A Parallel Evolutionary Algorithm for Prioritized Pairwise Testing of Software Product Lines

Roberto E. Lopez-Herrejon
Johannes Kepler University
Linz, Austria
roberto.lopez@jku.at

Javier Ferrer
Universidad de Málaga,
Andalucía Tech, Spain
ferrer@lcc.uma.es

Francisco Chicano
Universidad de Málaga,
Andalucía Tech, Spain
chicano@lcc.uma.es

Evelyn Nicole Haslinger
Johannes Kepler University
Linz, Austria
evelyn.haslinger@jku.at

Alexander Egyed
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

Enrique Alba
Universidad de Málaga,
Andalucía Tech, Spain
eat@lcc.uma.es

ABSTRACT

Software Product Lines (SPLs) are families of related software systems, which provide different feature combinations. Different SPL testing approaches have been proposed. However, despite the extensive and successful use of evolutionary computation techniques for software testing, their application to SPL testing remains largely unexplored. In this paper we present the Parallel Prioritized product line Genetic Solver (PPGS), a parallel genetic algorithm for the generation of prioritized pairwise testing suites for SPLs. We perform an extensive and comprehensive analysis of PPGS with 235 feature models from a wide range of number of features and products, using 3 different priority assignment schemes and 5 product prioritization selection strategies. We also compare PPGS with the greedy algorithm prioritized-ICPL. Our study reveals that overall PPGS obtains smaller covering arrays with an acceptable performance difference with prioritized-ICPL.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—
testing tools

Keywords

Combinatorial Interaction Testing, Software Product Lines,
Pairwise Testing, Feature Models

1. INTRODUCTION

A *Software Product Line (SPL)* is a family of related software systems, which provide different feature combinations [21]. The effective management and realization of *variability* – the capacity of software artifacts to vary – is crucial

to reap the benefits of SPLs such as increased software reuse, faster product customization, and reduced time to market.

Because of the large number of feature combinations in typical SPLs, variability modelling poses a unique set of challenging problems for software testing. In recent years many verification and testing SPL approaches, which rely on different techniques, have been proposed (e.g. [5, 6, 8, 16]). However, and despite the extensive and successful use of evolutionary computation techniques for software testing [12, 18], their potential application to SPL testing remains largely unexplored, in particular regarding test prioritization.

In this paper we present the *Parallel Prioritized product line Genetic Solver (PPGS)*, a genetic algorithm for the generation of prioritized pairwise testing suites for SPLs. PPGS receives as input a *feature model* that denotes a set of valid feature combinations and computes a set of products that covers the desired pairs of feature combinations according to a priority scheme that assigns different priority weights to a set of products. This scheme has been sketched in [16] and it is currently successfully applied in an industrial setting. We performed a comprehensive evaluation of PPGS with 235 feature models with a wide range of number of features and number of products, using 3 different weight priority assignment methods and 5 product prioritization selection strategies. In addition, we compared PPGS with prioritized-ICPL [16], an alternative greedy algorithm implementation. For the statistical comparison analysis both algorithms were executed 30 times for each feature model and combinations of priority assignment and product prioritization which yielded a total of 79800 independent runs that required about two weeks of computation on a 64-core dedicated cluster. Our study revealed that overall PPGS obtains smaller covering arrays with an acceptable performance difference with prioritized-ICPL. However, the performance difference tends to decrease as the number of products of the feature models increase. We believe these results shed light on the potential benefits that evolutionary algorithms and other search based techniques can bring for variability modeling problems such as testing.

In summary, the contributions of our work are: *i)* Formalization of the SPL testing prioritization scheme presented in [16], *ii)* Implementation of this scheme with the evolutionary algorithm PPGS, and *iii)* Comprehensive evaluation and comparison of PPGS and the prioritized-ICPL algorithm.

Figure 1: Aircraft SPL Feature Model

Prod	A	Wi	E	Ma	H	S	L	Pi	J	Me	Wo	Pl	C
p0	✓	✓	✓	✓	✓			✓				✓	
p1	✓	✓	✓	✓	✓			✓					✓
p2	✓	✓	✓	✓	✓				✓			✓	
p3	✓	✓	✓	✓	✓			✓					✓
p4	✓	✓		✓	✓						✓		
p5	✓	✓	✓	✓	✓			✓			✓		
p6	✓	✓	✓	✓	✓			✓		✓			
p7	✓	✓	✓	✓	✓				✓	✓			

Table 1: Sample Feature Sets of Aircraft SPL
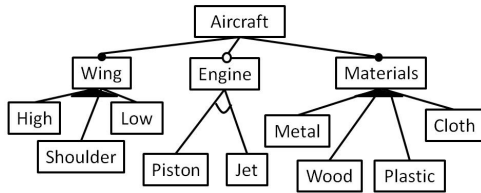
2. FEATURE MODELS AND RUNNING EXAMPLE

Feature models have become a *de facto* standard for modelling the common and variable features (depicted as labelled boxes) of an SPL and their relationships (depicted with lines) collectively forming a tree-like structure, which denotes the set of feature combinations that the products of an SPL can have [17]. Figure 1 shows the feature model of our running example, a hypothetical SPL of aircraft machines obtained from the SPLOT repository [19]. Each feature, apart from the root, has a single parent feature and can have a set of child features. Notice here that a child feature can only be included in a feature combination of a valid product if its parent is included as well.

