Test Effectiveness Evaluation of Prioritized Combinatorial Testing: A Case Study

Eun-Hye Choi[*], Shunya Kawabata[*†], Osamu Mizuno[†], Cyrille Artho[*], Takashi Kitamura[*]
[*]National Institute of Advanced Industrial Science and Technology (AIST), Ikeda, Japan
Email: {e.choi, c.artho, t.kitamura}@aist.go.jp
[†]Kyoto Institute of Technology, Kyoto, Japan
Email: s-kawabata@se.is.kit.ac.jp, o-mizuno@kit.ac.jp

***Abstract*—*Combinatorial testing* is a widely-used technique to detect system interaction failures. To improve test effectiveness with given priority weights of parameter values in a system under test, *prioritized combinatorial testing* constructs test suites where highly weighted parameter values appear earlier or more frequently. Such order-focused and frequency-focused combinatorial test generation algorithms have been evaluated using metrics called *weight coverage* and *KL divergence* but not sufficiently with fault detection effectiveness so far. We evaluate the fault detection effectiveness on a collection of open source utilities, applying prioritized combinatorial test generation and investigating its correlation with weight coverage and KL divergence.**

***Index Terms*—Prioritized combinatorial testing; Pairwise testing; Fault detection; Weight coverage; KL divergence.**

I. Introduction

Modern software systems have many parameters, and their interactions are too numerous to be fully tested. Combinatorial *pairwise testing* (generally, t-way testing) [13] addresses this problem by testing all interactions of two (t) parameters; it has been shown that t-way testing with small t ($2 \leq t \leq 4$) can efficiently detect system interaction failures while significantly reducing the number of test cases [11].

Recent papers ([1], [2], [3], [6], [10], [16]) have investigated *prioritized pairwise test generation*, which aims to achieve better test effectiveness. Prioritized pairwise test generation algorithms take a system under test (*SUT*) model with *priority weights* assigned to parameter values as an input and generate a pairwise test suite where highly weighted parameter values appear earlier and/or more frequently.

Prioritized pairwise test generation algorithms have been evaluated on two metrics: *weight coverage* and *KL divergence*. Approaches in [1], [2], [10] use weight coverage to evaluate the effectiveness of *order-focused* combinatorial test generation where given weights are used to order test cases. On the other hand, approaches in [2], [6], [16] use KL divergence to evaluate the effectiveness of *frequency-focused* combinatorial test generation where the weights are used to balance the frequency of parameter values in test cases. However, the relation of test effectiveness on fault detection with weight coverage and KL divergence has not been investigated so far.

To address this problem, we present a case study that evaluates the fault detection effectiveness with weight coverage and KL divergence and analyzes the correlation between them using nine variants [2] of prioritized pairwise test generation.

TABLE I
An example SUT model.

parameter	*value*; *weight*
p_1	a; 0.2, b; 0.1
p_2	c; 0.1, d; 0.1
p_3	e; 0.1, f; 0.1, g; 0.4
constraint	
p_1=b $\rightarrow$ $p_3 \neq$g	

TABLE II
An example pairwise test suite for the SUT in Table I with weight coverage *WC* and KL divergence *D*.

test case	p_1	p_2	p_3	*WC*	*D*
1	a	c	g	0.3182	1.5041
2	a	d	g	0.5000	0.8109
3	a	c	e	0.6136	0.6931
4	a	d	f	0.7273	0.4644
5	b	c	f	0.8636	0.2461
6	b	d	e	1.0000	0.2310

For empirical evaluation, we use twelve versions of three C projects, flex, grep, and make, from the Software artifact Infrastructure Repository (SIR) [4]. To generate prioritized pairwise test suites, we construct SUT models with constraints from test plans in Test Specification Language (TSL) [14] in the repository and extract priority weights of parameter values from bug reports of the repository. We show the results of examining weight coverage, KL divergence, and fault detection effectiveness of 108 (= 12 subjects × 9 prioritization methods) pairwise test suites, and analyze the correlation of the fault detection effectiveness with the order-focused and the frequency-focused prioritization effectiveness.

To our knowledge, this paper presents the first case study of evaluating not only weight coverage and KL divergence but also fault detection effectiveness of prioritized combinatorial test generation and investigating the correlation of them.

This paper is organized as follows: Section II explains the related work. Section III explains prioritized pairwise test generation and its evaluation metrics of weight coverage and KL divergence. Section IV describes the experimental setting, evaluation metrics we use, and experimental results. Section V concludes and proposes future work.

II. Related Work

Existing prioritized combinatorial test generation algorithms [1], [2], [6], [10], [16] have evaluated their test suites with weight coverage and KL divergence but not fault detection effectiveness as described in Section I.

On the other hand, X. Qu el. [17] evaluate fault detection effectiveness of test suites by an order-focused prioritized pairwise test generation algorithm called a deterministic density algorithm (DDA), which is a greedy algorithm proposed by R. Bryce and C. Colbourn [1]. X. Qu el. presented priority weight extractions from code coverage and specification and showed that combinatorial test generation by DDA based on their weights can find faults more effectively than exhaustive test cases. They evaluate neither weight coverage nor KL divergence with fault detection effectiveness, and their research purpose is different from ours.

