Marginal Distribution Algorithm for Feature Model Test Configuration Generation

Mohd Zanes Sahid, Mohd Zainuri Saringat, Mohd Hamdi Irwan Hamzah, Nurezayana Zainal
Faculty of Computer Science and Information Technology,
Universiti Tun Hussein Onn Malaysia (UTHM), Johor, Malaysia

***Abstract*—Generating test configuration for Software Product Line (SPL) is difficult, due to the exponential effect of feature combination. Pairwise testing can generate test input for a single software product that deviates from exhaustive testing, nevertheless proven to be effective. In the context of SPL testing, to generate minimal test configuration that maximizes pairwise coverage is not trivial, especially when dealing with a huge number of features and when constraints must be satisfied, which is the case in most SPL systems. In this paper, we propose an estimation of distribution algorithm, based on pairwise testing, to alleviate this problem. Comparisons are made against a greedy-based and a constraint handling based approach. The experiments demonstrate the feasibility of the proposed algorithm, such that it achieves better test configurations dissimilarity and at the same time maintain the test configuration size and pairwise coverage. This is supported by analysis using descriptive statistics.**

Keywords*—Estimation of distribution algorithm; marginal distribution algorithm; test configuration generation; pairwise testing; software product line

I. Introduction

Many software products developed for various domains carries some similar functionality. This software shares similar functionalities since they have been developed based on the same kind of input and output types. The similarity in the internal program structure due to identical user requirements also contributes to the commonalities among these software products. Because of this scenario, and based on the benefit of reuse principles, Software Product Line (SPL) has been developed as a software development paradigm to produce software inspired by product line approach. Developing an SPL system enables us to create a software structure that is customizable to various needs, by maximizing software artefacts reusability [1]. Due to the highly variable and reusable nature of SPL artefacts, it is uneconomic to develop software based on distinct requirements separately, as some of the functionalities are similar. However, it is difficult to employ single product development paradigm to build various software products that fulfil the needs of diverse users of a similar domain.

A unit of system function in an SPL is represented as a feature, and explicitly defined as common or variable features and utilized throughout the SPL development process. One way to model the commonalities and variabilities in an SPL is using a Feature Model (FM), based on feature modeling technique [1]. Two or more features are combined and utilized together in a single software product. This is known as feature configuration. The flexibility of feature configuration process could result in unspecified and unintended system behavior. This might lead to incorrect execution [1]. Hence, it is crucial to test all possible feature configurations to reduce the potential misbehavior of interacting features. But, to test all possible feature configurations is unfeasible. The number of feature configurations increases dramatically as the number of features increased, making full testing of feature configurations especially in large-scale FM impractical [2], [3]. In view of this, a number of techniques have been proposed to reduce the combinatorial explosion of feature configuration testing [4] that leveraged the potential of search-based techniques. More on this is presented in Section II.

In conventional meta-heuristics approaches, probabilities are implicitly employed in the selection and re-production operators, such as mutation operator, to produce offspring [5], [6]. We identified a research gap in feature configuration exhaustive testing, such that, one can explicitly build a probabilistic model of features distribution from an initial set of test configurations. This probabilistic model allows us to estimate the distribution of highly fit features and guide us in generating subsequent candidate solutions that maximize pairwise coverage. Towards that, in Section III we strategize the test configuration generation process, and our contributions are as follows:

1) We devise a set of algorithms based on bivariate marginal distribution in SPL context. This approach is perceived as a lightweight variant of estimation of distribution algorithm, in which only the statistics of the population are maintained across generation, instead of the actual population.

2) We introduce the notion of feature configuration dependency graph in part B of Section III, which contains the dependency information between pairs of features, extracted using statistical computation.

3) We implement the proposed approach using Java and conducted empirical studies. Results are reported and discussed in Sections IV and V.

II. Background and Related Works

A. Feature Model

Feature Modeling is a popular way to model SPL variability and it is by far the most reported in industry. In Feature Modeling, Feature Model (FM) notation has been developed to represent features and its dependencies [1]. The

tree representation of FM is known as Feature Diagram. It presents a feature as a node, and relationship between two features as an edge. Different types of edges can be assigned between features, which represent the relationship of type mandatory, optional, or, or alternative. Additionally, an FM might encompass some constraints which act as rules or conditions that limit the linking between features.

Fig. 1 shows a feature model named as OnlineBookstore SPL from Software Product Line Online Tools (SPLOT) [7]. This feature model defines some of the most common features that an online bookstore system should have. It consists of 25 features, including seven mandatory features. There is one cross-tree-constraint defined, which is ¬ Customer_Profiling ∨ Authorization. This constraint signifies that feature Customer_Profiling requires the presence of feature Authorization, but not conversely. Apart from that, a couple of variation points are defined. For example, we can choose to have Signup feature apart from Login; different types of Personalization features can be incorporated; and so on.

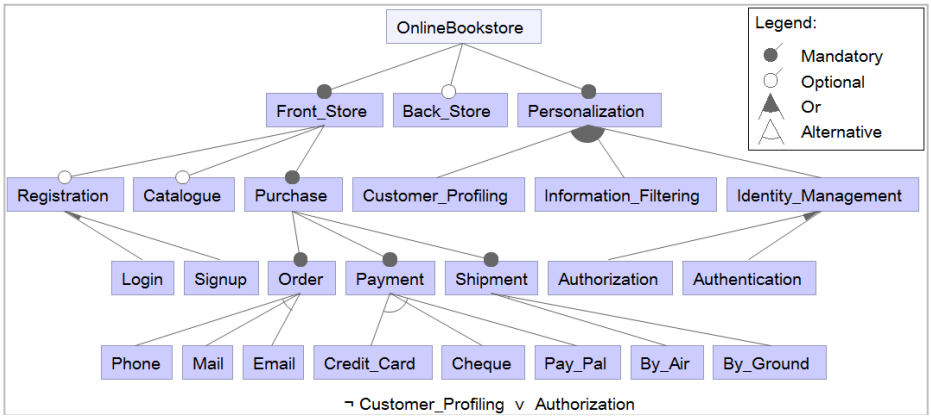

Fig. 1. A Feature Model of an OnlineBookstore SPL.

In the subsequent part of this paper, we refer each feature from our Feature Model as x. It is an integer where x ∈ [1, N], having N as the maximum number of features in the FM. x can be of positive prefix (not written) to indicate the feature is included or negative prefix (explicitly written) to indicate the feature is not included.

For brevity, each feature in the feature model shown in Fig. 1 is mapped to a unique number, assigned sequentially from top to bottom, left to right. This gives us a representation as shown in Fig. 2, which is used throughout this paper.

1: OnlineBookstore	2: Front_Store
3: Back_Store	4: Personalization
5: Registration	6: Catalogue
7: Purchase	8: Customer_Profiling
9: Information_Filtering	10: Identity_Management
11: Login	12: Signup
13: Order	14: Payment
15: Shipment	16: Authorization
17: Authentication	18: Phone
19: Mail	20: Email
21: Credit_Card	22: Cheque
23: Pay_Pal	24: By_Air
25: By_Ground	

Fig. 2. Number assignment of each feature.

B. Test Configuration

Software products of an SPL are configured and produced by combining several features. These artefacts are called feature configurations. In view of testing, test case(s) can be defined for each feature. Thus, to test a feature configuration, Test Configuration (TC), which consists of many test cases, can be generated in the same way the feature configuration is generated. A test configuration, TC, is a list of all features, represented by its feature number. Each feature in a test configuration can have either positive or negative prefix.

