Priority Integration for Weighted Combinatorial Testing

Eun-Hye Choi, Takashi Kitamura, Cyrille Artho, Akihisa Yamada, and Yutaka Oiwa
National Institute of Advanced Industrial Science and Technology (AIST), Japan
Email: {e.choi, t.kitamura, c.artho, akihisa.yamada, y.oiwa}@aist.go.jp

***Abstract*—Priorities (weights) for parameter values can improve the effectiveness of combinatorial testing. Previous approaches have employed weights to derive high-priority test cases either earlier or more frequently. Our approach integrates these order-focused and frequency-focused prioritizations. We show that our priority integration realizes a small test suite providing high-priority test cases early and frequently in a good balance. We also propose two algorithms that apply our priority integration to existing combinatorial test generation algorithms. Experimental results using numerous test models show that our approach improves the existing approaches w. r. t. order-focused and frequency-focused metrics, while overheads in the size and generation time of test suites are small.**

***Keywords*-Combinatorial testing; Pairwise testing; Prioritized Testing; Priority weight; Weight coverage; KL divergence.**

I. Introduction

Combinatorial testing (CT) [8] effectively detects system interaction failures in software. *Pairwise testing* (generally, t-wise testing) is a widely used CT technique, which stipulates to test all interactions between two parameters in a *system under test* (SUT) at least once. So far, various algorithms [8] have been proposed to generate pairwise *test suites* (i. e., lists of test cases) given an *SUT model* that comprises a list of parameters with their values and constraints over them. Combinatorial test generation algorithms are normally evaluated by their computation time and the following criterion:

CS: The size of a test suite should be as small as possible.

This criterion determines the cost of test execution.

Recent papers [1], [4], [6], [7], [9] have investigated *prioritized pairwise test generation,* which takes a *priority* notion into account in order to increase the quality of test suites. A prioritized test generation algorithm takes as input SUT models with priority weights assigned to parameter-values and generates test suites that consider such weights.

To our knowledge, every such algorithm proposed so far falls into one of the following two categories of priority criteria, depending on how weights are reflected in test suites:

CO: High-priority test cases should appear earlier [1], [7].

CF: High-priority parameter-values should appear more frequently [4], [6], [9].

For CO, weights are used to order test cases, presuming resources for test execution are limited in practice. For CF, weights are used to balance the frequency of appearance of certain values in test cases. Each of the two priority criteria in isolation increases the quality of a pairwise test suite. However, in practice one may want to integrate both to further improve the quality of test suites of limited size.

TABLE I: An example (weighted) SUT model.

Parameter	*Value*(*Weight*)
OS	Win(1), Unix(1), Mac(2)
CPU	Intel(1), AMD(3)
Browser	IE(1), Firefox(1), Safari(1), Opera(1)
Net	Wifi(2), LAN(2), 3G(2)

In earlier work [2], we proposed a novel idea for prioritized test suite generation that integrates all the three criteria: CO, CF, and CS. Our *priority integration* allows balancing the three criteria, obtaining a small test suite where important parameter values appear early and frequently. We also presented a basic algorithm that realizes priority integration as a global search, which is applicable only to small SUT models.

In this paper, we investigate the effectiveness of priority integration through analyzing experimental results for various priority orders. We also present two algorithms that realize practical test generation by applying our priority integration to existing practical algorithms, PICT [4] and the *density algorithm* [1]. We compare our priority integration algorithms with existing prioritization approaches, using metrics that measure CO and CF. Experimental results using numerous SUT models including ones based on large empirical benchmarks [3] and an actual system [5] show that our algorithms outperform the existing approaches w. r. t. the CO and CF.

This paper is organized as follows: Sections II and III describe preliminaries and state-of-the-art prioritized CT generation techniques, respectively. Sections IV and V present our priority integration algorithms and experimental results. Section VI concludes and proposes future work.

II. Preliminaries

A. Combinatorial Pairwise Testing

An *SUT* for CT is generally modeled from parameters, their values from a finite set, and constraints between parameter-values. In this paper, we do not consider constraints in SUT models for simplicity. An SUT is denoted as $\mathcal{S}(m; |V_1| \dots |V_m|)$, where m is the number of parameters, and for $1 \leq i \leq m$, V_i denotes the value domain of parameter p_i. We assume that $V_1, \dots, V_m$ are pairwise disjoint, and denote their disjoint union by V.

The SUT model in Table I has four parameters. Parameter "Browser" has four possible values, while both "OS" and

TABLE II: Example pairwise test suites by previous and proposed algorithms for the SUT model in Table I.

