Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 1323–1330

ISBN 978-83-60810-51-4

Comparison of Approaches to Prioritized Test Generation for Combinatorial Interaction Testing

Peter M. Kruse

Berner & Mattner Systemtechnik GmbH
Berlin, Germany
Email: peter.kruse@berner-mattner.com

Ina Schieferdecker

Fraunhofer FOKUS
Berlin, Germany
Email: ina.schieferdecker@fokus.fraunhofer.de

***Abstract*—Due to limited test resources, it is often necessary to prioritize and select test cases for a given system under test. Although test case prioritization is well studied and understood, its combination with test data generation is difficult and not completely solved yet. For example, the Classification Tree Method is a well established method for test data generation, however the application of prioritization techniques to it is a current research topic. We present an extension of the classification tree method that allows the generation of optimized test suites, containing test cases ordered according to their importance with respect to test goals. The presented algorithms are incorporated into the Classification Tree Editor and empirically evaluated on a set of benchmarks.**

***Keywords*-classification tree method, prioritized test case generation, test suite optimization**

I. Introduction

SOFTWARE testing is one of the activities in which limited resources are confronted with the need for exhaustive tests, for which infinitely many or too many tests need to be performed. Furthermore in typical industry-driven software development, testing time is often further reduced as the software under test may not be completed in time and release dates cannot be delayed. Hence, software testers have to deal with multiple challenges: Their resources are limited, the software comes too late, and scheduled time is not available. Testers have to handle such situations: not only due to time constraints, but also due to other resource constraints such as test personnel, test infrastructure, etc., not all test cases can be performed.

Hence, testers need to select those test cases with most relevance - by keeping a sufficient level of test coverage and failure-detection capability. Yet, there is no guarantee that selected test cases are more important than any of the others: there is no well-established way to determine the relevance and importance of test cases in a test suite and to select the best ones in relation to available resources.

In this article, we present an approach for test case selection and test suite optimization using the classification tree method. We introduce the weighting of classification tree elements allowing us to generate test suites with prioritized test cases. These test suites can be optimized by selecting only subsets of test cases when taking resource limits into account. Furthermore, coverage criteria are proposed for identifying the importance of a specific test case under given test aspects. Finally, an evaluation of the performance of prioritized test case generation as compared to plain test case sorting and two other existing approaches is given.

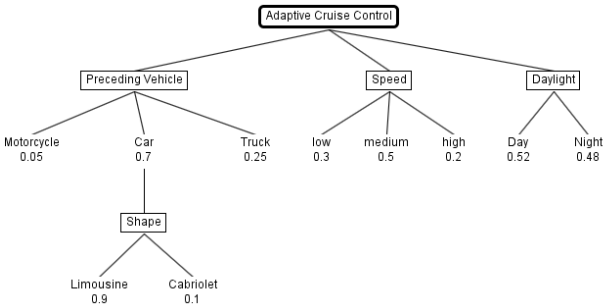

Fig. 1. Usage model for test object *ACC*

II. Definitions and Background

A. Classification Tree Method and Classification Tree Editor

The **Classification Tree Method** [1] aims at systematic and traceable test case identification for functional testing over all test levels (for example, component test or system test). It is based on the category partition method [2], which divides a test domain into disjunctive classes representing important aspects of the test object.

Applying the classification tree method involves two steps—designing the classification tree and defining test cases.

Design of the classification tree. The classification tree is based on the functional specification of the test object. Figure 1 shows an example tree for the adaptive cruise control (ACC) test object. For each aspect of interest (a so called classification), the input domain is divided into disjoint subsets (so called classes). In our example, classifications are *preceding vehicle*, *speed*, and *daylight*. Classes for speed are *low*, *medium*, and *high*. The class *car* is further refined into different *shapes*, which are *limousine* and *cabriolet*. Further refinements allow test objects to be described on any level of granularity.

Definition of test cases. Having composed the classification tree, test cases can be defined by combining classes from

different classifications. Since classifications only contain disjoint values—obviously speed cannot be low and high at the same time—test cases cannot contain several values of one classification.

The **Classification Tree Editor** (CTE XL) was introduced together with the classification tree method. Current versions of the CTE XL support automated test case generation and user-defined dependency rules [3]. Current test case generation offers four different coverage modes: *Minimal combination* creates a test suite that uses every class from each classification at least once in a test case. *Pairwise combination* creates a test suite that uses every class pair from disjunctive classifications at least once in a test case. *Threewise combination* ("triple-wise") creates a test suite that uses every triple of classes from disjunctive classifications at least once in a test case. *Complete combination* creates a test suite that uses every possible combination of classes from disjunctive classification in a test case.