To evaluate the efficiency of combinatorial t-way testing, Petke et al. [15] investigate fault detection effectiveness of combinatorial t-way test suites ($2 \leq t \leq 6$) that are generated by a simulated annealing algorithm, CASA [19], and a greedy algorithm, ACTS [18]. They also examine the fault detection rate of test prioritization of the t-way test suites w. r. t. *t'-way interaction coverage* with $2 \leq t \leq 6$, which means the test suites whose test cases are re-ordered in the descent order of t'-way coverage.

Henard et al. [8] also evaluate the fault detection availability of test prioritization of exhaustive test suites w. r. t. t-way coverage with $2 \leq t \leq 4$ in their comparison of white-box prioritization and black-box prioritization. In addition, Henard et al. [7] examine t-way coverage and the fault detection rate by test prioritization w. r. t. test case similarity for software product line systems.

While we in this paper explore weight coverage and KL divergence with fault detection effectiveness of prioritized combinatorial testing with weighted SUT, the work [15], [7], [8] consider combinatorial testing with non-weighted SUT and investigate neither weight coverage nor KL divergence.

III. Prioritized Combinatorial Testing

A. Prioritized pairwise testing

A system under test (SUT) for combinatorial testing is modeled from parameters, their associated values from finite sets, and constraints between parameter values. Table I shows an example SUT model with three parameters (p_1, p_2, p_3) and a constraint between p_1 and p_3; p_1 and p_2 have two values, p_3 has three values, and value pair (b, g) is not allowed by the constraint.

A *test case* for an SUT model assigns to each parameter a value that does not violate constraints in the SUT model. For example, a 3-tuple (a, c, e) is a test case for our example SUT model. We call a sequence of test cases a *test suite*.

A *pairwise test suite* for an SUT model is a test sequence to cover all *possible* value pairs between two parameters in the SUT model at least once. We say that a value pair is possible iff it does not violate SUT constraints. Table II shows an example pairwise test suite for the SUT model in Table I; it covers all possible 15 value pairs between two parameters, (a, c), (a, d), . . ., (d, g).

Prioritized pairwise testing takes an SUT whose parameter values are assigned a weight representing a relative importance in testing, e. g., error probability, occurrence probability, and risk [10], and constructs a pairwise test suite that considers the weights. Existing algorithms for prioritized pairwise test generation are classified, depending on how weights are reflected in a test suite, into order-focused approaches, frequency-focused approaches, and their integration.

B. Order-focused prioritization and weight coverage

The algorithms in the order-focused approach, e. g., DDA [1] and CTE-XL [10], consider that highly weighted values (value pairs) should appear early in a test suite. Hence, they use weights to let higher-priority values appear earlier in test generation.

To evaluate a test suite T, they use a metric called *weight coverage*, which is defined as

$$WC(T) = \frac{\textit{Sum of weights of value pairs covered by T}}{\textit{Sum of weights of all possible value pairs}}.$$

For example, weight coverage for the first two test cases of T in Table II for the SUT in Table I is 0.5 since the sum of weights of all possible 15 values pairs is 4.4 and that of value pairs covered by T is 2.2. In a test suite, order-focused prioritization uses higher-weighted values earlier, which implies obtaining higher weight coverage earlier.

C. Frequency-focused prioritization and KL divergence

The algorithms in the frequency-focused approach, e. g., PICT [3], the method by Fujimoto et al. [6], and FoCuS [16], consider that highly weighted values should appear frequently in a test suite. Hence, they use weights to utilize higher-priority values more often in test generation.

To evaluate a test suite T, they use *KL divergence* [12], which measures the difference between two probability distributions P and Q by

$$D(T) = \textstyle\sum_v P(v) \log(P(v)/Q(v)),$$

where $P(v)$ and $Q(v)$ respectively denote the current frequency of each parameter value v in T and the ideal occurrence frequency for v. The frequency-focused prioritization assumes that the number of occurrences of v is proportional to its weight.

For our example SUT in Table I, the ideal distribution $Q(v)$ is 2/3, 1/3, . . ., 2/3 for each value, a, b, . . ., g. On the other hand, the current distribution $P(v)$ of test suite T in Table II is 2/3, 1/3, . . ., 1/3, and the KL divergence $D(T)$ is 0.2310. By definition of KL divergence, $D(T)$ equals zero in the ideal situation, i. e., when $P = Q$, and it grows when the difference between P and Q is larger.

TABLE III
PROJECT DATA, NUMBER OF SEEDED FAULTS AND NUMBER OF DETECTED FAULTS.

No.	project	ver.	LoC	# of faults										
				seeded	detected by all tests	detected by prioritized pairwise tests by nine algo.								
						cs	co	cf	cs.co	co.cs	cs.cf	co.cf	cs.co.cf	co.cs.cf
1	flex	v1	12,160	19	16	16	16	16	16	16	16	16	16	16
2		v2	12,737	20	13	13	13	13	13	13	13	13	13	13
3		v3	12,781	17	9	9	9	9	9	9	9	9	9	9
4		v4	14,168	16	11	11	11	11	11	11	11	11	11	11
5		v5	12,893	9	5	5	5	5	5	5	5	5	5	5
6	grep	v1	12,507	18	4	4	4	4	4	4	4	4	4	4
7		v2	13,179	8	2	2	2	2	2	2	2	2	2	2
8		v3	13,291	18	**8**	7	7	**8**	7	7	7	**8**	7	**8**
9		v4	13,359	12	2	2	2	2	2	2	2	2	2	2
10	make	v1	18,460	19	4	4	4	4	4	4	4	4	4	4
11		v2	19,149	6	1	1	1	1	1	1	1	1	1	1
12		v3	20,340	5	1	1	1	1	1	1	1	1	1	1

TABLE IV
NUMBER OF ALL POSSIBLE TESTS, AND SIZES OF PAIRWISE TEST SUITES USED IN THE EXPERIMENT.