(a) Density Algorithm [1]

	OCBN	N	W	$WC(\%)$	D_{KL}
1	MAIW	6	24	14.81	3.466
2	MAFL	5	19	26.54	2.079
3	MASG	5	19	38.27	1.269
4	WAOW	5	16	48.15	0.651
5	UIOL	6	15	57.41	0.125
6	UIIG	5	13	65.43	0.102
7	WIFG	5	12	72.84	0.140
8	UISW	5	13	80.86	0.204
9	MIOG	3	9	86.42	0.284
10	WIIL	3	8	91.36	0.324
11	UAFW	3	9	96.91	0.264
12	WISL	2	5	100.00	0.306

(b) PI_{Dens} (CO.CF)

	OCBN	N	W	$WC(\%)$	D_{KL}
1	MAIW	6	24	14.81	3.466
2	MAFL	5	19	26.54	2.079
3	MASG	5	19	38.27	1.269
4	WAOW	5	16	48.15	0.651
5	UIOL	6	15	57.41	0.125
6	UIIG	5	13	65.43	0.102
7	WIFG	5	12	72.84	0.140
8	UISW	5	13	80.86	0.204
9	MIOG	3	9	86.42	0.284
10	WAIL	3	8	91.36	0.194
11	UAFW	3	9	96.91	0.164
12	WASL	2	5	100.00	0.123

(c) PICT Algorithm [4]

	OCBN	N	W	$WC(\%)$	D_{KL}
1	WIIW	6	15	9.26	5.257
2	UAIL	6	21	22.22	2.629
3	MIIG	5	16	32.10	1.827
4	MAFW	6	24	46.91	1.027
5	UIFL	5	12	54.32	1.085
6	WAFG	5	17	64.81	0.894
7	UISW	4	10	70.99	0.743
8	WASL	4	12	78.40	0.595
9	UISG	2	6	82.10	0.670
10	WIOW	3	7	86.42	0.549
11	UAOL	3	9	91.98	0.444
12	MISL	2	7	96.30	0.434
13	MIOG	2	6	100.00	0.392

(d) PI_{PICT} (CS.CO.CF)

	OCBN	N	W	$WC(\%)$	D_{KL}
1	MAIW	6	24	14.81	3.466
2	WIIL	6	15	24.07	2.282
3	UAIG	5	17	34.57	1.460
4	UIFW	6	15	43.83	1.200
5	WAFL	5	18	54.94	1.004
6	MIFG	5	16	64.81	0.894
7	WASW	4	12	72.22	0.587
8	MISL	4	12	79.63	0.509
9	WASG	2	6	83.33	0.471
10	MAOW	3	10	89.51	0.190
11	WIOL	3	7	93.83	0.218
12	UASL	2	5	96.91	0.193
13	UAOG	2	5	100.00	0.138

N: Number of newly covered parameter-value pairs. WC: Weight coverage.
W: Weight of newly covered parameter-value pairs. D_{KL}: KL divergence.

"Net" have three, and "CPU" has two possibilities. This SUT model (ignoring weights) is denoted by $\mathcal{S}(4; 2^1 3^2 4^1)$.

A *test case* for an SUT model $\mathcal{S}(m; |V_1| \dots |V_m|)$ assigns for each parameter p_i a value $v_i \in V_i$, and is expressed as an m-tuple $(v_1, \dots, v_m)$. For example, a 4-tuple (*Win, Intel, IE, Wifi*) in Table II-(c) is the first test case generated by PICT algorithm [4] for the example SUT in Table I. We call a sequence of test cases a *test suite*.

A *pairwise test suite* is a test suite to cover all parameter-value pairs in an SUT model at least once. Given SUT model $\mathcal{S}(m; |V_1| \dots |V_m|)$, it is defined as a $k \times m$ array, denoted by $\mathcal{T}(k; m, |V_1| \dots |V_m|)$, satisfying the following properties:

- For each element e in the j-th column with $1 \leq j \leq m$, $e \in V_j$.
- Each $k \times 2$ sub-array covers all pairs of values from the two columns at least once.

Each i-th row ($1 \leq i \leq k$) of $\mathcal{T}$ corresponds to the i-th test case. Each element in the j-th column ($1 \leq j \leq m$) corresponds to a value assignment for parameter p_j.

For the SUT model in Table I, there exist 6 parameter pairs, e. g. , (*OS, CPU*), . . . ,(*Browser, Net*), and totally 53 parameter-value pairs, e. g., (*Mac, Intel*), . . . ,(*Opera, 3G*). Every test suite in Table II is a pairwise test suite for the example SUT model, since it covers all the parameter-value pairs.

B. Weighted Pairwise Testing

For prioritized CT, we assume that a positive integer weight is assigned to each parameter-value in an SUT model. A weight represents a relative importance in the aspect of testing, e. g., occurrence probability, error probability, or risk of parameter-values [7]. We can easily extend our algorithm for the case where weights are probabilities.

Hereafter we call pairwise testing for a *weighted SUT* model *weighted pairwise testing*. A weighted SUT model is an SUT model with a *weighting function* w whose domain is V and range is the set of positive integers. The weighting function w is extended for parameter-value pairs as $w(v, v') = w(v) + w(v')$. We denote a weighted SUT by $\mathcal{S}(m; |V_1| \dots |V_m|; w)$.

For example, in the weighted SUT model in Table I, the weight of parameter-value *Win* is set to 1 and that of parameter-value *Mac* is set to 2. This represents that *Mac* is twice more important than *Win* in testing. For instance, the weight of a parameter-value pair (*Mac*, *Intel*) is calculated by $w(\textit{Mac}, \textit{Intel}) = w(\textit{Mac}) + w(\textit{Intel}) = 2 + 1 = 3$.