B. Prioritization

Prioritization is used to assign values of importance to classification tree elements. These values of importance are called *weights*. To cover various kinds of test aspects, these weights can differ. Higher and lower weights should reflect higher and lower importance, respectively. Consequently, one is able to compare the elements of the classification tree to determine their importance under a given test aspect and to guide test case generation by priorities. Elbaum et al. provide good overviews of existing approaches [4], [5]. There has been some work on test case prioritization that considered limited resources [6], [7]. In [8] a first approach for combining classification trees with priorities has been presented.

C. Combinatorial Interaction Testing

Combinatorial Interaction Testing (CIT) [9] is an effective testing approach for detecting failures caused by certain combinations of components or input values. The tester identifies the relevant test aspects and defines corresponding classes. These classes are called *parameters*, their elements are called *values*. We assume the parameters to be disjoint sets. A test case is a set of n values, one for each parameter. In the classification tree method, the parameters are called *classifications*, the values are called *classes*.

CIT is used to determine a smallest possible subset of tests that covers all combinations of values specified by a coverage criterion with at least one test case. A coverage criterion is defined by its strength t that determines the degree of parameter interaction and assumes that all parameters are considered.

The most common coverage criterion is 2-wise (or *pairwise*) testing, that is fulfilled if all possible pairs of values are covered by at least one test case in the result test set. A large number of CIT approaches have been presented in the past. A good overview and classification of approaches can be found in [10] and [11]. A good survey that focuses on CIT with constraints is given in [12]. Nearly all publications investigate pairwise combination methods, but most of them can be extended to arbitrary t-combinations.

There are only two known algorithms supporting prioritized test case generation. The first is the deterministic density algorithm (DDA) published in [13], which is an extension to [14]. The extended algorithm generates a test suite successively constructing single test cases. During test case construction it accounts for (1) uncovered pairs in the test suite generated so far and (2) user assigned weights. Pairs with higher weights are covered earlier than pairs with lower weights. For efficiency reasons, this algorithm does not consider explicit dependencies.

The other algorithm [15] maps the test problem to a binary decision diagram (BDD) and reads test cases in descending order of importance from it. The BDD is used to support explicit dependencies, too.

III. Contribution

We analyzed several existing prioritization techniques, from which the following three models were selected as a basis for prioritization:

Prioritization based on a usage model [16] tries to reflect usage distribution of all classes in terms of usage scenarios. Classes with a high occurrence have higher weights than classes with a low occurrence.

Prioritization based on an error model [4] aims to reflect distribution of error probabilities of all classes. Classes with a high probability of revealing an error have higher weights than classes with a low probability.

Prioritization based on a risk model [17] is similar to prioritization based on an error model but also takes error costs into account. Classes with a high risk have higher weights than classes with a low risk.

Example. Figure 1 shows the adaptive cruise control with assigned occurrence values, which represent values of importance. As the figure shows, *medium* is the most probable *speed*. *Low* and *high* are subsequent in descending order of importance. The values of all classes of one classification sum to `1` in the occurrence model. For example, class *car* has an occurrence rate of `0.7` of all *preceding vehicles*.

The values of refinements of classes are interpreted as conditional probabilities in the occurrence model. For example, if the *preceding vehicle* is a *car*, *limousines* have an occurrence rate of `0.9` and *cabriolets* of `0.1`. Hence, the resulting occurrence probability for a *limousine* being the *preceding vehicle* is `0.63 (= 0.7 * 0.9)`, and for a *cabriolet* it is `0.07`, accordingly.

The assigned values can also be taken to represent error probabilities, i.e. the probability that an error occurs in this class. Combined with costs associated to classes, risks can be calculated as well: as error probability times cost value. Since error values are independent of each other, an interpretation as conditional probabilities fails in the error model. The same applies for the risk values in the risk model. Therefore, the values in these two models are taken as absolute values, which we call *weights* in the following.

```
1: S //result list
2: M //set of all valid test cases
3: A //set of test cases containing pair p
4: P //set of uncovered class pairs
5: D //set of (test case, index value) pairs
6: while (|P| > 0) do
7:     p = select max weight pair from P
8:     A = filter M by p
9:     if |A| > 1 then
10:        D = calculateIndex(A, P)
11:        t = selectMaxIndex(D)
12:    else
13:        t = take single test case from M
14:    end if
15:    S = append(S, t)
16:    P = P − {classPairs(t)}
17:    M = M − {t}
18: end while
19: return S
```

Fig. 2. PPC algorithm

A. Coverage Criteria

We define coverage criteria to measure the degree to which the system under test is tested with an optimized test suite. Then, we compare optimized test suites of different sizes to evaluate coverage gains from additional test resources. At first, we define the criterion *weight coverage (WC)* that measures the degree to which the weights are covered:

$$WC = \frac{\text{sum of weights of covered class pairs}}{\text{sum of weights of all coverable class pairs}}$$

The metric is relative, i.e. considers the fact that classes may not be coverable because of dependencies.