No.	project	ver.	# of all tests	# of prioritized pairwise tests by nine algo.								
				cs	co	cf	cs.co	co.cs	cs.cf	co.cf	cs.co.cf	co.cs.cf
1	flex	v1	525	52	52	**51**	52	52	**51**	**51**	**51**	52
2		v2	525	52	52	**51**	52	52	**51**	**51**	**51**	52
3		v3	525	52	53	52	**51**	**51**	**51**	53	52	**51**
4		v4	525	52	52	**51**	**51**	**51**	**51**	52	**51**	**51**
5		v5	525	52	52	**51**	52	52	**51**	**51**	**51**	52
6	grep	v1	470	**75**	77	81	**75**	76	**75**	**75**	78	**75**
7		v2	470	**75**	**75**	81	76	76	78	**75**	76	79
8		v3	470	**75**	78	80	**75**	78	77	79	77	79
9		v4	470	75	**74**	80	75	75	78	76	77	77
10	make	v1	793	33	33	33	**32**	33	33	**32**	33	**32**
11		v2	793	**33**	**33**	35	34	**33**	**33**	**33**	**33**	**33**
12		v3	793	**33**	**33**	35	34	**33**	34	**33**	34	**33**

D. Pricot

The algorithm in [2], which we call *pricot*, integrates the order-focused prioritization (shortly co) and the frequency-focused prioritization (shortly cf) with a size-focused prioritization (shortly cs) which considers that the size of a test suite should be small. To realize a small test suite where high-priority test cases appear early and frequently in a good balance, pricot takes a prioritization order of cs, co, and cf (e. g., cs > co > cf, denoted by cs.co.cf) as an input and generates a pairwise test suite that considers the weights in the given order.

To evaluate test suites, pricot uses both weight coverage and KL divergence [2]. Table II shows a pairwise test suite that is generated by pricot with co.cf, together with cumulative weight coverage and KL divergence of its test cases. For our case study to investigate the relation of fault detection effectiveness with weight coverage and KL divergence, we use pairwise test suites generated by pricot with various prioritization orders.

IV. EXPERIMENTS

A. Research Questions

We set up the following two research questions to investigate the effectiveness of existing evaluation metrics of prioritized combinatorial testing.

RQ1. Do order-focused prioritized combinatorial test suites with higher weight coverage achieve better fault detection effectiveness?

RQ2. Do frequency-focused prioritized combinatorial test suites with better (lower) KL divergence achieve better fault detection effectiveness?

B. Experimental Setting

1) Subjects: For empirical experiments, we use three open source projects of C programs, flex, grep, and make, from the Software artifact Infrastructure Repository (SIR) [20]. Each project includes

- multiple versions of programs with seeded faults,
- a test plan in Test Specification Language (TSL) [14],
- all test cases satisfying the test plan, and
- a bug report for each version of the project that describes which test case detects a fault.