III. Related Work

Several techniques to generate weighted pairwise test suites have been proposed so far. They can be classified into those using weights for *ordering* test cases [1], [7], and those using weights for balancing the *occurrence frequency* of parameter-values [4], [6], [9]. They all adopt a "one-test-at-a-time" greedy style [8], which generates test cases one by one until all parameter-value pairs are covered, and consider weights when selecting each test case.

A. Order-focused (*CO*-based) Test Generation

The algorithms in the first category consider that high-weighted test cases should appear early in a test suite. The density algorithm [1] aims at covering parameter-value pairs of higher weights as early as possible, and thus generates test cases focusing on maximizing weights of newly covered parameter-value pairs. For our example SUT model, the density algorithm generates 12 test cases in Table II-(a). In contrast, CTE-XL [7] focuses on the weights of both uncovered and already-covered pairs, and generates test cases ordered according to their weights.

Both techniques share the advantage that important parameter-value pairs appear early in a test suite. However, they do not consider the test frequency aspect.

B. Frequency-focused (*CF*-based) Test Generation

The algorithms in the second category aim at testing highly weighted parameter-values frequently. PICT [4] generates a test case by choosing parameter-value pairs to maximize the number of newly-covered parameter-value pairs, and considers weights only when two choices of parameter-value pairs are equivalent according to an informal description in [4]. We observe that the frequency is reflected only mildly in PICT.

PictMaster [9] and Fujimoto et al. [6] aim at reflecting frequency more precisely; ideally, the number of occurrences of a parameter-value is proportional to its weight. In our example, for parameter *OS*, value *Mac* with weight 2 should appear twice as frequently as value *Win* with weight 1. PictMaster transforms a weighted SUT to a unweighted SUT by adding

redundant parameter-values according to their weights before passing it to PICT, and thus results in a larger test suite. For our example SUT, the PICT algorithm[1] generates 13 test cases in Table II-(c) while PictMaster generates 23 test cases. Fujimoto et al. developed another approach to add a given number of test cases to an existing test suite, in order to more accurately reflect given weights for value frequency.

The strength of these approaches is that more important values appear more frequently. On the other hand, they do not consider the order of important test cases, and often require large test suites.

C. Evaluation Metrics

The following metrics have been used to evaluate test suites:

- *Size*, i. e. number of test cases (denoted by $|\mathcal{T}|$) for CS,
- *Weight coverage* (denoted by WC) for CO, and
- *KL divergence* (denoted by D_{KL}) for CF.

To evaluate test suites w. r. t. CO, Bryce and Colbourn [1] and CTE-XL [7] use weight coverage, which is defined as

$$WC = \frac{\textit{Sum of weights of covered parameter-value pairs}}{\textit{Sum of weights of all parameter-value pairs}}.$$

For example, consider the first two test cases in Table II-(a). Since the sum of weights of covered parameter-value pairs is $24 + 19$ and that of all pairs is 162, WC is 26.54%.

To evaluate test suites w. r. t. CF, Fujimoto et al. [6] use KL divergence, which measures the difference between two probability distributions. KL divergence is defined as

$$D_{KL}(P||Q) = \textstyle\sum_{v \in V} P(v) \log(P(v)/Q(v)),$$

where $P(v)$ and $Q(v)$ respectively denote the current and the ideal occurrence frequencies for parameter-value v. In the ideal situation, the number of occurrences of each v is proportional to its weight. By definition, D_{KL} equals zero when $P = Q$, and it grows when the difference between P and Q is larger.

For example, in the test suite in Table II-(a), *Win*, *Unix*, and *Mac* appear four times, and thus the current distribution $P(i)$ is $4/12$ for each value i. The weight for each value is 1, 1, and 2, and thus the ideal distribution $Q(i)$ is $1/4$, $1/4$. and $2/4$ for each value i. By definition, D_{KL} is calculated as 0.306.

The sizes of generated test suites differ depending on algorithms. Let $|\mathcal{T}|_{min}$ denote the minimum size of such test suites. To evaluate CO and CF for the different-sized test suites generated by different algorithms, we consider WC and D_{KL} with the first $|\mathcal{T}|_{min}$ test cases. For example, the four test suites in Table II have different sizes and $|\mathcal{T}|_{min}$ is 12. Thus, we compare WC and D_{KL} of their first 12 test cases.

IV. Proposed Weighted Pairwise Testing

Our goal is to obtain a small test suite where important test cases appear early and frequently. In earlier work [2], we have proposed our basic idea of priority integration for weighted combinatorial testing, which integrates the three criteria of CS, CO, and CF. We assume that a priority order for combining the three criteria CS, CO, and CF is given, and we construct an algorithm for test generation according to the given prioritization order.

There are a lot of previous works on generating pairwise test suites focusing on CS and several works focusing on either CO or CF introduced in Section III. Our priority integration can exploit existing algorithms, and achieve better CO and CF at the same time. Here we present our weighted test generation algorithms applying priority integration to two existing algorithms, PICT [4] and the density algorithm [1].