B. Prioritized Pairwise Combination

A plain pairwise combination creates a test suite that covers each class pair in at least one test case. We extended this combination to a *prioritized pairwise combination (PPC)*. The combined weight for class pairs is calculated by multiplying either the occurrence or error probabilities. By contrast, the combined risk is the product of the summed individual costs and the multiplied individual errors. Test cases are selected and generated in descending order of importance.

Figure 2 provides the algorithm in pseudocode. The most weighted class pair from all class pairs not yet covered is chosen for a new test case to be added: We determine all candidate test cases containing this class pair and calculate the index values for these candidates. The index value of a test case is the sum of the weights of the added pairs and the quotient of the number of newly covered pairs and the overall number of coverable pairs in a test case.

PPC selects the test case with the highest assigned index value. By that, it guarantees at least coverage of the *n* most important class pairs by the *n* first test cases. The generated test suite may be slightly larger than the result of the plain pairwise combination since weights are taken into account. Please also note that the generation process using PPC is deterministic: the same test suite is generated for a given classification tree.

The result of PPC applied to our ACC example is given in Table I. The last column contains the value of the weight coverage. It shows that the first three test cases of the test suite cover already 70% of all class pairs' weights. For weight coverage of 90%, 95%, or 99%, only six, eight, or 10 test cases are to be executed. Depending on testers' needs or available resources, test efforts can be reduced in a controlled way.

TABLE I
RESULTING PPC TEST SUITE FOR USAGE MODEL

	Prec. Vehicle	Speed	Daylight	WC
#1	Limousine	medium	Day	0.32
#2	Limousine	low	Night	0.56
#3	Truck	medium	Night	0.70
#4	Truck	low	Day	0.8
#5	Limousine	high	Day	0.88
#6	Truck	high	Night	0.92
#7	Cabriolet	medium	Day	0.94
#8	Motorcycle	medium	Day	0.96
#9	Cabriolet	low	Night	0.98
#10	Motorcycle	low	Night	0.99
#11	Cabriolet	high	Night	0.99
#12	Motorcycle	high	Night	1

C. Plain Pairwise Sorting

In addition to PPC, we have analyzed *Plain Pairwise Sorting (PPS)* where a sorting approach based on class pair weights is applied to the results of a plain pairwise algorithm. The calculation of the weights is same as above.

The sorting brings all test cases into an order such that the weight covered by first test cases is maximized. The algorithm sorts all test cases by their absolute weight at first. Then, it applies as many discriminatory reorderings as there are test cases.

Please note that this approach does not guarantee coverage of any *n* most important class pairs by the *n* first test cases. However, the generated test suite will have exactly the same size as the plain pairwise combination, as the suite does not grow by sorting. The generation process using PPS is deterministic too; however its results differ from the PPC results.

Figure 3 and Figure 4 provide the algorithms in pseudocode. The insertion algorithm requires an initialization. For each class pair, the combined weight needs to be calculated by multiplication. A HashMap is filled with the class pair being the keys and the combined weights being the values.

The result of PPS applied to our ACC example is given in Table II.

IV. EMPIRICAL RESULTS

We evaluated the impact of weight considerations on weight coverage and on the absolute size of the generated test suites. A first impression is given in Figure 5. As can be seen, PPC and PPS perform similar, while the unsorted test suite does not perform as well. For a more detailed and systematic evaluation, we use the set of benchmarks proposed in [13] and also used in [15]: We compare (1) our PPC approach with our PPS approach, (2) PPC with the deterministic density algorithm (DDA) given in [13] and (3) PPC with the BBD approach

```
1: insert(List of testCases, testCase t)
2: double tcWeight = 0.0
3: /* Calculate insertion weight */
4: for all class pair in t do
5:     tcWeight += classPairWeights.get(class pair)
6: end for
7: /* Find position for insertion */
8: int i = 0
9: for all testCaseWeights do
10:    if (testCaseWeight ¡ tcWeight) then
11:        testCaseWeights.add(i, tcWeight)
12:        testCases.add(i, t)
13:        return
14:    end if
15:    i++
16: end for
17: /* All existing test cases have higher weights, append new test case at
    the end. */
18: testCaseWeights.add(tcWeight)
19: testCases.add(t)
```