C. Pairwise Testing

Complete testing of all possible feature configurations is not feasible. For n number of features, it requires 2^n number of test configurations to cover all possible combinations, because it is either selected or excluded. This makes it exponentially proportioned to the number of features. To alleviate this obstacle, pairwise testing has been widely used as a viable solution. The ultimate goal of pairwise testing is to cover all possible pair of features at least once [8]–[10]. Thus, testing can be focused on the interaction of both features. Pairwise testing is a kind of combinatorial testing, where we choose 2 features to be considered or included in our test pool. Generalization of pairwise testing is called t-wise testing, where t indicates the number of features to choose.

For its practicality and usability, SPL pairwise testing is governed by constraints. Considering two features (1 and 2) from OnlineBookstore SPL, four pairs of tuple have to be generated, i.e. (1,2), (1,-2), (-1,2) and (-1,-2), where negative prefix indicates that the feature is not selected in the feature configuration. Due to constraints (cross-tree-constraints and relationship of features in FM), some invalid pairs will be eliminated, e.g. (-1,2) is invalid, because root feature, i.e. 1, must always be selected. The same goes with mandatory features (2, 4, 7, and so on).

If we construct one Test Configuration, TC_i, for each pair of features, $pf_{v,w}$, (as an example, pair of feature 1 and -5), assigned as follows;

$pf_{1,-5}$ = (1, -5); TC_a = [1, ?, ?, ?, -5, ?, ?, ?, ?, ?]

we can set any arbitrary value for other variables in TC_a (marked as ?). However, these variables could possibly be matched with other pairs of features that we should cover. Thus, if we can systematically set the values of each variable in TC_i, we could maximize the number of valid pairs in each TC so that it can minimize the number of TC.

D. Related Works

SPL test configuration generation techniques that are based on greedy approach or applied meta-heuristics are discussed in the first part of this section. In the second part, few selected literatures of Estimation of Distribution Algorithms (EDA), including works related to its adoption in software engineering activities are presented.

Among others, multi-objective evolutionary algorithm has been proposed for SPL testing [6]. Their motivation was to minimize the tests suite by sorting the product lines. Henard et al. [2] also employed a search-based algorithm, (1+1) Evolution Strategy (ES), to generate and prioritise covering array, guided by a (dis)similarity measure. Henard et al. mentioned that t-wise approaches for SPLs are restricted to FMs of small sizes and t-wise coverage of low strength. Both are constrained by scalability issues that result from the intractable computation for very large FMs or high t-wise strength. Therefore, they formulated the feature configuration generation problem as a search-based where the search space consists of all valid feature configurations extracted from the FM. Dissimilarity between features is used as the fitness function during the searching for better populations.

Feature configuration testing is highly influenced by the effectiveness of constraint handling techniques that eliminate invalid test configurations. One of such prominent work is published by Yu, Duan et al. [11], whereby the validity checking of test configuration is achieved using minimum invalid tuples (MITs). This approach has been implemented as a tool named LOOKUP.

Hybrid of multi-objective crow search and fruitfly optimization has been studied and offers an optimal selection of the test suites at a fairly good convergence rate [12]. Haslinger et al. [13] applied a Simulated Annealing algorithm to generate t-wise covering array and demonstrated a tool to improve the performance of SPL testing. Haslinger et al. report a speedup of over 60% on 133 publicly available feature models, while preserving the coverage of the generated tests.

Johansen et al. published their solution [3] and a tool named ICPL, which capable of processing large feature models, better execution time and produced small covering array. They used the fact that a (t-1)-wise is always a subset of the t-wise, and employed this principle to recursively build up a higher strength covering array from a smaller one.

Estimation of Distribution Algorithms (EDA) is a kind of Evolutionary Algorithms that finds near optimal solutions based on the evolution of candidate solutions satisfying some fitness functions. EDA guide the search by explicitly building the probabilistic model of promising candidate solutions. The detail discussion on EDA is beyond the scope of this paper, but interested reader can refer to papers by Ceberio et al. [14], Shirazi et al. [15], Shakya and Santana [16], Simon [17] and Pelikan [18]. In the area of Search-Based Software Engineering (SBSE), to the best of our knowledge, no attempt has been made to employ any variant of high-order EDA (which includes bivariate or multivariate statistics) in SPL testing.

EDA have been adopted to solve many optimization problems in single software-product development such as to optimize test data generation and test suites generation [19]–[21] and refactoring [22]. The work in [19] employs bivariate EDA named as COMIT [23], in which the combination of pair of variables are viewed as tree, therefore it has a single root node. They proposed integration with data mining techniques to predict the performance of a test data generator. In the context of testing for concurrent software, detecting faults can be improved by exploiting information discovered in EDA exploration that can save future test efforts [24]. EDA has also been employed to improve software reliability prediction [25]. They reported that EDA-based approach can optimize the parameters of support vector regression in predicting the software reliability, by introducing a chaotic mutation operator into traditional EDA. Prior to that, they define the software reliability prediction problem as a combinatorial optimization problem with constraints, in which, search-based are known to be a viable solution to that problem.

III. Proposed Approach

This section presents an evolutionary-based algorithm that generates minimal and effective SPL test configuration that satisfies pairwise coverage of features, based on bivariate marginal distribution strategy.

A. Marginal Distribution Algorithms of EDAs

Estimation of Distribution Algorithms (EDAs) explores the space of potential solutions following the principle of survival of the fittest of individual and populations similar to Genetic Algorithm (GA) [16], [18]. However, in EDAs, crossover and mutation operators are removed and replaced by the estimation of a probability distribution. The Probability Distribution is a model of (1) the distribution of genes across all individuals, and (2) the dependence relations or independence relations of genes between individuals.

One way to estimate the distribution of genes from all the individuals in the population is by using marginal distribution. The simplest marginal distribution calculates the probability of each candidate solutions' genes independently. This strategy is called as univariate marginal distribution. This contrasts with bivariate marginal distribution, which calculates the estimation based on the dependency of two genes. The dependency that is of our interest is the statistically significant dependency, which can be computed using Pearson's chi-square statistics [26].

The probability distribution for the univariate marginal distribution is calculated using the frequency of each gene from all or truncated individuals and stored as Probability Vector (PV). The PV will be used to sample or generate new individuals in subsequent generations. For the bivariate marginal distribution, we start with calculating the frequency of each gene. Then, we calculate the joint probability of each pair of genes, using the previously calculated frequency value. Afterward, for each pair of genes, we calculate the Pearson's chi-square tests to establish links between interdependent

genes. The result of this is a set of genes dependency and we only consider two genes as interdependent if the value is statistically significant. Next, we generate a dependency graph to store this information. The graph is acyclic and not necessarily has to be connected. All genes that have no interdependency with other genes are assigned as a root node in the graph. Whereas, for those that have a link, if both genes are not yet added to the graph, choose any gene from that pair as the root node. Add the other gene as a child node and connect them using an edge. Among them, nodes that are added earlier are called as the parent node.

Based on the constructed graph, we generate a new population. First, populate genes for root nodes using univariate frequencies. Next, for each child nodes, populate the genes using conditional probability. Perform the same process for all child nodes. From there, the standard evolutionary step is applied, which is fitness evaluation of each individual in the new population. The population is truncated, and the process repeats until an acceptable solution is found. The intuition is, univariate-based EDA is considered as a lightweight evolutionary algorithm and require small memory footprint [27], whereas the bivariate EDA manifest possible variables interdependency [18].

B. Bivariate Distribution of SPL Features

This section presents the mechanism and illustration of a second-order EDA (bivariate distribution) to generate pairwise test configuration for SPL. Here, we named this approach as Combinatorial Testing using Estimation of Distribution (COTED).

The proposed strategy is generally outlined as follows:

1) Generate a set of test configuration as the initial population. Calculate the fitness of each test configuration using the number of covered pairwise. Then, we perform truncation to select highly fit candidate solutions.

