International Journal of Computer Applications (0975 – 8887)
Volume 160 – No 3, February 2017

A Univariate Marginal Approach for Pairwise Testing of Software Product Lines

Mohd Zanes Sahid
Universiti Tun Hussein Onn Malaysia (UTHM), Johor, Malaysia.

Abu Bakar Md Sultan
Universiti Putra Malaysia (UPM), Selangor, Malaysia

Abdul Azim Abdul Ghani
Universiti Putra Malaysia (UPM), Selangor, Malaysia

Salmi Baharom
Universiti Putra Malaysia (UPM), Selangor, Malaysia

ABSTRACT

Software Product Line (SPL) is a software engineering paradigm that is inspired by the concept of reusability of common features, formulated for different software products. Complete testing of all software products in SPL is known to be unfeasible. This is due to the very large number of possible products that can be produced or configured using a combination of features in the SPL. Pairwise Testing is a type of Combinatorial Testing, influenced by the perception that two factors (or features in the context of SPL testing) stimulate most faults. The effectiveness of SPL testing can be measured using the pairwise coverage of test configuration. However, to generate minimal test configuration that maximizes the pairwise coverage is not trivial, especially when dealing with a huge number of features and when constraints have to be satisfied, which is the case in most SPL scenarios. Therefore, it is the motivation of this work to investigate the feasibility of an Estimation of Distribution Algorithm (EDA), specifically the Univariate Marginal Distribution Algorithm (UMDA), in generating minimal test configuration for pairwise testing of SPL. The experimental results show that in certain problem instances, UMDA is able to compete with existing greedy and search-based algorithms.

General Terms

Algorithms, Software Product Lines Testing.

Keywords

product line testing, pairwise testing, combinatorial testing, univariate marginal distribution algorithm, estimation of distribution algorithm

1. INTRODUCTION

Software Product Line (SPL) is a software engineering paradigm that facilitates the development of software products that share common functionalities. The main goal of developing a system as a software product line is to create a software structure that is customizable to various needs, by maximizing software artefacts reusability [1]. Though belonging to a similar domain of usage, it is common for users to have distinctive requirements. 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 single software that fulfill the needs of diverse users of a similar domain.

In SPL, a unit of system function is represented as a feature. Features are 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],[2].

One of the critical activities in SPL is feature configuration; in which two or more features are combined and utilized together in a single software product. This could possibly result in unspecified and unintended system behavior and might lead to incorrect execution [3]. Hence, it is crucial to test all possible feature configurations in order 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 size of FM increased. Therefore, exhaustively testing all feature configurations especially in large-scale FM is not practical [4],[5].

A number of researchers had proposed a couple of prominent strategies to reduce the combinatorial explosion of feature configuration testing [6]. Based on our literature, most of the current approaches are based on greedy algorithms with only a few works leveraged the potential of search-based techniques, which have been widely used in the single software development testing. Additionally, it is common that software engineers develop an SPL with some concrete or predetermined software products as a subset of its final products [7]. Employment of conventional meta-heuristics techniques to generate minimized test configuration often requires these predefined valid software products as seeds or initial population. Probabilities are implicitly employed in the selection and re-production operators to produce offspring. In this sense, by explicitly building a probabilistic model of features distribution out of this seeds, it allows us to estimate the distribution of highly fit features in subsequent candidate solutions. This has remained uninvestigated. Therefore, this paper reports a first attempt to employ an Estimation of Distribution Algorithm (EDA) in the context of SPL testing. Based on the conducted empirical studies, we observed that our proposed solution is able to compete with other approaches in certain problem instances, hence suggested that an EDA is a viable approach towards better SPL testing.

This paper is organized as follows. Section II discusses the concept of feature modeling, feature configuration testing, SPL pairwise testing, Estimation of Distribution Algorithms and related works. In Section III, we present the proposed solution for SPL pairwise testing based on Univariate Marginal Distribution Algorithm (UMDA), illustration of the strategy, and the solution strategy. Section IV reports and discusses the empirical study. Finally, the Section V concludes this paper and highlights our future works.

