Code Coverage Analysis of Combinatorial Testing

Eun-Hye Choi*, Osamu Mizuno†, Yifan Hu†
* National Institute of Advanced Industrial Science and Technology (AIST), Ikeda, Japan
Email: e.choi@aist.go.jp
† Kyoto Institute of Technology, Kyoto, Japan
Email: o-mizuno@kit.ac.jp, y-hu@se.is.kit.ac.jp

***Abstract*—Combinatorial *t*-way testing with small *t* is known as an efficient black-box testing technique to detect parameter interaction failures. So far, several empirical studies have reported the effectiveness of *t*-way testing on fault detection abilities. However, few studies have investigated the effectiveness of *t*-way testing on *code coverage*, which is one of the most important coverage criteria widely used for software testing. This paper presents a quantitative analysis to evaluate the code-coverage effectiveness of *t*-way testing. Using three open source utility programs, we compare *t*-way testing with exhaustive (all combination) testing w. r. t. code coverage and test suite sizes.**

***Keywords*-Combinatorial testing; *t*-way testing; Exhaustive testing; Code coverage; Line coverage; Branch coverage.**

I. Introduction

Combinatorial testing [15], [20] is a common black-box testing to detect failures caused by parameter interactions. Modern software systems have a lot of parameters, and thus their interactions are too numerous to be exhaustively tested. Combinatorial *t-way testing* [15], [20], where *t* is called an *interaction strength*, addresses this problem by testing all value combinations of *t* parameters with small *t*, instead of testing all parameter-value combinations exhaustively. *t*-way testing has been applied to e. g. conformance testing for DOM (Document Object Model) Events standard [19], rich web applications [18], commercial MP3 players [25], and a ticket gate system for transportation companies [14].

Kuhn et al. [16] investigated the fault detection effectiveness of *t*-way testing; their result showed that *t*-way testing with small interaction strength t (≤ 4) can efficiently detect most interaction failures while significantly reducing the number of test cases compared to *exhaustive testing*, which tests all parameter-value combinations. Other studies [2], [8], [25] also supported the result by Kuhn et al. [16].

On the other hand, as far as we know, the only work by Giannakopoulou et al. [10] reported the effectiveness of *t*-way testing on *code coverage*. They compared code coverage between their model-checker based exhaustive testing and 3-way testing with two program modules for a NASA air transportation system.

Code coverage, which measures what percentage of source code is executed by a test suite, has been considered as one of the most important coverage metrics for software testing and is required by many industrial software development standards (e. g. [1]). Therefore, the code coverage effectiveness of *t*-way testing would be of big interest to practitioners who consider applying *t*-way testing to their software testing.

Note that *t*-way testing is a black-box testing and thus is difficult to achieve 100% code coverage and its code coverage depends on the *system under test (SUT)* model, e. g. parameters and their values, designed for *t*-way testing. Therefore, in order to evaluate the code coverage effectiveness of *t*-way testing, we compare code coverage obtained by *t*-way testing with that by exhaustive testing, similarly to [10].

In order to quantitatively analyze the code coverage effectiveness of *t*-way testing compared to exhaustive testing, we set up the following two research questions:

- RQ1: How high code coverage can *t*-way testing achieve compared to exhaustive testing? Can *t*-way testing obtain higher code coverage earlier compared to exhaustive testing? How large interaction strength *t* is necessary for *t*-way testing to achieve the code coverage close to that by exhaustive testing?
- RQ2: With the same number of test cases, how different are *t*-way testing and exhaustive testing on code coverage?

For evaluating the code coverage effectiveness of *t*-way testing, RQ1 compares *t*-way testing and exhaustive testing in their original sizes, while RQ2 compares *t*-way testing and exhaustive testing in the same sizes.

To answer the above research questions, we perform a case study that analyzes *t*-way test suites with $1 \leq t \leq 4$ on two kinds of widely used code coverage; *line coverage* (i. e., statement coverage) and *branch coverage* (i. e., decision coverage). For an empirical case study, we use seventeen versions of three C program projects, flex, grep, and make, from the Software-artifact Infrastructure Repository (SIR) [7]. To prepare *t*-way test suites, we first construct SUT models with constraints from test plans in Test Specification Language (TSL) [21] of the repository. We next generate *t*-way test suites for the SUT models using two state-of-the-art *t*-way test generation tools, ACTS [3], [26] and PICT [6], [31]. We evaluate the code coverage effectiveness of *t*-way testing by comparing the *t*-way test suites and exhaustive test suites on the examining line coverage and branch coverage with test suite sizes.

Paper Organization: Section II-A explains combinatorial *t*-way testing and Section II-B describes related work on the effectiveness evaluation of *t*-way testing. Section III describes our experimental setting, and Section IV explains experimental results which answer the research questions. Section V concludes this paper.