Algorithm 1, called *PI$_{PICT}$*, shows the pseudo code of our priority integration on PICT algorithm [4]. In the original PICT algorithm, a parameter pair is assigned to cover the most uncovered parameter-value pairs one by one until all parameters of a new test case are assigned. To generate a test case according to the given priority order, we consider the criteria one by one in order, and select the best parameter-value assignment w. r. t. each criterion (line 9) as follows: For CS, as PICT does, we choose the parameter-value pair that maximizes the number of newly covered pairs, denoted by

Algorithm 1: PICT-based Priority Integration (PI$_{\text{PICT}}$).

```
   Input: Weighted SUT model S, Priority order
   Output: Pairwise test suite T
1  UC = { All pairs of parameter-values in S };
2  while UC ≠ ∅ do
3      Choose the parameter pair with the most pairs in UC, and
       assign its first pair to a new test case t;
4      Remove the assigned pair from UC;
5      while unassigned parameter exists in t do
6          C_i ← the first priority criterion;
7          finished ← false;
8          while finished is false do
9              List the best pairs in UC w. r. t. C_i [C_i];
10             if there is one candidate pair or no priority
               criterion lower than C_i then
11                 Assign one p of the best pairs to t;
12                 Remove the pairs covered by the assignment
                   of p from UC;
13                 finished ← true;
14             else
15                 C_i ← the next priority criterion;
16     Add t to T;
```

Algorithm 2: Density-based Priority Integration (PI$_{\text{Dens}}$).

```
   Input: Weighted SUT model S, Priority order CO.CF
1  while unassigned parameter exists in t do
2      Choose a parameter p with the maximum parameter
       interaction weight, and list the best values of p that
       maximize weighted density [CO];
3      if there is one candidate value then Assign it to p;
4      else Assign any of the candidate values that minimizes
       D_KL [CF];
5      Remove the pairs covered by the assignment from UC;
```

[1]Note that the PICT tool by Microsoft generates different 13 test cases for the same SUT. In this paper, we use our own implementation of the PICT algorithm [4] to compare it with our priority integration on it.

TABLE III: Experimental results by Algorithms PI_{PICT} and PI_{Dens} with various priority orders (for all 60 models).

Algorithm	Priority Order	Test Suite Size ($\lvert\mathcal{T}\rvert$)			Generation Time (s)			Weight Coverage (WC^*)(%)			KL Divergence (D^*_{KL})		
		g-μ	g-σ	# of wins	g-μ	g-σ	# of wins	g-μ	g-σ	# of wins	g-μ	g-σ	# of wins
PI_{PICT}	CS: PICT	31.295	6.478	18	0.016	0.409	19	81.396	0.350	0	5.748	2.338	0
	CO	32.411	6.369	7	0.017	0.426	19	83.194	0.342	21	5.397	2.477	0
	CF	44.075	6.724	0	0.132	1.456	8	73.151	0.378	0	2.428	1.180	8
	CS.CO	31.288	6.376	20	0.017	0.434	19	82.413	0.341	3	5.070	2.357	0
	CO.CS	31.342	6.399	13	0.017	0.434	19	82.556	0.346	4	5.828	2.495	0
	CS.CF	31.564	6.388	16	0.069	0.972	9	81.595	0.351	0	2.605	1.284	3
	CO.CF	32.530	6.390	8	0.070	0.959	9	**83.247**	0.342	**22**	**2.194**	1.256	23
	CS.CO.CF	31.496	6.377	12	0.069	0.973	9	82.385	0.342	1	2.209	1.227	**24**
	CO.CS.CF	31.643	6.397	13	0.068	0.956	9	82.586	0.346	3	2.464	1.258	3
PI_{Dens}	CO: Density	**30.831**	6.418	**30**	**0.010**	0.286	**60**	82.922	0.350	9	3.420	1.679	0
	CO.CF	30.943	6.397	28	0.010	0.288	30	82.895	0.351	8	2.434	1.318	5

N. For CO, we choose the one that maximizes the weight of newly covered pairs, denoted by W, to obtain the maximal WC. For CF, we choose the one to minimize D_{KL}.

For example, consider the SUT model in Table I again and priority order CS>CO>CF, hereafter denoted by CS.CO.CF. PI_{PICT} generates the pairwise test suite in Table II-(d). For the first test case, pair (*IE, Win*) is used (line 3), since the parameter pair (*Browser, Net*) has the most parameter-value pairs. Next, the following assignment is determined by the given priority order. For CS, assignments of any parameter-value pair for (*OS, CPU*) increase the same number of newly covered pairs, $N = 5$ (line 9). Among them, for CO, the assignment of (*Mac, AMD*) maximizes the weight increase of newly covered pairs, $W = 21$ (line 10). Hence, (*Mac, AMD*) is assigned next (line 11) for the first test case.

Algorithm 2, called *PI$_{Dens}$*, shows another example of priority integration with priority order CO.CF on the density algorithm [1]. Lines 1–5 correspond to lines 3–15 of Algorithm 1. Density algorithm generates a test case by assigning a value to one parameter after another. It chooses a parameter-value to maximize *weighted density*[2] taking account of CO (line 2). We can integrate CF by choosing a parameter-value minimizing D_{KL} when there are several candidate values for CO (line 4).

For the example SUT model and priority order CO.CF, PI_{Dens} generates the pairwise test suite in Table II-(b). The first nine test cases generated by the density algorithm (considering only CO) and by PI_{Dens} considering CO.CF are the same. However, for test cases 10–12, their WC are the same, but D_{KL} are different; We can see that PI_{Dens} considering CO.CF improves Density w. r. t. CF.