2. BACKGROUND AND RELATED WORKS

2.1 Feature Modeling

Feature Modeling is a variability modeling technique that is commonly employed to produce Feature Model (FM) [1]. FM is a notation that represents features and its dependencies. The tree representation of FM is known as Feature Diagram. It presents a feature as a node, and relationships between features as edges. Basically, different types of edges can be assigned between features, which represent a relationship of type mandatory, optional, or, or alternative. Apart from that, FM might encompass some constraints as a rule or condition that limits the linking between features. Feature modeling is a popular way to model SPL variability and it is by far the most reported in industry. Figure 1 depicts a feature model for a simple ECommerce SPL [8].

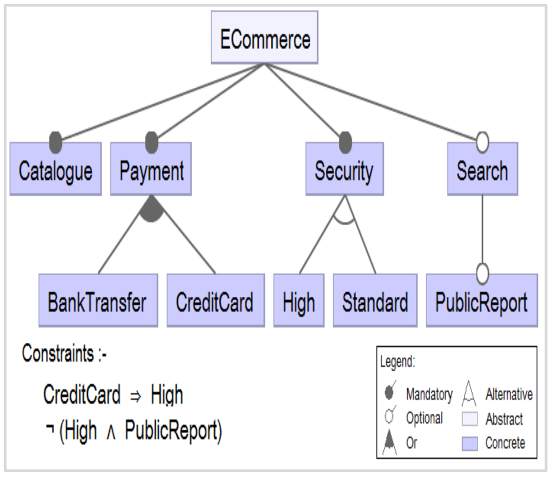

Fig 1: A Feature Model of an Ecommerce SPL

FM is created as the formalism to describe features, relationships and its constraints. The presence of constraint is unavoidable as it determines the usability and practicality of an SPL. FM can be translated into Propositional Logic. Formal methods can be used to analyze its structure. Boolean logic (AND, NOT, OR) can be used to represent relationships and constraints. A clause can be constructed to represent Conjunction/Disjunction of one or more feature. By transforming the FM into Propositional Logic, Boolean Satisfiability (SAT) solving techniques can be used to check the compliance of a given clause with respect to a particular FM.

2.2 Feature Configuration Testing

Products are configured and produced by combining several features. These artefacts are called feature configurations. In view of testing, the 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.

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. (why 2, because it is either selected or excluded). Consider an example given in Figure 1. A total of 1023 possible test configurations can be generated, conceiving that a test configuration requires at least one feature. Given the representation in Figure 2, we can construct a complete list of test configuration as shown partially in Table 1.

1→ECommerce, 2→Catalogue, 3→Payment,
4→Security, 5→Search, 6→Bank_Transfer,
7→Credit_Card, 8→High, 9→Standard,
10→Public_Report

Fig 2: Number assignment of each feature

Table 1. Partial list of all possible test configurations

Test Configuration	Feature									
	1	2	3	4	5	6	7	8	9	10
TC1	√	√	√	√	√	√	√	√	√	√
TC2	√	√	√	√	√	√	√	√	√	-
...										
TC500	√	-	-	-	-	-	√	√	-	-
…										
TC1023	-	-	-	-	-	-	-	-	-	√

Only partial are shown due to space constraint

2.3 SPL Pairwise Testing

The motivation of pairwise testing is to cover all possible pair of features, thus testing can be focused on the interaction of both features. To exhaustively cover all pairs of n number of features (from n choose r, where r=2), it can be calculated using the following combinatorial formula:

$$C(n,r) = \frac{n!}{r!(n-r)!} \qquad (1)$$

For each pair, each variable can take value of selected or unselected, thus the total number of pairs is:

$$C(n,r) * 2^2 \qquad (2)$$

Pairwise testing is a type of combinatorial testing, where we choose 2 elements to be considered or included in our test pool. It can be generalized to another type of combinatorial testing called as *t-wise* testing, where t indicates the number of elements to choose.

As with other context of pairwise testing, SPL pairwise testing is governed by constraints. Considering two features (1 and 2) from Ecommerce SPL, four pairs of tuple will have to be generated, i.e.(1,2), (1,-2), (-1,2) and (-1,-2), where negative sign 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. feature -1 is invalid because root feature must always be selected. The same goes with mandatory features (2, 3 and 4).

