Similarity-Based Prioritization in Software Product-Line Testing

Mustafa Al-Hajjaji
University of Magdeburg
Germany

Thomas Thüm
University of Magdeburg
Germany

Jens Meinicke
University of Magdeburg
Germany

Malte Lochau
TU Darmstadt
Germany

Gunter Saake
University of Magdeburg
Germany

ABSTRACT

Exhaustively testing every product of a software product line (SPL) is a difficult task due to the combinatorial explosion of the number of products. Combinatorial interaction testing is a technique to reduce the number of products under test. However, it is typically up-to the tester in which order these products are tested. We propose a similarity-based prioritization to be applied on these products before they are generated. The proposed approach does not guarantee to find more errors than sampling approaches, but it aims at increasing interaction coverage of an SPL under test as fast as possible over time. This is especially beneficial since usually the time budget for testing is limited. We implemented similarity-based prioritization in FeatureIDE and evaluated it by comparing its outcome to the default outcome of three sampling algorithms as well as to random orders. The experiment results indicate that the order with similarity-based prioritization is better than random orders and often better than the default order of existing sampling algorithms.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Testing—*Reusable Software*; D.2.13 [**Software Engineering**]: Reusable Software—*domain engineering*

General Terms

Reliability, Verification

Keywords

Software product lines, product-line testing, combinatorial interaction testing, prioritization

SPLC '14, September 15 - 19 2014, Florence, Italy
Copyright 2014 ACM 978-1-4503-2740-4/14/09 ...$15.00.
http://dx.doi.org/10.1145/2648511.2648532

1. INTRODUCTION

A software product line (SPL) is a set of products designed to make advantage of their common features [35]. These features meet specific needs of a given domain. Products are derived from a set of previously developed core assets. In addition to the common features, there are explicit variations which differentiate between products [40]. The adoption of SPLs in industry decreases implementation costs, reduces time to market, and improves the quality of derived products [27,29]. Thus, many software organizations change their software development from single systems to SPLs [41].

Evaluating the reliability of an SPL is important, because some of its features are constantly reused. Hence, testing becomes essential to avoid fault propagation to the derived products. Tevanlinna et al. [34] describe different strategies for SPL testing: *Product by product* (*a.k.a. product-based testing* [35]) is a traditional strategy that tests each product separately without reusing the test assets. The products are generated and tested individually, each using a standard testing technique. *Incremental testing* is a strategy that exploits the commonalities of products. In this strategy, only the first product is tested individually, while the following products are partially retested with regression techniques. In the *reusable asset instantiation* strategy, the test assets are created based on domain engineering and reused to test each product of an SPL.

Testing an SPL is a difficult task due to the high number of possible combination features which often leads to a combinatorial explosion of possible products that need to be tested. To overcome this problem, combinatorial interaction testing [8] has been proposed to reduce the number of products to test. Combinatorial interaction testing is based on the observation that most of the defects are expected to be caused by an interaction of few features [21]. Several approaches use t-wise testing to achieve the combination interaction between features [7, 31]. Several algorithms have been proposed to perform t-wise interactions such as AETG [8], CASA [12], Chvatal [5], ICPL [17,18], IPOG [25], and MoSo-PoLiTe [30]. The result of each algorithm is a set of products that need to be tested, which is a subset of all products.

The tester is responsible in which order these products are tested. Several approaches [2, 9, 15, 16, 19] have been proposed to prioritize the products based on different criteria. In this paper, we propose similarity-based prioritization

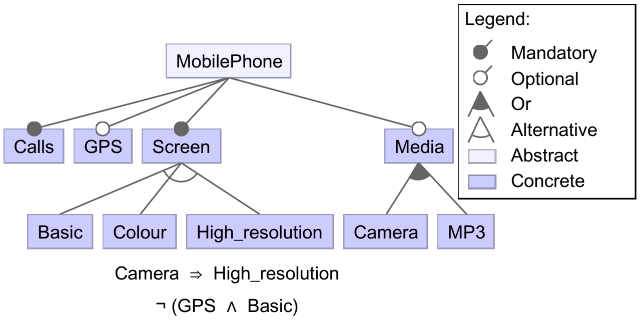


Figure 1: Feature diagram of SPL *Mobile Phone*

to prioritize the products. The goal is to increase the interaction coverage of the SPL under test as fast as possible over time. With our algorithm, we hope to reduce the time required to find defects, and thus to reduce the overall testing effort. In our approach, we select the product with maximum number of features to be the first one to test (a.k.a. *allyesconfig* [10]). We select the following product that has the minimum similarity with the previous product. Then, we select the next product that has the minimum similarity with the previous two products. The process continues until all products are ordered. The general idea of similarity-based testing is not new, Henard et al. combine the sampling with prioritization [15], but we adopt it by calculating the similarity between the products differently. In addition, the input of our approach can be all the valid configurations if the size of SPL is small, a set of configurations has been sampled using sampling algorithms, or a set of configurations has been given by domain experts. Our contributions are as follows:

- We propose similarity-based prioritization to order the products before they are generated and tested.
- We extend FeatureIDE [37] with support for product-line testing and similarity-based prioritization.
- We evaluate the effectiveness of similarity-based prioritization compared to the default order of three sampling algorithms, namely ICPL, CASA, and Chvatal and to random orders.

This paper is organized as follows. In Section 2, we give a brief introduction to feature models. We present the approach of similarity-based prioritization in Section 3. In Section 4, we describe our extension of FeatureIDE to support SPL testing. In Section 5, we evaluate our approach and discuss the results. In Section 6, we discuss the related work. Finally, we present our conclusion and future work in Section 7.