TABLE I
AN EXAMPLE SUT MODEL.

Parameter	*Values*
Debug mode (= p_1)	on, off
Bypass use (= p_2)	on, off
Fast scanner (= p_3)	FastScan (= 1), FullScan (= 2), off
Constraint: (Fast scanner = FullScan) → (Bypass use ≠ on)	

TABLE II
AN EXAMPLE OF ALL POSSIBLE PAIRS OF PARAMETER-VALUES.

Param. pairs	*Parameter-value pairs*
(p_1, p_2)	(on, on), (on, off), (off, on), (off, off)
(p_1, p_3)	(on, 1), (on, 2), (on, off), (off, 1), (off, 2), (off, off)
(p_2, p_3)	(on, 1), (on, off), (off, 1), (off, 2), (off, off)

TABLE III
A 2-WAY TEST SUITE $\mathcal{T}_1$.

	p_1	p_2	p_3
1	on	on	1
2	on	off	off
3	off	off	1
4	off	on	off
5	off	off	2
6	on	off	1

TABLE IV
AN EXHAUSTIVE TEST SUITE $\mathcal{T}_2$.

	p_1	p_2	p_3
1	on	on	1
2	on	on	off
3	on	off	1
4	on	off	2
5	on	off	off
6	off	on	1
7	off	on	off
8	off	off	1
9	off	off	2
10	off	off	off

TABLE V
RELATED WORK.

	Metrics studied	
	Code coverage	Fault detection
Kuhn et al. (2004) [16]		✓
Zhang et al. (2012) [25]		✓
Petke et al. (2015) [22]		✓
Henard et al. (2015) [11]		✓
Choi et al. (2016) [4]		✓
Giannakopoulou et al. (2011) [10]	✓	
This paper	✓	

II. BACKGROUND AND RELATED WORK

A. Combinatorial t-way Testing

The *System Under Test (SUT)* for combinatorial testing is modeled from parameters, their associated values from finite sets, and constraints between parameter-values. For instance, the SUT model shown in Table I, has three parameters (p_1, p_2, p_3); the first two parameters have two possible values and the other has three possibilities. Constraints among parameter-values exist when some parameter-value combinations cannot occur. The example SUT has a constraint such that $p_2 \neq on$ if $p_3 = 1$, i. e., the combination of $p_2 = on$ and $p_3 = 1$ is not allowed.

More formally, a model of an SUT is defined as follows:

Definition 1 (SUT model)**.** *An* SUT model *is a triple* $\langle P, V, \phi \rangle$*, where*

- P *is a finite set of parameters* $p_1, \ldots, p_{|P|}$*,*
- V *is a family that assigns a finite value domain* V_i *for each parameter* p_i $(1 \leq i \leq |P|)$*, and*
- ϕ *is a constraint on parameter-value combinations.*

A *test case* is a value assignment for the parameters that satisfies the SUT constraint. For example, a 3-tuple (p_1 *=on,* p_2 *=on,* p_3 *=1*) is a test case for our example SUT model. We call a sequence of test cases a *test suite*.

An *exhaustive test suite* (i. e. *all combination test suite*) is a sequence of all possible test cases, i. e., a test suite that covers all parameter-value combinations satisfying the SUT constraint. In general, exhaustive testing is impractical since it stipulates to test all possible test cases and thus its size (the number of test cases) increases exponentially with the number of parameters.

Combinatorial *t-way testing* (e. g., *pairwise*, when $t = 2$) alternatively stipulates to test all *t-way* parameter-value combinations satisfying the SUT constraint at least once. We call t an *interaction strength*. An exhaustive test suite corresponds to a t-way test suite with $t = |P|$.

Definition 2 (t-way test suite)**.** *Let* $\langle P, V, \phi \rangle$ *be an SUT model. We say that a tuple of* t $(1 \leq t \leq |P|)$ *parameter-values is* possible *iff it does not contradict the SUT constraint* ϕ*. A* t-way test suite *for the SUT model is a test suite that covers all possible* t*-tuples of parameter-values in the SUT model.*

Example 1. *Consider the SUT model in Table I and* $t = 2$*. There exist* 15 *possible t-tuples (pairs) of parameter-values, as shown in Table II. The test suites* $\mathcal{T}_1$ *in Table III is a 2-way (pairwise) test suite since it covers all the possible parameter-value pairs in Table II.* $\mathcal{T}_2$ *in Table IV is a 3-way test suite and corresponds to the exhaustive test suite since the number of parameters in the example model is three .*

Many algorithms to efficiently construct t-way test suites have been proposed so far. Approaches to generate t-way test suites for SUT models with constraints include greedy algorithms (e. g., AETG [5], PICT [6], [31], and ACTS [3], [26]), heuristic search (e. g., CASA [9], HHSA [12], and TCA [17]), and SAT-based approaches (e. g., Calot [23], [24]).