If we construct one test configuration, tc^i, for each pair of features, pf^i, (as an example, pair of feature 1 and -5), defined as follows;

$pf^a = (1, -5)$
$tc^a = \{1, ?, ?, ?, -5, ?, ?, ?, ?, ?\}$

we can set any arbitrary value for other variables (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 valid pairs in each tc so that it can minimize the number of required TC.

2.4 Estimation of Distribution Algorithms

Estimation of Distribution Algorithms (EDAs) is a class of Evolutionary Algorithms (EAs) that explore the space of potential solutions following the principle of survival of the fittest of individual and populations similar to Genetic Algorithm (GA) [9],[10]. 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. The Probability Distribution is calculated 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. In short, EDA-based algorithms utilize statistics and probability to create the next generation of individuals.

Generally, there are three categories of EDA according to the complexity of the statistical analysis they employ, (1) Univariate EDA, (2) Bivariate EDA, and (3) Multivariate EDA. Each category, respectively, assumes solution variables to have no dependency, pairwise dependencies, or multiple dependencies between variables [10].

The Univariate Marginal Distribution Algorithm (UMDA) belongs to the first category of EDA, and it is one of the most basic Univariate EDA. UMDA begins with a set of candidate solution, eliminate those less fit solutions, and then, calculate probability of each candidate solutions' elements. The result will be used to populate new candidate solutions for next generation [11]. One of the main advantages of Univariate EDA is it perform much simpler computation and require smaller memory footprint as compared to bivariate and multivariate EDAs [12].

2.5 Related Works

In this section, the paper discusses some of the notable works that are related to combinatorial testing of SPL, and application of EDAs in search-based software engineering.

Various approaches have been published towards a viable feature configuration testing of SPL systems. The multitudes of studies were mainly built around the combinatorial interaction testing approach, followed by its integration with other optimization approach such as search-based approach.

Recently, Alsariera et al. [13] proposed SPLs test reduction using Bat-inspired algorithm. Their motivation was to minimize the tests suite by formulating the test suite selection as a swarm of bats hunting for prey. Henard et al. [5] 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 current *t-wise* approaches for SPLs are restricted to a small number of FMs and low strength of *t-wise* coverage. Both are constrained by scalability issues that results from intractable computation for very large FMs or high *t* values. Therefore, they formulated the feature configuration generation problem as a search-based where the search space is defined as all the valid feature configurations extracted from the FM. Thus, meta-heuristic techniques can be used to systematically explore this space. In view of this, dissimilarity between features are used as a fitness function towards searching for populations of feature configurations in this space.

Wang et al. [14] use a weighted Genetic Algorithm to minimize SPL test suites, and at the same time maintain fault detecting power. Haslinger et al. [15] 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 speed up of over 60% on 133 publicly available feature models, while preserving the coverage of the generated tests.

Ensan et al. [16] highlighted that comprehensive testing of all potential products results in exponential test suites in test space. They proposed to use a Genetic Algorithm with SAT solver to search for SPL feature interactions. They managed to automatically generate and find appropriate and minimal test suites for a given SPL while maintaining practical resource utilization and achieving acceptable fault detection capability. However, the limitation of their approach is it does not manage to scale for FM with over 300 features.

Johansen et al. published their solution [4] and a tool named ICPL, which capable of processing large feature models, better execution time and most importantly 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.

It is deemed necessary to mention works related to EDA, as our solution is partly based on EDA. EDAs have been adopted to solve optimization problems in software engineering. We noticed that EDAs have been utilized in optimizing test data generation and test suites generation [17],[18],[19] and fault detection [20]. EDA has also been employed to improve software reliability prediction [21].

3. UMDA FOR SPL PAIRWISE TESING

This paper reports the first attempt to employ UMDA to generate a minimal set of SPL test configuration that satisfies pairwise coverage of features. We define our fitness function as the number of pairs of features covered by each test configuration. Hence, the more pairs are covered, the better the fitness. The algorithm of the original UMDA is shown in Figure 3.

```
1.    Initialize a population of candidate solutions $\{x_i\}, i \in [1, N]$
2.    Note that each $x_i$ includes $n$ bits $x_i(1), \ldots, x_i(n)$
3.    While not (termination criterion)
4.      Select M fittest individuals from $\{x_i\}$, where $M < N$
5.      Index the M selected individuals as $\{x_i\}, i \in [1, M]$
6.      $\Pr(x(k) = 1) \leftarrow \sum_{i=1}^{M} \delta(x_i(k) - 1)/M$, for $k \in [1, n]$
7.      For $i = 1$ to $N$ (population size)
8.        For $k = 1$ to $n$
9.          $r \leftarrow U[0,1]$
10.         If $r < \Pr(x(k) = 1)$
11.             $x_i(k) \leftarrow 1$
12.         else
13.             $x_i(k) \leftarrow 0$
14.         End if
15.       Next bit
16.     Next individual
17.   Next generation
```