2. FEATURE MODELING

Not all combinations of features in an SPL are considered valid. Such a valid combination is called configuration [20]. Each configuration can be used to generate a product. A feature model is used to define the valid combinations of features. A feature diagram is the graphical representation of a feature model [20].

There are several types of relations between the features and the subfeatures as illustrated in *Figure 1*. These connections are distinguished as: *and*-group, *or*-group, and *alternative*-group [3]. In the *and*-group, children can be *optional*, such as *GPS* and *Media*, or *mandatory*, such as *Calls* and *Screen*. The selection of a feature implies the selection of its parent feature and its mandatory subfeatures of an *and*-group. In an *or*-group, at least one subfeature has to be selected, when its parent is selected. In an *alternative*-group, exactly one subfeature must be selected [37], when its parent is selected. For instance, a mobile phone can include only one of the following features: *Basic*, *Colour*, or *High_resolution*. The features are either *abstract* or *concrete:* if implementation artifacts are mapped to a feature, it is *concrete*; otherwise, it is *abstract* [38].

Furthermore, features can have additional dependencies that cannot be described using only the hierarchical structure; cross-tree constraints are used to define such dependencies. Cross-tree constraints are propositional formulas usually shown below the feature diagram [1]. As illustrated in *Figure 1*, the feature model of the SPL *Mobile Phone* contains cross-tree constraints *requires* and *excludes*. An example of a *requires* constraint is, if feature *camera* is selected to be included in a mobile phone, feature *High_resolution* must be selected. An example of an *excludes* constraint is that a mobile phone cannot support the features *GPS* and *Basic* at the same time.

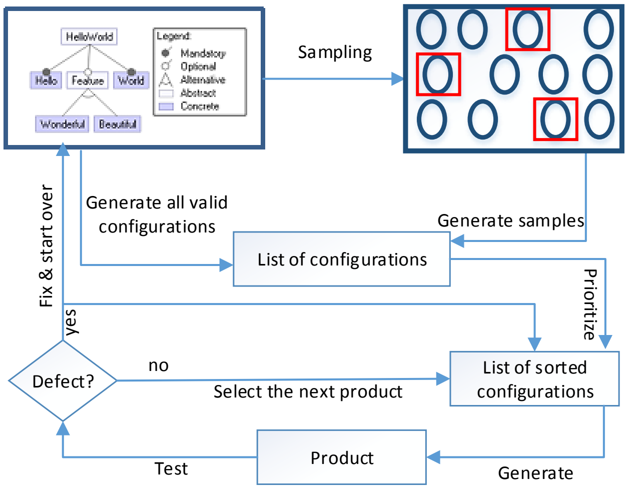


Figure 2: Similarity-based prioritization approach

3. SIMILARITY-BASED PRIORITIZATION

Similarity-based prioritization gets a set of configurations as input and orders them based on their similarity. The input of similarity-based prioritization can be all valid configurations if the size of SPL is small. For larger SPLs, the input is a set of configurations created from a sampling algorithm, given by a domain expert, or the productively used configurations. The output of the algorithm is a prioritized list of configurations.

Similarity-based prioritization select one configuration at a time to be generated and tested. As illustrated in *Figure 2*, if defects are found in the first generated product, the user can test the next configuration, or fix the defects and start the process again. If no defects are found in that product, the second configuration will be selected and generated.

	c_1	c_2	c_3	c_4
c_1	0	0.125	0.250	0.625
c_2	0.125	0	0.375	0.500
c_3	0.250	0.375	0	0.375
c_4	0.625	0.500	0.375	0

Table 1: Distances between example configurations

Selecting the following configuration will be based on the similarity between the configurations. A configuration is selected that has the minimum similarity with all previously tested configurations is selected. The rationale of this strategy is that similar products are likely to contain the same defects. This assumption based on the observation that dissimilar test cases are likely to detect more defects than similar ones [14].

We use Hamming distance [13] to evaluate the degree of similarity between configurations. The values of distance between configurations are between 0 and 1. If the value is 0, it indicates that the configurations are identical (similar). If the value is 1, it indicates that the configurations are completely different from each other. We define the distance of two given configurations c_i and c_j relative to the set of features F

$$d(c_i, c_j, F) = 1 - \frac{|c_i \cap c_j| + |(F \backslash c_i) \cap (F \backslash c_j)|}{|F|} \tag{1}$$

The general idea of calculating the distance between configurations is not new. However, we compute the distance between configurations different than Henard et al. [15]. They do not consider the deselected features when they calculate the distance. The rationale of taking the deselected features into account is that some defects are caused because some of features are not selected. In the following example, we show how we calculate the distance between the configurations. Assume, we have four configurations created from a combination of eight features:

$c_1 = \{f_2, f_8\}$.
$c_2 = \{f_1, f_2, f_8\}$.
$c_3 = \{f_2, f_5, f_7, f_8\}$.
$c_4 = \{f_1, f_2, f_4, f_5, f_6, f_7, f_8\}$.

We list all distances between configurations in *Table* 1. The distance between configurations c_1 and c_2 is calculated as follows.

$$d(c_1,\ c_2,\ F) = 1 - \frac{2+5}{8} = 0.125 \tag{2}$$

We select at each step the configuration which is the most dissimilar to the already selected configurations. To achieve this, as illustrated in *Algorithm 1*, we have a set C of configurations from a feature model and a list S which is initially empty. In list S, the ordered configurations are stored. The first step is to select a configuration to calculate its distance to the other configurations. We select the configuration with the maximum number of selected features because it covers most defects in individual features and enables selection of the next configuration with large distance. This strategy is common in the Linux community to test the configuration with the maximum number of selected features (a.k.a. *allyesconfig*) [10]. We add this configuration to S, and remove it from C. All other configurations in C are selected based on the similarity with the configurations in S. The configuration with maximum distance to the first selected configuration (*allyesconfig*) will be added to S and removed from C. The process continues until all configurations in C are added to S.