Fig. 3. Sort algorithm insertion

```
1: finalize(List of testCases)
2: if (|testCases| < 2) then
3:     return
4: end if
5: testCase firstElem = testCases.getFirst
6: /* Set weight of all covered pairs to 0 */
7: for all class pair in firstElem do
8:     classPairWeights.put(class pair, 0)
9: end for
10: List of testcases tail = l.subList(1, |testCases|)
11: /* Sort tail using insertion sort */
12: List of testcases newTail = new List
13: testCaseWeights.clear
14: for all testcase in tail do
15:     insert(newTail,testcase)
16: end for
17: /* Finalize tail */
18: finalize(newTail) // recursion
19: testCases = firstElement + newTail
```

Fig. 4. Sort algorithm finalization

given in [15]. The benchmark uses four different weight distributions applied to eight scenarios. The distributions are:

- d_1 *(Equal weights).* All classes have the same weight,
- d_2 *(*$50/50$ *split).* Half of the weights for each classification are set to `0.9` the other half to `0.1`,
- d_3 *(*$(1/v_{max})^2$ *split).* All weights of classes for a classification are equal to $(1/v_{max})^2$, where v_{max} is the number of classes associated with the classification,
- d_4 *(Random).* Weights are randomly distributed.

TABLE II
RESULTING SORTING TEST SUITE FOR USAGE MODEL

	Prec. Vehicle	Speed	Daylight	WC
#1	Limousine	medium	Night	0.31
#2	Limousine	high	Day	0.52
#3	Truck	medium	Day	0.66
#4	Limousine	low	Night	0.77
#5	Truck	high	Night	0.85
#6	Motorcycle	low	Day	0.9
#7	Cabriolet	medium	Day	0.93
#8	Cabriolet	low	Night	0.96
#9	Truck	low	Night	0.98
#10	Motorcycle	medium	Night	0.99
#11	Cabriolet	high	Night	0.99
#12	Motorcycle	high	Night	1

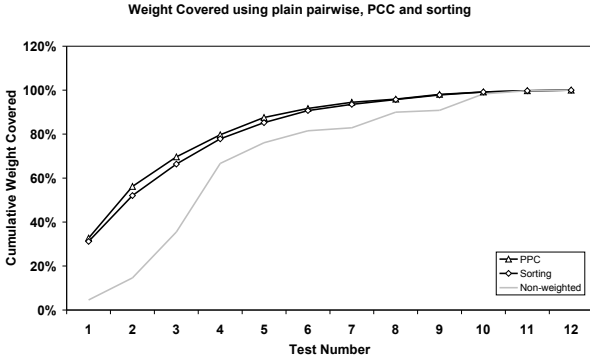

Fig. 5. Comparison of PPC, PPS, and of non-weighted generation

The scenarios $s_1, ..., s_8$ with their resulting test suite sizes are given in Table III. The classifications are given in a shorthand notation, where for example s_5 with $8^2 7^2 6^2 2^4$ consists of 2 classifications with 8 classes, 2 classifications with 7 classes, 2 classifications with 6 classes, and 4 classifications with 2 classes.

A. Comparison of PPC vs. PPS

Size. As shown in Table IV in the 100% weight column, the test suites generated by the PPC are larger than those generated using the PPS approach. Table IV also contains values for the BDD generation, which we will discuss later. The PPC generated scenarios s_3 and s_7 in d_1 and d_3 are up to 300% larger than their PPS counterparts. The problem here seems to be the combination of scenario and distribution. For any other distribution or scenario, the PPC results are much smaller, therefore, closer to the PPS results. For the rest, the resulting PPC test suites are up to 50% larger in some cases (s_3 d_4 or s_7 d_2). On average, the PPC test suites are 42% larger than the PPS test suites. Ignoring the extreme values (s_3 and s_7 from d_1 and d_3), the PPC test suites are 12% larger on average.

Applying equal sorting (d_1), the PPC algorithm results in smaller test suites for some scenarios. For s_2, it is generally smaller. The normal test case generation serving as input for the test case sorting, seems to have problems with this particular scenario. The reasons behind this behavior need further investigation. For s_6 d_3, the PPC test suite has the same size as the PPS.

For the majority of scenario-distribution combinations, PPS results in smaller test suites, which is not surprising since in prioritization selection focuses on weights first; in contrast to PPS, which tries to get the smallest test suite possible.