Fig 3: Algorithm for UMDA [11]

The intuition behind this work is, during the search for fitter test configurations, the more frequent a particular feature present in the current fittest test configuration, the more frequent it should be included in the subsequent list of test configuration. For example, if feature 5 appears more frequent in our 10 best test configurations, we should create more test configurations with feature 5 instead of -5 for 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.

3.1 Illustrating Test Configuration Generation using UMDA

To demonstrate the strategy, the following illustrations are presented for a single iteration of test configuration generation based on the Probability Vector (PV) calculated from the first randomly generated candidate test configurations. Assuming we have the following set of valid pairs of features:

Pair of features, *pf* = { (1,2), (1,3), (1,-3), (1,4), (1,-4), (1,5), (1,-5), (2,3), (2, 3), (2,4), (2,-4), (2,5), (2, 5), (2,6), (3,4), (3,-4), (3,5), (3,-5), (3,6), (4,7) }

First, we start with randomly generated five valid set of test configurations as our first population of candidate solutions:

```
Population 1:
  TC1    {  1,  2,  3, -4, -5,  6, -7 }
  TC2    {  1,  2, -3,  4,  5, -6,  7 }
  TC3    {  1,  2,  3,  4,  5, -6, -7 }
  TC4    {  1,  2, -3, -4, -5, -6, -7 }
  TC5    {  1,  2,  3, -4, -5,  6,  7 }
```

Next, for each test configuration, we calculate its fitness value. We count how many pairs from *pf* are matched with the respective pair of features in each test configuration, which results in the following fitness values:

```
Population 1:
  TC1    {  1,  2,  3, -4, -5,  6, -7 }      → fitness=11
  TC2    {  1,  2, -3,  4,  5, -6,  7 }      → fitness=8
  TC3    {  1,  2,  3,  4,  5, -6, -7 }      → fitness=9
  TC4    {  1,  2, -3, -4, -5, -6, -7 }      → fitness=7
  TC5    {  1,  2,  3, -4, -5,  6,  7 }      → fitness=11
```

Truncation selection operator from GA is borrowed and applied here. It is a parent selection mechanism one can use to select potential candidate solutions for reproduction [22]. For this illustration, we truncate the population by choosing three most fit individuals (test configurations), i.e.

```
Truncated Population 1:
  TC1    {  1,  2,  3, -4, -5,  6, -7 }      → fitness=11
  TC3    {  1,  2,  3,  4,  5, -6, -7 }      → fitness=9
  TC5    {  1,  2,  3, -4, -5,  6,  7 }      → fitness=11
```

Next, we calculate the PV by finding the percentage of each positive numbered feature out of all values by its position across all selected test configurations. For example, for position 7, the value from TC1, TC3 and TC5 are -7, -7 and 7, respectively. Hence, the probability for positive number of feature 7 is 1/3, i.e. ≈0.33. The PV for first population is calculated as follows:

PV = {1.0, 1.0, 1.0, 0.33, 0.33, 0.67, 0.33}

Subsequently , the second population of five candidate solutions is generated using the estimated distribution calculated in PV. Populate each position with the respective value using probability PV(*i*) where *i* refers to the position of probability value in PV. For example, for position 6, we should have value 6 appear ≈67% times in our five test configurations. A possible distribution of features in position 6 is -6, 6, 6, -6 and 6. A possible set of test configurations is generated as follows:

```
Population 2:
  TC1    {  1,  2,  3, -4,  5, -6, -7 }
  TC2    {  1,  2,  3, -4,  5,  6,  7 }
  TC3    {  1,  2,  3, -4, -5,  6, -7 }
  TC4    {  1,  2,  3,  4, -5, -6, -7 }
  TC5    {  1,  2,  3,  4, -5,  6,  7 }
```