Table III shows the lines of code (LoC) including comments, the number of seeded faults, the number of detected faults by all test cases. Table IV shows the number of all test cases for each version of the projects we use. The faults in the repository were hand-seeded by multiple developers to reflect real types of faults based on their experience [4]. We choose the versions whose number of detected faults is not zero from the repository.

```
Parameters:
...
 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.

TABLE V
Sizes of SUT models.

project	model	constraint
flex	$29; 3^{23}4^{4}6^{2}$	$97; 2^{712}22^{1}24^{2}25^{17}26^{9}$
grep	$14; 2^{4}3^{1}4^{3}5^{1}6^{1}9^{1}11^{1}13^{1}20^{1}$	$87; 2^{433}3^{27}4^{8}7^{5}16^{1}24^{1}27^{1}28^{1}31^{10}$
make	$22; 2^{2}3^{12}4^{4}5^{2}6^{1}7^{1}$	$79; 2^{526}21^{1}22^{1}23^{1}24^{3}25^{7}26^{9}$

2) SUT models: For each project, we construct an SUT model whose parameters, values, and constraints are fully extracted from the TSL specification. For example, Fig. 1 show a part of the test plan in TSL for project flex included in SIR. From the TSL specification, we construct the SUT model for flex whose parameters include Bypass use$(= p_x)$ and Fast scanner$(= p_y)$, values for p_x includes Bypass_on$(= v_a)$, values for p_y includes FullScan$(= v_b)$, and constraints include $(p_y = v_b) \rightarrow (p_x \neq v_a)$. Table V shows the size of the SUT model for each project. In the table, the size of a model 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 $l; h_1^{l_1} h_2^{l_2} \ldots h_m^{l_m}$ which indicates that the constraint is described in conjunctive normal form (CNF) with l variables whose Boolean value represents an assignment of a value to a parameter and for each j there are h_j clauses that have l_j literals.

3) Weights: For each version of the project, we extract the weight of each parameter value v, denoted by $w(v)$, from the bug report. We define $w(v)$ as the conditional probability that a test case t detects a fault given that v is assigned to the test case t. $w(v)$ is then calculated using the Bayesian inference as follows [9]:

$$w(v) = P(t \text{ detects a fault} \mid v \text{ is assigned to } t) \quad (1)$$

$$= \frac{P(v \text{ is assigned to } t \mid t \text{ detects a fault})}{P(v \text{ is assigned to } t)} \quad (2)$$

We compute the above equation (2) and determine the weight for each parameter value v using the information in the bug report of SIR that describes whether each test case t detects a fault or not.

4) Test suites: We use prioritized pairwise test suites generated by pricot [2] for the constructed SUT models with constraints and weights. For each model, we use nine variants of test suites generated with the following prioritization orders: 1) cs, 2) co, 3) cf, 4) cs.co, 5) co.cs, 6) cs.cf, 7) co.cf, 8) cs.co.cf, and 9) co.cs.cf. In Tables III and IV, we show the size of each test suite and the number of faults detected by the test suite. We highlight the case where more faults are detected in Table III, and highlight the case where the size of the test suite is minimum in Table IV. For all subjects except grep v3, all the pairwise test suites detect all faults detected by all test cases, while sizes of the pairwise test suites are less than 18% of those of exhaustive test suites.

C. Evaluation metrics

To evaluate the fault detection effectiveness of a test suite T, we use the metric called *NAPFD* (Normalized Average Percentage of Faults Detected) [17], which is defined by

$$NAPFD(T) = p - \frac{F_1 + F_2 + \ldots + F_m}{m \times n} + \frac{p}{2n}$$