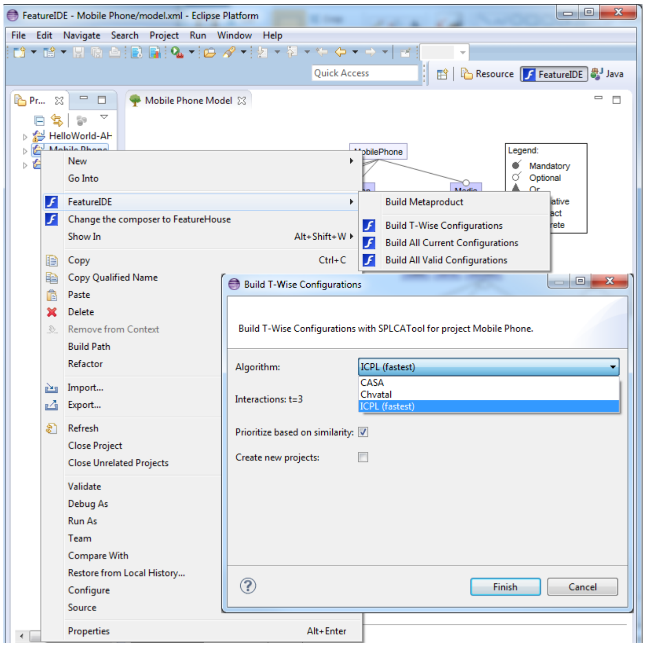
Algorithm 1

```
1: Input: $C$={$c_1, c_2, c_3, ....., c_n$} (set of configurations),
2:        $F$ ( set of features)
3: Output: $S$ (list of sorted configurations)
4: S ← [ ]
5: Select $c_{max} \in C$ with $\exists\ c_i \in C$,$|c_i| > |c_{max}|$)
6: $S.add(c_i)$
7: $C.remove(c_i)$
8: while $C$ not empty do
9:        Select$c_i \in C$ where
          $min(d(c_i, s_j, F)) = \max_{c_i \in C}(\min_{s_j \in S}(d(c_i, s_j, F)))$
10:          $S.add(c_i)$
11:          $C \leftarrow \quad C \backslash \{c_i\}$
12: end while
13: return S
```

With reference to the previous example, c_4 is the first configuration selected, because it contains the maximum number of features. Next, configuration c_1 is selected, because it has the maximum distance with configuration c_4 (0.625). Configuration c_1 is added to list S and removed from set C. Then, we have two configurations in each (i.e, list $S = (c_1, c_4)$, set $C = \{c_2, c_3\}$). We calculate the minimum distance for each configuration $c_i \in C$ with all configurations in S, and we select c_i with the maximum distance. The distances between configuration c_2 and (c_1, c_4) are (0.125, 0.5), and the distances between configuration c_3 and (c_1, c_4) are (0.25, 0.375), respectively. The minimum distance is 0.125 for c_2 and 0.25 for c_3, and thus the next configuration is c_3. The last selected configuration is c_2. The new order of the configurations is: c_4, c_1, c_3, c_2. In case two or more configurations have the same value of the maximum of minimum distance, we select the first of these configurations that got this value of distance.

We select the maximum of minimum distances between each configuration in C and all the selected configurations in S to make sure that the next selected configuration is the less similar to *all* already selected configurations. We did not choose the sum of distances between the configurations as Henard et al. [15], because it is misleading in some cases. For instance, with reference to *Table 1*, the summations of distances for configurations c_2 and c_3 with other configurations are 0.625 and 0.625, respectively. We expect that the number of defects will be detected by c_3 are more than the defects by c_2, because c_2 is similar to c_1 (distance is 0.125), and c_1 already detect most of the defects that can be detected by c_2.

We evaluate *Algorithm 1* by describing the amount of time it takes to finish the process. To calculate the distance between the configurations ($d(c_i, s_j, F)$), the features need to be checked whether they are selected in both configurations. The same features need to be checked in both configurations whether they are not selected (see *Equation* 1). The equation is executed in $\mathcal{O}(|F|)$. In *Line 9*, for each configu-



Build T-Wise Configurations with SPLCATool for project Mobile Phone.

Algorithm: ICPL (fastest)

Interactions: t=3

Prioritize based on similarity: ☑

Create new projects: □

Figure 3: Sampling and prioritization with FeatureIDE

ration in set C, we calculate the distance with each configuration in list S. Hence, *Line 9* is executed in $\mathcal{O}(|C|^2|F|)$. The algorithm has a loop, which keep the algorithm running until all configurations C are ordered. Hence, the complexity of *Algorithm 1* is $\mathcal{O}(|C|^3|F|)$ where |C| is the number of configurations to be prioritized and |F| is the number of features of the SPL.

4. SPL TESTING WITH FEATUREIDE

FeatureIDE [37] is an open-source framework based on Eclipse, which supports all phases of the development of SPLs from domain analysis to software generation. The main focus of FeatureIDE is to cover the whole development process and incorporate tools for the implementation of SPLs into an integrated development environment.

Feature models can be constructed in FeatureIDE by adding, editing, and removing features using a graphical editor. With FeatureIDE, a user can create and edit configurations with s special editor. However, creating dozens of configurations manually for testing purposes is not efficient and should be automated.