2) Calculate univariate frequencies and bivariate frequencies of each feature number.

3) Create a feature configuration dependency graph using the calculated frequency.

4) Generate new test configurations based on the graph.

5) Repeat until termination criteria are matched.

We define our fitness function as the number of pairs of features covered by each test configuration, whereby, the more pairs covered, the better the fitness. The intuition is, during the search for fitter test configurations, the stronger the dependency of a particular pair of features present in the current fittest test configuration, the more frequent it should be included in the subsequent list of test configuration. For example, if the dependency of features 5 and 7 are statistically significant in our 10 best test configurations, we should create more test configurations using the calculated conditional distribution of features 5 and 7 in the next iteration. The definition of best test configurations refers to those that cover a higher number of pairs from our list of all valid pairs.

By modeling the non-dependency between two features in a set of test configurations, we can search for possible dependency between two features. This dependency would suggest that the two features should be paired, and those that have strong dependency should be considered first. A variant of EDA that has the capability to find this dependency is called the Bivariate Marginal Distribution Algorithm (BMDA) [17], [28]. It uses a factorization of the univariate marginal and joint probability distribution that able to expose second-order dependencies. For our test configuration generation problem, we define the univariate marginal and joint probability distribution as follows:

Definition 1 (univariate marginal probability)

The probability of a feature is selected, $p(x_i)$, is unconditional to other features. For example, $p(5) = 0.7$, means that the probability of feature number 5 to be selected is 70 per cent from all test configurations.

Definition 2 (joint probability)

Joint probability between feature v and feature w, $JP_{v,w}$, is defined as the probability of feature v and feature w been considered (either selected or not selected).

In the remaining part of this section, we elaborate the details of the mechanism to generate test configuration that satisfies pairwise coverage of feature configuration.

Step 1. Feature configurations in FM are governed by constraints, so that only valid test configurations are generated. A SAT solver is utilized to populate the seed of our search space. Once a collection of valid test configurations is available, we calculate their fitness using pairwise coverage and remove unfit test configurations. To illustrate this, we make a list of valid pair of features that needs to be covered in Listing 1. We start by populating 20 test configurations from SAT solver and calculate the number of pairwise covered by each test configuration as its fitness value. We sort and select 10 fittest test configurations as our truncated initial population, which is presented in Fig. 3.

```
Pair of features, pf = { (-3,20), (-3,21), (5,19), (-5,20), (5,22), (-5,23),
(-6,18), (8,9), (8,19), (8,21), (-8,23), (-9,19), (9,23), (-10,20), (-10,23),
(11,19), (11,22), (-11,23), (12,19), (12,22), (-12,23), (-16,23), (19,23),
(20,22), (21,24), (21,-25), (22,-24), (22,25) }
```

Listing 1. Pair of valid features that needs to be covered.

Test Configuration, TC	Fitness
01:[1,2, 3,4, 5, 6,7, 8, 9, 10,-11, 12,13,14,15, 16, 17,-18, 19,-20,-21, 22,-23,-24, 25]	8
02:[1,2, 3,4, 5, 6,7, 8,-9, 10, 11,-12,13,14,15, 16,-17,-18, 19,-20,-21, 22,-23,-24, 25]	8
03:[1,2,-3,4, 5, 6,7,-8,-9, 10, 11, 12,13,14,15,-16, 17,-18,-19, 20,-21, 22,-23,-24, 25]	7
04:[1,2,-3,4, 5, 6,7, 8, 9, 10, 11,-12,13,14,15, 16, 17,-18, 19,-20,-21, 22,-23, 24,-25]	6
05:[1,2, 3,4,-5, 6,7,-8, 9, 10,-11,-12,13,14,15, 16, 17,-18,-19, 20,-21,-22, 23,-24, 25]	6
06:[1,2,-3,4, 5,-6,7,-8, 9,-10, 11, 12,13,14,15,-16,-17, 18,-19,-20,-21, 22,-23,-24, 25]	6
07:[1,2,-3,4,-5,-6,7,-8,-9, 10,-11,-12,13,14,15, 16, 17,-18,-19, 20, 21,-22,-23, 24,-25]	5
08:[1,2,-3,4,-5,-6,7,-8,-9, 10,-11,-12,13,14,15, 16,-17, 18,-19,-20,-21,-22, 23, 24,-25]	5
09:[1,2, 3,4, 5,-6,7,-8, 9, 10, 11,-12,13,14,15,-16, 17,-18,-19, 20,-21,-22, 23,-24, 25]	4
10:[1,2,-3,4, 5,-6,7,-8, 9, 10, 11,-12,13,14,15, 16,-17,-18,-19, 20,-21,-22, 23, 24,-25]	4

Fig. 3. List of initial truncated population with fitness value.

Step 2. We calculate the univariate distribution for each feature. Each feature in each test configuration has either positive or negative prefix. We compute the mean of positive number for each feature from all test configurations. These processes are presented in Algorithm 1. The result of this process is the Probability Vector, PV of our initial population, as shown in Fig. 4.

Algorithm 1. Calculate Probability Vector

1. Load a population of candidate solutions
2. Select 10 test configurations according to its pairwise fitness
3. Let *n* be the length of a test configuration
4. For *i* = 1 to *n*
5. Calculate the mean of positive *i* as *mean_i*
6. Set the probability vector for feature *i*, *P(i) = mean_i*
7. Next *i*

PV:[1.0,1.0,0.4,1.0,0.7,0.5,1.0,0.3,0.6,0.9,0.6,0.3,
1.0,1.0,1.0,0.7,0.6,0.2,0.3,0.5,0.1,0.5,0.4,0.4,0.6]

Fig. 4. Probability vector of initial population.

The next step is to calculate the joint probability of all possible value in each pair of the feature. As an example, for features 5 and 11, we calculate the occurrences of all four pairs; i.e. (5,11), (5,-11), (-5,11) and (-5,-11). Based on the population in Fig. 3, we get the joint probability value of 0.6, 0.1, 0.0 and 0.3, respectively. This process is defined in Algorithm 2, line 2 to 8.

After that, calculate the Pearson's chi-square statistics, $C_{v,w}$, for each pair of features using the following equation:

$$C_{v,w} = n * \sum_{\alpha,\beta} \frac{[JP_{\alpha v,\beta w} - P(\alpha v)P(\beta w)]^2}{P(\alpha v)P(\beta w)}$$

where n is the number of test configuration

α is either positive or negative prefix

β is either positive or negative prefix

For example, for v=5 and w=11:

$$C_{5,11} = 10 * \left(\frac{\left(JP_{5,11} - P(5)P(11)\right)^2}{P(5)P(11)} + \frac{\left(JP_{5,-11} - P(5)P(-11)\right)^2}{P(5)P(-11)} + \frac{\left(JP_{-5,11} - P(-5)P(11)\right)^2}{P(-5)P(11)} + \frac{\left(JP_{5,11} - P(-5)P(-11)\right)^2}{P(-5)P(-11)} \right) = 10 * \left(\frac{(0.6 - (0.7*0.6))^2}{0.7*0.6} + \frac{(0.1 - (0.7*0.4))^2}{0.7*0.4} + \frac{(0.0 - (0.3*0.6))^2}{0.3*0.6} + \frac{(0.3 - (0.3*0.4))^2}{0.3*0.4} \right) = 6.4$$

This step is defined in Algorithm 2, line 9 to 14. Based on our sampled population, the calculated bivariate frequencies are shown in Fig. 5. Here, we are only interested in chi-square value of at least 3.84 [17], based on the degree of freedom of 1 and p value of 0.05. By calculating the chi-square values of the initial population, we choose 11 feature pairs. These pairs are conceived as having a strong dependency, due to the high degree of correlation.