B. Related Work: Effectiveness evaluation of t-way testing

The effectiveness of t-way testing with small interaction strength t on fault detection have been reported by several empirical studies so far [13], [15], but the code coverage of t-way testing has not been studied well. Table V summarizes the effectiveness metrics studied in related work.

Kuhn et al. [16] investigated parameter interactions inducing actual failures of four systems; a software for medical devices, a browser, a server, and a database system. As a result, 29–68% of the faults involved a single parameter; 70–97% (89–99%) of the faults involved up to two (three) parameter interactions; 96–100% of the faults involved up to four and five parameter

interactions; no fault involved more than six parameters. From the result, the authors concluded that most failures are triggered by parameter interactions with small t (at most four to six) and thus t-way testing with $4 \leq t \leq 6$ could provide the fault detection ability of "pseudo-exhaustive" testing.

Zhang et al. [25] also explored that failures of actual commercial MP3 players are triggered by t-way parameter interactions with at most $t = 4$.

Petke et al. [22] more thoroughly studied the efficiency of early fault detection by t-way testing with $2 \leq t \leq 6$. They used six projects, flex, make, grep, gzip, nanoxml, and siena, from the Software artifact Infrastructure Repository (SIR) and showed the number of faults detected after 25%, 50%, 70%, and 100% of test cases are executed.

Henard et al. [11] used five projects, grep, sed, flex, make, and gzip, also from SIR and compared the number of faults detected by test suite prioritization with t-way coverage ($2 \leq t \leq 4$) and other black-box and white-box prioritization.

Choi et al. [4] used three projects, flex, grep, and make, from SIR and investigated a correlation of the fault detection effectiveness with two evaluation metrics, called weight coverage and KL divergence, for prioritized t (= 2)-way testing.

To our knowledge, the only work by Giannakopoulou et al. [10] reported code coverage of t-way testing. Their target system is a component of the Tactical Separation Assisted Flight Environment (TSAFE) of the Next Generation Air Transportation System (NextGen) by the NASA Ames Research Center. In their work, t (= 3)-way testing and their model-checker (JPF [30]) based exhaustive testing are compared w. r. t. code coverage; line coverage, branch coverage, loop coverage, and strict condition coverage, which are computed using CodeCover [28].

Giannakopoulou et al. reported that for two program modules, the differences of code coverage by 3-way testing and exhaustive testing are 0–2% for the four coverage metrics they used, while the numbers of test cases are 6,047 for 3-way testing but 9.9×10^6 for exhaustive testing. In this paper, we more thoroughly analyze the code coverage effectiveness of t-way testing with $1 \leq t \leq 4$ using three open source utility programs.

III. Experiments

A. Subject Programs

To investigate code coverage of t-way testing, we use three open source projects of C programs, flex, grep, and make, from the Software artifact Infrastructure Repository (SIR) [32]. flex is a lexical analysis generator. grep is a program to search for text matching regular expressions. make is a program to control the compile and build process. The programs have been widely used to evaluate testing techniques by researchers in studies including [4], [11], [22]. Table VI shows for each version of programs we use, the version identifier, the year released, and the lines of code (LoC) calculated using cloc [27].

TABLE VI
Subject programs.

Proj.	Ver.	Identifier	Year of release	LoC
flex	v0	2.4.3	1993	10,163
	v1	2.4.7	1994	10,546
	v2	2.5.1	1995	13,000
	v3	2.5.2	1996	13,048
	v4	2.5.3	1996	13,142
	v5	2.5.4	1997	13,144
grep	v0	2.0	1996	8,163
	v1	2.2	1998	11,945
	v2	2.3	1999	12,681
	v3	2.4	1999	12,780
	v4	2.4.1	2000	13,280
	v5	2.4.2	2000	13,275
make	v0	3.75	1996	17,424
	v1	3.76.1	1997	18,525
	v2	3.77	1998	19,610
	v3	3.78.1	1999	20,401
	v4	3.79.1	2000	23,188

```
Parameters:
...
 Debug mode:    # -d
   Debug_on.
   Debug_off.

 Bypass use:    # -Cr
   Bypass_on.   [property Bypass]
   Bypass_off.

 Fast scanner:  # -f, -Cf
   FastScan.    [property FastScan]
   FullScan.    [if !Bypass][property FullScan]
   off.         [property f&Cfoff]
...
```

Fig. 1. A part of the test plan for flex in TSL.

