Softw Syst Model
DOI 10.1007/s10270-016-0569-2

REGULAR PAPER

Effective product-line testing using similarity-based product prioritization

Mustafa Al-Hajjaji[1] · Thomas Thüm[2] · Malte Lochau[3] · Jens Meinicke[4] · Gunter Saake[1]

Received: 30 July 2015 / Revised: 21 September 2016 / Accepted: 11 November 2016

Abstract A software product line comprises a family of software products that share a common set of features. Testing an entire product-line product-by-product is infeasible due to the potentially exponential number of products in the number of features. Accordingly, several sampling approaches have been proposed to select a presumably minimal, yet sufficient number of products to be tested. Since the time budget for testing is limited or even a priori unknown, the order in which products are tested is crucial for effective product-line testing. Prioritizing products is required to increase the probability of detecting faults faster. In this article, we propose similarity-based prioritization, which can be efficiently applied on product samples. In our approach, we incrementally select the most diverse product in terms of features to be tested next in order to increase feature interaction coverage as fast as possible during product-by-product testing. We evaluate the gain in the effectiveness of similarity-based prioritization on three product lines with real faults. Furthermore, we compare similarity-based prioritization to random orders, an interaction-based approach, and the default orders produced by existing sampling algorithms considering feature models of various sizes. The results show that our approach potentially increases effectiveness in terms of fault detection ratio concerning faults within real-world product-line implementations as well as synthetically seeded faults. Moreover, we show that the default orders of recent sampling algorithms already show promising results, which, however, can still be improved in many cases using similarity-based prioritization.

Keywords Software product lines · Product-line testing · Model-based testing · Combinatorial interaction testing · Test-case prioritization

Communicated by Prof. Hassan Gomaa.

✉ Mustafa Al-Hajjaji
mustafa.al-hajjaji@st.ovgu.de

Thomas Thüm
t.thuem@tu-braunschweig.de

Malte Lochau
Malte.Lochau@es.tu-darmstadt.de

Jens Meinicke
meinicke@ovgu.de

Gunter Saake
saake@iti.cs.uni-magdeburg.de

1 University of Magdeburg, Magdeburg, Germany

2 TU Braunschweig, Braunschweig, Germany

3 TU Darmstadt, Darmstadt, Germany

4 METOP GmbH, University of Magdeburg, Magdeburg, Germany

1 Introduction

Most software project managers view testing as a necessary evil that occurs during or at the end of a project. According to Ludewig and Lichter, testing a single system consumes between 25 and 50% of the development costs [29,46]. Model-based testing aims to minimize the testing costs by automating test generation from abstract models of the software under test [60].

Testing a software product line consumes even more resources [23,43]. A software product line comprises a set of similar software products that share common features and differ in variable features. Using a product line, the customer is able to compose software systems from a large space of configuration options [6]. Recently, software product lines are gaining widespread acceptance in industry. Several companies such as Bosch, Philips, Siemens, gen-

Published online: 17 December 2016

eral Motors, Hewlett Packard, Boeing, and Toshiba apply product-line approaches to reduce the costs for development and maintenance, to increase quality, and to reduce time to market [6,48,63].

Given a product-line implementation, one major challenge concerning its quality assurance is that it is difficult to guarantee that all its features and every possible combination of them in a product will work as expected. This difficulty is because of the following reasons:

- The number of possible products potentially grows exponentially in the number of features.
- Testers usually have a limited amount of time and resources to run tests on a specific product.
- Every further derivation of a product under test causes additional costs (may include assembling of software and hardware, in case of industrial systems).

Product-line testing is more critical than testing a single system because a fault in a single feature can exist in thousands or even millions of products. Two testing strategies have recently been pursued to test product lines [58]. First, product-by-product testing follows a traditional strategy where concrete products are generated and tested individually using established testing techniques from single product testing. Second, family-based testing is a strategy where all products are checked in a single run whether they satisfy their specifications [14]. In the family-based strategy, testing is achieved without considering any particular product. It uses a virtual representation of the product line that simulates all products (i.e., by superimposing test specifications of all products). However, effective testing should not only investigate the software but also its interplay with the hardware and environment, which is not supported by recent family-based testing techniques. In addition, family-based testing can be time-consuming and even incomplete (e.g., IO) as it requires a complex and specialized execution environment [7,44,47]. To overcome those limitations, we consider product-by-product testing instead of family-based testing. However, testing a product line exhaustively product-by-product is infeasible or even impossible due to aforementioned exponential explosion of products and many redundant testing steps due to the commonality among products. To overcome these difficulties, test engineers are seeking techniques to reduce the effort and time of product-by-product testing.