where m denotes the number of faults detected by the all test cases, n denotes the number of test cases of T, F_i $(1 \leq i \leq m)$ denotes the number of the test cases where fault i is detected, and p denotes the number of faults detected by T divided by m. For example, assume that there are two faults and the first test case and the third test case of T in Table II detect each of the two faults. (We call this assumption X in the following.) *NAPFD* of T is $0.75(= 1 - 4/12 + 1/12)$.

NAPFD is a normalized *APFD* [5], which is a common metric to evaluate fault detection effectiveness of test prioritization in regression testing, for evaluating test suites with different sizes and thus different numbers of faults detected (See [17] for further details). *NAPFD* measures the area under the curve when the percent of detected faults is on the y-axis and the percent of test cases is on the x-axis; higher *NAPFD* implies faster and more effective fault detection.

To evaluate weight coverage and KL divergence for prioritized test suites with different sizes, we use normalized values of weight coverage *WC* and KL divergence *D* following *NAPFD*, which we call *Normalized Weight Coverage* (*NWC*) and *Normalized KL divergence* (*NKLD*) respectively. We define *NWC* and *NKLD* of a test suite T as follows:

$$NWC(T) = \frac{p_w}{n}\Big(\sum_{1 \leq i \leq n} WC(T_i)/WC(T) - \frac{1}{2}\Big),$$

$$NKLD(T) = \frac{p_d}{n}\Big(\sum_{1 \leq i \leq n} D(T_i)/D_{max}(T) - \frac{1}{2}\Big),$$

where

- n denotes the number of test cases in T,
- T_i denotes the test suite having the first i test cases in T,
- p_w denotes $WC(T)$ divided by the maximum value of weight coverage, i.e., 1.
- $D_{max}(T)$ denotes the maximum value of $D(T_i)$ for $1 \leq i \leq n$,
- p_d denotes $D_{max}(T)$ divided by d_{max}, where d_{max} denotes the maximum value of $D_{max}(T')$ for each T' of all test suites for evaluation.

For the test suite T in Table II, *NWC* is $0.5871(= 4.0227/6 - 1/12)$. *NKLD* is $0.3543(= 3.9496/(6 \times 1.5041) - 1/12)$ where $d_{max} = 1.5041$.

TABLE VI
NAPFD, *NWC*, and *NKLD* for sample subjects.

subject	test suites	*NAPFD*	*NWC*	*NKLD*
flex v1	co	**0.9772**	0.6349	0.5629
	cf	0.9571	**0.7038**	**0.4105**
	co.cf	0.9767	0.6381	0.5493
grep v3	co	0.7492	0.6799	0.2882
	cf	0.8266	0.6447	0.3492
	co.cf	**0.8608**	**0.6864**	**0.2801**
make v1	co	0.8106	0.7439	0.3771
	cf	**0.9242**	0.7131	**0.3648**
	co.cf	0.8047	**0.7373**	0.3839

NWC (resp. *NKLD*[1]) measures the area under the curve when the percent of *WC* (resp. *D*) is on the *y*-axis and the percent of test cases is on the *x*-axis; higher *NWC* (resp. lower *NKLD*) implies better test effectiveness on order-focused (resp. frequency-focused) prioritization.

D. Results

Fig. 2 shows the cumulative numbers of faults detected, weight coverage, and KL divergence by the pairwise test cases generated by pricot. Due to space limitations, we show the results by three methods co (order-focused prioritization), and cf (frequency-focused prioritization), and co.cf (their integration) for three subjects flex v1, grep v3, and make v1; we selected a subject whose number of detected faults is the maximum in each project. Table VI gives *NAPFD*, *NWC*, and *NKLD* for each case.

From Table VI, for grep v3, method co.cf, which provides the best *NWC* and *NKLD* among the three methods, obtains the best *NAPFD*. For make v1, method cf, which provides the best *NKLD*, obtains the best *NAPFD* but the worst *NWC*. For flex v1, method co obtains the best *NAPFD* but the worst *NWC* and *NKLD*. However, looking at the first 10 test cases for flex v1 in Fig. 2, where all faults are detected, co and co.cf achieve better fault detection with better weight coverage and KL divergence compared to cf.