TABLE III
BENCHMARK SCENARIOS

s_1	3^4	s_4	$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1$	s_7	$3^{50} 2^{50}$
s_2	10^{20}	s_5	$8^2 7^2 6^2 2^4$	s_8	$20^2 10^2 3^{100}$
s_3	3^{100}	s_6	$15^1 10^5 5^1 4^1$		

TABLE IV
BENCHMARK DETAILED RESULT, PART 1

		50% weight			75% weight			100% weight		
		BDD	PPC	PPS	BDD	PPC	PPS	BDD	PPC	PPS
Distribution 1	s_1	5	5	5	7	8	7	10	10	9
	s_2	62	56	60	112	95	110	251	218	245
	s_3	6	6	6	11	9	10	33	130	34
	s_4	26	24	22	47	45	42	94	92	94
	s_5	19	18	17	37	35	34	74	71	74
	s_6	62	52	49	105	89	92	184	172	177
	s_7	4	2	2	8	7	8	28	70	29
	s_8	13	3	9	35	18	21	400	400	400
Distribution 2	s_1	1	1	2	3	3	4	14	12	9
	s_2	21	18	36	49	36	79	278	228	245
	s_3	1	1	4	4	4	8	35	51	34
	s_4	10	8	11	22	16	23	100	100	94
	s_5	7	6	8	15	13	21	83	78	74
	s_6	28	17	24	55	31	53	207	190	177
	s_7	1	1	3	2	2	5	29	42	29
	s_8	3	3	3	16	12	13	400	403	400
Distribution 3	s_1	5	5	5	7	8	7	10	10	9
	s_2	62	56	60	112	95	110	249	218	245
	s_3	6	6	6	11	9	10	33	130	34
	s_4	8	8	8	18	16	19	137	109	94
	s_5	5	4	4	10	9	10	101	86	74
	s_6	27	26	26	53	46	53	222	177	177
	s_7	4	3	4	7	6	7	33	84	29
	s_8	6	6	8	11	9	11	463	420	400
Distribution 4	s_1	4	3	3	6	6	5	13	12	9
	s_2	44	31	47	94	65	95	286	229	245
	s_3	5	3	5	11	7	10	47	52	34
	s_4	17	15	18	33	29	36	108	106	94
	s_5	10	8	10	24	19	24	88	80	74
	s_6	35	27	36	70	56	73	208	187	177
	s_7	3	2	4	7	5	7	42	33	29
	s_8	8	5	8	26	15	21	406	404	400

TABLE V
BENCHMARK DETAILED RESULT, PART 2

		25%		66%		90%		95%		99%	
		PPC	PPS	PPC	PPS	PPC	PPS	PPC	PPS	PPC	PPS
Distribution 1	s_1	3	3	7	6	9	9	10	9	10	9
	s_2	27	27	79	89	132	162	152	190	180	228
	s_3	3	3	8	8	15	16	20	20	37	26
	s_4	11	10	36	34	64	63	74	74	86	88
	s_5	8	7	28	27	50	51	57	59	66	68
	s_6	23	22	74	74	123	131	139	149	159	169
	s_7	2	2	6	6	11	12	14	15	20	21
	s_8	3	3	13	15	64	46	92	62	145	120
Distribution 2	s_1	1	1	1	3	7	5	8	7	11	8
	s_2	9	12	27	60	87	131	121	163	169	212
	s_3	1	2	1	7	9	13	12	16	19	23
	s_4	3	4	12	17	36	38	53	51	76	77
	s_5	2	3	10	15	28	37	43	46	61	62
	s_6	7	9	25	39	73	92	106	114	148	153
	s_7	1	2	1	4	6	9	9	12	14	18
	s_8	1	1	7	8	30	30	56	47	114	122
Distribution 3	s_1	3	3	7	6	9	9	10	9	10	9
	s_2	27	27	79	89	132	162	152	190	180	228
	s_3	3	3	8	8	15	16	20	20	37	26
	s_4	4	4	12	14	30	35	42	46	72	70
	s_5	2	2	7	7	15	19	23	25	52	54
	s_6	12	12	38	41	83	92	107	120	149	158
	s_7	2	2	5	5	10	10	13	13	21	19
	s_8	3	3	8	9	15	19	19	26	28	45
Distribution 4	s_1	2	2	5	5	8	7	10	8	11	9
	s_2	12	19	50	74	104	150	129	181	169	223
	s_3	1	3	5	8	12	15	15	19	21	26
	s_4	6	7	23	28	46	56	59	67	82	82
	s_5	3	4	14	17	33	41	44	50	60	65
	s_6	11	14	43	57	90	111	112	131	148	160
	s_7	1	2	4	6	9	11	11	14	17	21
	s_8	2	3	10	15	35	43	55	60	120	137

Weight coverage. The detailed results are given in Table IV and V. Table IV contains results of the BDD approach and our own two contributions for the coverage levels of 50%, 75% and 100%. Table V contains additional result values for the coverage levels of 25%, 66%, 90%, 95%, and 99% for what no BDD results are known.