Combinatorial interaction testing is an approach used in product-by-product testing to systematically reduce the number of products under test [49–51]. In particular, T-wise combinatorial interaction testing is based on the observation that most faults are caused by interactions among a fixed number of at most T features [35,38]. Consequently, combinatorial interaction testing is used to sample a presumably minimal subset of configurations that cover each T-wise interaction by at least one product under test [18]. This optimization problem is NP-hard, and several heuristics have been proposed to perform T-wise sampling such as CASA [26], Chvatal [17], ICPL [34,35], IPOG [42], and IncLing [4]. The output of these algorithms is a presumably minimal product sample to be tested to obtain T-wise combinatorial interaction testing coverage.

Although the sampling algorithms can reduce the number of products significantly, the number can still be large, especially if the product line has a large number of features. For instance, Johanson et al. [35] report that 480 products are selected when 2-wise coverage is applied to the Linux kernel with 6888 features. As the time budget in practice is limited in testing, the order in which products are tested matters to presumably find faults as soon as possible within the available amount of time. For this purpose, several approaches have been proposed to prioritize products based on different criteria, such as statistical analysis of usage models [19], cost and profit of selecting a set of test cases [11], dissimilarity among products [3,4,32], and domain knowledge [36,55].

In this article, we propose similarity-based prioritization for sample-based product-line testing. With regard to the previously tested products, we incrementally select the most diverse products in terms of features to be tested next. Identifying the feature selections that differentiate products and prioritize them based on the degree of similarity can yield a meaningful order for a subset of potential products for testing. The idea of prioritizing products is to increase the probability to find faults as fast as possible. Henard et al. [32] propose a related search-based approach to combine sampling and prioritization techniques using a similarity among configurations. However, similarity-based prioritization is a heuristic approach focus on prioritizing products and can be combined with any sampling technique and may even be applied to the set of productively used configurations. Sánchez et al. [55] use non-functional attributes to prioritize products. In our approach, we do not require more domain knowledge than the feature selection, as the domain knowledge is often not available in practice. Instead, we follow a purely model-based black-box approach, solely incorporating the feature model (i.e., knowledge about valid combinations of features) as domain knowledge, if available. The input of our approach can be all valid configurations, a set of configurations that is generated using sampling algorithms, or configurations given by domain experts.

With similarity-based prioritization, we aim to cover as many combinations and therefore interactions as early as possible which increases the probability to improve the fault detection ratio. We select and test products one at a time. At first, we select the product with maximum number of features (a.k.a. *all-yes-config* [21]). Second, we select the

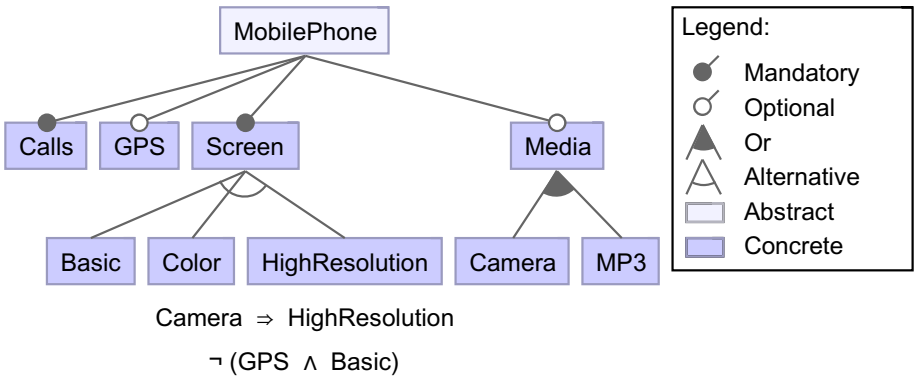

Fig. 1 Feature diagram of product line *MobilePhone*

subsequent product that has the minimum similarity with the previous product. Then, we select the next product that has the minimum similarity with *all* previous products. The process continues until all products are selected or the testing time budget for this product line is consumed. By following the previous steps, we increase the feature interaction coverage of the product line under test. Our contributions are as follows:

- We propose similarity-based prioritization [3] to order the products before they are materialized for testing.
- We integrate the implementation of our approach into FeatureIDE [59] with support for product-line testing and similarity-based prioritization.
- We evaluate the increase in effectiveness (i.e., finding faults as soon as possible) of similarity-based prioritization compared to random orders by executing test cases on three product-line implementations with real faults.
- We evaluate potential improvements in effectiveness of similarity-based prioritization concerning feature interaction faults compared to random orders as well as an interaction-based approach and the default order of three sampling algorithms, namely ICPL, CASA, and Chvatal.