The root feature is always included. There are four different kinds of hierarchical feature relationships: *i) Optional features* are depicted with an empty circle and indicate that they may or may not be selected if their respective parent feature is selected. An example is feature `Engine`; *ii) Mandatory features* are depicted with a filled circle and indicate that they have to be selected whenever their respective parent feature is selected. For example, features `Wing` and `Materials`; *iii) Inclusive-or relations* are depicted as filled arcs crossing over a set of lines connecting a parent feature with its child features. They indicate that at least one of the features in the inclusive-or group must be selected if the parent is selected. An example is feature `Materials` that if selected at least one of the features `Metal`, `Wood`, `Plastic`, and `Cloth` must be selected; *iv) Exclusive-or relations* are depicted as empty arcs crossing over a set of lines connecting a parent feature with its children features. They indicate that exactly one of the features in the exclusive-or group must be selected whenever the parent feature is selected. For example, if feature `Engine` is selected, then either feature `Piston` or feature `Jet` must be selected.

Besides the parent-child relations, features can also relate across different branches of the feature model with so called *Cross-Tree Constraints (CTC)*. These constraints as well as those implied by the hierarchical relations between features are usually expressed and checked using propositional logic, for further details refer to [3].

Definition 1. *Feature List (FL) is the list of features in a feature model.*

Definition 2. *Feature Set (FS) is a 2-tuple [sel,$\overline{sel}$] where sel and $\overline{sel}$ are respectively the set of selected and not-selected features of a member product. Let FL be a feature list, thus sel, $\overline{sel} \subseteq FL$, sel $\cap \overline{sel} = \emptyset$, and sel $\cup \overline{sel} = FL$. The terms p.sel and p.$\overline{sel}$ respectively refer to the set of selected and unselected features of product p.*

Definition 3. *A feature set `fs` is valid in feature model `fm`, i.e. `valid(fs, fm)` holds, iff `fs` does not contradict any of the constraints introduced by `fm`.*

The `FL` for the Aircraft feature model is [`Aircraft`, `Wing`, `Engine`, `Materials`, `High`, `Shoulder`, `Low`, `Piston`, `Jet`, `Metal`, `Wood`, `Plastic`, `Cloth`]. For example, the feature set p0=[{`Aircraft`, `Wing`, `Engine`, `Materials`, `High`, `Piston`, `Plastic`}, {`Shoulder`, `Low`, `Jet`, `Metal`, `Wood`, `Cloth`}] is valid. As another example, a feature set with features `Piston` and `Jet` would not be valid because it violates the constraint of the exclusive-or relation which establishes that these two features cannot appear selected together in the same feature set. In our running example, the feature model denotes 315 valid feature sets. Some of them are depicted in Table 1, where for any given feature set its selected features are ticked (✓) and its unselected features are empty. In this table, we use as column labels the shortest distinguishable prefix of the feature names (e.g. `Wi` for feature `Wing`).

3. PRIORITIZED PAIRWISE COVERING ARRAYS IN SPL

In this section we provide a formal definition of the priority scheme implemented by PPGS based on the sketched description provided in [16] and its supporting code.

Definition 4. *A prioritized product `pp` is a 2-tuple [`fs`, `w`], where `fs` represents a valid feature set in feature model `fm` and $w \in \mathbb{R}$ represents its weight.* Let pp_i and pp_j be two prioritized products. We say that pp_i has higher priority than pp_j for test-suite generation iff pp_i's weight is greater than pp_j's weight, that is $pp_i.w > pp_j.w$.

As an example, let us say that we would like to prioritize product p1 with a weight of 17. This would be denoted as pp1=[p1,17].

Definition 5. *A pairwise configuration pc is a 2-tuple [sel, $\overline{sel}$] representing a partially configured product, defining the selection of 2 features of feature list FL, i.e. $pc.sel \cup pc.\overline{sel} \subseteq FL \land pc.sel \cap pc.\overline{sel} = \emptyset \land |pc.sel \cup pc.\overline{sel}| = 2$. We say a pairwise configuration pc is covered by feature set fs iff $pc.sel \subseteq fs.sel \land pc.\overline{sel} \subseteq fs.\overline{sel}$.*

Consider for example the pairwise configuration that indicates that feature `Plastic` is selected while feature `Cloth` is deselected pc1=[{`Plastic`},{`Cloth`}]. Notice that pc1 is covered by products p0 and p2 of Table 1. Another example is pairwise configuration pc2=[{`High`, `Wood`},{}] with features `High` and `Wood` selected and no feature unselected. This configuration is covered by products p4 and p5 of Table 1. A last example is pc3=[{},{`Shoulder`, `Low`}], with no selected features and `Shoulder` and `Low` as unselected features. This configuration is covered by all the products

shown in Table 1. In total, for all the 315 products denoted by the feature model of Figure 1, there exist 240 valid pairwise configurations.