Table VII presents the results of *NAPFD*, *NWC*, and *NKLD* for all 108 test suites by nine variants of prioritization for all 12 subjects. Fig. 3 shows box plots for the results. Each box plot shows the mean (triangle in the box), median (thick horizontal line), the first/third quartiles (hinges), and highest/lowest values within 1.5 × inter-quatile range of the hinge (whiskers). Points outside the range (dots) are considered outliers. Table VIII shows the average and the number of *wins*, which indicates the number of times that each method obtains the best value among the nine methods, of *NAPFD*, *NWC*, and *NKLD* for all subjects.

Although the result shows arbitrary orders on *NAPFD*, *NWC*, and *NKLD* for the nine methods, co.cf, which provides the maximum *NAPFD* (0.8943) on average, obtains the maximum number of wins for *NWC* and *NKLD* among the nine methods; co.cf achieves the best *NWC* for 5 subjects and the best *NKLD* for 6 subjects among 12 subjects. On the other hand, co.cs.cf, which obtains the maximum number of wins (5 times) for *NAPFD*, achieves the maximum *NWC* (0.7294) and the best *NAPFD* (0.3426). On the contrary, cs (size-focused prioritization, which does not consider weights of values in test generation) provides the minimum *NAPFD* (0.7018) on average and achieves the best *NWC* or *NKLD* for no subject.

Fig. 4 shows scatter plots with regression lines and coefficients R for the correlation between *NWC* and *NAPFD* and that between *NKLD* and *NAPFD*, using the 108 test suites. From the result, *NAPFD* is correlated with *NWC* ($R = 0.389$) although no correlation is found between *NAPFD* and *NKLD* ($R = -0.101$). We also investigated *NWC*, *NKLD*, and *NAPFD* of the minimum test suite T_i having the first i test cases of each test suite T that detect all faults detected by T. (For example, assuming X in Section IV, the minimum test suite of T is the one having the first three test cases.) Fig. 5 shows the correlation using the minimum test suites. The result shows that *NAPFD* is more significantly correlated with *NWC* ($R = 0.556$) but is still not correlated with *NKLD* ($R = 0.146$).

The experimental results answer to the research questions, RQ1 and RQ2, as follows: Combinatorial test generation that achieves higher weight coverage can provide better (faster) fault detection but that with better KL divergence might not. Basically, frequency-focused prioritization aims to provide more effective fault detection while order-focused prioritization aims to provide earlier fault detection. Therefore, to investigate the fault detection effectiveness of frequency-focused combinatorial test generation, examining the correlation of KL divergence to the number of faults detected is also our interest. Unfortunately, the numbers of faults detected by test suites used in our experiments are almost the same, and thus further case studies on more software projects will be included in future work.

V. Conclusion and Future Work

This paper investigates the fault detection effectiveness with weight coverage and KL divergence of prioritized combinatorial test generation. In our empirical evaluation using a collection of open source utilities, order-focused combinatorial test generation with higher weight coverage achieves the best *(fastest)* fault detection while the frequency-focused combinatorial test generation with better KL divergence fares worse. The correlation between KL divergence and the test effectiveness w. r. t. detecting *more* faults will be investigated in future work. In addition, further case studies on software projects with real faults is an important future work. We are also investigating automated methods of extracting priority weights for prioritized combinatorial testing to achieve better fault detection effectiveness.