Algorithm 2. Calculate Bivariate Frequencies

1. Initialize joint probability, *JP*
2. For *v* = 1 to *n* - 1
3. For *w* = 2 to *n*
4. For each test configuration, *tc*
5. Calculate joint probability between feature *v* and *w*, $JP_{v,w}$, group by combination of positive and negative prefix
6. Next *tc*
7. Next *w*
8. Next *v*
9. Initialize chi-square, *C*
10. For *v* = 1 to *n*-1
11. For *w* = 2 to *n*
12. Calculate the Pearson's chi-square statistics $C_{v,w}$
13. Next *w*
14. Next *v*

Feature Pair	Chi-square
(8,19)	10.0
(24,25)	10.0
(22,23)	6.6
(5,11)	6.4
(10,18)	4.5
(3,24)	4.5
(5,22)	4.4
(8,22)	4.4
(12,22)	4.4
(8,20)	4.4
(6,8)	4.4
Other pair	<3.84

Fig. 5. Bivariate frequencies of the initial population.

Step 3. The succeeding step is to create a Feature Configuration Dependency Graph (FCDG). We define FCDG as a forest and are specified in Definition 3.

Definition 3. (Feature Configuration Dependency Graph, FCDG).

FCDG = (V,E), where V is the set of all features available in the forest, and E is the set of all edges between some ordered pairs of features. FCDG contains a collection of possibly disconnected trees.

Each feature is represented by a node, and dependency between the pair of features is represented by an edge. The dependency between features is to be calculated based on conditional probability, thus its relationship is of type directional. Therefore, we link the respective nodes in our FCDG using directed edges.

We define the following six properties for the FCDG:

1) The indegree of a node is the number of edges directing to that node. Each node has zero or one indegree.

2) The outdegree of a node is the number of edges leading away from that node. Each node has zero or more outdegree.

3) A node with zero indegree and non-zero outdegree is called as a root node. FCDG can have more than one root node.

4) A node with non-zero indegree is called as a child node.

5) A node with non-zero outdegree is called as a parent node.

6) A node without a degree is called as a standalone node.

The generation process of FCDG starts by selecting a random feature and adds it to our graph. Then, add a dependent feature by finding another feature having the highest chi-square value of at least 3.84, and add it to the graph. Repeat this step until no more features fulfil this criterion. Then, select another random feature and repeat the whole process until all features are added to the graph. This process is defined in Algorithm 3.

Algorithm 3. Create Feature Configuration Dependency Graph

1. Let *W* as the set of all features
2. Let *F* as an empty graph, consists of empty *V* and *E*
3. Select a random feature, *r*, from *W*
4. Add *r* to the graph, *V*
5. Remove *r* from *W*
6. If there are no more features in *W*, goto end
7. For each remaining features in *W*
8. Find a feature, *s*, that has the highest dependency to feature *r*
9. If found
10. add *s* to *V*
11. removes *s* from *W*
12. add set {*r*,*s*} into *E*
13. if not found
14. goto step 3
15. end if
16. End

Executing this algorithm against the values from Fig. 5 will result in a graph with the following attributes:

- The set of all features, V = {1,2,3,…,25}
- Edges between some ordered feature pairs, E = { {3,24}, {5,11}, {8,6}, {8,19}, {8,20}, {18,10 }, {22,5}, {22,8}, {22,12,}, {22,23}, {24,25} }

In this case, the FCDG for our running example consists of 25 nodes with 11 edges. This can be graphically presented using a forest with three disconnected trees as shown in Fig. 6. All nodes in white colour are standalone nodes, in which no dependency to other nodes is discovered. The rest (coloured nodes) are nodes with dependency. As an example, it is shown that features 18 and 10 are highly dependent. From Fig. 3, feature 18 is always negative whenever feature 10 is positive, and the other way round. Another example is between feature 8 and 19. It is of high frequency that both having negative values in the same row. The same relationship (of a certain pattern) can be observed for the rest of the pairs.

Step 4. Once we have the dependency graph, we can proceed with generating a new population. It consists of two parts, (1) to populate root nodes, and (2) to populate child node.

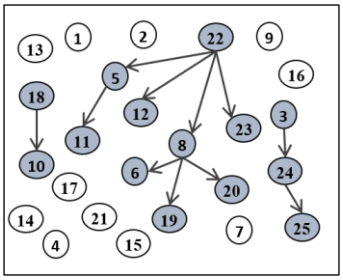

Fig. 6. The feature configuration dependency graph of the initial population.

We start by populating all features correspond to the root nodes in our graph. The features are assigned with positive or negative values using univariate probabilities. Then, we populate the remaining features that correspond to the child nodes. This is performed by calculating the conditional probabilities of the child nodes given its parent nodes. We define the conditional probabilities for our strategy as follows:

Definition 4. (conditional probability)

Conditional probability of feature s and feature r, *CP(s|r)* is defined as the probability of feature *s* to be selected, given feature *r* been selected. It is calculated using the joint probability of *s* and *r*, $JP_{s,r}$, divide by the univariate probability of *r*, i.e *P(r)*.

$$\mathrm{CP}(\mathrm{s} \mid \mathrm{r}) = \frac{JP_{s,r}}{P(r)}$$

This process is defined from line 3 to line 8 of Algorithm 4.

Algorithm 4. Populate New Generation

1. For each root nodes, *r*, in *G*
2. Populate new generation having positive/negative value of *r* using univariate frequencies
3. For other nodes, *s*, in *G*
4. If parent node of *s* has been populated
5. Populate positive/negative value of *s* based on the conditional probability of *s* given parent of *s*
6. If all features have been populated, goto end
7. Next root node
8. End

To demonstrate the first part, which is populating all the root nodes, Fig. 7 shows a possible assignment for 20 test configurations of our new generation. For example, for feature 16, from our initial generation, the PV value for feature 16 is 0.7, hence 70% of the new generation should have the positive value of 16. This can be achieved by using random numbers generated from a uniform distribution between 0 and 1. As per shown in Fig. 7, the outcome of this strategy is the assignment of a positive value of 16 for test configurations TC01, TC04-TC08, TC10, TC11, TC14 TC16 and TC18 TC20. The remaining test configurations are assigned with -16. We apply the same strategy to populate the remaining root node features, and values are presented in Fig. 7. For non-root node features, which we mark with unfilled squares (□), will be populated later.

The second part populates the remaining features, with respect to the child nodes from our dependency graph, i.e. features 5, 6, 8, 10, 11, 12, 19, 20, 23, 24, 25. Let us choose feature 12 as an example. Since feature 22 has been assigned with values, we assign feature 12 given the respective values of feature 22, using conditional distribution. It can be calculated using the joint probability of both features having positive values in the initial population, i.e. 0.3. Then divide by the probability vector of feature 22, i.e. 0.5. This equates to 0.6. Thus, we populate 60% of feature 12 with positive values for test configuration having positive 22. Similarly, calculate the probability of positive 12 given the negative value of feature 22, and use the result to populate the value of remaining test configurations. Once all values for feature 12 have been

assigned, we use the same strategy to populate the remaining features. A possible outcome of this process is shown in Fig. 8.