Definition 6. *A weighted pairwise configuration wpc is a 2-tuple [pc,w] where pc is a pairwise configuration and $w \in \mathbb{R}$ represents its weight computed as follows. Let PP be a set of prioritized products and PP_{pc} be a subset, $PP_{pc} \subseteq PP$, such that PP_{pc} contains all prioritized products in PP that cover pc of wpc, i.e. $PP_{pc} = \{pp \in PP | pp.fs \text{ covers } wpc.pc\}$. Then $w = \sum_{p \in PP_{pc}} p.w$*

Let us consider the following set of prioritized products from Table 1. Let `PP={[p0,17], [p1,17], [p2,15], [p3,15], [p4,13] , [p5,13], [p6,6], [p7,6]}` with `pp`$_i$ = `[fs`$_i$`,w`$_i$`]`, and assume that the remaining 307 products of our feature model in Figure 1 (i.e. 315 minus 8 shown in the table) have priority weight values of 0. The weight of pairwise configuration `pc1=[{Plastic},{Cloth}]` is then `wpc1.w= pp0.w+pp2.w = 17+15 = 32`, that is, the summation of the weights of the products whose feature sets cover `pc1` with weight greater than zero, namely `p0` and `p2`. Similarly, the weight for `pc2` (`High` and `Wood` selected) is `wpc2.w= pp4.w+pp5.w = 13+13 = 26`, and for `pc3` `wpc3.w=102` computed by summing all the product weights because their feature sets all cover `pc3`. In total, the eight products shown in Table 1 generate 169 weighted pairwise configurations with value greater than zero.

Definition 7. *A prioritized pairwise covering array ppCA for a feature model fm and a set of weighted pairwise configurations WPC is a set of valid feature sets FS that covers all weighted pairwise configurations in WPC whose weight is greater than zero: $\forall wpc \in WPC \ (wpc.w > 0 \Rightarrow \exists fs \in ppCA \text{ such that } fs \text{ covers } wpc.pc)$.*

Let us consider the prioritized products set `PP` just described above. The following table lists six products that together constitute a `ppCA`:

A	Wi	E	Ma	H	S	L	Pi	J	Me	Wo	Pl	C
✓	✓	✓	✓	✓			✓					✓
✓	✓	✓	✓	✓		✓					✓	
✓	✓	✓	✓	✓			✓			✓		
✓	✓	✓	✓	✓				✓	✓	✓		✓
✓	✓	✓	✓	✓	✓		✓				✓	
			✓	✓						✓		

Notice that the first three rows correspond to products p1, p2 and p5 of Table 1. Also notice that the last three products are not part of that table even though their weights are considered zero in our running example. That is, their selection led to a ppCA with a smaller number of products than the original set. Next we describe how this pairwise prioritization scheme was implemented by PPGS.

The optimization problem we are interested in consists of finding a prioritized pairwise covering array, *ppCA*, with the minimum number of feature sets, that is: find *ppCA* with minimum |*ppCA*|. What makes the problem far from trivial is the constraints imposed to *ppCA* by Definition 7.

4. ALGORITHM DESCRIPTION

The *Parallel Prioritized product line Genetic Solver (PPGS)* is a novel constructive genetic algorithm which follows the master-slave model to parallelize the evaluation of the individuals [1]. It computes a ppCA as defined in the previous section. In each iteration, PPGS adds one new product to a partial solution until all pairwise combinations are covered. Algorithm 1 sketches the pseudocode of PPGS.

It uses as inputs a feature model `FM` and the set of prioritized products for the test suite generation. At the beginning, the test suite is initialized with an empty list (Line 2) and the set of remaining pairs (`RP`) is initialized with the weighted pairwise configurations present in at least one of the given prioritized products (Line 3), as it was described in Section 3. In each iteration of the external loop (Lines 4-23) the algorithm creates a random initial population of individuals (Line 6) and enters an inner loop which applies the traditional steps of a generational evolutionary algorithm (Lines 7-20). That is, some individuals (products in our case) are selected from the population $P(t)$, recombined, and mutated. PPGS represents a product by only the list of selected features, so the operators only affect the selected features. If a generated offspring individual is not a valid product (i.e., it violates any constraint derived from the feature model), it is transformed into a valid product by applying a `Fix` operation (Line 13) provided by the FAMA tool [23].

Since the evaluation is performed in parallel in this algorithm, the fixed individuals are stored in a structure for later evaluation of them (Line 14). After we leave the inner loop (Lines 7-20), the evaluation is performed in parallel (Line 16), and finally the individuals are inserted in the offspring population Q. The fitness value of an offspring individual (Line 16) is the sum of the weights of the weighted pairwise configurations that would remain to be covered after adding the offspring solution to the test suite. Thus, this fitness function must be minimized in order to first select the product that covers weighted pairwise configurations with higher weights. Notice that, as the search progresses, the cost of computing the fitness function is reduced because every time less weighted pairwise configurations remain uncovered.

In Line 18, the best individuals of $P(t)$ and Q are kept for the next generation $P(t+1)$. The internal loop is executed until a maximum number of evaluations is reached. Then, the best individual (product) found is included in the test suite (Line 21) and `RP` is updated by removing the weighted pairwise configurations covered by the selected best solution (Line 22). Then, the external loop starts again until there is no weighted pair left in the `RP` set.

We set the configuration parameters of PPGS with values frequently observed in the literature for genetic algorithms: crossover strategy one-point with a probability of 0.8, selection strategy binary tournament, population size of 10 individuals, mutation that iterates over all selected features of an individual and replaces a feature by another randomly chosen feature with a probability of 0.1, and termination condition of 1,000 fitness evaluations and full weight coverage in the external loop.