Hence, we extended FeatureIDE to generate all valid products of the SPL. However, because generating all products may not be feasible. For a large SPLs, we extended FeatureIDE to support the generation of products using different sampling algorithms. As illustrated in *Figure 3* with our extensions, FeatureIDE integrates several t-wise algorithms to create configurations.

The user has several options to build configurations. As illustrated in *Figure* 3, the user can build all valid configurations, all the current configurations (the productively used configurations)that have been created, or samples using t-wise algorithms. For sampling, we integrated CASA [12], Chvatal [5], and ICPL [17, 18] algorithms to FeatureIDE. The user can select the sampling algorithm and the value of t to control the t-wise interaction coverage. Furthermore, the user can select whether to build the configurations to the same Eclipse project or build configurations to a different Eclipse project by checking the option of "Create new project".

We present an example of creating configurations using ICPL algorithm [17] with t=3. It takes the feature model in *Figure* 1 as input and returns the following configurations represented by the model:

$c1=\{MobilePhone, Calls, Screen, Colour, Media, MP3, GPS\}$
$c2=\{MobilePhone, Calls, Screen, High\_resolution,$
$Media, Camera\}$
$c3=\{MobilePhone, Calls, Screen, High\_resolution\}$
$c4=\{MobilePhone, Calls, Screen, Basic\}$
$c5=\{MobilePhone, Calls, Screen, High\_resolution, GPS\}$
$c6=\{MobilePhone, Calls, Screen, Basic, Media, MP3\}$
$c7=\{MobilePhone, Calls, Screen, Colour\}$
$c8=\{MobilePhone, Calls, Screen, High\_resolution,$
$Media, Camera, MP3, GPS\}$
$c9=\{MobilePhone, Calls, Screen, Colour, GPS\}$
$c10=\{MobilePhone, Calls, Screen, Colour, Media, MP3\}$
$c11=\{MobilePhone, Calls, Screen, High\_resolution,$
$Media, MP3, GPS\}$
$c12=\{MobilePhone, Calls, Screen, High\_resolution,$
$Media, Camera, GPS\}$
$c13=\{MobilePhone, Calls, Screen, High\_resolution,$
$Media, MP3, Camera\}$

In addition, we implemented similarity-based prioritization in FeatureIDE. The user can order the configurations by checking on the option "Prioritize based on similarity". If the option is not chosen, the configurations will be ordered based on the default order of the selected sampling algorithm.

5. EVALUATION

In this section, we present our experiments to answer the following research questions:

- **RQ1:** Can similarity-based prioritization detect interactions faster than with the default order of sampling algorithms?
- **RQ2:** What is the sampling algorithm with the fastest interaction coverage rate?
- **RQ3:** Does similarity-based prioritization detect feature interactions faster than the random orders?
- **RQ4:** What is the computation time of similarity-based prioritization compared to sampling algorithms?

We begin by defining the experimental setting, and then we present and explain the results of our experiment.

5.1 Experiment Design

We use two SPLs of different size in our experiment: *Mobile Phone* and *Smart Home*. The SPL *Mobile Phone* has ten features and has been previously used before as illustrative example for SPL testing [4,16]. In *Figure 1*, we already showed the features of SPL *Mobile Phone* and the dependencies between them. We choose SPL *Mobile Phone* because it contains all common properties of feature models [36]. We use the *Smart Home* as a larger SPL in the experiment. *S*PL Smart Home consists of 60 features and has also been used before as illustrative example for SPL testing [17, 42].

The goal of t-wise testing is to find defects caused by the interactions of up-to t features. The highest value of t for some algorithms is 3 (e.g., ICPL algorithm). Hence, we apply t-wise testing where $t = 3$ for all the sampling algorithms.

We use the output of the following algorithms: CASA [12], Chvatal [5], and ICPL [17, 18] as input for the experiment. We compare similarity-based prioritization with the default order of each algorithm and random orders of the configurations. In SPLs, most of the defects are expected to be caused by an interaction of few features [21]. We evaluate our approach by a simulation of defects based on the feature model. We assume that, if the combination of features causing a defect is covered in a configuration, the defects will be detected. Basically, each defect can be detected by a different subset of all products (i.e., DS (Defect Simulation) $\subseteq SPL$). We simulate defects occurring because of pairwise interactions and defects caused by single features.

- A defect occurs whenever a specific feature is selected, for instance, a defect can occur at the implementation level (e.g., division by zero):
  $DS = \{c|c \in SPL \wedge f \in c\}$
- A defect occurs whenever a specific feature is not selected. For instance, the feature initializes a variable, when the feature is removed, a defect can occur:
  $DS = \{c|c \in SPL \wedge f \notin c\}$
- A defect occurs whenever two specific features are selected. For instance, one feature calls a method in another feature and the retrieved value is wrong:
  $DS = \{c|c \in SPL \wedge f_1 \wedge \in c \wedge f_2 \wedge \in c\}$
- A defect occurs whenever a specific feature is selected while the another feature is not selected. For example, one feature calls a method from a feature that is not selected: $DS = \{c|c \in SPL \wedge f_1 \in c \wedge f_2 \notin c\}$
- A defect occurs whenever two specific features are selected. For instance, if one or both features are not selected, a defect can be found:
  $DS = \{c|c \in SPL \wedge f_1 \wedge \notin c \wedge f_2 \wedge \notin c\}$

We perform our experiments on the configurations before and after the prioritization for the algorithms mentioned previously. We use the defect simulation to generate 97 potential defects for SPL *Mobile Phone* and 4721 potential defects for SPL *Smart Home*.