Hereafter, we denote by $\text{PI}_{\text{PICT}}{}^{O}$ (resp. $\text{PI}_{\text{Dens}}{}^{O}$) the variant of PI_{PICT} (resp. PI_{Dens}) that considers priority order O.

V. Experiments and Analysis

We set up experiments to investigate the effectiveness of our priority integration on existing algorithms, by comparing the PICT and density algorithms with our PI_{PICT} and PI_{Dens}. We implemented PI_{PICT} and PI_{Dens} in C, and simulated the PICT and density algorithms by $\text{PI}^{\text{CS}}_{\text{PICT}}$ and $\text{PI}^{\text{CO}}_{\text{Dens}}$, since their original implementation is not open.

We prepared 60 weighted SUT models; 35 of them are collected from an existing benchmark set [3], 18 are from an industrial case study [5], and the other seven are from related work [1]. From these benchmarks, we removed the constraints and assigned weights using random numbers following a normal distribution with $\mu = 5.0$ and $\sigma = 1.0$. The size of an SUT model is expressed as $g_1^{k_1} g_2^{k_2} \ldots g_n^{k_n}$, which indicates that for each i there are k_i parameters that have g_i values.

We evaluate the algorithms with test generation time and the three metrics explained in Section III-C; test suite size $|\mathcal{T}|$, weight coverage WC, and KL divergence D_{KL}. Since the test suite size differs depending on the SUT model, we use the normalized values of WC and D_{KL}, denoted by WC^* and D^*_{KL}, respectively; i. e., WC^* and D^*_{KL} denote WC and D_{KL} per test case. We also measure the *geometric mean* and *geometric standard deviation* (g-μ and g-σ) and the number of times that the algorithm has obtained the best results (# of wins).[3] We highlight the best results in Tables III and IV. Experiments were performed using a computer with Quad-Core Intel Xeon E5 3.7G Hz, with 64 GB memory running on Mac OS 10.9.4.

A. Effect of Priority Integration

In this section, we compare different priority orders using nine variants; three single-priority approaches (CS, CO, and CF), and six variants of priority integration (CS.CO, CO.CS, CS.CF, CO.CF, CS.CO.CF, and CO.CS.CF). Table III summarizes the results by PI_{PICT} and PI_{Dens} for all 60 models.

The results show that algorithms using a single priority (CS, CO, and CF) obtain good results for their metric ($|\mathcal{T}|$, WC^*, and D^*_{KL}, respectively). However, they get poor results for the other metrics. In particular, CF incurs quite large test suites, long computation time, and also low weight coverage. The reason is that CF disregards interaction coverage. For similar reasons, CS achieves low weight coverage (with high divergence), and CO also results in relatively high divergence. From the same reason, priority integration without CO (resp. CF) gives rise to high KL divergence (resp. low weight coverage).

On the other hand, priority integration of both CO and CF, such as CO.CF, CS.CO.CF, and CO.CS.CF, obtains a good bal-

[2]Due to space limitations, we elide detail of the density algorithm. Please refer to related work [1] for details on *weighted density*.

[3]The geometric mean avoids favoring larger benchmarks over smaller ones, which would be the case with the arithmetic mean.

TABLE IV: Comparison of existing algorithms PICT and Density, and our priority integration of CO.CF.