Acknowledgments

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

[1]Strictly the area under the curve for KL divergence is calculated by $p_d/n \times (\sum_{1 \le i \le n} D(T_i)/D_{max}(T) - D(T_1)/2 + \sum_{2 \le i \le n}(D(T_{i-1}) - D(T_i))/2)$. We use the simplified formula in this paper.

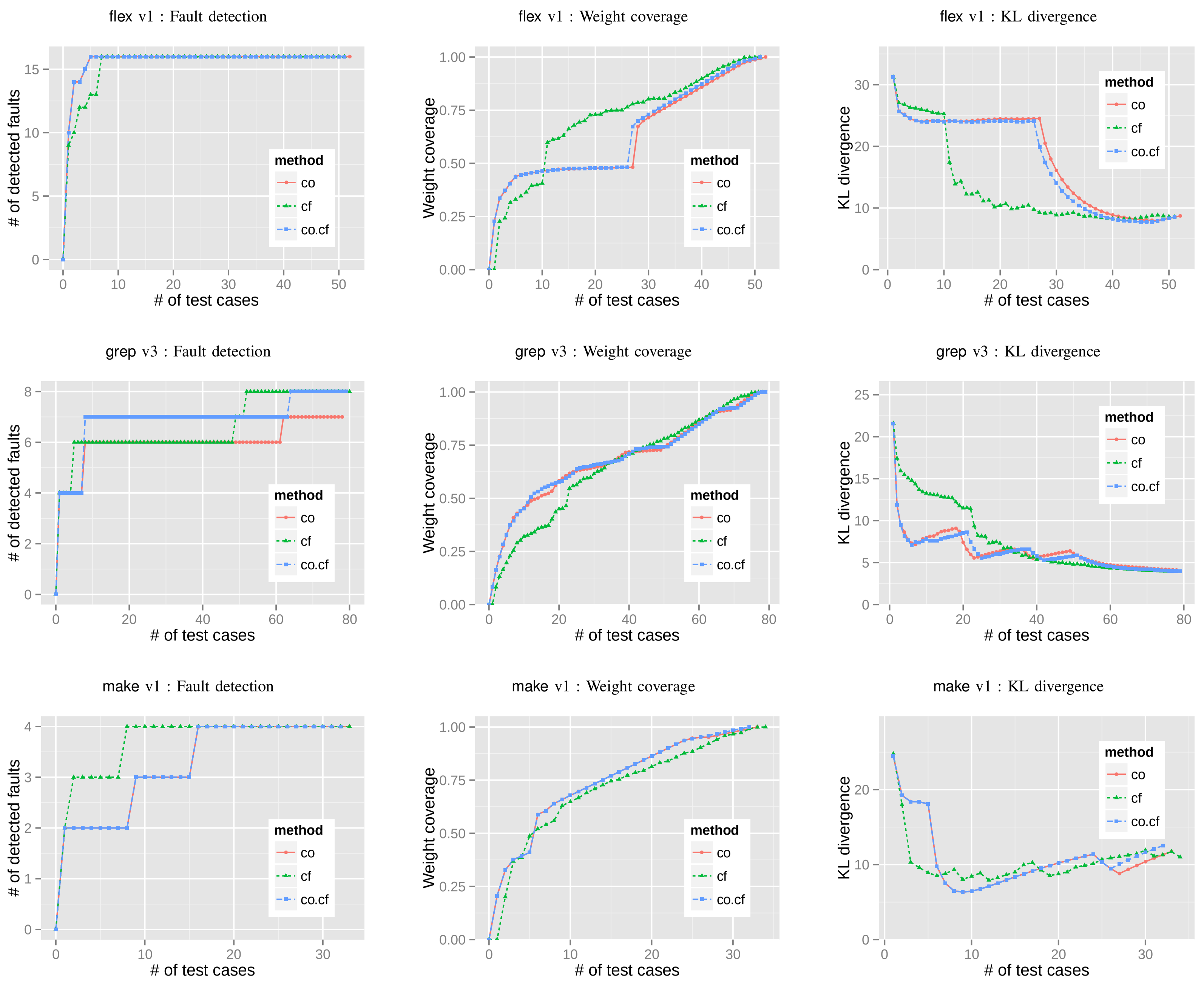

Fig. 2. Number of faults detected, weight coverage, and KL divergence for sample subjects.

TABLE VII
NAPFD, *NWC*, AND *NKLD* OF NINE VARIANTS OF TEST SUITES FOR EACH SUBJECT.

flex v1				flex v2				flex v3			
test suites	*NAPFD*	*NWC*	*NKLD*	test suites	*NAPFD*	*NWC*	*NKLD*	test suites	*NAPFD*	*NWC*	*NKLD*
cs	0.9111	0.7204	0.4107	cs	0.8277	0.7140	0.4137	cs	0.6442	0.6357	0.4756
co	**0.9772**	0.6349	0.5629	co	0.7907	0.6261	0.5676	co	**0.8606**	**0.8034**	0.2916
cf	0.9571	0.7038	0.4105	cf	0.8107	0.6976	0.4137	cf	0.6741	0.6195	0.4894
cs.co	0.9111	0.7234	0.4076	cs.co	0.8277	0.7169	0.4107	cs.co	0.6765	0.7170	0.3953
co.cs	0.9772	0.6745	0.5096	co.cs	0.8025	0.6666	0.5132	co.cs	0.7505	0.7609	0.3079
cs.cf	0.9632	0.7204	0.4103	cs.cf	0.8137	0.7140	0.4133	cs.cf	0.6656	0.6289	0.4794
co.cf	0.9767	0.6381	0.5493	co.cf	0.8544	0.6294	0.5541	co.cf	**0.8606**	**0.8034**	**0.2914**
cs.co.cf	0.9632	**0.7234**	**0.4072**	cs.co.cf	0.8137	**0.7169**	**0.4103**	cs.co.cf	0.6784	0.7167	0.3952
co.cs.cf	**0.9772**	0.6738	0.5046	co.cs.cf	**0.8602**	0.6659	0.5083	co.cs.cf	0.7505	0.7609	0.3077

flex v4				flex v5				grep v1			
test suites	*NAPFD*	*NWC*	*NKLD*	test suites	*NAPFD*	*NWC*	*NKLD*	test suites	*NAPFD*	*NWC*	*NKLD*
cs	0.8488	0.7352	0.4125	cs	**0.9904**	0.7165	0.4110	cs	0.5900	0.6537	0.3550
co	0.9205	0.6547	0.5538	co	**0.9904**	0.6261	0.5676	co	0.7013	0.6137	0.3889
cf	**0.9207**	0.7198	**0.4117**	cf	0.9902	0.7001	0.4109	cf	**0.8025**	0.6513	**0.3396**
cs.co	0.8387	**0.7353**	0.4291	cs.co	**0.9904**	**0.7178**	0.4087	cs.co	0.6933	**0.6545**	0.3418
co.cs	0.8565	0.6826	0.5143	co.cs	0.9904	0.6391	0.5447	co.cs	0.6842	0.6088	0.3947
cs.cf	**0.9207**	0.7352	0.4120	cs.cf	0.9902	0.7165	0.4106	cs.cf	0.7867	0.6509	0.3507
co.cf	0.9205	0.6547	0.5494	co.cf	0.9902	0.6294	0.5541	co.cf	0.6933	0.6174	0.3831
cs.co.cf	0.8244	**0.7353**	0.4290	cs.co.cf	0.9902	**0.7178**	**0.4083**	cs.co.cf	0.7949	0.6492	0.3440
co.cs.cf	0.8565	0.6826	0.5081	co.cs.cf	**0.9904**	0.6393	0.5392	co.cs.cf	0.6767	0.6150	0.3846

grep v2				grep v3				grep v4			
test suites	*NAPFD*	*NWC*	*NKLD*	test suites	*NAPFD*	*NWC*	*NKLD*	test suites	*NAPFD*	*NWC*	*NKLD*
cs	0.7200	0.4555	0.6314	cs	0.8242	0.6446	0.3587	cs	0.7467	0.7203	0.4737
co	0.8733	**0.8171**	**0.5009**	co	0.7492	0.6799	0.2882	co	0.9257	0.7977	0.5393
cf	0.9444	0.4691	0.6535	cf	0.8266	0.6447	0.3492	cf	0.9000	0.7126	**0.4549**
cs.co	0.8553	0.6764	0.8030	cs.co	0.8242	0.6446	0.3587	cs.co	0.8733	0.7982	0.5585
co.cs	0.8487	0.7614	0.5659	co.cs	0.7492	0.6799	0.2882	co.cs	0.9133	0.8035	0.5425
cs.cf	0.9423	0.4518	0.6156	cs.cf	0.7752	0.6463	0.3545	cs.cf	0.8974	0.7211	0.4732
co.cf	0.8733	**0.8171**	0.5091	co.cf	**0.8608**	**0.6864**	**0.2801**	co.cf	**0.9276**	0.8024	0.5377
cs.co.cf	0.9539	0.6798	0.8108	cs.co.cf	0.7752	0.6463	0.3545	cs.co.cf	0.9091	0.7982	0.5549
co.cs.cf	**0.9557**	0.7709	0.5785	co.cs.cf	**0.8608**	0.6864	**0.2801**	co.cs.cf	0.9156	**0.8041**	0.5406

make v1				make v2				make v3			