5.2 Experiment Results

We divide this section into two parts. In the first part, we answer RQ1 and RQ2. In the second part, we answer RQ3 and RQ4.

Similarity-based prioritization vs default order of sampling algorithms. In order to answer RQ1 and RQ2, we compare similarity-based prioritization with the default order of the algorithms ICPL, Chvatal, and CASA for the SPL *Mobile Phone*.

Configs	ICPL (t=3)		Chvatal (t=3)		CASA (t=3)	
	D	P	D	P	D	P
1st	76	84	58	84	69	84
2nd	11	7	22	4	13	6
3rd	1	0	13	7	0	7
4th	4	0	1	0	5	0
5th	2	4	2	0	10	0
6th	2	2	0	2	0	0
7th	0	0	0	0	0	0
8th	1	0	1	0	0	0
9th	0	0	0	0	0	0
10th	0	0	0	0	0	0
11th	0	0	0	0	0	0
12th	0	0	0	0	0	0
13th	0	0	0	0	0	0

Table 2: Number of defects found in the nth configuration for SPL *Mobile Phone* with default order of each algorithm (D) and similarity-based prioritization approach (P).

We show in *Table 2*, the number of configurations tested to find the first defect. We show the number of defects that have been found in the first configuration. From *Table 2*, we can observe that for each sampling algorithm (CASA, Chvatal, and ICPL), the number of detected defects with prioritization (84, 84, 84) is higher than the number of the detected defects in the first configuration without prioritization (76, 58, 69) for each sampling algorithm, respectively. We illustrate in *Table 2* that *allyesconfigs* configuration [10] can detect most of the defects, which validates our decision to use it as the first configuration.

In order to answer RQ1 and RQ2, we show in the box plots of *Figure* 4 the distribution of defects for SPL *Mobile Phone*. We illustrate in the box plots the median levels and distribution of configurations for the various defects in each sampling algorithm for both cases, the default order and similarity-based prioritization. Each sampling algorithm shows few different patterns. For instance, the median for all the sampling algorithms, in both cases (default order and similarity-based prioritization) is 1. However, the upper quartile and the outliers indicate a slightly different distribution. For example, the upper quartile of the default order and similarity-based prioritization for Chvatal are 2 and 1 respectively. The results indicate that similarity-based prioritization is better than the default order of Chvatal algorithm. The number of outliers of similarity-based prioritization for algorithm ICPL indicates that it is better than the default order. Regarding RQ2, from the median and upper quartile values for each algorithm, it shows that the default order of algorithm ICPL is better that the default order of the algorithms CASA and Chvatal.

We show in *Figure* 5 the distribution of defects for SPL *Smart Home*. It illustrates that similarity-based prioritization is better than the default order of algorithms CASA and Chvatal. It indicates that the default order of algorithm ICPL is better that the other order of algorithms CASA and Chvatal. *Figure* 5 illustrates also that the similarity-based prioritization and the default order of algorithm ICPL have the same median, upper quartile values, and even the same outliers.

SPL	Sampling algorithm	Default order	Similarity-based prioritization	Improvement
Mobile Phone	ICPL	1.50	1.30	13%
	Chvatal	1.70	1.30	24%
	CASA	1.70	1.20	29%
Smart Home	ICPL	1.08	1.08	0%
	Chvatal	1.80	1.50	17%
	CASA	1.90	1.60	16%

Table 3: Average number of configurations to detect a defect

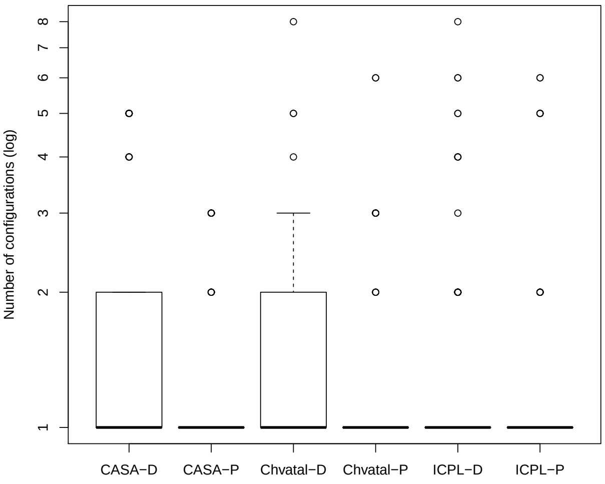


Figure 4: The number of configurations to detect all defects in SPL *Mobile Phone*; D- default order of each algorithm, P- similarity-based prioritization approach.

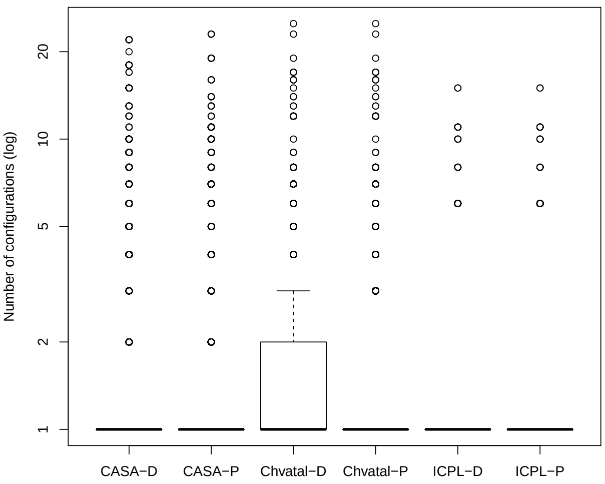


Figure 5: The number of configurations to detect all defects in SPL *Smart Home*; D- default order of each algorithm, P- similarity-based prioritization approach.