B. Subject Test Suites

1) SUT Models: For each project, flex, grep, and make, we construct an SUT model for t-way testing whose parameters, values, and constraints are fully extracted from the test plan in TSL (Test Specification Language), which is included in SIR. For example, Figure 1 shows a part of the test plan in TSL for project flex. From the TSL specification, we construct the SUT model for flex whose parameters include Debug mode($= p_1$), Bypass use($= p_2$) and Fast scanner($= p_3$), p_2 has two values including Bypass_on($= on$), p_3 has three values including FullScan($= 2$), and constraints include $(p_3 = 2) \rightarrow (p_2 \neq on)$. Table I corresponds to a part of the SUT model for flex, which is constructed from the part of the test plan in Figure 1.

Table VII shows the size of the SUT model constructed for each project. In the table, the size of parameter-values is expressed as $k; g_1^{k_1} g_2^{k_2} \ldots g_n^{k_n}$, which indicates that the number of parameters is k and for each i there are k_i parameters that have g_i values. The size of constraints is expressed as $l; l_1^{h_1} l_2^{h_2} \ldots l_m^{h_m}$, which indicates that the constraint is described in conjunctive normal form (CNF) with l variables whose Boolean value

TABLE VII
CONSTRUCTED SUT MODELS.

Proj.		Model size
flex	Parameter-values	29; $3^{23}4^{4}6^{2}$
	Constraints	97; $2^{712}22^{1}24^{2}25^{17}26^{9}$
grep	Parameter-values	14; $2^{4}3^{1}4^{3}5^{1}6^{1}9^{1}11^{1}13^{1}20^{1}$
	Constraints	87; $2^{433}3^{27}4^{8}7^{5}16^{1}24^{1}27^{1}28^{1}31^{10}$
make	Parameter-values	22; $2^{2}3^{12}4^{4}5^{2}6^{1}7^{1}$
	Constraints	79; $2^{526}21^{1}22^{1}23^{1}24^{3}25^{7}26^{9}$

TABLE VIII
SIZES AND CODE COVERAGE OF EXHAUSTIVE TEST SUITES.

Proj.	Size		Line coverage	Branch coverage
flex	525	Avg.	0.7968	0.8544
		Min.	0.7789	0.8151
		Max.	0.8312	0.9316
grep	470	Avg.	0.4961	0.4948
		Min.	0.4726	0.4746
		Max.	0.5900	0.5826
make	793	Avg.	0.4543	0.5373
		Min.	0.4234	0.5126
		Max.	0.4726	0.5494

represents an assignment of a value to a parameter and for each j there are h_j clauses that have l_j literals. For the example SUT model in Figure 1, the size of parameter-values is $3; 2^2 3^1$ and the size of constraints is $2; 2^1$.

2) Test Suites: We use t-way test suites with $1 \leq t \leq 4$ that are generated by ACTS [3], [26] and PICT [6], [31] for our constructed SUT models with constraints. The tools ACTS and PICT are state-of-the-art open source t-way test generation tools developed by NIST (National Institute of Standards and Technology) and Microsoft, respectively. For comparison, we also use exhaustive test suites each of which obtains all possible test cases. The exhaustive test suite for the test plan of each project is included in SIR.

3) Evaluation Metrics: To evaluate code coverage of each test suite, we analyze the following two kinds of code coverage, which are computed using gcov [29]:

- Line coverage: the percentage of program lines executed.
- Branch coverage: the percentage of branches of conditional statements executed.

gcov is a source code analysis tool, which is a standard utility delivered with the GNU C/C++ Compiler and reports how many lines and branches are executed.

IV. RESULTS

Table VIII shows the size, i. e. the number of test cases, and the code coverage (line coverage and branch coverage) of the exhaustive test suite for each project, while Table IX and Table X show those of the subject t-way test suites ($1 \leq t \leq 4$) generated by ACTS and PICT. Table IX shows the sizes of the subject t-way test suites with the ratio of them over the sizes of exhaustive test suites. Table X summarizes line coverage and branch coverage of the subject t-way test suites for each project.

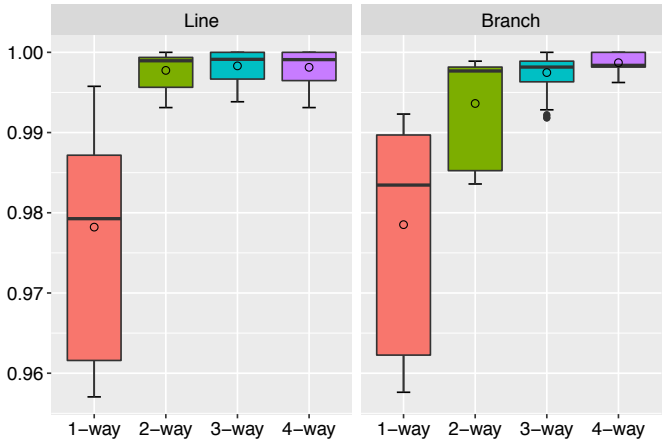

Fig. 2. Ratio of code coverage of t-way testing ($1 \leq t \leq 4$) over that of exhaustive testing for all versions of projects.