In d_1, PPS starts very strong. Reaching 25% coverage is always better than or equal to PPC. For 50% and 66%, PPC is only better in s_2, where PPS seems to have a general weakness, and s_8. For the rest of measured weights, PPC becomes better and better, while PPS loses its advantages from the lower values.

In d_2, the prioritization performs better in all scenarios for the 25%–75% target weights. Starting at 90%, PPS gets better with s_1 and s_8. For 95%, PPS also needs fewer test cases with s_4. This case needs further investigation, maybe it is just a good weight distribution for the test case generator creating the PPS input. For 99%, PPS only performs best for s_1. For 100%, PPS perfoms better in all but s_2.

In d_3, PPS comes close to the PPC. For 25% coverage, both approaches perform equally. For 50%, the PPC is only better for s_2 and s_7. For 66% weight coverage, the prioritization is better for s_2, s_4, s_6, and s_8. For 75% coverage, PPC is always better for large scenarios (s_2, ..., s_7). For 90% and 95% coverage, PPS performs equally both for s_7 and s_3 for the latter. For 99% coverage, PPS surpasses PPC for half of all scenarios.

In d_4, the prioritization approach has the best weight coverage. The random distribution of weights leads to a high weight coverage when performing test case composition with its pair selection.

In general, the PPC gives better weight coverage. There are, however, three exceptions: Very small scenarios, problematic distributions, and very high weight marks. For *very small scenarios*, PPS has a good starting point, since any pairwise-covering test suite has a good chance for containing combinations of any class pair, so as a consequence, even combinations of only high-weight pairs. In these cases, PPS can sort the good combinations to the beginning. The influence of *problematic distributions* has already been analyzed in detail. PPC performs better on random and on $(1/v_{max})^2$ split while it has some problems on equal distributions (with low target covering marks) and is on par with the PPS approach on the $50/50$ split. For higher target marks on equal distributions, it becomes better again since PPS has a general problem here. For *very high weight marks* starting at around 95% or 99% and even higher, PPC loses its advances gradually. Since PPS performs better for 100% coverage, there obviously must be a point $\leq 100\%$ where both approaches perform equally.

To conclude. Having two algorithms which both generate test suites covering all possible pair combinations, the PPC covers weights better than the PPS approach, because it tries

to combine high weight pairs into early test cases. The PPS approach is worse, because it has no influence on the actual composition of test cases with their contained pairs. So while both test suites contain all possible class pairs, PPC does early weight coverage.

B. Comparison of PPC with DDA

Furthermore, we compared PPC with DDA with respect to size and weight coverage.

Size. Comparing PPC and DDA test suite sizes, there is no clear result. From 32 test suites generated with DDA and PPC, 18 DDA suites are smaller than the PPC test suites. For the remaining 14 scenario-distribution combinations, PPC generates smaller test suites. As already stated, PPC produces a very large test suite for the s_3 and s_7 in d_1 and d_3 combinations. The DDA produces smaller test suites for these combinations, similar to the PPS results. The DDA, however, has two outliers with s_2 and s_6: The result set is 50% larger for s_2 and 25% larger for s_6 compared with PPC.

For the majority of results, both algorithms perform similarly. For d_2, the PPC has some advantages, for d_1 and d_3 DDA performs better. The scenario s_6 seems to be a good PPC scenario, while s_3 and s_7 are handled well by DDA. There is no general tendency for one or the other to produce considerably different test suite sizes since both algorithms aim to cover high weight instead of generating small test suites.

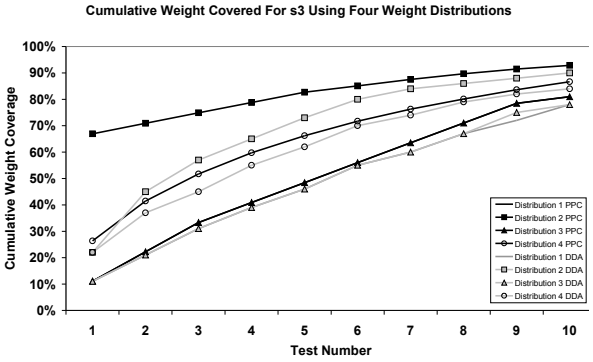

Fig. 6. Cumulative weight covered in the first 10 tests using input s_3

Weight coverage. In [13], the weight results are only given for s_3, s_4, s_5, s_7, and s_8. We analyzed the given figures, the results are given in Figures 6-10. The solid black lines give the PPC results while the light gray lines give the DDA results. Both, black and gray lines without any markers stand for d_1; d_2 lines carry small solid squares; the solid triangles represent d_3; and d_4 has circles.