The average numbers of all approaches in *Table 3* shows that similarity-based prioritization is better than each default order of each algorithm in SPL *Mobile Phone*. *Table 3* indicates that the average number of configurations to detect a defect of similarity-based prioritization and the defaults order of ICPL algorithm is equal in SPL *Smart Home*. In *Table 3*, the average number of configurations to detect a defect in the SPL *Mobile Phone* for ICPL with similarity-based prioritization is 1.30 configuration. If we do not use our approach, the average number of configurations for the default order is 1.50. We show the percentage of improvement for each algorithm when we use similarity-based prioritization. We also show that the average number of similarity-based prioritization is less than the average number of Chvatal and CASA in SPL *Smart Home*. Regarding RQ1, the results show that similarity-based prioritization is often better than the default order of each algorithm. Regarding RQ2, the results suggest that ICPL has the best default order among the three sampling algorithms.

Similarity-based prioritization vs random orders.
We performed several experiments to compare similarity-based prioritization against random orders. We perform experiments on the configurations created by each sampling algorithm (CASA, Chvatal, and ICPL). In *Figure* 6, we show the distribution of the detected defects for 200 random order based on samples by Chvatal. *Figure* 6 indicates that only one random experiment (the orange bar) is better than our approach and the other 199 experiments are worse than similarity-based prioritization. In *Figure 7*, we show the distribution of the detected defects for 200 random orders using Chvatal algorithm on the SPL *Smart Home*. *Figure 7* indicates that similarity-based prioritization outperform 99 % (dark gray bars) of all the random experiments.

We show only the result of the random experiments for Chvatal algorithm because the results of the random experiments of the other algorithms (CASA and ICPL) lead to similar results. Hence, regarding RQ3, similarity-based prioritization is better than the random orders. From all experiments, we observe that the default order of CASA algorithm is non-deterministic, the order changes every time we run the experiments using the same parameters. In contrast, the default order of the other algorithms (ICPL and Chvatal) seems to be deterministic.

The experiments were performed on a PC with an Intel Core i5 CPU @ 3.33 GHz, 8 GB RAM, and Windows 7. We show the average execution time of similarity-based prioritization and sampling algorithms for 10 runs using SPLs *Mobile Phone* and *Smart Home* as illustrative examples in *Table 4*. Regarding RQ4, comparing the average execution time of our algorithm to the average execution time of sam-

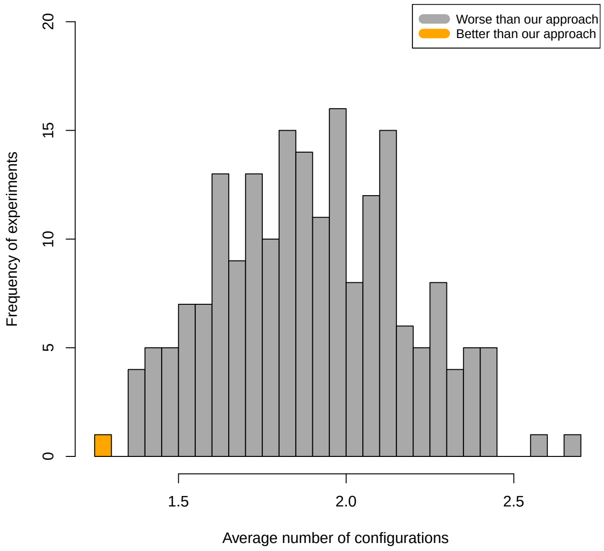


Figure 6: Average number of configurations to detect a defect in SPL *Mobile Phone* for 200 random orders

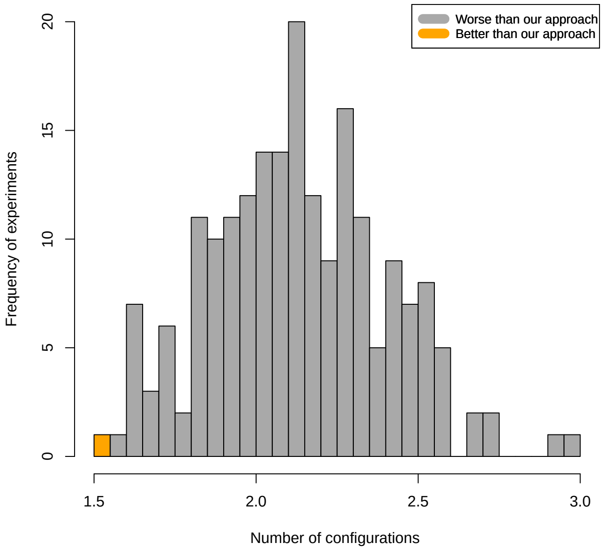


Figure 7: Average number of configurations to detect a defect in SPL *Smart Home* for 200 random orders

pling algorithms in *Table 4*, the required time for similarity-based prioritization is negligible. The benefit of spending time for similarity-based prioritization is that our approach will help detect defects earlier.

From our experiments, regarding RQ1, the results in *Figures 4* and *5* indicate the rate of early interaction coverage of similarity-based prioritization is often better than the default order in all the sampling algorithms. Regarding RQ2, we show in *Figures 4* and *5* that the best default order among the three sampling algorithms is ICPL. Regarding RQ3, the result show that similarity-based prioritization is significantly better than random orders. Regarding RQ4, the required time for similarity-based prioritization is trifling compared to the time is required for the sampling task.

Further explorations.