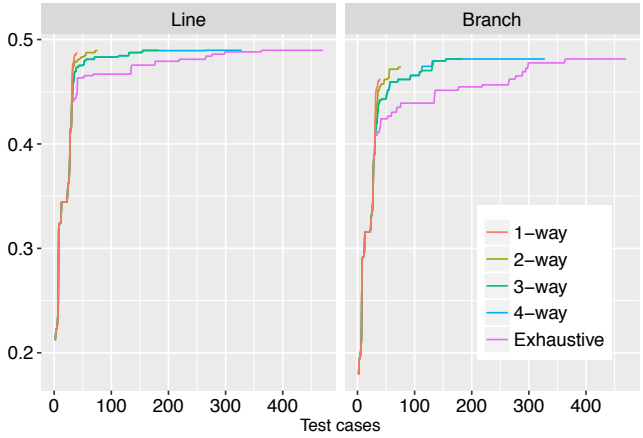

Fig. 3. Example code coverage growths of t-way testing ($1 \leq t \leq 4$) and exhaustive testing for one version (v1) of grep.

In Table VIII and Table X, we show the average, minimum, and maximum values of code coverage for versions of each project.

For example, for project flex, the sizes of 2-way test suites (52 by ACTS and 51 by PICT) are less than 10% of the size of the exhaustive test suite (525) from Table VIII and Table IX. On the other hand, for flex, line coverage and branch coverage of 2-way test suites (the exhaustive test suite) are on average 0.7927 (0.7968) and 0.8522 (0.8544) from Table VIII and Table X.

A. RQ1: t-way testing vs. exhaustive testing

To compare the code coverage between t-way testing and exhaustive testing, we investigate the following metric

$$R_{Cov}(T_t, EX) = Cov(T_t) \ / \ Cov(EX),$$

which denotes the ratio of code coverage of t-way test suite T_t over that of exhaustive test suite EX.

Table XI summarizes the values of $R_{Cov}(T_t, EX)$ with $1 \leq t \leq 4$ for line coverage and branch coverage for each project and

TABLE IX
SIZES OF t-WAY TEST SUITES ($1 \leq t \leq 4$).

Proj.		# of test cases (ratio over # of the exhaustive test cases)							
		1-way		2-way		3-way		4-way	
flex	ACTS	30	(5.71 %)	52	(9.90 %)	91	(17.33 %)	155	(29.52 %)
	PICT	30	(5.71 %)	51	(9.71 %)	90	(17.14 %)	154	(29.33 %)
grep	ACTS	40	(8.51 %)	76	(16.17 %)	183	(38.94 %)	328	(69.79 %)
	PICT	40	(8.51 %)	77	(16.38 %)	180	(38.30 %)	326	(69.36 %)
make	ACTS	27	(3.40 %)	34	(4.29 %)	44	(5.55 %)	68	(8.58 %)
	PICT	27	(3.40 %)	34	(4.29 %)	45	(5.67 %)	69	(8.70 %)

TABLE X
CODE COVERAGE OF t-WAY TEST SUITES ($1 \leq t \leq 4$).

Proj.		Line coverage				Branch coverage			
		1-way	2-way	3-way	4-way	1-way	2-way	3-way	4-way
flex	Avg.	0.7683	0.7927	0.7934	0.7931	0.8407	0.8522	0.8522	0.8522
	Min.	0.7481	0.7755	0.7763	0.7755	0.8018	0.8136	0.8136	0.8136
	Max.	0.8145	0.8264	0.8267	0.8267	0.9225	0.9281	0.9281	0.9281
grep	Avg.	0.4917	0.4959	0.4961	0.4961	0.4769	0.4880	0.4933	0.4948
	Min.	0.4668	0.4723	0.4726	0.4726	0.4549	0.4676	0.4712	0.4746
	Max.	0.5875	0.5897	0.5900	0.5900	0.5726	0.5786	0.5845	0.5826
make	Avg.	0.4451	0.4539	0.4540	0.4540	0.5321	0.5364	0.5366	0.5366
	Min.	0.4168	0.4230	0.4230	0.4230	0.5053	0.5117	0.5117	0.5117
	Max.	0.4628	0.4724	0.4724	0.4726	0.5442	0.5484	0.5484	0.5484

TABLE XI
COMPARISON OF CODE COVERAGE BETWEEN t-WAY TESTING ($1 \leq t \leq 4$) AND EXHAUSTIVE TESTING.