For s_3, PPC works better than DDA for all distributions (Figure 6). For d_2, the advantage is remarkably high at the beginning, although at later test cases, DDA approaches the PPC values. For s_4, the DDA has a clear advantage for d_3 and a small advantage for d_1 (Figure 7). For d_2 and d_4, PPC performs slightly better. For s_5, the PPC generally performs better than the DDA (Figure 8). In this scenario, PPC gives

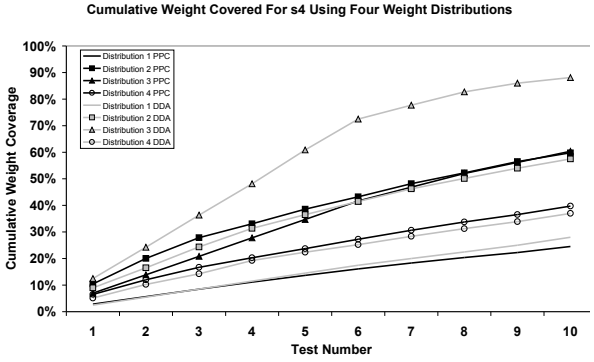

Fig. 7. Cumulative weight covered in the first 10 tests using input s_4

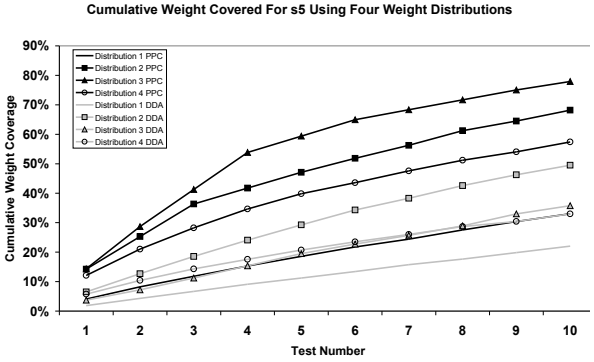

Fig. 8. Cumulative weight covered in the first 10 tests using input s_5

its best results for the d_3. For s_7 (Figure 9), the results are very similar to s_3. The PPC generally performs better than the DDA. The PPC has a very strong start for d_2, while the DDA start is quite similar to the other distribution starts. For s_8, the DDA starts better than PPC for d_4 (Figure 10). For the remaining distributions, the PPC starts better, d_2 again very much better. In later test cases, the DDA surpasses PPC for d_1 and d_2. PPC surpasses DDA for d_4 and stays ahead for d_3 all the time.

Comparing all 240 generated test cases (4 scenarios with 4 distributions with 10 test cases each and 1 scenario with

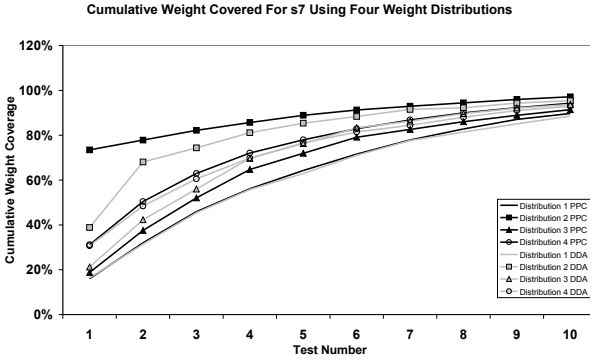

Fig. 9. Cumulative weight covered in the first 10 tests using input s_7

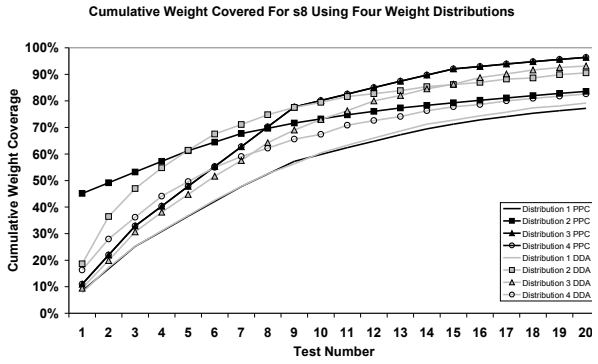

Fig. 10. Cumulative weight covered in the first 20 tests using input s_8

4 distributions with 20 test cases each) for both DDA and PPC, the PPC performs better for 192 test cases. The DDA performs better for 47 test cases. So while the values for DDA in [13] are not 100% accurate, a tendency can be seen: PPC performs better than DDA in 4 of 5 cases with respect to weight coverage. Unfortunately, no results for DDA with s_1, s_2, and s_6 are given. Comparing PPC's good performance in s_6 with DDA results would be quite interesting.

To conclude. PPC performs better than DDA in terms of early weight coverage; the resulting test suites are slightly larger. However, further analysis is needed for *missing* scenarios.