The general concept of the approach presented in this article has been already presented in previous work [3,5]. In this article, we substantially extend the experimental evaluation by results gained from a series of experiments. We investigate the potential increase in effectiveness of similarity-based prioritization using three real-world product lines, larger feature models, improved tool support, and two types of artificial seeded faults.

The remainder of this article is structured as follows: We describe relevant background on feature modeling, combinatorial interaction testing, and test-case prioritization in Sect. 2. In Sect. 3, similarity-based prioritization is introduced in detail. We discuss how FeatureIDE supports sample-based product-line testing in Sect. 4. We present our evaluation in Sect. 5 and discuss related work in Sect. 6. Finally, we conclude our work in Sect. 7.

2 Background on product sampling and prioritization

In this section, we present necessary background on feature models, combinatorial interaction testing for product lines, and test-case prioritization.

2.1 Feature modeling

Feature models consist of a tree-like hierarchical structure that captures the information of all possible configurations of product line in terms of features and relationships among them. These models are usually represented graphically by feature diagrams [37]. Figure 1 shows an example of feature diagrams of a product line *MobilePhone*. Feature diagrams are used to restrict the variability of a product line as not all combinations of features are valid. A valid combination is called a *configuration*. Each configuration can be used to generate a product implementing the selected features. The selection of a feature implies the selection of its parent feature. In particular, the relations among features are classified into the following groups.

- *Or*-group, at least one subfeature, such as *Camera* and *MP3*, has to be selected, if its parent is selected.
- *Alternative*-group, exactly one subfeature must be selected, if their parent is selected. For instance, a mobile phone may include only one of the following features: *Basic, Color*, or *HighResolution*.
- *And*-group, If a subfeature is neither in an *Or*-group nor in an *Alternative*-group, it is either *mandatory* or *optional*.

The features are either *abstract* or *concrete*. If implementation artifacts are mapped to a feature, such as feature *Calls*, it is *concrete*; otherwise, it is *abstract*, such as feature *MobilPhone* [57]. Furthermore, there are additional dependencies among features which cannot be described with a hierarchical structure, i.e., cross-tree constraints that define such dependencies:

- *Require* constraint, for instance, the selection of feature *Camera* in a mobile phone implies the selection of feature *HighResolution*.
- *Exclude* constraint, for example, features *GPS* and *Basic* cannot be supported for the same mobile phone.
- Additional constraints can also be defined as propositional formulas using logical operators, such as $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$.

The configurations, which are captured as selection of features from a feature model, serve as the input of our similarity-based prioritization. The number of all valid configurations for product line *MobilePhone* (consisting of nine concrete features) is 14. For example, the following constitutes a possible configuration of product line *MobilePhone*.

c = {Calls, GPS, Screen, HighResolution, Media, MP3}

2.2 Combinatorial interaction testing for product lines

The number of products may increase exponentially in the number of features; hence, testing every product of a product line is often infeasible. To alleviate this problem, combinatorial testing is used to reduce the effort of product-line testing by selecting a minimal, yet sufficient subset of products [15,49,50]. A feature interaction occurs when the integration of two or more features modifies or influences the behavior of other features [33]. Using combinatorial interaction testing, the faults that are triggered by erroneous interactions between a certain number of features can be detected. The configuration generation in product-line testing may be defined as an instance of a Boolean combinatorial testing problem with constraints [32].

In pairwise combinatorial interaction testing, each combination of two features appears in at least one configuration. The possible combinations of feature B and feature C, where $B = Basic$, and $C = Color$ in our running example, are $B \wedge C$, $B \wedge \neg C$, $\neg B \wedge C$, $\neg B \wedge \neg C$. While the valid combinations $B \wedge \neg C$, $\neg B \wedge C$, and $\neg B \wedge \neg C$, according to the feature model, need to be part of at least in one configuration, the invalid combination $B \wedge C$ does not need to be covered.