Step 5. The final step in this iteration is to calculate the fitness of each individual in the new generation. We count how many pairs from Listing 1 matched with each pair of features in each test configuration. The fitness values are shown in the right column of Fig. 8. It is observed that three test configurations (TC10, TC17, and TC18) have better fitness value (marked with *) compared to the previous generation of test configuration (refer Fig. 3). Truncation is again applied to select only ten highly fit individuals. The whole process repeats from Algorithm 1 and continue until the intended pairwise coverage has been met.

```
List of root node features=
   [1, 2, 3, 4, 7, 9, 13, 14, 15, 16, 17, 18, 21, 22]
PV (for root node features)=
 [1.0, 1.0, 0.4, 1.0, 1.0, 0.6, 1.0, 1.0, 1.0, 0.7, 0.6, 0.2, 0.1, 0.5]
New Generation of Test Configuration, TC:
01:[1,2,-3,4,□,□,7,□, 9,□,□,□,13,14,15, 16,-17,-18,□,□,-21,-22,□,□,□]
02:[1,2,-3,4,□,□,7,□,-9,□,□,□,13,14,15,-16, 17, 18,□,□,-21,-22,□,□,□]
03:[1,2,-3,4,□,□,7,□, 9,□,□,□,13,14,15,-16, 17,-18,□,□,-21, 22,□,□,□]
04:[1,2, 3,4,□,□,7,□,-9,□,□,□,13,14,15, 16, 17, 18,□,□,-21, 22,□,□,□]
05:[1,2, 3,4,□,□,7,□, 9,□,□,□,13,14,15, 16, 17,-18,□,□,-21,-22,□,□,□]
06:[1,2,-3,4,□,□,7,□, 9,□,□,□,13,14,15, 16, 17, 18,□,□,-21,-22,□,□,□]
07:[1,2,-3,4,□,□,7,□,-9,□,□,□,13,14,15, 16, 17,-18,□,□,-21, 22,□,□,□]
08:[1,2, 3,4,□,□,7,□, 9,□,□,□,13,14,15, 16, 17,-18,□,□,-21,-22,□,□,□]
09:[1,2,-3,4,□,□,7,□, 9,□,□,□,13,14,15,-16, 17, 18,□,□,-21,-22,□,□,□]
10:[1,2, 3,4,□,□,7,□, 9,□,□,□,13,14,15, 16, 17, 18,□,□,-21,-22,□,□,□]
11:[1,2, 3,4,□,□,7,□, 9,□,□,□,13,14,15, 16,-17, 18,□,□, 21, 22,□,□,□]
12:[1,2,-3,4,□,□,7,□, 9,□,□,□,13,14,15,-16,-17,-18,□,□,-21,-22,□,□,□]
13:[1,2,-3,4,□,□,7,□, 9,□,□,□,13,14,15,-16,-17,-18,□,□,-21, 22,□,□,□]
14:[1,2, 3,4,□,□,7,□,-9,□,□,□,13,14,15, 16, 17,-18,□,□, 21,-22,□,□,□]
15:[1,2,-3,4,□,□,7,□, 9,□,□,□,13,14,15, 16, 17,-18,□,□,-21,-22,□,□,□]
16:[1,2, 3,4,□,□,7,□, 9,□,□,□,13,14,15, 16,-17, 18,□,□,-21,-22,□,□,□]
17:[1,2,-3,4,□,□,7,□,-9,□,□,□,13,14,15,-16,-17,-18,□,□,-21, 22,□,□,□]
18:[1,2, 3,4,□,□,7,□,-9,□,□,□,13,14,15, 16, 17,-18,□,□,-21, 22,□,□,□]
19:[1,2,-3,4,□,□,7,□, 9,□,□,□,13,14,15, 16, 17,-18,□,□,-21, 22,□,□,□]
20:[1,2, 3,4,□,□,7,□,-9,□,□,□,13,14,15, 16,-17,-18,□,□,-21,-22,□,□,□]
```

Fig. 7. Populated root node features using univariate frequencies.

```
List of Non-Root node features =
[5, 6 , 8 , 10, 11, 12, 19, 20, 23, 24, 25]
New Generation of Test Configuration, TC:                                                                    Fitness
01:[1,2,-3,4,-5,-6,7,-8, 9, 10,-11,-12,13,14,15, 16,-17,-18,-19,-20,-21,-22, 23, 24,-25]      5
02:[1,2,-3,4, 5,-6,7,-8,-9, 10, 11,-12,13,14,15,-16, 17, 18,-19, 20,-21,-22, 23, 24,-25]      5
03:[1,2,-3,4, 5, 6,7, 8, 9, 10, 11, 12,13,14,15,-16, 17,-18, 19,-20,-21, 22,-23, 24,-25]      8
04:[1,2, 3,4, 5, 6,7, 8,-9,-10,-11,-12,13,14,15, 16, 17, 18, 19,-20,-21, 22,-23,-24,-25]      5
05:[1,2, 3,4, 5,-6,7,-8, 9, 10, 11,-12,13,14,15, 16, 17,-18,-19,-20,-21,-22, 23,-24, 25]      3
06:[1,2,-3,4, 5, 6,7,-8, 9, 10, 11,-12,13,14,15, 16, 17, 18,-19,-20,-21,-22, 23, 24,-25]      3
07:[1,2,-3,4, 5, 6,7, 8,-9, 10, 11,-12,13,14,15, 16, 17,-18, 19,-20,-21, 22,-23, 24,-25]      6
08:[1,2, 3,4, 5,-6,7,-8, 9, 10, 11,-12,13,14,15, 16, 17,-18,-19, 20,-21,-22, 23,-24,-25]      3
09:[1,2,-3,4, 5,-6,7,-8, 9,-10,-11,-12,13,14,15,-16, 17, 18,-19,-20,-21,-22, 23, 24,-25]      7
10:[1,2, 3,4,-5,-6,7,-8, 9,-10,-11,-12,13,14,15, 16, 17, 18,-19, 20,-21,-22, 23,-24,-25]      9 *
11:[1,2, 3,4, 5, 6,7,-8, 9, 10, 11, 12,13,14,15, 16,-17, 18,-19,-20, 21, 22,-23,-24, 25]      5
12:[1,2,-3,4,-5,-6,7,-8, 9, 10, 11,-12,13,14,15,-16,-17,-18,-19, 20,-21,-22, 23, 24,-25]      7
13:[1,2,-3,4, 5, 6,7, 8, 9, 10, 11,-12,13,14,15,-16,-17,-18, 19,-20,-21, 22,-23,-24, 25]      8
14:[1,2, 3,4,-5, 6,7,-8,-9, 10,-11,-12,13,14,15, 16, 17,-18,-19, 20, 21,-22, 23,-24,-25]      6
15:[1,2,-3,4,-5, 6,7,-8, 9, 10,-11,-12,13,14,15, 16, 17,-18,-19, 20,-21,-22, 23, 24,-25]      7
16:[1,2, 3,4,-5,-6,7,-8, 9,-10, 11,-12,13,14,15, 16,-17, 18,-19, 20,-21,-22, 23,-24,-25]      8
17:[1,2,-3,4, 5, 6,7, 8,-9, 10, 11, 12,13,14,15,-16,-17,-18, 19,-20,-21, 22,-23,-24,-25]      9 *
18:[1,2, 3,4, 5, 6,7, 8,-9, 10, 11, 12,13,14,15, 16, 17,-18, 19,-20,-21, 22,-23,-24, 25]     10 *
19:[1,2,-3,4, 5, 6,7, 8, 9, 10, 11, 12,13,14,15, 16, 17,-18, 19,-20,-21, 22,-23, 24,-25]      8
20:[1,2, 3,4,-5,-6,7,-8,-9, 10,-11,-12,13,14,15, 16,-17,-18,-19,-20,-21,-22, 23,-24, 25]      4
```

Fig. 8. Populated non-root node features using conditional distribution and calculated fitness value.

