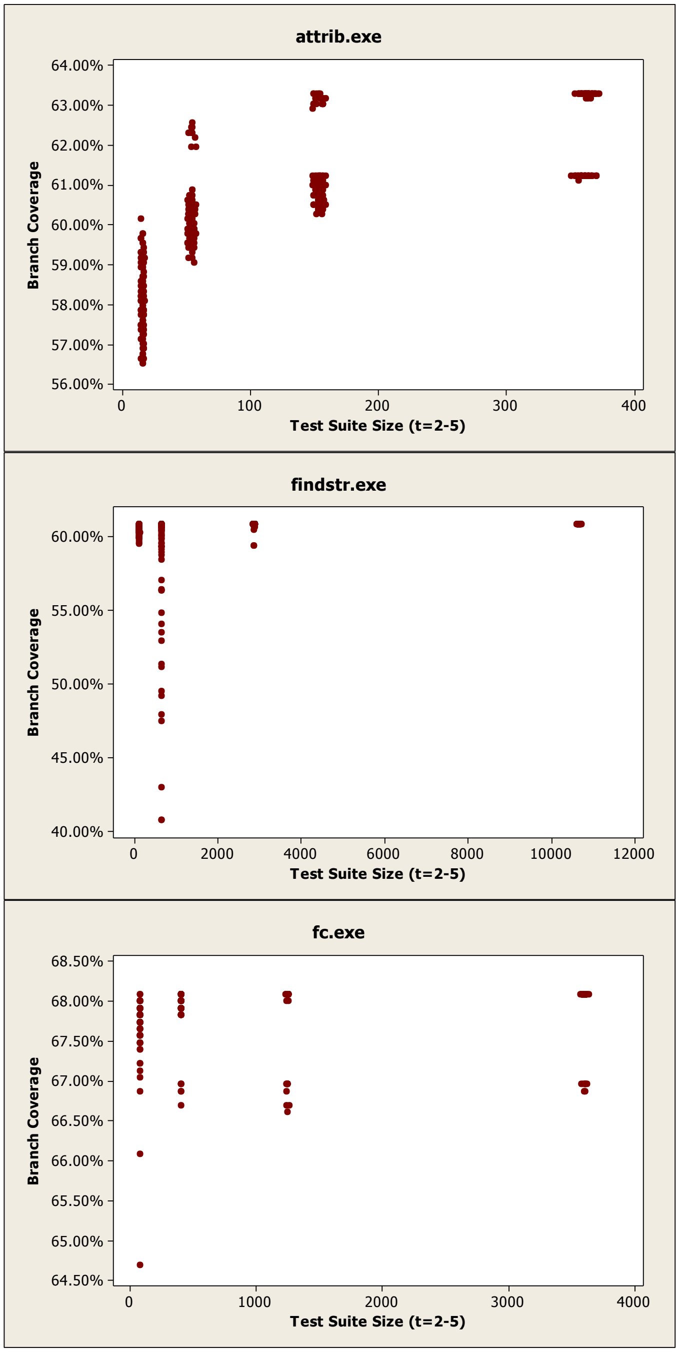
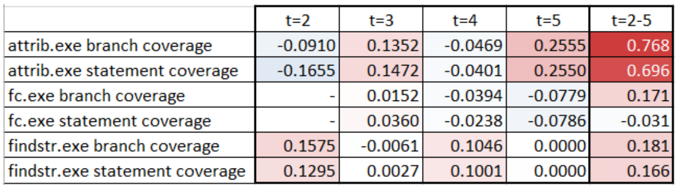


Figure 8. Scatterplots of achieved branch coverage relative to the test suite size.

In addition, Figure 9 quantitatively summarizes correlations between test suite size and coverage *within* each category determined by t and also overall relationship of coverage with test suite size when all results for t=2-5 are considered together.



	t=2	t=3	t=4	t=5	t=2-5
attrib.exe branch coverage	-0.0910	0.1352	-0.0469	0.2555	0.768
attrib.exe statement coverage	-0.1655	0.1472	-0.0401	0.2550	0.696
fc.exe branch coverage	-	0.0152	-0.0394	-0.0779	0.171
fc.exe statement coverage	-	0.0360	-0.0238	-0.0786	-0.031
findstr.exe branch coverage	0.1575	-0.0061	0.1046	0.0000	0.181
findstr.exe statement coverage	0.1295	0.0027	0.1001	0.0000	0.166

Figure 9. Correlation between the size of generated test suites and coverage

Within each category determined by the parameter t, we don't find a correlation between the size of a test suite and achieved coverage for any of the programs. In other words, no matter how varied test suite sizes are for each t, the achieved coverage has little to do with the size of the resulting suite.

The results of quantifying the same relationship for t=2-5 as a whole are more mixed. For **fc.exe** and **findstr.exe**, there isn't any correlation between the size of test suite and achieved coverage. This is consistent with the results shown in Table 5 in that it supports the claim that there are not statistically significant differences in branch coverage achieved by **fc.exe** for t=3-5 and by findstr.exe for t=4-5; no matter the test suite size.

For **attrib.exe**, we find evidence of correlation between the size of a test suite and coverage when t=2-5 are evaluated together. This result suggests a possibility that **at least for some programs, it is not the t-wise fault model but rather the ability of the t-wise methods to generate large and diversified test suites that enables achieving greater coverage**.

Threads to Validity

This study reflects the results collected when testing four relatively similar utilities. All are command-line tools with small to medium size and complexity of code. All models were developed using black-box approach i.e. by examining the inputs of the programs and not their implementation neither was the environment in which they run modelled in any way.