T-wise testing has been generalized from pairwise testing to cover all T-wise combinations of features [18,35]. A trade-off exists between the time computation and test coverage. The higher T, the higher is the test coverage, the more computation time is required to find a minimum set of configurations that covers all combinations of T features, and as a result the number of configurations to be generated and tested increases. These configurations can also be represented as *covering array*. Generating covering arrays is an NP-complete problem where the main challenge of generating them is to find the minimal set of configurations that covers the T-wise combinations of features. Existing works have shown that it is feasible to generate the covering arrays from feature models [17,26,34,35]. In the following, we give an overview on some algorithms that we use in our evaluation.

Chvatal is a greedy algorithm proposed by Chvatal [17] to solve the covering array problem. The original version of the algorithm was not specialized to create configurations. Recently, Johansen et al. [34] adapted and improved the algorithm to incorporate feature models. At first, all T-wise feature combinations are generated. Second, an empty configuration is created. Then, the algorithm iterates through all feature combinations to add the maximum possible number of them to the configuration. Each time a new combination is added to the configuration, the algorithm checks the configuration validity using SAT solver. If the configuration is invalid, the combination is removed from the configuration. A configuration is added to the final set of configurations if it at least contains one uncovered combination. The process continues until all T-wise feature combinations are covered at least once.

ICPL [35] is an efficient algorithm for computing the T-wise covering array. The goal of ICPL is to cover the T-wise combination of features as fast as possible. ICPL is based on the Chvatal algorithm with several improvements, such as identifying the invalid feature combinations at an early stage. In addition, the parallelization of the algorithm shortens the computation time significantly. Similar to the Chvatal algorithm, ICPL tries to cover the maximum number of uncovered feature combinations, each time a configuration is created.

CASA [26] is a simulated annealing algorithm that was proposed to generate T-wise covering arrays of product lines. CASA algorithm iterates three nested strategies to reduce the number of created configurations. CASA is a non-deterministic algorithm. Thus, different configurations are created with different orders for the same feature model.

As the testing time in practice is limited, the order in which products are tested matters to find faults as soon as possible. Table 1 exemplarily lists nine configurations that are created from feature model *MobilePhone* using pairwise sampling with ICPL [34,35]. Each sampling algorithm typically outputs an ordered list of configurations. The orders of the aforementioned sampling algorithms may already be effective as they all aim to cover as many interactions as fast as possible.

2.3 Test-case prioritization

In a single system, it may require a large amount of time and effort to run all test cases in an existing test suite, especially

Table 1 Configurations of *MobilePhone* product line created using pairwise sampling with ICPL

ID	Configurations
c_1	{Calls, Screen, Color}
c_2	{Calls, GPS, Screen, HighResolution, Media, MP3}
c_3	{Calls, Screen, HighResolution, Media, Camera}
c_4	{Calls, Screen, Basic}
c_5	{Calls, Screen, HighResolution, Media, Camera, MP3}
c_6	{Calls, GPS, Screen, Color, Media, MP3}
c_7	{Calls, GPS, Screen, HighResolution, Media, Camera}
c_8	{Calls, Screen, Basic, Media, MP3}
c_9	{Calls, GPS, Screen, HighResolution}

for larger-scale real-world software. Rothermel et al. [53] report that running test cases of their 20 KLOC industrial software takes approximately 7 weeks. To reduce the number of executed test cases, different techniques have been developed, such as test set minimization, test-case selection, and test-case prioritization [64]. While minimization and selection, the test cases aim to reduce the test sets, test-case prioritization aims to reorder test cases based on their priority while keeping the same size. According to some criteria, the test cases with highest priority should be executed earlier than those with lower priority in order to find faults as soon as possible. Several goals of prioritization can be achieved, such as reducing the costs of testing or increasing the rate of fault detection of test cases (i.e., finding faults as soon as possible). To meet the goals of prioritization, various criteria can be applied to test cases, such as the coverage of the test cases that cover a large part of the code and the estimated ability of test cases to reveal faults [53]. Another technique is time-aware prioritization, which uses a genetic algorithm [62] to prioritize test cases to be executed within a given time budget. Regarding the interaction coverage, Bryce and Memon [13] propose to prioritize test cases by examining all T-wise interactions of a software system. We refer to the latter in our paper as *interaction-based*.