We compare between two options in *Algorithm 1* either we use the sum of distances between configurations or the maximum of minimum distance between the configuration and all the selected configurations. We found that, for instance, in SPL *Mobile Phone*, for each sampling algorithm (CASA, Chvatal, and ICPL) the early rates of interaction coverage for the maximum of minimum distance between the configurations $(1.3, 1.3, 1.2)$ are better than the early rates of for the summation of configurations distances $(1.6, 1.6, 1.7)$ for each sampling algorithm, respectively.

A question is raised that whether similarity-based prioritization can scale to handle large feature models. To answer this question, we conduct experiments on all feature models in S.P.L.O.T. repository [28] that have more than 140 features. We apply 2-wise sampling to sample the configurations. The generated configurations from these feature models are large for some feature models (e.g., BattleofTanks leads to 663 configurations sampled by algorithm CASA). We found that similarity-based prioritization is better than the default order of each sampling algorithm (CASA, Chvatal, and ICPL) in all SPLs. The average percentages of improvement for CASA, Chvatal, and ICPL in all SPLs are 11.6%, 10.1%, and 10.2%, respectively.

5.3 Threats to Validity

There is a potential threat that may affect the external validity related to the size of feature models. We cannot ensure that the proposed approach will provide the same results for larger feature models. This validity threat comes from that the size of feature models and the constraints may affect the results of our similarity-based prioritization. To reduce this threat, in addition to the two SPLs (*Mobile Phone* and *Smart Home* that have been used as illustrative examples in this paper, we conduct experiments with larger feature models (larger than 140 features). The inputs to our approach for some feature models are large (e.g., BattleofTanks leads to 663 configurations sampled by algorithm CASA),

There is another potential threat related to the distribution of defects. We assume that the defects are equally distributed over the features and their interaction in an SPL. This is rather problematic in testing as defects are often found where they are not expected. Still, assuming equally distributed defects seems to be better than to build on potentially, wrong distributions.

We evaluate our approach against the default order of each sampling algorithm, but we do not know whether the default orders of algorithms are already good enough. Hence, we compare our approach random orders. To overcome the threat related to the random experiments and its values, we run 200 experiments for a random approach to reduce risks due to random effects.

There is a potential threat that may affect the internal validity related to the implementation of our approach. To overcome this threat, we applied our algorithm on a small

SPL	Sampling algorithm	Sampling	Similarity-based prioritization	Percentage of prioritization compared to sampling
Mobile Phone	ICPL	175ms	1ms	0.6%
	Chvatal	245.1ms	1ms	0.4%
	CASA	528.6ms	1ms	0.2%
Smart Home	ICPL	1929.5ms	21.3ms	1.1%
	Chvatal	31900.7ms	20ms	0.1%
	CASA	641702.5ms	23.1ms	0.004%

Table 4: Average execution time of the sampling algorithms and similarity-based prioritization

SPLs and analyzed the results step by step. We make our implementation and experiments data publicly available.[1]

6. RELATED WORK

Several approaches have been proposed for testing SPLs. However, most approaches focus on reduction of the test space [6, 7, 22, 26, 31, 33]. Combinatorial testing is a strategy used to select a small subset of all products in which most interaction faults occur. Kuhn et al. [24] mention that 70% and 95% of defects are found for pairwise and 3-wise coverage respectively. They also indicate that almost all the defects are found for 6-wise coverage. Several approaches use t-wise sampling to achieve the combination interaction between features [31, 32]. For instance, perrouin et al. [32] propose an approach to generate t-wise test cases based on Alloy SAT solver. The output of all these approaches can be used as input to similarity-based prioritization, to detect defects earlier.

Another strategy to reduce the test space is selecting which test cases need to be run. Lochau et al. [26] explore deriving test suites to test SPL incrementally. Kowal et al. [23] add additional information to feature model about the actual source of feature interaction. They use these information to reduce the number of products that need to be tested. Shi et al. [33] present a compositional symbolic execution technique to analyze the SPL. They reduce the number of products that need to be tested by means of the interactions between features. Our similarity-based prioritization approach is not alternative for these techniques but it can be combined to increase the efficiency of SPL testing

Recently, a few prioritization approaches have been proposed to prioritize the products based on different criteria. Baller et al. [2] propose a framework to prioritize the products under test based on the selection of adequate test suites with regard to cost/profit objectives. In contrast, we prioritize our approach based on similarity between configurations. Henard et al. [15] combine sampling and prioritization during the products generation. We adopt it by calculating the similarity between the configurations in different way by taking the deselected features into account. In addition, we select the configuration with maximum of minimum distance. Furthermore, our approach can be combined with any sampling algorithm. They use a search-based approach to generate the product.

In addition, several approaches have been proposed to prioritize configurations based on domain knowledge [9, 19]. Johansen et al. [19] combine the sampling and prioritization based on weights. In their approach, they use domain knowledge to assign weights on t-wise interaction. An approach based on the behavior of SPL is proposed by Devroey et al. [9]. They use statistical testing based on a usage model to extract products with high probability to be executed. Ensan et al. [11] propose a goal-oriented approach by giving configurations that contain the most desirable features priority over less important features. Prioritizing these configurations is based on the expectations of the domain experts. Uzuncaova et al. [39] propose an approach by generating test cases from SPL specifications. In contrast to all these approaches, similarity-based prioritization does not require domain knowledge or any further effort of additional statistics, such as usage models. The user can apply similarity-based prioritization regardless of the SPL's domain.

7. CONCLUSION AND FUTURE WORK

In this paper, we proposed similarity-based prioritization for SPL testing. There are many sampling algorithms, which are used to generate a subset of all valid configurations. Testing these configurations may take a considerable amount of time depending on the number of configurations. In this paper, we prioritized the configurations to find products containing defects earlier, based on the assumption that most errors are caused by an interaction of few features.