		Test Suite Size ($\|\mathcal{T}\|$)				Generation Time (s)				Weight Coverage (WC^*)(%)				KL Divergence (D^*_{KL})			
		PI_{PICT}		PI_{Dens}		PI_{PICT}		PI_{Dens}		PI_{PICT}		PI_{Dens}		PI_{PICT}		PI_{Dens}	
	SUT model	CS	CO.CF	CO	CO.CF	CS	CO.CF	CO	CO.CF	CS	CO.CF	CO	CO.CF	CS	CO.CF	CO	CO.CF
1	$2^{13}4^5$ (spins)	**21**	24	**21**	**21**	**0.001**	0.004	**0.001**	**0.001**	77.941	78.592	**78.951**	78.915	2.253	**1.294**	1.495	1.348
2	$2^{42}3^24^{11}$ (spinv)	32	32	32	**31**	0.014	0.091	**0.008**	**0.008**	85.410	**86.467**	86.412	86.440	6.327	**2.359**	3.624	2.610
3	$2^{189}3^{10}$ (gcc)	21	22	**19**	**19**	0.355	2.489	**0.166**	**0.166**	86.186	88.009	88.032	**88.034**	12.431	**9.903**	11.951	10.790
4	$2^{158}3^84^45^16^1$ (apache)	33	37	**32**	33	0.273	1.767	**0.129**	0.130	89.752	**91.039**	90.982	90.976	14.088	**5.592**	9.004	6.258
5	$2^{49}3^14^2$ (bugzilla)	**18**	20	**18**	19	0.008	0.044	**0.005**	**0.005**	82.861	**85.332**	85.318	85.316	4.282	**2.825**	3.473	3.166
6	$2^{86}3^34^15^56^2$	**43**	47	45	44	0.067	0.383	**0.031**	0.032	89.368	90.601	**90.610**	90.574	13.170	**2.683**	5.848	3.181
7	$2^{86}3^34^35^16^1$	**31**	36	**31**	32	0.049	0.290	**0.023**	0.024	88.205	**90.084**	89.960	89.950	8.845	**3.249**	5.135	3.773
8	$2^{27}4^2$	**17**	18	**17**	**17**	0.002	0.008	**0.002**	**0.002**	80.930	84.235	84.283	**84.296**	3.825	**1.987**	2.387	2.157
9	$2^{51}3^44^25^1$	25	26	**24**	**24**	0.012	0.067	**0.007**	**0.007**	85.758	87.172	**87.216**	87.215	5.402	**2.709**	3.642	2.998
10	$2^{155}3^74^35^56^4$	**52**	55	**52**	53	0.376	2.186	**0.164**	0.169	91.639	**92.583**	92.463	92.456	23.419	**3.960**	10.499	4.730
11	$2^{73}4^36^1$	**26**	29	27	27	0.025	0.152	**0.013**	0.014	87.352	**89.254**	88.737	88.736	6.856	**3.134**	4.445	3.718
12	$2^{29}3^1$	12	**11**	12	12	0.002	0.008	**0.002**	**0.002**	78.191	**80.912**	80.787	80.790	2.842	**2.319**	2.556	2.454
13	$2^{109}3^24^25^36^3$	45	46	45	**43**	0.116	0.635	**0.051**	0.052	90.278	**91.317**	91.292	91.306	13.932	**3.231**	6.644	3.764
14	$2^{57}3^14^15^16^1$	31	31	**30**	**30**	0.014	0.080	**0.008**	**0.008**	87.524	**89.941**	89.743	89.740	6.467	**2.226**	3.723	2.650
15	$2^{130}3^64^55^26^4$	**48**	51	49	**48**	0.219	1.313	**0.100**	0.102	90.927	**91.940**	91.772	91.771	18.526	**3.710**	8.650	4.290
16	$2^{84}3^44^25^26^4$	48	49	**47**	**47**	0.066	0.376	**0.031**	0.032	89.843	90.892	**90.917**	90.898	13.167	**2.699**	5.957	3.003
17	$2^{136}3^44^34^16^3$	44	46	**41**	**41**	0.192	1.158	**0.086**	0.087	90.863	92.145	**92.151**	**92.151**	15.151	**3.785**	8.425	4.605
18	$2^{124}3^44^15^26^2$	39	44	**37**	**37**	0.137	0.808	**0.061**	0.062	90.186	91.546	**91.696**	91.695	13.575	**3.795**	7.631	4.376
19	$2^{81}3^54^36^3$	44	43	41	**39**	0.054	0.308	**0.025**	0.026	89.061	**90.710**	90.476	90.472	12.067	**2.615**	5.652	3.312
20	$2^{50}3^44^15^26^1$	34	34	**32**	**32**	0.014	0.076	**0.008**	**0.008**	86.734	**88.463**	88.098	88.099	7.016	**2.161**	3.640	2.607
21	$2^{81}3^34^26^1$	**26**	30	**26**	**26**	0.036	0.217	**0.018**	**0.018**	87.408	**89.227**	89.105	89.105	8.094	**3.645**	5.118	4.161
22	$2^{128}3^34^25^16^3$	43	44	**40**	**40**	0.154	0.902	**0.068**	0.069	90.550	91.705	91.935	**91.937**	16.644	**3.814**	8.489	4.401
23	$2^{127}3^23^34^66^2$	**46**	50	47	47	0.198	1.132	**0.087**	0.089	90.664	91.755	**91.698**	91.693	17.044	**3.658**	7.938	4.218
24	$2^{172}3^94^95^36^4$	52	56	**50**	**50**	0.552	3.217	**0.237**	0.241	91.576	**92.418**	92.408	92.386	26.401	**4.619**	12.529	5.447
25	$2^{138}3^44^55^46^7$	**58**	61	**58**	60	0.331	1.855	**0.141**	0.145	91.411	**92.008**	92.004	92.004	24.935	**3.540**	10.858	4.169
26	$2^{76}3^34^25^16^3$	43	43	**40**	**40**	0.045	0.233	**0.020**	0.021	88.956	**90.563**	90.497	90.449	11.630	**2.677**	5.553	3.089
27	$2^{72}3^44^16^2$	37	37	**36**	**36**	0.030	0.169	**0.015**	**0.015**	88.804	90.824	91.084	**91.106**	9.815	**2.603**	5.026	2.853
28	$2^{25}3^16^1$	**18**	19	**18**	**18**	**0.002**	0.007	**0.002**	**0.002**	80.606	82.899	**84.069**	84.039	3.580	**1.609**	1.966	1.659
29	$2^{110}3^25^36^4$	**49**	52	**49**	50	0.112	0.653	**0.051**	0.052	90.951	**92.098**	91.910	91.926	14.911	**2.782**	7.195	3.432
30	$2^{118}3^64^25^26^6$	**54**	55	55	**54**	0.179	1.046	**0.082**	0.084	91.446	**92.230**	92.187	92.183	21.244	**3.034**	9.302	3.486
31	$2^{87}3^14^35^4$	**33**	35	**33**	35	0.052	0.320	**0.026**	**0.026**	88.696	89.772	**89.913**	**89.913**	10.735	**3.073**	5.810	3.640
32	$2^{55}3^24^25^16^2$	**37**	38	38	**37**	0.017	0.094	**0.009**	0.010	87.464	**89.584**	89.508	89.502	7.723	**2.037**	3.676	2.340
33	$2^{167}3^{16}4^25^36^6$	56	58	**53**	54	0.522	3.054	**0.234**	0.238	91.909	**92.625**	92.606	92.596	27.883	**4.519**	11.961	5.444
34	$2^{134}3^75^3$	32	31	29	**28**	0.153	0.990	**0.073**	0.074	88.913	**90.489**	90.350	90.347	10.532	**5.082**	7.441	5.701
35	$2^{73}3^34^3$	21	24	**20**	**20**	0.026	0.167	**0.013**	**0.013**	85.036	87.003	**87.012**	87.008	6.010	**3.758**	4.836	4.296
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:
	g-μ	31.295	32.530	**30.831**	30.943	0.016	0.070	**0.010**	**0.010**	81.396	**83.247**	82.922	82.895	5.748	**2.194**	3.420	2.434
	g-σ	6.478	6.390	6.418	6.397	0.409	0.959	0.286	0.288	0.350	0.342	0.350	0.3501	2.338	1.256	1.679	1.318
	# of wins	18	8	**30**	28	19	9	**60**	30	0	**22**	9	8	0	**23**	0	5