In the context of product lines, interaction-based is a search approach that systematically enumerates all possible configurations and selects the configuration which covers the highest number of feature interactions to be tested next. In particular, the interaction-based approach works as follows. We derive all feature combinations from the given configurations. Then, we incrementally select one configuration at a time covering the largest number of uncovered feature combinations. In case we have two or more configurations covering the same number of uncovered combinations, we randomly select the next configuration to be tested next. The interaction-based approach does not scale to complex product lines due to the expensive computation of the T-wise interactions as well as the exponential growth of the configurations number which must be enumerated each time a configuration is selected [32]. In this article, in addition to random orders and the default orders of sampling algorithms, we used the interaction-based approach to measure the effectiveness of similarity-based prioritization, where we prioritize configurations based on the dissimilarity among them, as we show in the next section.

3 Similarity-based prioritization

In model-based testing, a similarity heuristic is used in some cases to prioritize test cases based on similarity measures between them. Hemmati et al. [30] observe that dissimilar test cases in the context of state machine-based testing are likely to detect different faults. Hence, we propose similarity-based prioritization to order configurations based on the similarity among them. As illustrated in Fig. 2, the input of similarity-based prioritization is a set of configurations. For small product lines, all valid configurations can be used as input to prioritization. For larger product lines, these configurations can be reduced using sampling algorithms or given by domain experts. The outcome of our approach is a prioritization of configurations in terms of a list.

In similarity-based prioritization, we select one configuration at a time to be used to generate and test a product. We

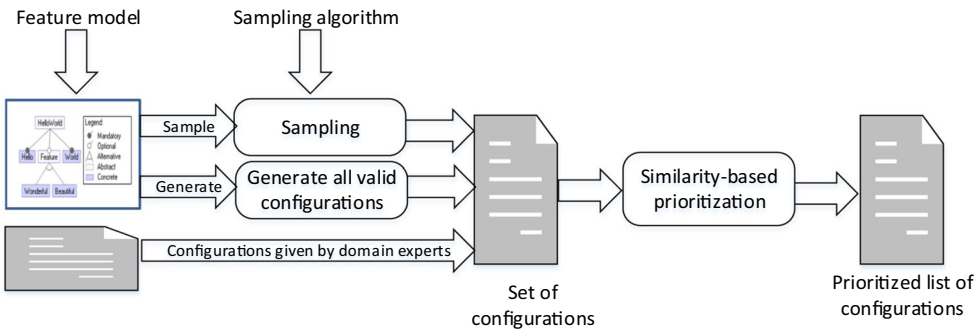

Fig. 2 Overview of similarity-based prioritization approach

Table 2 Distances between the nine configurations listed in Table 1

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
c_1	0	0.556	0.444	**0.222**	0.556	0.334	0.556	0.444	0.334
c_2	0.556	0	**0.334**	0.556	**0.222**	**0.222**	**0.222**	**0.334**	**0.222**
c_3	0.444	**0.334**	0	0.444	0.111	0.556	0.111	0.444	0.334
c_4	**0.222**	0.556	0.444	0	0.556	0.556	0.556	0.222	0.337
c_5	0.556	**0.222**	0.111	0.556	0	0.444	0.222	0.334	0.444
c_6	0.334	**0.222**	0.556	0.556	0.444	0	0.444	0.334	0.444
c_7	0.556	**0.222**	0.111	0.556	0.222	0.444	0	0.556	0.222
c_8	0.444	**0.334**	0.444	0.222	0.334	0.334	0.556	0	0.556
c_9	0.334	**0.222**	0.334	0.337	0.444	0.444	0.222	0.556	0

select the configuration that has the lowest value of similarity compared to *all* previously selected configurations in terms of feature selections. Prioritizing configurations by identifying the feature selections that differentiate them can yield a meaningful order for a subset of potential products for testing. Prioritizing test cases based on similarity has previously been proposed in single-system testing [16,30] as well as in product-line testing [32]. However, compared to the latter, our approach differs in the following: selection of *all-yes-config* as first product to be tested, usage of the distance function to measure similarity, and the way prioritization is computed. We describe the aforementioned differences in detail later in this section.