Let us show an example of the execution of the inner loop of PPGS, assuming `RP` contains the following entries [`wpc1, wpc2, wpc3`], where `wpc1.pc = [{Engine, Piston},{}]`, `wpc2.pc = [{Cloth}, {Plastic}]` and `wpc3.pc= [{Jet}, {Plastic}]`. Figure 2(a) shows the selected features of two valid products. PPGS applies the evolutionary operators only to the selected features. The dashed square indicates the cross-over point for the two individuals. Figure 2(b) shows the result of the crossover. Figure 2(c) illustrates mutation of the first recombined individual, the mutated element `Jet` is underlined. Notice that this product is invalid because features `Piston` and `Jet` are mutually exclusive (see Figure 1). This situation requires the application of a

Algorithm 1: Pseudocode of PPGS.

```
 1: proc Input:feature model FM, prioritized products prods
 2: TS ← ∅    // Initialize the test suite
 3: RP ← weighted_pairs_to_cover(prods)
 4: while not empty(RP) do
 5:     t=0
 6:     P(t) ← Create_Population() // P = population
 7:     while evals < totalEvals do
 8:         Q ← ∅    // Q = auxiliary population
 9:         for i ← 1 to (PPGS.popSize / 2) do
10:             parents←Selection(P(t))
11:             offspring←Recombination(PPGS.Pc,parents)
12:             offspring←Mutation(PPGS.Pm,offspring)
13:             Fix(offspring)
14:             ParallelEvaluator.addSolution(offspring)
15:         end for
16:         solutions←ParallelEvaluator.evaluate();
17:         Insert(solutions,Q)
18:         P(t+1) := Replace (Q,P(t))
19:         t= t + 1
20:     end while   //internal loop
21:     TS ← TS ∪ best_solution(P(t))
22:     RemovePairs(RP, best_solution(P(t)))
23: end while   //external loop
24: return TS
25: end_proc
```

fix operation to provide an actual valid product shown in Figure 2(d). Once a valid product is found, the algorithm generates all its possible pairs and removes them from a copy of the `RP` set, from which the fitness value is computed as the number of pairs that remain still uncovered. In our example, `wpc2` and `wpc3` are covered by the product on Figure 2(d), thus the fitness function for this individual is `wpc3.w`, the remaining weight that has not yet been covered. The inner loop is repeated until the algorithm reaches 1,000 evaluations. Then, the best individual is removed from `RP`, the covered pairs, and the product is added to the current test suite `TS`. When there are no more weighted pairs to be covered or the maximum number of fitness evaluations has been reached, the algorithm returns its current `TS` value.

PPGS has been implemented using jMetal [7], a Java framework aimed at the development, experimentation, and study of metaheuristics for solving optimization problems. Additionally and as mentioned before, PPGS uses a framework for the analysis of feature models called FAMA [23]. This framework provides a simple, easy-to-use, and extensible programming API that supports a common and basic set of feature model operations.
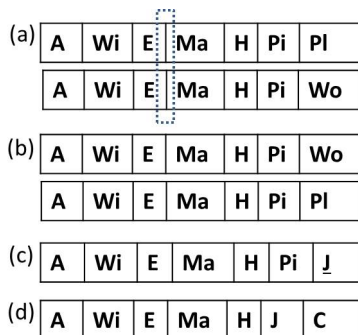
Figure 2: Evolutionary Operations

5. EVALUATION

This section describes how our evaluation was carried out. We start with the algorithm used to compare and contrast PPGS, followed by the methods employed to assign priorities, the selection of the products to prioritize, the experimental corpus of feature models, and the software and hardware infrastructure used.

5.1 Prioritized-ICPL (pICPL) Algorithm

To compare and contrast our PPGS algorithm we employed prioritized-ICPL, a greedy algorithm to generate **n**-wise covering arrays developed by Johansen et al. [16]. Prioritized-ICPL does not compute covering arrays with full coverage but rather covers only those **n**-wise combinations among features that are present in at least one of the prioritized products, as was described in Section 3. We must highlight here that the pICPL algorithm uses *data parallel execution*, supporting any number of processors. Their parallelism comes from simultaneous operations across large sets of data. For further details on prioritized-ICPL please refer to [16].

We should remark that an earlier and more well-known version of a greedy algorithm for SPL pairwise testing by the same authors Johansen et al. is also called ICPL [15]. However that version did not consider prioritization. To avoid any confusions and as a short hand notation, henceforth we will use the term pICPL to refer to prioritized-ICPL.

5.2 Weight Priority Assignment Methods

We considered three methods to assign weight values to prioritized products.

5.2.1 Measured values

For this type, the weights were derived from non-functional properties values obtained from 16 real SPL systems, from different problem domains and implemented using different technologies, that were measured with the SPL Conqueror approach [22]. This approach aims at providing reliable estimates of measurable non-functional properties such as performance, main memory consumption, and footprint. It works by performing a set of actual property measurements on different products (usually a proper subset of all the feature combinations denoted by a feature model) with different feature interaction types. The measured values are then used to compute the estimated property values for the feature combinations that were not measured. This choice of weight priority assignment allows us to emulate more realistic scenarios whereby software testers need to schedule their testing effort giving priority, for instance, to products or feature combinations that exhibit higher footprint or performance.

For our work, we use the actual values taken on the measured products considering pairwise feature interactions. Table 2 summarizes the SPL systems evaluated, their measured property (`Prop`), number of features (`NF`), number of products (`NP`), number of configurations measured (`NC`), and the percentage of prioritized products (`PP%`) used in our comparison as explained shortly.