In addition, even though, wherever appropriate, a number of interactions was excluded from models by using constraints, there was no attempt to model negative testing scenarios. This study is focused on two often used code coverage metrics, namely statement and branch coverage and even though the input models used reflect real-world test configurations for each, they are dependent on a skill of the domain modeler.

Conclusions and Future Work

The primary criterion of combinatorial test designs, namely to cover all t-way combinations, does not preclude from generating test suites that result in differences in achieved code coverage. Researchers and practitioners must realize this fundamental property and adjust expectations accordingly. In addition, when assessing effectiveness of combinatorial testing with coverage metrics using methodologies that might result in creating variable test suite sizes, it would be prudent to generate many t-way test suites and report the distributions of coverage as to provide to the reader more precise information about the stability of achieved coverage.

As a result of this study, we found and described several properties of the combinatorial test generation technique that are related to the fact that many different and still conforming to t-wise criterion test suites, can be generated for the same description of the SUT. We included general advice for researchers and practitioners that stem from these findings.

An attempt to quantify if the same effects are exhibited when using "defects found" as a measure of test suite effectiveness is left as possible future work. As well as the matter of understanding the exact relationship between sizes of test suites generated by t-wise methods and coverage; including identifying the conditions under which this relationship supports the claim of t-wise methods of being more efficient and effective than other techniques like random testing. This study can also be extended to cover negative testing scenarios which could help better understand the coverage properties of testing the error paths in relation to the combinatorial strength of test suites.

In addition, it will be advantageous to conclusively determine reasons of the large coverage variance exhibited by some of the test targets and reasons for sometimes generating low coverage suites that nominally still cover all t-way combinations.

Lastly, following the finding that achieved levels of coverage for one of the programs varied very little in comparison to other programs, a follow up study should explore this attribute as a more general measure of the input model quality.

References

[1] http://www.pairwise.org/tools.asp.

[2] P. E. Ammann and A. J. Offutt. Using formal methods to derive test frames in category-partition testing. In Ninth Annual Conference on Computer Assurance (COMPASS'94), Gaithersburg MD, pages 69–80, 1994.

[3] J. Bach and P. Shroeder. Pairwise testing - a best practice that isn't. In Proceedings of the 22nd Pacific Northwest Software Quality Conference, pages 180–196, 2004.

[4] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation,
and test coverage. In Proceedings of the International Conference on Software Testing, Analysis, and Review (STAR), San Diego CA, 1998.

[5] K. Burroughs, A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In Proceedings of the IEEE International Conference on

Communications (Supercomm/ICC'94), May 1-5, New Orleans, Louisiana, pages 745–752, 1994.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. IEEE Transactions On Software Engineering, 23(7), 1997.

[7] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. IEEE Software, 13(5):83–87, 1996.

[8] C. J. Colbourn, M. B. Cohen, and R. C. Turban. A deterministic density algorithm for pairwise interaction coverage. In Proceedings of the IASTED International Conference on Software Engineering, 2004.

[9] S. R. Dalal, A. J. N. Karunanithi, J. M. L. Leaton, G. C. P. Patton, and B. M. Horowitz. Model-based testing in practice. In Proceedings of the International Conference on Software Engineering (ICSE 99), New York, pages 285–294, 1999.

[10] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In Proceedings of the International Conference on Software Engineering (ICSE 97), New York, pages 205–215, 1997.

[11] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies - a survey. GMU Technical Report, 2004.

[12] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. Discrete Mathematics, 284(1-3):149–56, 2004.

[13] R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In Proceedings of the 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center, 2002.

[14] Y. Lei and K. C. Tai. In-factor-order: a test generation strategy for pairwise testing. In Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium, pages 254–261, 1998.

[15] Y. K. Malaiya. Antirandom testing: getting the most out of black-box testing. In Sixth International Symposium on Software Reliability Engineering, Oct. 24-27, 1995, pages 86–95, 1996.

[16] R. Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. Communications of the ACM, 28(10):1054–1058, 1985.

[17] G. J. Myers. The Art of Software Testing. John Wiley and Sons, 1978.

[18] G. Sherwood. Effective testing of factor combinations. In Proceedings of the Third International Conference on Software Testing, Analysis and Review, Washington, DC, pages 133–166, 1994.

[19] H. Shimokawa and S. Satoh. Method of setting software test cases using the experimental design approach. In Proceedings of the Fourth Symposium on Quality Control in Software Production, Federation of Japanese Science and Technology, pages 1–8, 1984.

[20] B. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in testing the remote agent planner. In Proceedings of AIPS, 2000.

[21] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. IEEE Transactions of Software Engineering, 28(1), 2002.

[22] K. Tatsumi. Test case design support system. In Proceedings of the International Conference on Quality Control (ICQC), Tokyo, 1987, pages 615–620, 1987.

[23] D. R. Wallace and D. R. Kuhn. Failure modes in medical device software: an analysis of 15 years of recall data International Journal of Reliability, Quality and Safety Engineering, 8(4), 2001.

[24] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In Proceedings of the 13th International Conference on Testing Communicating Systems (Test-Com 2000), pages 59–74, 2000.

[25] A. W. Williams and R. L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), page 246, 1996.

[26] J. Czerwonka. Pairwise testing in real world. Practical extensions to test case generators. Proceedings of 24th Pacific Northwest Software Quality Conference, 2006.

[27] J. Yan, J. Zhang. Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing. 30th Annual International Computer Software and Applications Conference (COMPSAC'06), 2006.

[28] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. Communications of the ACM 31, Issue 6. June 1988, pages 676-686.

[29] C. Nie and H. Leung. A survey of combinatorial testing. ACM Computing Surveys, Volume 43 (2), Article 11, February 2011.