We consider the selected *and deselected* features when we calculate the distance (cf. Eq. 1). The rationale of taking the deselected features into account is based on the observation that some faults can be triggered because some features are *not* selected [1]. In particular, we use an adaptation of Hamming distance [28] to measure the similarity among configurations. The distance values among configurations range from 0 to 1. Values close to 0 indicate similar feature selections, while values close to 1 indicate that the configurations differ in every feature. We define the distance between two given configurations c_i and c_j relative to a set of all features F as

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

where $|c_i \cap c_j|$ is the number of features selected in configurations c_i and c_j, $|(F \backslash c_i) \cap (F \backslash c_j)|$ is the number of features deselected in configurations c_i and c_j, and F is the set of all features. The definition of $distance(c_i, c_j, F)$ is symmetric with regard to c_i and c_j, which can be used to reduce the number of computational steps of our algorithms. Devroey et al. [20] investigate different types of distance measurements in product-line testing, and they report that Hamming distance is more effective than other distance types for the most investigated case studies.

Previous work reports that dissimilar test cases are likely to detect new faults more than the similar ones [30]. We assume that the observation applies to the generated configurations. To ensure that the next selected configuration is the most dissimilar to *all* already selected configurations S, we consider the minimum distances between each configuration c_i in C with all configurations in S as

$$D_{min}(c_i, S, F) = \underset{s_j \in S}{min}(distance(c_i, s_j, F)) \tag{2}$$

where $D_{min}(c_i, S, F)$ are the minimum distances between each configuration $s_j \in S$ and configuration $c_i \in C$ and $distance(c_i, s_j, F)$ is obtained from Eq. 1. Then, we select the configuration with the maximum value from these minimum values as

$$D_{max}(C, S) = \underset{c_i \in C}{max}(D_{min}(c_i, S, F)) \tag{3}$$

where $D_{max}(C, S)$ is the maximum distance between each configuration $c_j \in C$ and all the minimum distances and $D_{min}(c_i, S, F)$ is obtained from Eq. 2. We do not consider the sum or average of distances between the configurations, as it is misleading in some cases as we show in the following example. We use the nine configurations that are listed in Table 1 from our running example. For instance, the distance among configurations c_1 and c_2 is 0.556. In Table 2, we list the distances among the nine configurations. In each step, we select a not yet selected configuration with the maximum distance to already selected configurations.

Since considering the distance does not help to select the first configuration to test, we select the configuration that has the maximum number of selected features to be tested first. If more than one configuration has the same maximum number of selected features, we take the first one we find. The rationale of selecting the configuration with the maximum number of selected features as the first to test is that it is assumed to cover most faults, which may exist in an individual feature. Therefore, it is a common strategy in the Linux community to test this configuration first (a.k.a. *all-yes-config*) [21]. We explain the steps of *Algorithm 1* with our running example. As illustrated in Table 1, we have three configurations c_2, c_5, and c_7 that have the maximum number of features (six features). We select c_2 to be the first configura-

Algorithm 1: Similarity-Based Prioritization

1: **Input:** $C=\{c_1, c_2, c_3, \ldots, c_n\}$ (set of configurations),
2: F (set of features)
3: **Output:** S (list of prioritized configurations)
4: S ← [ ]
5: **for** i← 1 to $C.size()$ **do**
6: **for** j← i+1 to $C.size()$ **do**
7: $AllDistances$[i, j]= $\underset{c_i \in C,\ c_j \in C}{distance}(c_i, s_j, F)))$
8: **end for**
9: **end for**
10: Select $c_{max} \in C$ with $\forall\ c_i \in C$: $|c_{max}| \geq |c_i|$
11: $S.add(c_i)$
12: $C.remove(c_i)$
13: $TestProduct(c_i)$
14: **while** C *not empty* **do**
15: $c_i = SelectConfiguration(C, S, AllDistances)$ (Algorithm 2)
16: $S.add(c_i)$
17: $C.remove(c_i)$
18: $TestProduct(c_i)$
19: **end while**
20: **return S**

tion (*all-yes-config*) to be tested (*Algorithm 1*, Lines 10–13). The selected configuration is added to the list S, which contains the prioritized configurations, and removed from set C, which contains the remaining configurations. In particular, c_2 is added to list S and removed from set C. Then, the configuration with the maximum distance to c_2 is selected (as shown in *Algorithm* 2).