IV. Experiment and Results

COTED has been implemented and executed on a set of feature models from Software Product Line Online Tools (SPLOT) [7]. The objective is to measure the efficiency and effectiveness of bivariate distribution approach based on EDA in generating test configuration satisfying pairwise testing. The comparison has been made against (1) a greedy-based approach, ICPL [3] and (2) a constraint handling approach based on the minimum-invalid-tuple strategy, LOOKUP [29].

The first part assesses the efficiency by measuring the minimum number of test configurations that the three approaches able to generate. The second part measures the quality of the generated test configuration, in terms of the frequency of pairwise tuple, and test configuration similarity. During the experiments, 8 datasets of various sizes of constrained Feature Models (FMs) have been selected from SPLOT. COTED has been executed with the population of size 800 with truncation size 100, stagnancy count of 3 executions, maximum generations were 5000 and execution timeout of 1800 seconds.

A. Minimum Number of Test Configurations

This is the most used metric that evaluates the efficiency of the solution for SPL test configuration generation [12]. It calculates the number of test configurations generated using a particular approach that either fully satisfies the pairwise

coverage, or partially fulfil the coverage with a decent percentage. However, the latter does not conform to the definition of pairwise testing, i.e. to have all pairs covered at least once. Therefore, a complete pairwise coverage is often of the goal in any SPL test configuration exercise.

Fig. 9 shows that LOOKUP is the most outstanding tool in generating the most minimal test configurations. For J2EEWebArch and CocheEcologico, it outperforms the other techniques. For others, it produces an equal number of test configuration as generated by COTED, except for SPLSimulES dataset.

B. t-wise Frequency

This measure has been devised by Perrouin et al. [30] as the ratio between the occurrences of t-wise and the number of test configurations generated. This can be used to check whether the solution satisfies the t-wise principle, i.e., in the solution, every valid combination of t factors must present at least once. An optimum solution consists of combination of t factors once. This, however, is hard to achieve.

Fig. 10 shows the box plots of all evaluated techniques calculated based on the median of t-wise frequencies of the generated test configurations for each benchmark datasets. In general, the average and the dispersion of the t-wise frequencies are stable for the three techniques. Most of the results show that the frequencies are maintained low, as depicted by the concentration on the low end of the scale, except for Ecommerce (Fig. 10(a)) and Billing (Fig. 10(g)) datasets. Low frequency of t-wise in the generated test configurations indicates that there are less pairwise occurrences; hence lower the redundancy of feature of pairs. This is useful in the event of limited time and resources available for testing, which is often the case in SPL testing.

On average, as shown in Table I, the median and standard deviation (σ) of the proposed techniques resides on the decent level, which is on par with the other approaches. Even though, on average, ICPL can demonstrate lower t-wise frequency (0.288), the deviation of the overall solution is worse than the rest. On the other side of the coin, LOOKUP and COTED managed to cover pairwise steadily, with low variations, on average, however, it covers higher frequency than ICPL. The differences between COTED and LOOKUP are relatively low. 50 per cent of the overall medians are equal for both techniques.

Datasets	Num. of Features	Num. of Constraints	TC Generation Techniques		
			COTED	ICPL	LOOKUP
Ecommerce	10	10	**6**	7	**6**
Cellphone	11	14	**7**	8	**7**
GraphProductLine	20	30	**15**	17	**15**
SPLSimulES	32	25	**10**	**10**	11
ArcadeGame	61	87	**16**	18	**16**
J2EEWebArch	77	86	19	18	**17**
Billing	88	89	**13**	14	**13**
CocheEcologico	94	131	92	93	**90**

Fig. 9. Minimum number of test configuration generated.

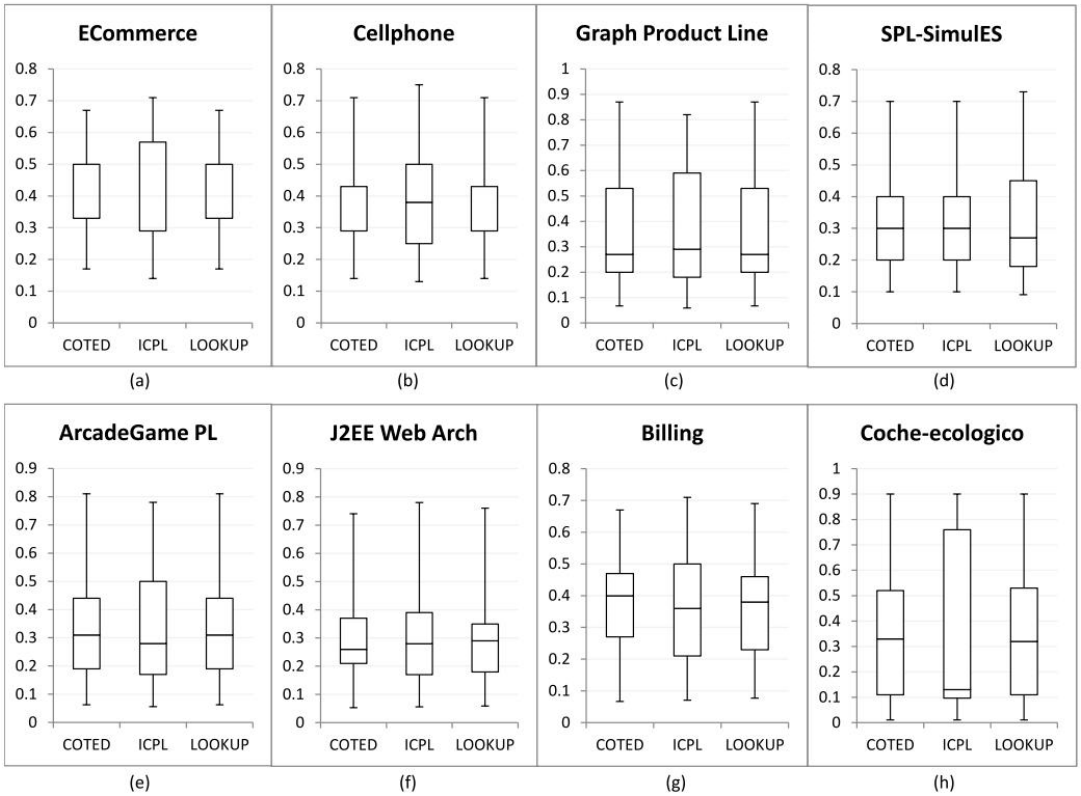

Fig. 10. Box plots for the median of t-wise frequency of the three approaches.

TABLE I. MEDIAN AND STANDARD DEVIATION (σ) OF T-WISE FREQUENCY

Techniques / Datasets	COTED		ICPL		LOOKUP	
	Median	σ	*Median*	σ	*Median*	σ
ECommerce	0.33	0.1634	0.29	0.1910	0.33	0.1634
Cellphone	0.29	0.1615	0.38	0.1636	0.29	0.1615
SPL-SimulES	0.3	0.1498	0.3	0.1543	0.27	0.1470
ArcadeGamePL	0.31	0.1695	0.28	0.2101	0.31	0.1771
Graph Product Line	0.27	0.2280	0.29	0.2289	0.27	0.2263
J2EE Web Arch	0.26	0.1440	0.28	0.1566	0.29	0.1391
Billing	0.4	0.1530	0.36	0.1782	0.38	0.1553
Coche-ecologico	0.33	0.2488	0.13	0.3353	0.32	0.2530
Average	0.311	0.177	0.288	0.202	0.307	0.177

C. Test Configurations Similarity

The third measurement is test configuration similarity [30]. The objective is to assess the degree of similarity between test configurations among a different set of solutions. The similarity between two test configurations is calculated using Jaccard Index, *Jac*. Given a and b as the two test configurations, we calculate Jac(a,b) as follows:

$$\text{Jac(a,b)} = \frac{|a \cap b|}{|a \cup b|}$$

The presence of all mandatory features is a must in all test configurations. Since all solutions from the three techniques are of valid test configurations, we omit the similarity checking for mandatory features. Only optional features are observed.