5.2.2 Rank based values

For this second type of weight values, we selected the products to prioritize based on how dissimilar they are when compared to all other products of an SPL, and assigned them

SPL Name	Prop	NF	NP	NC	PP%
Prevayler	F	6	32	24	75.0
LinkedList	F	26	1440	204	14.1
ZipMe	F	8	64	64	100.0
PKJab	F	12	72	72	100.0
SensorNetwork	F	27	16704	3240	19.4
BerkeleyDBF	F	9	256	256	100.0
Violet	F	101	$\approx$ 1E20	101	$\approx$ 0.0
Linux subset	F	25	$\approx$ 3E24	100	$\approx$ 0.0
LLVM	M	12	1024	53	5.1
Curl	M	14	1024	68	6.6
x264	M	17	2048	77	3.7
Wget	M	17	8192	94	1.15
BerkeleyDBM	M	19	3840	1280	33.3
SQLite	M	40	$\approx$ 5E7	418	$\approx$ 0.0
BerkeleyDBP	P	27	1440	180	12.50
Apache	P	10	256	192	75.0

Footprint, **M**ain memory consumption, **P**erformance, **N**umber of **F**eatures, **N**umber of **P**roducts, **N**umber of **C**onfigurations, **P**ercentage of **P**rioritized products.

Table 2: Measured Values Case Studies Summary

priority weights based on their rank values. For further details please refer to the associated materials. The intuition behind this assignment choice is that by giving the same weight value to two of the most SPL-wide dissimilar products, the weight values will be more likely spread among a larger number of pairwise configurations making the covering array harder to compute. In addition, this enables us to select different percentages of the number of products for prioritization as elaborated in Subsection 5.3.

5.2.3 Random Values

For this type of weight values, we randomly generate weights between the minimum and maximum values obtained with the ranked based values approach.

5.3 Product Prioritization Selection

We selected the products for prioritization based on the priority assignment method. For the measured values assignment method, all the measured products were used as our prioritization products. In three cases this meant including all the products of the product line. Please refer to Table 2 for further details. For the rank based values and the random values assignment methods, a percentage of the products denoted by each individual feature model was used for product prioritization. The selected percentages are: 5%, 10%, 20%, 30%, and 50%.

5.4 Experimental Corpus

We created three groups based on both the number of products denoted by the feature models and how their priority was assigned as shown in Table 3. Group G1 was formed with 160 feature models, whose number of products ranges from 16 to 1000 products, and that were evaluated with rank based and random values. Group G2 was formed with 59 feature models, whose number of products ranged from 1000 to 80000 that were also evaluated with rank based and random values. The threshold value to divide groups G1 and G2, and the selected percentages were chosen to provide an

	G1	G2	G3	Summary
NFM	160	59	16	235
NP	16-1K	1K-80K	32- $\approx$3E24	16- $\approx$3E24
NF	10-56	14-67	6-101	6-101
WPA	RK,RD	RK, RD	M	
PP%	20,30,50	5,10,20	$\approx$0.0 $-$ 100	
PI	960	354	16	1330

NFM: Number Feature Models, **NP**: Number Products, **NF**: Number Features, **WPA**: Weight Priority Assignment, **RK**: Rank based, **RN**: Random, **M**: Measured, **PP**%: Prioritized Products Percentage, **PI**: Problem Instances

Table 3: Evaluation Case Studies Summary

ample variety of number of products to prioritize. Group G3 was formed with 16 feature models, with number of products ranging from 16 to $\approx$3E24 that were evaluated with measured values.

We obtained 16 feature models from SPL Conqueror, 5 from Johansen et al. [16], and 201 from the SPLOT website [19] (a repository for the feature model analysis research community). Thus in total we employed 222 distinct feature models. Please notice that we also incorporated 5 SPL Conqueror feature models to G1 and 8 to G2. This yields a grand total of 235 feature models to analyze. For G1 and G2 the problem instances are computed considering that for each feature model two priority assignment methods are used with three different prioritization selection percentages. For example, this yields for G1 $160\times2\times3 = 960$ instances. In total 1330 problem instances were analyzed, with two algorithms PPGS and pICPL, with 30 independent executions. This means that the data of a total of 79800 independent runs was generated and analyzed.

5.5 Hardware and Software Setup

PPGS and pICPL are non-deterministic algorithms, so we performed 30 independent runs for a fair comparison between them. As performance measures we analyzed both the number of products required to test the SPL and the time required to run the algorithm. In both cases, the lower the value the better the performance, since we want a small number of products to test the SPL and we want the algorithm to be as fast as possible. All the executions were run in a cluster of 16 machines with Intel Core2 Quad processors Q9400 (4 cores per processor) at 2.66 GHz and 4 GB memory running Ubuntu 12.04.1 LTS and managed by the HT Condor 7.8.4 cluster manager. Since we have four cores available per processor, we have executed only one task per single processor, so we have used four parallel threads in each independent execution of the analyzed algorithms. The total 79800 independent runs of our evaluation required about two weeks of computation on a 64-core dedicated cluster.

6. RESULTS AND ANALYSIS

In this work we have used two different statistical techniques to measure different aspects of the comparison.

6.1 Wilcoxon Test

In order to check if the differences between the algorithms are statistically significant or just a matter of chance, we

applied the non-parametric Wilcoxon rank-sum test. We highlight in the tables the statistically significant differences with a confidence level of 95% (p-value under 0.05).