We illustrate the algorithms based on the distance values in Table 2. Configurations c_1 and c_4 have the maximum distance (0.556) to configuration c_2 (largest values in the second row). In case we have two or more configurations with the same distance value, we select the first configuration that gets this value of distance. Hence, we select c_1 to be added to list S and removed from set C (highlighted with circle). As a result, two configurations exist in list $S = (c_2, c_1)$, and all other configurations remain in set C (i.e., $C = \{c_3, c_4, c_5, c_6, c_7, c_8, c_9\}$). We consider the distance for each configuration $c_i \in C$ to all configurations in S. The distances between both configurations (c_2, c_1) and all configurations are listed in the first two rows of Table 2.

According to Eq. 2, we consider the minimum distance between each configuration $c_i \in C$ and all configurations in S (minimum distances highlighted with bold letters). According to Eq. 3, we select the first configuration $c_i \in C$ with the maximum of these distances. The maximum distance is 0.334 for c_3, which is thus selected next (highlighted with circle and bold letters).

In contrast, considering the sum or the average of distances between the configurations might be misleading in some cases. For instance, in Table 2, the summations of distances for each configuration (c_3 and c_4) with configurations (c_2 and c_1) are equivalent (0.778). Although these distances are identical, we expect that the configuration c_3 detects more

Algorithm 2: Select the next Configuration

1: **Input:** $C=\{c_1, c_2, c_3, \ldots, c_n\}$ (set of configurations),
2: S (list of tested/prioritized configurations)
3: $AllDistances$[ ] (store all distances among configurations)
4: **Output:** $c_i \in C$ (configuration)
5: $NextConfig := null$
6: $NextConfigDistance := 0$
7: **for** i← 1 to $C.size()$ **do**
8: $TempDistance := 1$
9: **for** j← 1 to $S.size()$ **do**
10: **if** $(AllDistances[i, j] < TempDistance)$ **then**
11: $TempDistance := AllDistances[i, j]$
12: **end if**
13: **end for**
14: **if** $(TempDistance > NextConfigDistance)$ **then**
15: $NextConfigDistance := TempDistance$
16: $NextConfig := C.get(i)$
17: **end if**
18: **end for**
19: **return** $NextConfig$

new faults than configuration c_4, because configuration c_4 is more similar to configuration c_1 (distance is 0.222) than configuration c_3, and c_1 already detects most of the faults that can be detected by c_4. Thus, we select c_3 to be tested before c_4.

We continue this process until all configurations are ordered. The resulting order of all configurations is $S = (c_2, c_1, c_3, c_8, c_4, c_6, c_9, c_5, c_7)$. Performing similarity-based prioritization is computationally cheap, because we essentially rely on Hamming distance, which is a linear operation in the number of features, to calculate distances among configurations. We consider only the number of selected and deselected features of each configuration. However, the complexity factor is the overall number of configurations which might be exponential in the number of features in the worst

case (i.e., if no sampling is applied before). Similarity-based prioritization does not require further domain knowledge than the feature selection. It requires a set of configurations, which is created using sampling algorithms or given by domain experts. In addition, our approach is compatible with any sampling algorithm producing an explicit list of products as output. We implemented our approach and combined it to existing sampling algorithms in FeatureIDE [59] as will be described in the next section.

4 Similarity-based prioritization for product-line testing with featureIDE

FeatureIDE [59] is a set of Eclipse plug-ins that support all phases of product-line development from domain analysis to software generation. FeatureIDE covers the entire product-line development process and integrates with tools for the implementation of product lines.

In this section, we focus on the functionality of FeatureIDE that is related to feature modeling, configuration creation, and product-by-product testing. In FeatureIDE, feature models can be created using a graphical editor. With regard to configuration creation, users can create and edit configurations with an editor. However, it is not practical to create many configurations manually. For this purpose, we extended FeatureIDE as follows:

- Automated product generation for all user-defined configurations.
- Generation of all valid configurations
- Integration of existing sampling algorithms covering up to 2-wise (IncLing [4]), 3-wise (ICPL [35]), 4-wise (Chvatal [17]), and 6-wise interactions (CASA [26])
- Similarity-based prioritization and interaction-based prioritization of configurations retrieved by any of the three previous steps
- Product-by-product testing with JUnit testing being executed and illustrated in the JUnit view
- Propagation of error markers from generated products to original source code

Each sampling algorithm produces some implicit order as part of its output that is given by the positioning of the products within the output data structure. However, this order is not explicitly mentioned by the authors who proposed these sampling algorithms and is somehow forced by coverage criteria. Moreover, the order produced is even non-deterministic (i.e., different runs with the same input often lead to different orders).