	Proj.	Line coverage				Branch coverage			
		1-way	2-way	3-way	4-way	1-way	2-way	3-way	4-way
Avg. of $R_{Cov}(T_t, EX)$ ($R = Cov(T_t)/Cov(EX)$)	flex	96.41 %	99.49 %	99.58 %	99.54 %	98.40 %	99.74 %	99.74 %	99.74 %
	grep	99.11 %	99.95 %	100.00 %	100.00 %	96.32 %	98.58 %	99.67 %	100.00 %
	make	97.97 %	99.91 %	99.93 %	99.92 %	99.03 %	99.84 %	99.88 %	99.87 %
	Avg.	97.82 %	99.77 %	99.83 %	99.81 %	97.85 %	99.36 %	99.76 %	99.87 %
# ($R \geq 99.5$ %) / # all cases	flex	0 / 12	8 / 12	8 / 12	8 / 12	0 / 12	12 / 12	12 / 12	12 / 12
	grep	4 / 12	12 / 12	12 / 12	12 / 12	0 / 12	0 / 12	7 / 12	12 / 12
	make	0 / 10	10 / 10	10 / 10	10 / 10	0 / 10	10 / 10	10 / 10	10 / 10
	Total	4 / 34	30 / 34	30 / 34	30 / 34	0 / 34	22 / 34	29 / 34	34 / 34

all projects. In the table, we also show the numbers of cases where $R_{Cov}(T_t, EX) \geq 99.5\%$, i. e. t-way testing achieves more than 99.5% of the coverage obtained by exhaustive testing, over the numbers of all cases (versions) for projects.

Figure 2 presents the box plots for the results of $R_{Cov}(T_t, EX)$ for all projects. Each box plot shows the mean (circle in the box), median (thick horizontal line), the first/third quartiles (hinges), and highest/lowest values within 1.5 × the inter-quartile range of the hinge (whiskers).

- How high code coverage can t-way testing achieve compared to exhaustive testing?

From Table XI and Figure 2, we can see that t-way testing with even small t can achieve high values of $R_{Cov}(T_t, EX)$, i. e. high ratios of code coverage over the code coverage of exhaustive testing.

In the result of our case study, 1-way (2-way) testing covers avg. 97.82% (99.77%) of line coverage of exhaustive testing and avg. 97.85% (99.36%) of branch coverage of exhaustive testing. With 3-way (4-way) testing, line coverage is avg. 99.83% (99.81%) and branch coverage is avg. 99.76% (99.87%) of the coverage of exhaustive testing.

- Can t-way testing obtain higher code coverage earlier compared to exhaustive testing?

Figure 3 shows example line coverage growths and branch coverage growths of t-way test suites ($1 \leq t \leq 4$) and the exhaustive test suites for one version (v1) of project grep. (The coverage growths represent the typical cases of our experiment results.) We can see that t-way testing with smaller t obtains higher code coverage earlier compared to exhaustive testing and t-way testing with larger t.

For the example case in Figure 3, to obtain 48% line coverage (46% branch coverage), 1-way, 2-way, 3-way, and 4-way testing respectively require 35, 42, 56, and 56 (36, 47, 71, and 71) test cases, while exhaustive testing requires 219 (265) test cases.

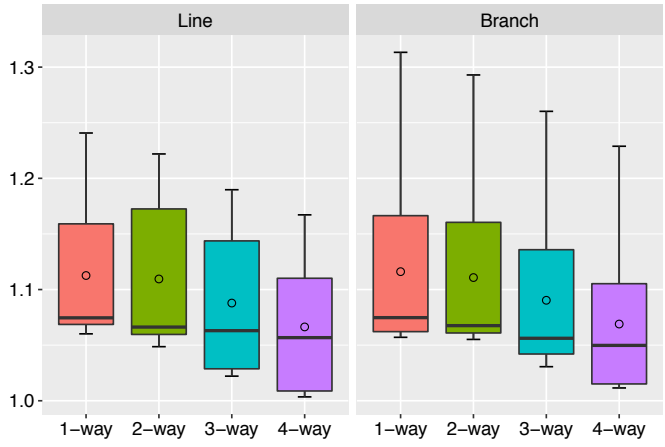

Fig. 4. Ratios of code coverage of t-way testing ($1 \leq t \leq 4$) over that of exhaustive testing with the same sizes for all versions of projects.

- How large t is necessary for t-way testing to achieve the code coverage close to that by exhaustive testing?