In Table 4 we summarize the results obtained for group G1, feature models with up to 1000 products. Each column corresponds to one algorithm and in the rows we show the number of products required to reach 50% up to 100% of total weighted coverage. The data shown in each cell is the mean and the standard deviation of the 30 independent runs. We highlight with a light gray background those values that are better with respect to the other algorithm with a statistically significant difference. We can observe that PPGS requires a smaller number of products to test the SPL with a significant difference when we consider a coverage level of 80% up to 99%. In the rest of the cases the differences are not statistically significant, so we cannot claim that one algorithm is better than the other. Regarding the time, pICPL is around 6 times faster than PPGS with a statistically significant difference. The time is given in milliseconds in the tables.

Cov.	PPGS	pICPL	Cov.	PPGS	pICPL
50%	$1.20_{0.40}$	$1.20_{0.40}$	96%	$4.00_{1.23}$	$4.37_{1.42}$
75%	$1.92_{0.51}$	$1.98_{0.58}$	97%	$4.38_{1.32}$	$4.71_{1.54}$
80%	$2.15_{0.59}$	$2.25_{0.68}$	98%	$4.83_{1.46}$	$5.18_{1.74}$
85%	$2.47_{0.72}$	$2.58_{0.81}$	99%	$5.58_{1.71}$	$5.87_{1.99}$
90%	$2.88_{0.86}$	$3.13_{1.03}$	100%	$7.56_{2.85}$	$7.56_{3.03}$
95%	$3.72_{1.14}$	$4.06_{1.33}$	TIME	23897_{28669}	10116_{18842}

Table 4: Mean and standard deviation of 30 indep. runs for G1 (significant differences are highlighted)

Table 5 shows the results for group G2: feature models with 1000 to 80000 products. We use the same legend and notation as for Table 4. In this case the advantage of PPGS over pICPL is larger than in the previous case. First, we can observe that PPGS is better than pICPL with statistically significant difference in all the coverage percentages except 100%. Regarding the computation time, PPGS is faster than pICPL but without statistically significant difference. From these results, the trend we can observe is that as the number of products of the SPL grows PPGS is still better in quality than pICPL while it is also better in runtime. Part of our future work is to verify if this trend holds for feature models with a larger number of products.

Cov.	PPGS	pICPL	Cov.	PPGS	pICPL
50%	$1.16_{0.36}$	$1.36_{0.83}$	96%	$4.98_{0.97}$	$5.83_{3.14}$
75%	$2.09_{0.42}$	$2.47_{1.65}$	97%	$5.55_{1.10}$	$6.43_{3.27}$
80%	$2.39_{0.52}$	$2.86_{1.79}$	98%	$6.34_{1.34}$	$7.23_{3.48}$
85%	$2.73_{0.59}$	$3.27_{2.08}$	99%	$7.66_{1.88}$	$8.59_{4.11}$
90%	$3.36_{0.76}$	$3.98_{2.38}$	100%	$14.57_{10.65}$	$13.79_{9.98}$
95%	$4.59_{0.90}$	$5.42_{3.12}$	TIME	$273728_{7.2E+5}$	$638164_{2.1E+6}$

Table 5: Mean and standard deviation of 30 indep. runs for G2(significant differences are highlighted)

Let us now focus on group G3, feature models with measured weight values. Table 6 shows the average number of products required to cover each SPL and the time for both pICPL and PPGS over the 16 models. According to the statistically significant differences the conclusions in this group of models are similar to the conclusions in the previous ones: PPGS is better in quality (lower number of products) while pICPL is faster. In detail, regarding the quality

of the solutions, PPGS is better than pICPL in 68 model-coverage combinations with statistical significant difference, while pICPL is better than PPGS only in 19 model-coverage combinations. Regarding the time, pICPL is usually faster than PPGS with a statistically significant difference, with the only exception of the `SensorNetwork` model, in which they do not have statistically significant difference.

If we take a closer look at the data in the table and taking into account the statistical significant differences, we can observe that PPGS is overall better than pICPL in 8 out of the 16 models, namely: `Apache`, `BerkeleyDBF`, `Berkeley-DBP`, `LLVM`, `PkJab`, `SensorNetwork`, `Violet` and `Wget`. On the other hand, pICPL is better only for 2 models when all coverage percentages are considered: `Prevayler` and `ZipMe`. In the remaining 6 models pICPL is better for some percentages while PPGS is better for others.

As a general conclusion of this first analysis we can say that if the number of products to test is a critical aspect for the Testing Engineer, PPGS should be applied to generate these products instead of pICPL. The time required by PPGS is usually no longer than a few minutes, which is a reasonable time to generate a better quality test suite. We argue this is the most common scenario in software companies with SPLs, where carrying on each product test can require hours if not days to perform, specially if they involve complex software and hardware setups [16]. On the other hand, pICPL could be employed when the number of products to test is not a critical issue and a slightly faster generation of the test suite is preferable.