C. Comparison of PPC with BDD

Finally, we compared PPC with BDD with respect to size and weight coverage.

Size. From 32 test suites generated with BDD and PPC, 20 PPC suites are smaller than the BDD test suites. In eight cases BDD generated suites are smaller, while for the remaining 4 scenario-distribution combinations, the result set have equal sizes. As already stated, PPC produces a very large test suite for the s_3 and s_7 in d_1 and d_3 combinations. The BDD produces smaller test suites for these combinations, similar to the PPS and DDA results. These special cases need further investigation.

Weight coverage. The BDD weight results are only given for 50%, 75% and 100% [15]. Their and our results can be found in Table IV. Unfortunately, they have not evaluated results for other coverage levels as we did in Table V.

For 50% coverage, the PPC performs better than the BDD in 22 cases when comparing all 32 generated results (8 scenarios with 4 distributions each) for both BDD and PPC. For the remaining 10 cases BDD is on a par with PPC. The BDD does not perform better for scenario-distribution combination at a coverage level of 50%.

For 75% coverage, PPC increases its advantage over the BDD. The BDD performs better for only 2 cases while PPC performs better in 26 cases and is on par with BDD in 4 cases.

Comparing all 96 generated results (8 scenarios with 4 distributions with 3 different weights each) for both BDD and PPC, the PPC performs better in 68 cases. The BDD performs better for only 10 cases. For the remaining 18 cases BDD is on a par with PPC. PPC performs better than BDD in more than 2 of 3 cases with respect to weight coverage.

To conclude. PPC performs clearly better than BDD in terms of early weight coverage and resulting test suite size.

V. Conclusions

This paper presents the PPC approach and PPS approach for comparison. Both approaches have been successfully implemented in the classification tree editor. A set of benchmarks using eight scenarios with four different weight distributions has successfully been applied to both algorithms and the results have been compared with two other approaches, DDA and BDD, which demonstrated the usability of our work.

Based on the comparison between our two approaches, the following guideline on when to use which technique can be given: If a full pairwise coverage is already established, PPS can help to select subsets of test suites. If the weight distribution is equal or scenarios are small, there is no reason to use prioritized test case generation. If however, scenarios are large and distributions tend to be non-uniform, the application of prioritized test generation becomes valuable in all cases, where subsets of test suites are needed. Then, a subset selection based on weights is more successful using prioritized test generation.

Comparing our results with others: PPC performs clearly better than BDD in terms of early weight coverage and resulting test suite size. PPC performs better than DDA in terms of early weight coverage; the resulting test suites are, however, slightly larger.

Future work will analyze efforts for test generation when using the approaches presented in this paper.

Acknowledgments

This work is partly supported by EU grant ICT-257574 (FITTEST).

References

[1] M. Grochtmann and K. Grimm, "Classification trees for partition testing," *Softw. Test., Verif. Reliab.*, vol. 3, no. 2, pp. 63–82, 1993.

[2] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[3] E. Lehmann and J. Wegener, "Test case design by means of the CTE XL," *Proceedings of the 8th European International Conference on Software Testing, Analysis and Review (EuroSTAR 2000), Kopenhagen, Denmark, December*, 2000.

[4] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[5] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, pp. 185–210, 2004.

[6] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE Transactions on Software Engineering*, vol. 36, pp. 593–617, 2010.

[7] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 international symposium on Software testing and analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 1–12.

[8] A. Hoffmann, A. Rennoch, I. Schieferdecker, and N. Radziwill, "A generic approach for modeling test case priorities with applications for test development and execution," in *MOTES*, 2009.

[9] M. B. Cohen, J. Snyder, and G. Rothermel, "Testing across configurations: implications for combinatorial testing," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–9, November 2006.

[10] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Softw. Test., Verif. Reliab.*, vol. 15, no. 3, pp. 167–199, 2005.

[11] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.

[12] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2007, pp. 129–139.

[13] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pairwise coverage with seeding and constraints," *Information & Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[14] C. J. Colbourn and M. B. Cohen, "A deterministic density algorithm for pairwise interaction coverage," in *Proc. of the IASTED Intl. Conference on Software Engineering*, 2004, pp. 242–252.

[15] E. Salecker, R. Reicherdt, and S. Glesner, "Calculating prioritized interaction test sets with constraints using binary decision diagrams," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 278–285.

[16] G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical testing of software based on a usage model," *Softw. Pract. Exper.*, vol. 25, no. 1, pp. 97–108, 1995.

[17] S. Amland, "Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study," *Journal of Systems and Software*, vol. 53, no. 3, pp. 287–295, 2000.