Evaluate our second population by calculating the fitness value of each test configurations:

```
Population 2:
  TC1    {  1,  2,  3, -4,  5, -6, -7 }      → fitness=9
  TC2    {  1,  2,  3, -4,  5,  6,  7 }      → fitness=11
  TC3    {  1,  2,  3, -4, -5,  6, -7 }      → fitness=11
  TC4    {  1,  2,  3,  4, -5, -6, -7 }      → fitness=9
  TC5    {  1,  2,  3,  4, -5,  6,  7 }      → fitness=12
```

In general, it is shown that the second population consists of better (fitter) test configurations as compared to the first population. Additionally, in the second population, one test configuration (TC5) has fitness value 12, greater than any previous test configuration. It means that, we managed to find a test configuration that covers more pairs than others.

This strategy can be repeated until we find no more improvement. Then, we select the best test configuration and store it in our hall-of-fame, remove pairs that are already covered and start again with new random candidate solutions. This process continues until all pairs have been covered, or when generation limit is achieved. In the case of generation limit is achieved, we consider that we only managed to cover partial pairwise test configuration.

3.2 The SPL-Pairwise-UDMA Strategy

Few adjustments have been incorporated to UMDA in our proposed strategy. Firstly, we changed the initial population generation approach. Instead of generating random test configuration naively, we employ Boolean Satisfiability (SAT) solver to generate random test configurations in a controlled manner. This controlled generation is crucial to ensure that only valid test configurations are populated.

Secondly, as we iterate on a number of generations, the fitness of test configurations in newer generations may be getting stagnant or no improvement. When this happened, we put aside the fittest test configuration discovered so far in a space called as hall-of-fame, and we replace the current generation with new generation using SAT solver. Then we recalculate the PV based on the new generation, without considering the historical distribution. This is repeated until all valid pairs are covered, or when iteration limit is achieved.

Thirdly, for every new candidate solutions generated using the PV, we omit all candidate solutions that are invalid. This is done by checking the clause satisfiability using SAT solver. This is significant to ensure that the constraints of the FM are satisfied, and eventually, we only have valid test configurations. The whole strategy for test configuration generation using UMDA is formulated as an algorithm presented in Figure 4.

```
1.  Initialize a population of candidate solutions $\{x_i\}, i \in [1, N]$
2.  Note that each $x_i$ includes $n$ bits $x_i(1), \ldots, x_i(n)$
3.  Generate set of feature pairs to be covered, $pf$
4.  Set maximum generation, G, generation counter, $g$
5.  While not (termination criterion)
6.     Select M fittest individuals from $\{x_i\}$, where M < N
7.     Index the M selected individuals as $\{x_i\}, i \in [1, M]$
8.     Set fittest individual as generation winner, $w$
9.     Break loop if $|pf| = 0$ or $g = G$
10.    $\Pr(x(k) = 1) \leftarrow \sum_{i=1}^{M} \delta(x_i(k) - 1)/M$, for $k \in [1, n]$
11.    Insert Pr in tracking queue, $\{QPr_z\}, z \in [1, 3]$
12.    If $|QPr_a - QPr_b| = 0$, where $a \mathrel{!=} b$ and $a, b \in z$
13.       Insert $w$ in hall-of-fame, $\{hf_j\}, j > 0$
14.       Reset the population of candidate solutions $\{x_i\}, i \in [1, N]$
15.       Reset Pr and $g$
16.    Else
17.       For $i = 1$ to $N$ (population size)
18.          For $k = 1$ to $n$
19.             $r \leftarrow U[0,1]$
20.             If $r < \Pr(x(k) = 1)$
21.                $x_i(k) \leftarrow 1$
22.             Else
23.                $x_i(k) \leftarrow 0$
24.             End if
25.          Next bit
26.       Next individual
27.    End If
28. Next generation
29. List in $hf$ is the solution
```

Fig 4: Algorithm of the proposed strategy

4. EMPIRICAL STUDY

The algorithm has been implemented using Java, and we have performed experiments to evaluate its effectiveness.