This similarity measure can be used to measure the degree of diversity of the generated solutions. Lower Jaccard Index value indicates that the test configurations are less likely to be similar, hence more diversified. Fig. 11 shows the box plots calculated based on the median of the test configuration similarity from the generated solutions for each benchmark datasets. Overall, the averages of the test configuration similarity are low and encouraging among all techniques, and the dispersions of the median are stable for all techniques. This is depicted in Fig. 11 based on the trend of right skewness, as most medians are closer to the first quartile than the third quartile. COTED performance is on par with LOOKUP, and in fact, it managed to outperform LOOKUP at SPL-SimulES dataset. Overall, COTED and LOOKUP outperform ICPL for most datasets.

With respect to the average and measure of dispersion, as shown in Table II, LOOKUP performed better than the rest, with the exception to three datasets (Cellphone, SPL SimulES and ArcadeGamePL) where COTED has a bit lower median values. Meanwhile, the median of COTED is better than ICPL, with lower median and σ on five datasets (ECommerce, Cellphone, ArcadeGamePL, Graph Product Line and Coche ecologico). This suggests, on average, it produces more dissimilar sets of test configurations.

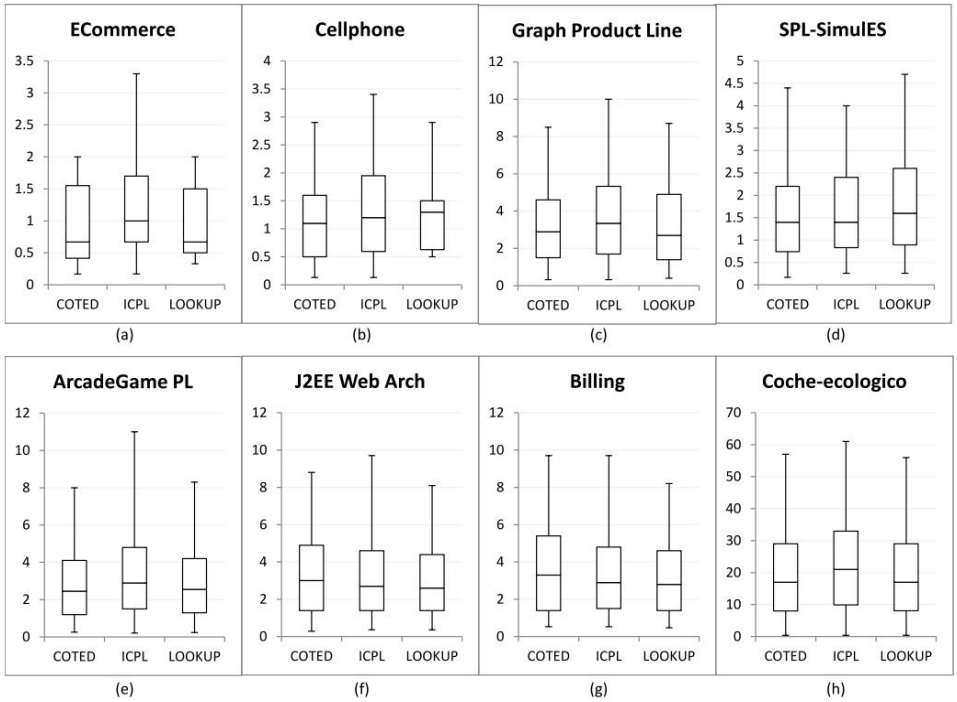

Fig. 11. Box plots for the median of t-wise similarity of the three approaches.

TABLE II. MEDIAN AND STANDARD DEVIATION (σ) OF TEST CONFIGURATION SIMILARITY

Techniques / Datasets	COTED		ICPL		LOOKUP	
	Median	*σ*	*Median*	*σ*	*Median*	*σ*
ECommerce	0.67	0.6581	1	0.8064	0.67	0.6489
Cellphone	1.1	0.8134	1.2	0.8955	1.3	0.7188
SPL-SimulES	1.4	1.0682	1.4	1.0199	1.6	1.1632
ArcadeGamePL	2.45	1.9168	2.9	2.2197	2.55	1.9063
Graph Product Line	2.9	2.1399	3.35	2.4305	2.7	2.1715
J2EE Web Arch	3	2.2429	2.7	2.1194	2.6	1.9755
Billing	3.3	2.3822	2.9	2.2185	2.8	2.0049
Coche-ecologico	17	13.6009	21	14.3204	17	13.3103
Average	3.977	3.102	4.556	3.253	3.902	2.987

V. DISCUSSIONS

By calculating the marginal distribution between every two features in a particular sample, we can infer its connection. And based on that strong assumption, the population evolve towards more frequently connected features. This can directly be translated to more pairwise coverage. The ability to maximize pairwise coverage at each evolution cycle results in the reduction in the overall cycles of exploration, and subsequently, reduce the number of generated test configuration that fulfil pairwise coverage.

This strategy has been evaluated against two current approaches, i.e. greedy-based and minimum invalid tuple based. Of the three strategies, the minimum invalid tuple-based strategy performed the best, but, competitively challenged by COTED, and this is supported by results analysis using descriptive statistics.

Even though the performance of COTED is shown to be comparable, if not better than other approach, it provides us with a set of knowledge on the problem structure. By analysing the evolution of the probability models during test configuration generation, we discover a set of data on how the problem is being solved. We also gain knowledge on how features are distributed in the population with respect to other features. We explicitly acquire this in the form of feature configuration dependency graph which stores a set of feature pairs that have strong dependency. This information is deemed crucial as it could help us (1) decide how to prioritize the test configurations in pairwise testing, and (2) infer a higher order marginal distribution based on the collection of dependency knowledge.

As compared to test generation, previous literature highlighted that test prioritization for SPL is insufficiently researched, especially on one that is based on feature reusability [31]. Reusable features are features that appear more frequently in final software products than the others. Hence, calculating the frequency might help in extracting the most reusable one. In view of interaction testing, two interacting features are of one main concern. Thus, to find those reusable interactions could mean to find common feature interactions.

The dependency knowledge in the form of a collection of feature configuration dependency graphs are acquired iteratively from second-order probabilistic model. As opposed to computing a higher-order probabilistic model (which involves multivariate computation), this process is more viable as it incurs much lower cost. Additionally, a higher-order probabilistic model is achievable by grouping or clustering lower-order dependencies which contains highly interacting sets of variables [32]. Therefore, we could leverage a lightweight second-order iterative computation for practical higher-order computation. This remains to be investigated and thus motivate our future work.

VI. CONCLUSION AND FUTURE WORKS

Generating efficient and effective test configurations for SPL is difficult. One way to feasibly tackle the combinatorial explosions of feature configuration testing is by leveraging pairwise testing.

Based on the work we conducted throughout this study, we found that the marginal distribution algorithm-based approach is a feasible and competitive strategy. It allows us to reduce the number of required test configuration from an exhaustive approach by leveraging pairwise coverage as its fitness function. Our proposed strategy managed to generate the solution of similar quality in terms of t-wise frequency and test configuration diversity, compared to those generated by state-of-the-art approaches. The outcome of the proposed strategy is two-fold. First, it generates minimized test configuration for pairwise testing. Secondly, the inherent ability of the strategy to extract the dependency knowledge in the form of feature configuration dependency graphs. As per our knowledge, this is the first time a combinatorial interaction testing in software product line problem is being modelled and tackled by using probability based evolutionary algorithm.