ance of weight coverage and KL divergence. Among the three priority orders, CO.CF achieves the highest improvement rates over CS for WC^* and D^*_{KL}; $2.3\% = (83.247 - 81.396)/81.396$ for WC^* and $61.8\% = (5.748 - 2.194)/5.748$ for D^*_{KL}. Recall that WC^* is the normalized weight coverage per test case, and hence the total improvement can be large for big test suites.

We can also see that applying CO.CF integration improves PICT, achieving better weight coverage than Density does. On the other hand, w. r. t. test suite size, CS.CO.CF and CO.CS.CF are better than CO.CF, and the difference in test suite size (resp. generation time) with PICT is less than one (resp. 0.1 seconds) on average.

To summarize, compared to existing algorithms considering a single priority, our priority integration considering both CO and CF improves weight coverage and KL divergence simultaneously on the same-sized test suites, with a small overhead for test generation time.

B. Detailed Results for Priority Order CO.CF

In this section, we further investigate the results of our priority integration with CO.CF, which seems to be a promising priority order from the result of Table III. Table IV compares $|\mathcal{T}|$, test generation time, WC^*, and D^*_{KL} of PICT, Density, and priority integration with CO.CF on them. Recall that PI^{CS}_{PICT} and PI^{CO}_{Dens} respectively correspond PICT and Density algorithms. Due to space limitation, we show the results for 35 benchmark models [3] among all 60 models we used.[4]

From the results, $PI^{CO.CF}_{PICT}$ improves PICT on both weight coverage and KL divergence for all models. The (geometric) mean improvement rate for all 60 models is 2.3% and 61.8% for WC^* and D^*_{KL}, respectively. On the other hand, $PI^{CO.CF}_{Dens}$ improves Density on KL divergence for all models. The improvement rate of D^*_{KL} is 28.8% on average. Since Density already considers CO, WC^* is almost the same after priority integration. We confirmed that the difference of $PI^{CO.CF}_{PICT}$ and PICT for WC^* and D^*_{KL} is significant, while that of $PI^{CO.CF}_{Dens}$ and Density is significant for D^*_{KL} with $p < 0.01$, using the Wilcoxon signed-rank test [10].

C. Detailed Results for Selected Benchmarks

In this section, we look into the change of weight coverage and KL divergence on test cases by four algorithms, PICT, Density, $PI^{CO.CF}_{PICT}$, and $PI^{CO.CF}_{Dens}$, for four large empirical models:

[4]See http://staff.aist.go.jp/e.choi/compsac2015/results.html for the entire results and larger sized graphs.