4.1 Objectives

An empirical study has been carried out in order to gauge the effectiveness of applying a univariate estimation distribution algorithm in generating test configuration fulfilling pairwise coverage. Two measures of effectiveness have been evaluated, i.e.:

i. Minimum number of test configurations, *eff_min(TC)*

ii. Maximum pairwise coverage, *eff_max(PC)*

4.2 Experiment Operations

The experiments have been performed on 9 datasets of various number of features and constraints, obtained from SPLOT [8]. The parameters configured for the experiment are 200 population size with truncation size 5, stagnancy count is 3, maximum iterations is 5000 and timeout is 1800 seconds. Stagnancy count is the number of consecutive iterations having the fittest value unchanged. This is used to signal that the population is stagnant. Comparisons were made against ICPL [4] and (1+1) ES-based tool [5], for measuring the *eff_min(TC)* and *eff_max(PC)* respectively.

4.3 Results and Discussions

The first experiment was conducted to measure the effectiveness of the proposed strategy in minimizing the number of generated test configurations. Results have been recorded and presented in Table 2. Pairwise coverage was set to 100% coverage. Cells in the third and fourth columns that are greyed indicate smaller number of test configuration, hence better solutions. Our UDMA-based solution managed to generate a smaller number of test configurations on majority datasets (small size SPL, having feature count less than 100). However, we cannot compete with ICPL for the largest dataset (eShop), with a significant difference.

Table 2. Number of test configurations

Feature Models	Number of Features	Number of Test Config ***eff_min(TC)*** UMDA-SPL	ICPL
Ecommerce	10	**6**	7
Cellphone	11	**7**	8
GPL	20	**15**	17
SPL-SimulES	32	10	10
Arcade Game PL	61	**16**	18
J2EE-Web-Arch	77	20	**18**
Billing	88	**13**	14
Coche ecologico	94	**91**	93
eShop	287	39	**24**

The second experiment measures the effectiveness in maximizing the number of pairwise coverage. The result is presented in Table 3. The execution of (1+1) ES-based tool was driven by the number of required test configuration (product configuration), thereby, we execute that tool based on the number of test configuration returned by our UMDA-based solution, and observed its pairwise coverage. Our UDMA-based solution was always better in maximizing the number of pairwise coverage (always 100%), compared to the (1+1) ES-based tool. However, the effectiveness gain is too small. Therefore, it is not really fair to compare both in terms of pairwise coverage. Additionally, the (1+1) ES-based tool is claimed to perform prioritization, apart from test configuration generation. Hence, further investigation may have to be performed to compare both in terms of its prioritization effectiveness, and probably efficiency (including, but not limited to execution time and number of fitness function calculations).

Table 3. Number of pairwise coverage

Feature Models	*Number of Pairwise Coverage* ***eff_max(PC)*** UMDA-SPL (100%)	(1+1) - ES	
Ecommerce	106	101	95.28%
Cellphone	151	149	98.68%
GPL	499	489	98.00%
SPL-SimulES	1448	1434	99.03%
Arcade Game PL	5209	5188	99.60%
J2EE-Web-Arch	9837	9820	99.83%
Billing	8725	8715	99.89%
Coche ecologico	11075	11035	99.64%
eShop	147534	147449	99.94%

4.4 Threats to Validity

This work is intended to demonstrate the feasibility and effectiveness of UMDA in SPL pairwise testing, and experiment was conducted on small and moderate-sized SPLs. As we aware of its generalizability concern, we put aside its scalability aspect at this moment, which we noticed in order to achieve high scalability, it requires a more sophisticated program structure and execution platform (such as parallel execution during fitness function calculation [23]).

Evaluation of our proposed strategy was only conducted against two existing tools, in which one might raise concern on the significance of the results obtained. We believe it is sufficient as the two tools are chosen due to their outstanding performance as reported in several literatures.

5. CONCLUSION AND FUTURE WORKS

This first attempt to employ (univariate) EDA shows that the first variant of EDA is able to compete with a greedy approach for small SPLs, however, less convincing for large SPL. In view of this, we planned to conduct further works based on (i) other variant of first class EDA, such as Compact Genetic Algorithm (cGA), and Population-Based Incremental Learning (PBIL) algorithm, and, (ii) SPL having a higher number of features to substantiate these findings.