6.2 $\hat{A}_{12}$ statistic

In order to properly interpret the results of statistical tests, it is always advisable to report effect size measures. For that purpose, we have also used the non-parametric effect size measure $\hat{A}_{12}$ statistic as recommended by Arcuri and Briand [2]. Given a performance measure M, $\hat{A}_{12}$ measures the probability that running algorithm A yields higher M values than running another algorithm B. If the two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. If $\hat{A}_{12} = 0.3$ entails one would obtain smaller results 70% of the time with A. Table 7 shows the $\hat{A}_{12}$ statistic to assess the practical significance of the results. In this table a value lower than 0.5 means that PPGS is better than pICPL, a value greater than 0.5 means pICPL is better than PPGS and 0.5 means a draw. At a first glance, we can see that most of the times (31), PPGS obtains smaller test suites for all percentages of coverage, meanwhile pICPL is only better than PPGS twice. We have highlighted with dark and light gray background the lowest and highest values of the table (0.2497 and 0.5157). The lowest value indicates that PPGS obtains a better test suite than pICPL for 98% of coverage in a model of G2 in more than 75% of the cases. The highest value indicates that pICPL obtains a better test suite for 100% coverage in a model of G1 with a probability near 0.5. In general, this statistic reconfirms that PPGS gets better test suites than pICPL in terms of the number of products.

7. THREATS TO VALIDITY

We identified two main threats to validity in our work. First, we used a single assignment for the parameters values of PPGS based on the authors' experience. A change in the values of these parameters could have an impact in the results of the algorithm. Thus, we can only claim that the

Model	Alg.	50%	75%	80%	85%	90%	95%	96%	97%	98%	99%	100%	TIME
Apache	PPGS	2	3	3	4	4	6	6	6	7	7	7	10394
	pICPL	2	3	3	4	5	6	7	7	7	8	8	7582
Berk.DBF	PPGS	2	4	4	5	5.97	6.97	6.97	6.97	7.97	8	8.17	11213
	pICPL	2	4	5	6	7	8	8	8	8	9	9	8152
Berk.DBM	PPGS	2	3	3	4	4.73	6.87	7.80	8.77	9.97	11.90	23.33	117607
	pICPL	2	3	3	4	6	7	8	8	10	11	21	94512
Berk.DBP	PPGS	1	2	2	3	3	4	4.83	5	5.93	7	10.60	47361
	pICPL	1	2	3	3	4	6	6	6	6	7	12	57291
Curl	PPGS	2	3	3	3.97	4.03	5.83	6	6.50	7.37	8.07	9.63	17454
	pICPL	2	3	3	4	4	6	6	6	7	7	8	6382
LinkedList	PPGS	1	2	2	2	3	4.23	5	5	6.13	7.79	13.37	60684
	pICPL	1	2	2	3	3	4	4	5	7	11	14	71151
Linux	PPGS	2	4	4	5	6	7	7.67	8	8.37	9.40	11.10	49385
	pICPL	2	4	5	5	6	8	8	8	8	9	10	30522
LLVM	PPGS	2	3	3.03	4	5	6	6	6.07	7	8	8.17	12805
	pICPL	2	3	4	4	5	6	7	7	7	8	8	9032
PKJab	PPGS	1	2	2	3	3.07	4	5	5	5	6	7	11439
	pICPL	1	2	3	3	3	5	5	6	7	8	8	4661
Prevayler	PPGS	2	3	3	3	4	5	5	5.60	6	6	6	8091
	pICPL	2	3	3	3	4	5	5	5	6	6	6	2412
S.Network	PPGS	1	3	3	3	4	5.03	5.47	6	6.97	7.87	13.97	71971
	pICPL	1	3	4	5	6	8	9	9	10	11	17	74181
SQL.Mem	PPGS	1	2.17	2.90	3.23	4.07	6.14	6.97	7.93	9.23	11.70	31.53	903118
	pICPL	1	3	4	4	5	8	8	9	11	14	28	407991
Violet	PPGS	1	1	1	2	2	2.93	3	3.07	3.30	4.53	12.83	31376054
	pICPL	1	1	1	2	2	3	3	4	4	6	15	2471691
Wget	PPGS	2	2.13	3	3.07	4	5.43	6	6.40	7	8.03	11.37	31525
	pICPL	2	3	3	4	4	6	6	7	7	9	11	19612
x264	PPGS	1.23	2.23	3	3.07	4	5.30	6	6.50	7.23	8.47	12.10	37368
	pICPL	1	2	3	3	4	5	6	7	7	9	13	13441
ZipMe	PPGS	2	3	3	4	5	6	6	7	7	7	7.03	13035
	pICPL	2	3	3	4	5	6	6	6	7	7	7	6142

Table 6: Group G3. When considering array sizes PPGS is statistically better than pICPL in 69 cases, and pICPL is better in 18 cases.

Group	50%	75%	80%	85%	90%	95%
G1	0.4985	0.4729	0.4511	0.4473	0.3785	0.3501
G2	0.4529	0.4193	0.3760	0.3726	0.3436	0.2887
G3	0.5104	0.4562	0.2844	0.3563	0.3198	0.3239

Group	96%	97%	98%	99%	100%	
G1	0.3410	0.3703	0.3634	0.4000	0.5157	
G2	0.2847	0.2647	0.2497	0.2595	0.4945	
G3	0.3312	0.3135	0.3927	0.3068	0.4166	

Table 7: $\hat{A}_{12}$ statistical test results for all groups. PPGS yields better test suite size values.

conclusions are valid for the combination of parameter values that we used. Second, the selection of feature models for the experiment corpus can indeed bias the results obtained. To counteract this threat, we used 3 different sources for our feature models. From them we selected a number of feature models as large as possible, with the widest ranges of both number of features and number of products. We should point out that beyond the ranges of our groups G1 and G2, the feature analysis tool FAMA that we employ for PPGS presents a performance and scalability bottleneck. Addressing these limitations is part of our future work.