Surprisingly, as the result of our case study, all t-way test suites with $1 \leq t \leq 4$ obtain more than 95% of code coverage of exhaustive test suites. Especially, for project grep, 4-way test suites obtain the same line coverage and branch coverage with the exhaustive test suite. As described in Table XI, in 30 cases of all 34 cases, 2-way, 3-way, and 4-way test suites achieve more than 99.5% of line coverage of exhaustive test suites. For branch coverage, in all cases, 4-way test suites achieve more than 99.5% of coverage of exhaustive test suites.

> *From the results, t-way testing with small t ($1 \leq t \leq 4$) can efficiently obtain code coverage close to that by exhaustive testing while requiring smaller test cases.*

B. RQ2: t-way testing vs. exhaustive testing in the same sizes

To compare the code coverage between t-way testing and exhaustive testing with the same sizes, we investigate the following metric

$$R_{Cov}(T_t, EX*) = Cov(T_t) \ / \ Cov(EX*),$$

which denotes the ratio of code coverage of t-way test suite T_t over that of a subset, hereafter denoted by $EX*$, of exhaustive test suite EX whose size is same with T_t. In our experiments, we constructed $EX*$ 100 times by randomly selecting $|T_t|$ test cases from exhaustive test suite EX and use the average value of the code coverage for the 100 $EX*$.

Table XII summarizes the values of $R_{Cov}(T_t, EX*)$ with $1 \leq t \leq 4$ for line coverage and branch coverage for each project and all projects. In the table, we also show the numbers of cases where $R_{Cov}(T_t, EX*) \geq 105\%$, i. e. t-way testing achieves more than 105% of the coverage obtained by exhaustive testing with the same size, over the numbers of all cases for projects. Figure 4 presents the box plots for the results of $R_{Cov}(T_t, EX*)$ for all projects.

- With the same number of test cases, how different are t-way testing and exhaustive testing on code coverage?

From Table XII and Figure 4, we can see that t-way test suites with $1 \leq t \leq 4$ achieve higher line coverage and branch coverage compared to exhaustive test suites in the same sizes. Especially, t-way testing with smaller t obtains higher values of $R_{Cov}(T_t, EX*)$, i. e. higher ratios of code coverage over that of exhaustive testing in the same size.

As described in Table XII, for all cases, 1-way and 2-way testing achieve more than 105% of code coverage of exhaustive testing with the same size. For 3-way and 4-way testing, the numbers of cases that achieve more than 105% of line (branch) coverage of exhaustive testing with the same sizes are 24 and 20 (24 and 16) cases among all 34 cases.

> *From the results, t-way testing with smaller t can obtain higher code coverage compared to exhaustive testing with the same number of test cases.*

V. Conclusion

This paper analyzes the code coverage effectiveness of combinatorial t-way testing with small t. As a result of our empirical evaluation using a collection of open source utility programs, t-way testing with small t ($1 \leq t \leq 4$) efficiently covers more than 95% of code coverage achieved by exhaustive testing, while requiring much smaller test cases. In addition, comparing in the same test suite sizes, t-way testing with smaller t obtains higher ratio of code coverage over that by exhaustive testing.

In this paper, we evaluate two kinds of widely used code coverage metrics, line coverage and branch coverage. Further work includes evaluating other metrics such as loop coverage, condition coverage, etc. Another further work is to investigate both the code coverage effectiveness and the fault detection effectiveness of t-way testing and analyze the relation between them on real software projects.

Acknowledgments

The authors would like to thank anonymous referees for their helpful comments and suggestions to improve this paper. This work was partly supported by JSPS KAKENHI Grant Number 16K12415.

References

[1] International Standardization Organization, ISO26262: Road vehicles - Functional safety, November 2011.

[2] K. Z. Bell and M. A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. In *Proc. of International Conference on Information and Communication Technology*, pages 221–235. IEEE, 2005.

[3] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial testing of ACTS: A case study. In *Proc. of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 591–600. IEEE, 2012.

[4] E. Choi, S. Kawabata, O. Mizuno, C. Artho, and T. Kitamura. Test effectiveness evaluation of prioritized combinatorial testing: a case study. In *Proc. of the International Conference on Software Quality, Reliability & Security (QRS)*, pages 61–68. IEEE, 2016.

TABLE XII
COMPARISON OF CODE COVERAGE BETWEEN t-WAY TESTING ($1 \leq t \leq 4$) AND EXHAUSTIVE TESTING WITH THE SAME SIZES.