We applied and evaluated our approach on the sampling algorithms CASA, Chvatal, and ICPL. We performed our experiments on two SPLs of different size. We showed that the rate of early interaction coverage of similarity-based prioritization is better than random, CASA order, and Chvatal order. In our evaluation, we identified that the rate of early interaction coverage of default order of ICPL is better than the default order of CASA and Chvatal algorithms. From the result, it appears existing sampling algorithms already have quite good order, which, however, can even be improved in many cases using similarity-based prioritization. In addition, to evaluate the scalability of our approach, we performed new experiments on all feature models in S.P.L.O.T. repository that have more than 140 features. The result indicated that similarity-based prioritization is better than the default order of each sampling algorithm.

In future work, we will include the effect of feature interactions in our prioritization approach. We plan to compare our similarity-based prioritization approach to the order of other sampling algorithms, such as AETG [8], IPOG [25],

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/spl-testing/

and MoSo-PoLiTe [30]. Furthermore, we are going to execute real test cases against our approach or generate real defects into the source code of existing SPLs using techniques from mutation testing.

8. ACKNOWLEDGMENTS

We thank Martin Fagereng Johansen for his support in integrating the SPLCATool into FeatureIDE, Reimar Schröter for helpful comments on a draft. Finally, we want to thank Fabian Benduhn and Hauke Baller for interesting discussions. This work is partially supported by the German Research Foundation (DFG SA 465/34-2).

9. REFERENCES

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[2] H. Baller, S. Lity, M. Lochau, and I. Schaefer. Multi-Objective Test Suite Optimization for Incremental Product Family Testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 303–312. IEEE Computer Science, 2014.

[3] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005.

[4] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[5] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[6] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and Adequacy in Software Product Line Testing. pages 53–63. ACM, 2006.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. pages 129–139. ACM, 2007.

[8] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering (TSE)*, 34(5):633–650, 2008.

[9] X. Devroey, G. Perrouin, M. Cordy, P.-Y. Schobbens, A. Legay, and P. Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 10:1–10:7. ACM, 2014.

[10] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. Understanding Linux Feature Distribution. In *Proceedings of the 2012 Workshop on Modularity in Systems Software (MISS)*, pages 15–20. ACM, 2012.

[11] A. Ensan, E. Bagheri, M. Asadi, D. Gasevic, and Y. Biletskiy. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *Proceedings of the International Conference on Information Technology:New Generations (ITNG)*, pages 291–298. IEEE Computer Science, 2011.

[12] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. 16(1):61–102, 2011.

[13] R. W. Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[14] H. Hemmati and L. Briand. An Industrial Investigation of Similarity Measures for Model-Based Test Case Selection. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 141–150. IEEE Computer Science, 2010.

[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Suites for Large Software Product Lines. *CoRR*, abs/1211.5451, 2012.

[16] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Multi-Objective Test Generation for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 62–71. ACM, 2013.

[17] M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. pages 638–652. Springer, 2011.

[18] M. F. Johansen, Ø. Haugen, and F. Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 46–55. ACM, 2012.

[19] M. F. Johansen, Ø. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. pages 269–284. Springer, 2012.

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

[21] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 181–190, 2009.

[22] C. H. P. Kim, D. Batory, and S. Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 57—68. ACM, 2011.

[23] M. Kowal, S. Schulze, and I. Schaefer. Towards Efficient SPL Testing by Variant Reduction. In *Conference of the International workshop on Variability & composition (VariComp)*, pages 1–6. ACM, 2013.

[24] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering (TSE)*, 30(6):418–421, 2004.

[25] Y. Lei, R. N. Kacker, D. R. Kuhn, V. Okun, and

J. Lawrence. IPOG: A General Strategy for T-Way Software Testing. In *Proceedings of the International Conference on Engineering of Computer-Based Systems (ECBS)*, pages 549–556. IEEE Computer Science, 2007.

[26] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental Model-based Testing of Delta-oriented Software Product Lines. pages 67–82. Springer, 2012.

[27] J. McGregor. Testing a Software Product Line. In *Testing Techniques in Software Engineering*, pages 104–140. Springer, 2010.

[28] M. Mendonça, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 761–762. ACM, 2009.

[29] L. Northrop and P. Clements. A Framework for Software Product Line Practice, Version 5.0. *SEI*, 2007.

[30] S. Oster. *Feature Model-Based Software Product Line Testing*. PhD thesis, TU Darmstadt, Darmstadt, 2012.

[31] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journa l (SQJ)*, 20(3-4):605–643, 2012.

[32] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 459–468. IEEE Computer Science, 2010.

[33] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 270–284. Springer, 2012.

[34] A. Tevanlinna, J. Taina, and R. Kauppinen. Product Family Testing: A Survey. *Software Engineering Notes (SEN)*, 29:12–17, 2004.

[35] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. 47(1):6:1–6:45, 2014.

[36] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE Computer Science, 2009.

[37] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming (SCP)*, 79(0):70–85, 2014.

[38] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–200. IEEE Computer Science, 2011.

[39] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. A Specification-Based Approach to Testing Software Product Lines. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT (ESEC)*, pages 525–528. ACM, 2007.

[40] F. J. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[41] D. M. Weiss. The Product Line Hall of Fame. In *Proceedings of the International Software Product Line Conference (SPLC)*, page 395. IEEE Computer Science, 2008.

[42] N. Weston, R. Chitchyan, and A. Rashid. A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 211–220, 2009.