8. RELATED WORK

There exists substantial literature on both search based testing and SPL testing. In this section, we briefly summarize those pieces of work closest to ours. Within the area of Search-Based Software Engineering a major research focus has been software testing [12]. A recent overview by McMinn highlights the major achievements made in the area and some of the open questions and challenges [18]. Relevant in this realm is the work by Ferrer et al. who propose a test prioritization genetic algorithm [10]. However, in clear contrast with our work their algorithm is not for SPLs but for systems without variability. Some of the very few applications of search based techniques to SPL are summarized next. Garvin et al. applied simulated annealing to combinatorial interaction testing for computing n-wise coverage for SPLs [11]. Ensan et al. propose a genetic algorithm approach for test case generation for SPLs [9]. In contrast with our work, they use as fitness function a variation of cyclomatic complexity metric adapted to feature models, their goal is not n-wise coverage and they do not consider priorities. Henard et al. propose an approach that uses a dissimilarity metric that favors individuals whose n-wise coverage varies the most from the current population thus increasing the chances of wider coverage [13]. A key difference with our work is that the prioritization is not based on assigned weights that have a perceived market or quality value. Recent surveys on SPL testing [6,8], attest the increasing relevance of the topic within the SPL community but also confirm that the latent potential of search based testing techniques remains largely untapped. Perrouin et al. propose an approach that translates t-wise coverage problems into Alloy programs and rely on its automatic instance generation to obtain covering arrays [20]. Hervieu et al. employ constraint programming for computing pairwise coverage from feature models [14]. *In sharp contrast with our work, none of these algorithms considers any prioritization criteria.*

9. CONCLUSIONS AND FUTURE WORK

In this paper we formalized a SPL testing prioritization scheme and presented its implementation with PPGS. We evaluated PPGS with 235 feature models of different characteristics using different selection criteria for product prioritization. Furthermore, we compared PPGS with greedy algorithm pICPL, a comparison that totalled 79800 independent runs. Our analysis showed that while PPGS obtains overall shorter covering arrays it exhibits a performance difference with pICPL that tends to decrease for the feature models with larger number of products. It should be noted though that the current PPGS implementation is not fine-tuned for performance, so part of our future work is to evaluate alternative representations of population individuals and evolutionary operations, and to streamline the feature model analysis support. Addressing these two limitations will extend our study to include feature models with more than 80,000 prioritized products. Also, recall that the current stopping condition of PPGS is a fixed number of evaluations. We plan to study the impact on performance of different approaches to adapt this stopping condition based on the characteristics of feature models. A starting point is the work of Bhandari et al. [4]. We believe our work sheds light on the potential of search based techniques for SPL testing.

10. ACKNOWLEDGEMENTS

11. REFERENCES

[1] E. Alba and G. Luque. *Parallel genetic algorithms*, volume 367 of *Studies in Computational Intelligence*. Springer-Verlag, 2011.

[2] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verif. Reliab*, 2012.

[3] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[4] D. Bhandari, C. A. Murthy, and S. K. Pal. Variance as a stopping criterion for genetic algorithms with elitist model. *Fundam. Inform.*, 120(2):145–164, 2012.

[5] H. Cichos, S. Oster, M. Lochau, and A. Schürr. Model-based coverage-driven test suite generation for software product lines. In *MoDELS*, pages 425–439, 2011.

[6] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. A systematic mapping study of software product lines testing. *Information & Software Technology*, 53(5):407–423, 2011.

[7] J. J. Durillo and A. J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760 – 771, 2011.

[8] E. Engström and P. Runeson. Software product line testing - a systematic mapping study. *Information & Software Technology*, 53(1):2–13, 2011.

[9] F. Ensan, E. Bagheri, and D. Gasevic. Evolutionary search-based test generation for software product line feature models. In *CAiSE*, pages 613–628, 2012.

[10] J. Ferrer, P. M. Kruse, J. F. Chicano, and E. Alba. Evolutionary algorithm for prioritized pairwise test data generation. In T. Soule and J. H. Moore, editors, *GECCO*, pages 1213–1220. ACM, 2012.

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

[12] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11, 2012.

[13] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test suites for large software product lines. *CoRR*, abs/1211.5451, 2012.

[14] A. Hervieu, B. Baudry, and A. Gotlieb. Pacogen: Automatic generation of pairwise test configurations from feature models. In T. Dohi and B. Cukic, editors, *ISSRE*, pages 120–129. IEEE, 2011.

[15] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *SPLC (1)*, pages 46–55, 2012.

[16] M. F. Johansen, Ø. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating better partial covering arrays by modeling weights on sub-product lines. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *MoDELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2012.

[17] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[18] P. McMinn. Search-based software testing: Past, present and future. In *ICST Workshops*, pages 153–163. IEEE Computer Society, 2011.

[19] M. Mendonca. Software Product Line Online Tools(SPLOT), 2013. http://www.splot-research.org/.

[20] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. L. Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *ICST*, pages 459–468. IEEE Computer Society, 2010.

[21] K. Pohl, G. Bockle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[22] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information & Soft. Technology*, 55(3):491–507, 2013.

[23] P. Trinidad, D. Benavides, A. Ruiz-Cortes, S. Segura, and A. Jimenez. Fama framework. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 359–359, Sept.