	Proj.	Line coverage				Branch coverage			
		1-way	2-way	3-way	4-way	1-way	2-way	3-way	4-way
Avg. of $R_{Cov}(T_t, EX*)$ ($R* = Cov(T_t)/Cov(EX*)$)	flex	116.45 %	117.26 %	114.45 %	111.04 %	116.65 %	116.12 %	113.65 %	110.57 %
	grep	109.93 %	108.39 %	105.36 %	103.31 %	111.26 %	110.44 %	107.36 %	104.94 %
	make	106.63 %	106.45 %	106.11 %	105.36 %	105.99 %	105.80 %	105.51 %	104.85 %
	Avg.	111.26 %	110.95 %	108.79 %	106.64 %	111.61 %	111.08 %	109.04 %	106.90 %
# ($R* \geq 105$ %) / # all cases	flex	12 / 12	12 / 12	12 / 12	12 / 12	12 / 12	12 / 12	12 / 12	12 / 12
	grep	12 / 12	12 / 12	2 / 12	2 / 12	12 / 12	12 / 12	2 / 12	2 / 12
	make	10 / 10	10 / 10	10 / 10	6 / 10	10 / 10	10 / 10	10 / 10	2 / 10
	Total	34 / 34	34 / 34	24 / 34	20 / 34	34 / 34	34 / 34	24 / 34	16 / 34

[5] 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.

[6] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case senarios. In *Proc. of the 24th Pacific Northwest Software Quality Conference*, pages 419–430. Citeseer, 2006.

[7] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[8] G. B. Finelli. NASA software failure characterization experiments. *Reliability Engineering & System Safety*, 32(1):155–169, 1991.

[9] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.

[10] D. Giannakopoulou, D. Bushnell, J. Schumann, H. Erzberger, and K. Heere. Formal testing for separation assurance. *Annals of Mathematics and Artificial Intelligence*, 63(1):5–30, 2011.

[11] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. Comparing white-box and black-box test prioritization. In *Proc. of the 38th International Conference on Software Engineering (ICSE)*, pages 523–534. ACM, 2016.

[12] Y. Jia, M. B. Cohen, M. Harman, and J. Petke. Learning combinatorial interaction testing strategies using hyperheuristic search. In *Proc. of the 37th International Conference on Software Engineering (ICSE)*, pages 540–550. IEEE/ACM, 2015.

[13] R. N. Kacker, D. R. Kuhn, Y. Lei, and J. F. Lawrence. Combinatorial testing for software: An adaptation of design of experiments. *Measurement*, 46(9):3745–3752, 2013.

[14] T. Kitamura, A. Yamada, G. Hatayama, C. Artho, E. Choi, N. T. B. Do, Y. Oiwa, and S. Sakuragi. Combinatorial testing for tree-structured test models with constraints. In *Proc. of the International Conference on Software Quality, Reliability & Security (QRS)*, pages 141–150. IEEE, 2015.

[15] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to combinatorial testing*. CRC Press, 2013.

[16] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004.

[17] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang. TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation. In *Proc. of the 30th International Conference on Automated Software Engineering (ASE)*, pages 494–505. ACM/IEEE, 2015.

[18] C. Maughan. Test case generation using combinatorial based coverage for rich web applications. *PhD Thesis. Utah State University*, 2012.

[19] C. Montanez, D. R. Kuhn, M. Brady, R. M. Rivello, J. Reyes, and M. K. Powers. An application of combinatorial methods to conformance testing for document object model events. *NISTIR-7773*, 2010.

[20] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11, 2011.

[21] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, 1988.

[22] J. Petke, M. Cohen, M. Harman, and S. Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Trans. Software Eng.*, 41(9):901–924, 2015.

[23] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E. Choi. Greedy combinatorial test case generation using unsatisfiable cores. In *Proc. of the 31st International Conference on Automated Software Engineering (ASE)*, pages 614–624. IEEE/ACM, 2016.

[24] A. Yamada, T. Kitamura, C. Artho, E. Choi, Y. Oiwa, and A. Biere. Optimization of combinatorial testing by incremental SAT solving. In *Proc. of the 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[25] Z. Zhang, X. Liu, and J. Zhang. Combinatorial testing on id3v2 tags of mp3 files. In *Proc. of the 5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 587–590. IEEE, 2012.

[26] ACTS, Available: http://csrc.nist.gov/groups/SNS/acts/.

[27] cloc – Count Lines of Code, Available: http://cloc.sourceforge.net.

[28] CodeCover – an open-source glass-box testing tool, Available: http://codecover.org.

[29] gcov – a test coverage program, Available: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[30] JavaPathfinder, Available: http://babelfish.arc.nasa.gov/trac/jpf.

[31] Pairwise Independent Combinatorial Tool, Available: http://github.com/Microsoft/pict.

[32] Software-artifact Infrastructure Repository, Available: http://sir.unl.edu/.