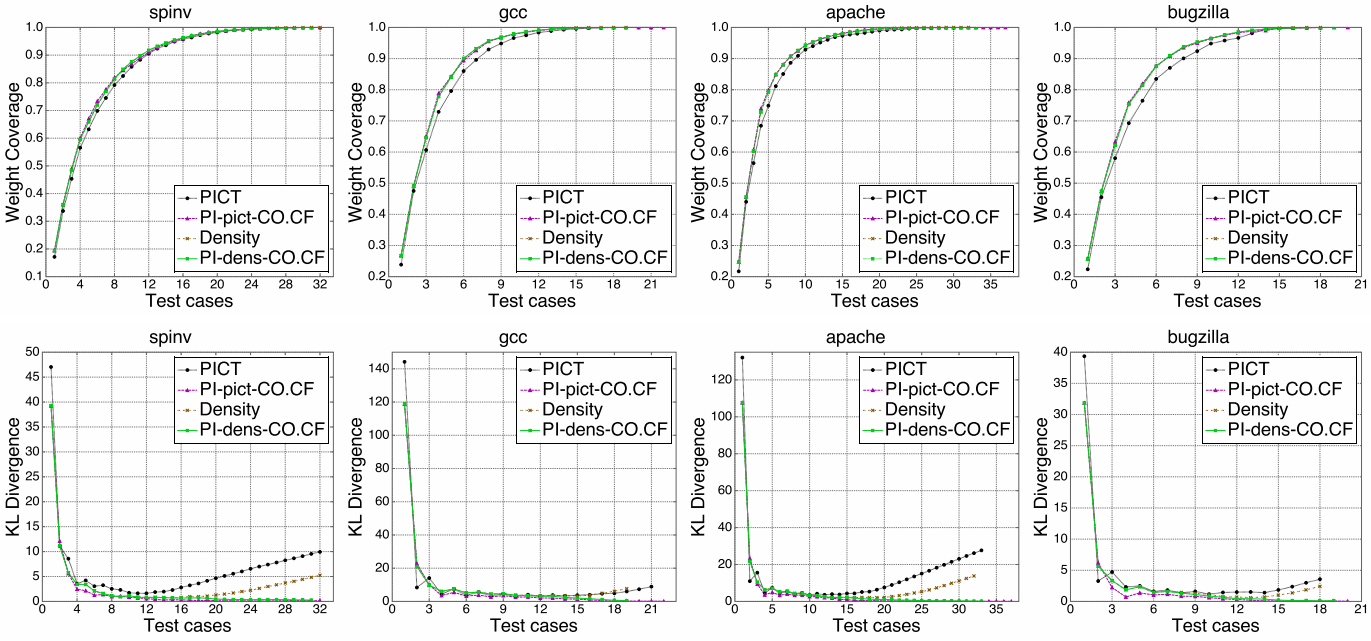

Fig. 1: Weight coverage and KL divergence by PICT, Density, $\text{PI}^{\text{CO.CF}}_{\text{PICT}}$, and $\text{PI}^{\text{CO.CF}}_{\text{Dens}}$ for 4 large empirical models.

spinv, gcc, apache, and bugzilla (models 2–5 in Table IV). The results are shown in Fig. 1.[4]

The results show that $\text{PI}^{\text{CO.CF}}_{\text{PICT}}$ obtains high weight coverage earlier than PICT. The final difference in coverage may not appear significant. However, we can notice that CO.CF integration by PI_{PICT} improves weight coverage of PICT to about the same level as Density with CO. For instance, PICT requires 10 test cases to achieve 95% weight coverage for the gcc model, but $\text{PI}^{\text{CO.CF}}_{\text{PICT}}$ requires 8 test cases, the same number that Density requires.

KL divergence shows astounding improvements by $\text{PI}^{\text{CO.CF}}_{\text{PICT}}$ and $\text{PI}^{\text{CO.CF}}_{\text{Dens}}$. We can see that the divergence by PICT and Density becomes larger, i. e., more distant from the ideal state of parameter-value frequency, when the number of test cases grows. $\text{PI}^{\text{CO.CF}}_{\text{PICT}}$ and $\text{PI}^{\text{CO.CF}}_{\text{Dens}}$ bring the divergence close to the ideal value, 0. For example, KL divergence of the test suites with the minimum number ($= 31$) generated by PICT, $\text{PI}^{\text{CO.CF}}_{\text{PICT}}$, Density, and $\text{PI}^{\text{CO.CF}}_{\text{Dens}}$ for the spinv model are 9.536, 0.162, 4.831, and 0.278, respectively. Hence, the improvement on KL divergence by $\text{PI}^{\text{CO.CF}}_{\text{PICT}}$ (resp. $\text{PI}^{\text{CO.CF}}_{\text{Dens}}$) reaches 98.3% (resp. 94.2%).

VI. Conclusion

In this paper, we propose algorithms that integrate order-focused and frequency-focused prioritizations for weighted pairwise testing. Experimental results show that we realize a practical way of constructing a small pairwise test suite achieving better weight coverage and also significantly better divergence by applying our priority integration to existing one-test-at-a-time test generation algorithms.

We consider several directions for the future. Even though we consider pairwise testing in this paper, priority integration for t-wise testing with a higher combinatorial strength t can also be managed by enumerating parameter-value t-tuples instead of pairs in our algorithms. In addition, for practical SUTs, we need to avoid test cases violating constraints between parameter-values. We are also interested in investigating an efficient way of extracting weights for SUT models, which are important inputs to determine the test effectiveness of weighted combinatorial testing.

Acknowledgments: The authors would like to thank anonymous referees for their helpful comments. This work is partly supported by JST A-Step grant AS2524001H.

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, and Y. Oiwa. Design of prioritized N-wise testing. In *Proc. of the 26th IFIP International Conference on Testing Software and Systems (ICTSS)*, pages 186–191, 2014.

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

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

[5] N. Do, T. Kitamura, N. Tang, G. Hatayama, S. Sakuragi, and H. Ohsaki. Constructing test cases for N-wise testing from tree-based test models. In *Proc. of the fourth International Symposium on Information and Communication Technology (SoICT)*, 2013.

[6] S. Fujimoto, H. Kojima, and T. Tsuchiya. A value weighting method for pair-wise testing. In *Proc. of the 20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 99–105, 2013.

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

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

[9] PictMaster. http://sourceforge.jp/projects/pictmaster/.

[10] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.