ACKNOWLEDGMENT

This research was supported by Universiti Tun Hussein Onn Malaysia (UTHM) through Tier 1 (Vote Q103).

REFERENCES

[1] D. Hinterreiter, K. Feichtinger, L. Linsbauer, H. Prähofer, and P. Grünbacher, “Supporting feature model evolution by lifting code-level dependencies: A research preview,” in Requirements Engineering:

Foundation for Software Quality: 25th International Working Conference, REFSQ 2019, Essen, Germany, March 18--21, 2019, Proceedings 25, 2019, pp. 169–175.

[2] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. le Traon, "Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines," Softw. Eng. IEEE Trans., vol. 40, no. 7, pp. 650–670, 2014, doi: http://doi.org/10.1109/TSE.2014.2327020.

[3] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in Proceedings of the 16th International Software Product Line Conference-Volume 1, 2012, pp. 46–55. doi: 10.1145/2362536.2362547.

[4] A. Bajaj and O. P. Sangwan, "A systematic literature review of test case prioritization using genetic algorithms," IEEE Access, vol. 7, pp. 126355–126375, 2019.

[5] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, vol. 1, pp. 517–528.

[6] N. Khoshniat, A. Jamarani, A. Ahmadzadeh, M. Haghi Kashani, and E. Mahdipour, "Nature-inspired metaheuristic methods in software testing," Soft Comput., pp. 1–42, 2023.

[7] M. Mendonca, M. Branco, and D. Cowan, "SPLOT: software product lines online tools," in Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, 2009, pp. 761–762.

[8] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," IEEE Softw., vol. 13, no. 5, p. 83, 1996.

[9] D. Gupta and L. Sharma, "Improved Combinatorial Algorithms Test for Pairwise Testing Used for Testing Data Generation in Big Data Applications," in Artificial Intelligence, Chapman and Hall/CRC, 2021, pp. 81–90.

[10] J. Ferrer, F. Chicano, and J. A. Ortega-Toro, "CMSA algorithm for solving the prioritized pairwise test data generation problem in software product lines," J. Heuristics, vol. 27, pp. 229–249, 2021.

[11] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Combinatorial Test Generation for Software Product Lines Using Minimum Invalid Tuples," in 15th International Symposium on High-Assurance Systems Engineering (HASE), 2014, pp. 65–72. doi: 10.1109/HASE.2014.18.

[12] P. Ramgouda and V. Chandraprakash, "Constraints handling in combinatorial interaction testing using multi-objective crow search and fruitfly optimization," Soft Comput., vol. 23, no. 8, pp. 2713–2726, 2019, doi: 10.1007/s00500-019-03795-w.

[13] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "Improving CASA runtime performance by exploiting basic feature model analysis," arXiv Prepr. arXiv1311.7313, 2013.

[14] J. Ceberio, A. Mendiburu, and J. A. Lozano, "A roadmap for solving optimization problems with estimation of distribution algorithms," Nat. Comput., pp. 1–15, 2022.

[15] A. Shirazi, J. Ceberio, and J. A. Lozano, "EDA++: Estimation of distribution algorithms with feasibility conserving mechanisms for constrained continuous optimization," IEEE Trans. Evol. Comput., vol. 26, no. 5, pp. 1144–1156, 2022.

[16] S. Shakya and R. Santana, "A Review of Estimation of Distribution Algorithms and Markov Networks," in Markov Networks in Evolutionary Computation, vol. 14, S. Shakya and R. Santana, Eds. Springer Berlin Heidelberg, 2012, pp. 21–37. doi: 10.1007/978-3-642-28900-2_2.

[17] D. Simon, "Estimation of Distribution Algorithms," in Evolutionary Optimization Algorithms, John Wiley & Sons, 2013, pp. 313–347.

[18] M. Pelikan, M. Hauschild, and F. Lobo, "Estimation of Distribution Algorithms," in Springer Handbook of Computational Intelligence, J. Kacprzyk and W. Pedrycz, Eds. Springer Berlin Heidelberg, 2015, pp. 899–928. doi: 10.1007/978-3-662-43505-2_45.

[19] R. Sagarna and J. Lozano, "Software Metrics Mining to Predict the Performance of Estimation of Distribution Algorithms in Test Data Generation," in Knowledge-Driven Computing, vol. 102, C. Cotta, S. Reich, R. Schaefer, and A. Ligęza, Eds. Springer Berlin Heidelberg, 2008, pp. 235–254. doi: 10.1007/978-3-540-77475-4_15.

[20] R. Sagarna, A. Arcuri, and Y. Xin, "Estimation of distribution algorithms for testing object oriented software," in Evolutionary Computation, 2007. CEC 2007. IEEE Congress on, 2007, pp. 438–444. doi: 10.1109/cec.2007.4424504.

[21] R. Sagarna and J. A. Lozano, "Scatter Search in software testing, comparison and collaboration with Estimation of Distribution Algorithms," Eur. J. Oper. Res., vol. 169, no. 2, pp. 392–412, 2006, doi: http://dx.doi.org/10.1016/j.ejor.2004.08.006.

[22] N. Sadat Jalali, H. Izadkhah, and S. Lotfi, "Multi-objective search-based software modularization: structural and non-structural features," Soft Comput., vol. 23, no. 21, pp. 11141–11165, 2019, doi: 10.1007/s00500-018-3666-z.

[23] S. Baluja and S. Davies, "Fast probabilistic modeling for combinatorial optimization," in AAAI/IAAI, 1998, pp. 469–476.

[24] J. Staunton and J. Clark, "Applications of Model Reuse When Using Estimation of Distribution Algorithms to Test Concurrent Software," in Search Based Software Engineering, vol. 6956, M. Cohen and M. Ó Cinnéide, Eds. Springer Berlin Heidelberg, 2011, pp. 97–111. doi: 10.1007/978-3-642-23716-4_12.

[25] C. Jin and S.-W. Jin, "Software reliability prediction model based on support vector regression with improved estimation of distribution algorithms," Appl. Soft Comput., vol. 15, pp. 113–120, 2014, doi: http://dx.doi.org/10.1016/j.asoc.2013.10.016.

[26] M. Pelikan and H. Mühlenbein, "Marginal distributions in evolutionary algorithms," in Proceedings of the International Conference on Genetic Algorithms Mendel, 1998, vol. 98, pp. 90–95.

[27] M. Hauschild and M. Pelikan, "An introduction and survey of estimation of distribution algorithms," Swarm Evol. Comput., vol. 1, no. 3, pp. 111–128, 2011, doi: http://dx.doi.org/10.1016/j.swevo.2011.08.003.

[28] M. Pelikan and H. Mühlenbein, "The bivariate marginal distribution algorithm," in Advances in Soft Computing, Springer, 1999, pp. 521–535.

[29] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Combinatorial test generation for software product lines using minimum invalid tuples," in High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on, 2014, pp. 65–72.

[30] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Pairwise testing for software product lines: comparison of two approaches," Softw. Qual. J., vol. 20, no. 3–4, pp. 605–643, 2012, doi: 10.1007/s11219-011-9160-9.

[31] M. Z. Sahid, A. B. M. Sultan, A. A. A. Ghani, and S. Baharom, "Combinatorial Interaction Testing of Software Product Lines: A Mapping Study," J. Comput. Sci., vol. 12, no. 8, pp. 379–398, 2016, doi: http://dx.doi.org/10.3844/jcssp.2016.379.398.

[32] R. Santana, P. Larranaga, and J. A. Lozano, "Learning factorizations in estimation of distribution algorithms using affinity propagation," Evol. Comput., vol. 18, no. 4, pp. 515–546, 2010.