test suites	*NAPFD*	*NWC*	*NKLD*	test suites	*NAPFD*	*NWC*	*NKLD*	test suites	*NAPFD*	*NWC*	*NKLD*
cs	0.7727	0.7339	0.4084	cs	0.3788	0.5274	0.2821	cs	0.1667	0.5063	0.2873
co	0.8106	0.7439	0.4238	co	0.9848	0.8523	**0.2693**	co	0.9848	**0.8574**	0.3715
cf	0.9242	0.7131	**0.4054**	cf	0.9571	0.5361	0.3335	cf	0.9571	0.5468	0.3151
cs.co	0.8516	0.6567	0.6259	cs.co	**0.9853**	0.8388	0.4171	cs.co	**0.9853**	0.8439	0.4870
co.cs	0.8788	**0.7513**	0.4506	co.cs	0.9848	0.8522	0.2694	co.cs	0.9848	0.8574	0.3716
cs.cf	0.9091	0.7342	0.4184	cs.cf	0.9545	0.5291	0.2758	cs.cf	0.9559	0.5075	**0.2815**
co.cf	0.8047	0.7373	0.4315	co.cf	0.9848	**0.8523**	0.2696	co.cf	0.9848	**0.8574**	0.3720
cs.co.cf	**0.9470**	0.6666	0.6184	cs.co.cf	0.9848	0.8388	0.4174	cs.co.cf	**0.9853**	0.8439	0.4874
co.cs.cf	0.8750	0.7439	0.4470	co.cs.cf	0.9848	0.8522	0.2697	co.cs.cf	0.9853	0.8574	0.3720

TABLE VIII
NAPFD, *NWC*, AND *NKLD* FOR ALL SUBJECTS.

test suites	*NAPFD*		*NWC*		*NKLD*	
	avg	# wins	avg	# wins	avg	# wins
cs	0.7018	1	0.6469	0	0.4060	0
co	0.8808	3	0.7256	4	0.3537	0
cf	0.8887	2	0.6429	0	0.4063	3
cs.co	0.8594	3	0.7269	5	0.3518	0
co.cs	0.8684	2	0.7282	1	0.3460	0
cs.cf	0.8812	1	0.6463	1	0.4039	0
co.cf	**0.8943**	3	0.7271	**5**	0.3491	**6**
cs.co.cf	0.8850	2	0.7277	4	0.3506	3
co.cs.cf	0.8907	**5**	**0.7294**	2	**0.3426**	1

REFERENCES

[1] R. Bryce and C. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.

[2] E. Choi, T. Kitamura, C. Artho, A. Yamada, and Y. Oiwa. Priority integration for weighted combinatorial testing. In *Proc. of COMPSAC*, pages 242–247. IEEE, 2015.

[3] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test case generators. *Microsoft Corporation, Software Testing Technical Articles*, 2008.

[4] 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.

[5] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Software Eng.*, 28(2):159–182, 2002.

[6] S. Fujimoto, H. Kojima, and T. Tsuchiya. A value weighting method for pair-wise testing. In *Proc. of APSEC*, pages 99–105, 2013.

[7] 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.

[8] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: Using similarity

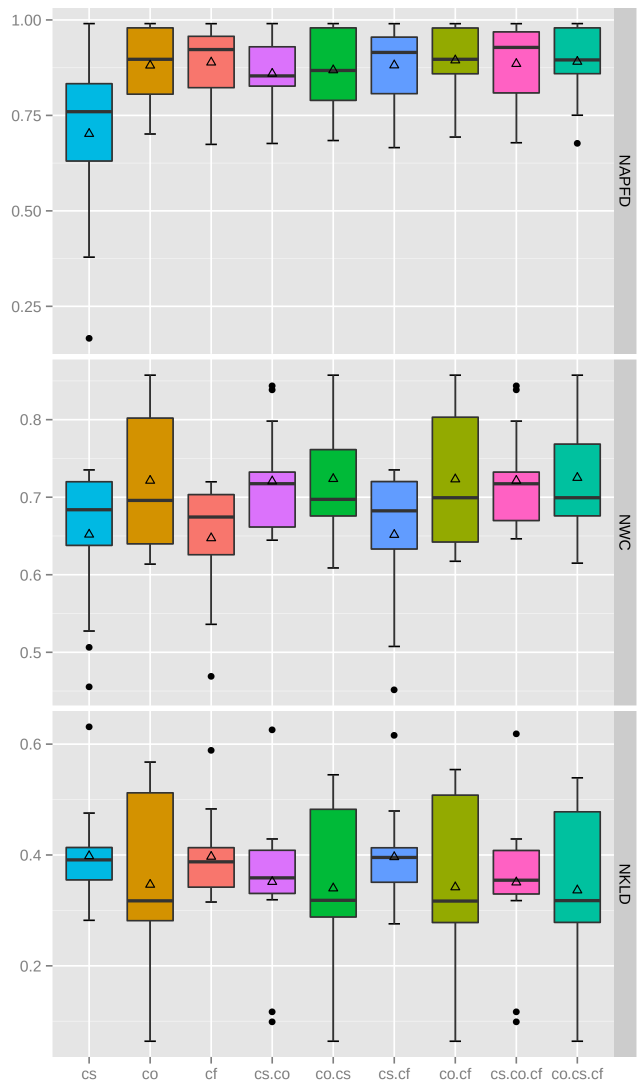

Fig. 3. *NAPFD*, *NWC*, and *NKLD* for all subjects.

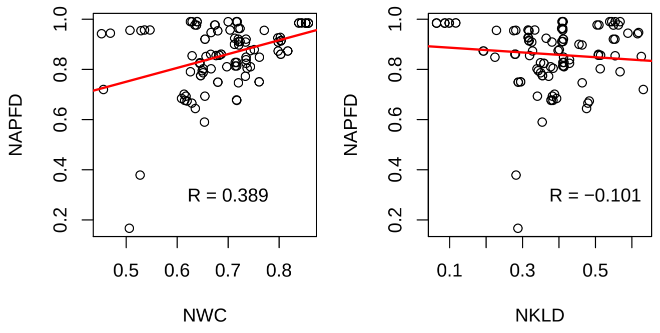

Fig. 4. Correlation of *NAPFD* with *NWC* and *NKLD* for full test suites.

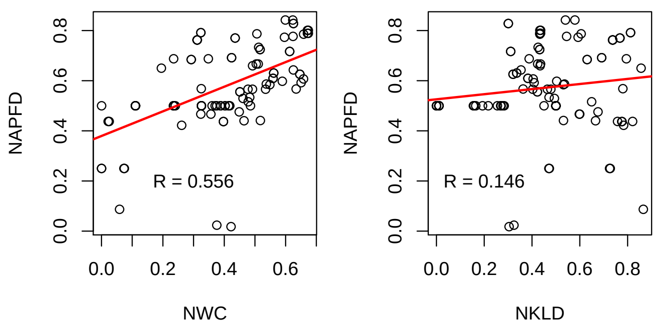

Fig. 5. Correlation of *NAPFD* with *NWC* and *NKLD* for the minimum test suites.

to generate and prioritize t-wise test configurations for software product lines. *IEEE Trans. Software Eng.*, 40(7):650–670, 2014.

[9] S. Kawabata, E. Choi, and O. Mizuno. A prioritization of combinatorial testing using Bayesian inference. *Technical Report of IEICE (in Japanese)*, 115(SS2015-95):115–120, 2016.

[10] P. Kruse and M. Luniak. Automated test case generation using classification trees. *Software Quality Professional*, pages 4–12, 2010.

[11] 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.

[12] S. Kullback and R. A. Leibler. The annals of mathematical statistics. *On information and sufficiency*, pages 79–86, 1951.

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

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

[15] 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.

[16] I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proc. of ISSTA*, pages 254–264, 2011.

[17] Q. Xiao, M. Cohen, and K. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *Proc. ICSM'07*, pages 255–264. IEEE, 2007.

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

[19] CASA, Available: http://cse.unl.edu/citportal/.

[20] SIR, Available: http://sir.unl.edu/.