The strategy presented in this paper exploits the simple probability distribution calculated on each variable, individually. However, considering a pair of features as the element of the fitness function, we observe that there exists some dependency between features in our problem space. Hence, a more rigor statistical analysis (bivariate or multivariate EDAs) is deemed to be more appropriate. Thus, our other future works will be carried out to employ Bivariate Marginal Distribution Algorithm (BMDA) to find dependency between two features, and exploit this information to improve the effectiveness of test configuration generation in SPL.

6. REFERENCES

[1] K. Lee, K. Kang, and J. Lee, "Concepts and Guidelines of Feature Modeling for Product Line Software Engineering," in *Software Reuse: Methods, Techniques, and Tools*. vol. 2319, C. Gacek, Ed., ed: Springer Berlin Heidelberg, 2002, pp. 62-77.

[2] M. M. Alam, A. I. Khan, and A. Zafar, "A Comprehensive Study of Software Product Line Frameworks," *International Journal of Computer Applications,* vol. 151, pp. 11-17, 2016.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake, "Feature Interactions," in *Feature-Oriented Software Product Lines: Concepts and Implementation*, ed Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 213-241.

[4] 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.

[5] C. Henard, *et al.*, "Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines," *Software Engineering, IEEE Transactions on,* vol. 40, pp. 650-670, 2014.

[6] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed, "A first systematic mapping study on combinatorial interaction testing for software product lines," in *Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1-10.

[7] S. Oster, F. Markert, and P. Ritter, "Automated Incremental Pairwise Testing of Software Product Lines," in *Software Product Lines: Going Beyond*. vol. 6287, J. Bosch and J. Lee, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 196-210.

[8] 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.

[9] M. Pelikan, M. Hauschild, and F. Lobo, "Estimation of Distribution Algorithms," in *Springer Handbook of Computational Intelligence*, J. Kacprzyk and W. Pedrycz, Eds., ed: Springer Berlin Heidelberg, 2015, pp. 899-928.

[10] 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., ed: Springer Berlin Heidelberg, 2012, pp. 21-37.

[11] D. Simon, "Estimation of Distribution Algorithms," in *Evolutionary Optimization Algorithms*, First ed: John Wiley & Sons, 2013, pp. 313-34 7.

[12] M. Hauschild and M. Pelikan, "An introduction and survey of estimation of distribution algorithms," *Swarm and Evolutionary Computation,* vol. 1, pp. 111-128, 2011.

[13] Y. A. Alsariera, M. A. Majid, and K. Z. Zamli, "SPLBA: An interaction strategy for testing software product lines using the Bat-inspired algorithm," in *Software Engineering and Computer Systems (ICSECS), 2015 4th International Conference on*, 2015, pp. 148-153.

[14] S. Wang, S. Ali, and A. Gotlieb, "Minimizing test suites in software product lines using weight-based genetic algorithms," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, Amsterdam, The Netherlands, 2013, pp. 1493-1500.

[15] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, "Improving CASA runtime performance by exploiting basic feature model analysis," *arXiv preprint arXiv:1311.7313,* 2013.

[16] F. Ensan, E. Bagheri, and D. Gašević, "Evolutionary Search-Based Test Generation for Software Product Line Feature Models," in *Advanced Information Systems Engineering*. vol. 7328, J. Ralyté, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 613-628.

[17] 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, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2008, pp. 235-254.

[18] 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.

[19] R. Sagarna and J. A. Lozano, "Scatter Search in software testing, comparison and collaboration with Estimation of Distribution Algorithms," *European Journal of Operational Research,* vol. 169, pp. 392-412, 2006.

[20] 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., ed: Springer Berlin Heidelberg, 2011, pp. 97-111.

[21] C. Jin and S.-W. Jin, "Software reliability prediction model based on support vector regression with improved estimation of distribution algorithms," *Applied Soft Computing,* vol. 15, pp. 113-120, 2014.

[22] H. Mühlenbein and D. Schlierkamp-Voosen, "Predictive Models for the Breeder Genetic Algorithm I. Continuous Parameter Optimization," *Evolutionary Computation,* vol. 1, pp. 25-49, 1993.

[23] R.-Z. Qi, Z.-J. Wang, and S.-Y. Li, "A Parallel Genetic Algorithm Based on Spark for Pairwise Test Suite Generation," *Journal of Computer Science and Technology,* vol. 31, pp. 417-427, 2016.