In FeatureIDE, the user chooses the way to order configurations, either by the default order of these sampling algorithms or by similarity-based prioritization. To evaluate our approach, we used this implementation to conduct our experiments.

5 Experiment and results

Based on our implementation of similarity-based prioritization in FeatureIDE, we carried out several experiments. For a given set of configurations, similarity-based prioritization aims at detecting more faults earlier (with respect to random and sampling orders) by faster increasing interaction coverage. It aims to increase the probability of detecting faults as soon as possible for product lines under test by increasing the interaction coverage as fast as possible over time. We measured the potential improvements of effectiveness in terms of the fault detection rate compared to random orders, to the default order of sampling algorithms, and to the interaction-based approach [13]. This interaction-based approach is commonly used to measure the effectiveness of CIT approaches [32]. The rationale of the comparison to the interaction-based approach is that our approach aims to increase the interaction coverage for a product line under test over time. In our experiments, we consider the interaction of only up to $t = 2$ in the interaction-based approach, as it does not scale well to large t. In our evaluation, we focus on the following research questions.

- **RQ1:** Can similarity-based prioritization detect faults faster compared to random orders and the interaction-based approach?
- **RQ2:** Do sampling algorithms produce samples with an acceptable effective order?
- **RQ3:** Is the computational overhead required for similarity-based prioritization neglectable compared to the efforts required for sampling?

Regarding RQ1, we ran experiments using three available subject product lines with real faults located in the feature source code. These subjects have already been used in previous studies on product-line verification [7,44]. In Table 3, we summarize statistics on these product lines. As these product lines are rather small in terms of the number of features, we ran further experiments to evaluate our approach using real-world feature models and artificial ones of vari-

Table 3 Overview of subject product lines

Product lines	LOC	Features	Specifications	Products	Faults
Elevator	1046	6	9	20	8
Mine-pump	580	7	5	64	4
E-mail	1233	9	9	40	9

ous sizes. With respect to the experiment of product lines with real faults and to the experiment of feature models, we derived the following, more specific research questions from $RQ1$:

- **RQ1.1:** Can similarity-based prioritization detect faults faster than random orders and the interaction-based approach *for product lines with real faults*?
- **RQ1.2:** Can similarity-based prioritization cover the *feature interactions* faster than random orders and the interaction-based approach?

In Sect. 5.2, we describe the experiment to address $RQ1.1$, where we used three subject product lines with real faults in the source code. In Sect. 5.3, we present the settings and the results of the experiment with feature models to address $RQ1.2$, $RQ2$, and $RQ3$. In Sect. 5.1, we describe APFD metric to evaluate our results.

5.1 Weighted average percentage of faults detected (APFD)

We use a metric called APFD developed by Elbaum et al. [22] to evaluate the pace of fault detection. The APFD is calculated by measuring the weighted average of the number of faults detected for the system under test. APFD values range from 0 to 1; higher values of APFD indicate faster fault detection rates. APFD can be calculated as

$$\text{APFD} = 1 - \frac{tf_1 + tf_2 + \cdots + tf_n}{n * m} + \frac{1}{2n} \tag{4}$$

where n is the number of test cases, which represent configurations in our case, m is the number of faults, and tf_i is the position of first test t that exposes the fault.

We show how to calculate APFD using our running example. We have nine configurations $C = \{c_1, \ldots, c_9\}$, as listed in Table 1, and we estimated the distribution of six faults $f_1, \ldots, f_6$, as listed in Table 4. Assume we have two orderings of these test suites, ordering O_1 : $c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9$ and ordering O_2 : $c_3, c_1, c_6, c_4, c_8, c_7, c_2, c_5, c_9$. Incorporating the data from Table 4 into the APFD calculation yields

$$\text{APFD} = 1 - \frac{4+1+2+3+3+3}{6*9} + \frac{1}{2*9} = 0.76$$

for O_1 and

$$\text{APFD} = 1 - \frac{4+2+6+1+1+1}{6*9} + \frac{1}{2*9} = 0.78$$

for O_2. Thus, the ordering O_2 yields a better fault detection rate (0.78) than O_1 (0.76).

Based on guidelines for reporting statistical tests [8], we use the Mann–Whitney U test to analyze the APFD values. The Mann–Whitney U test is a nonparametric statistical test of the null hypothesis that two samples come from the same population against an alternative hypothesis (i.e., a particular population tends to have larger values than the other). From this test, we obtain the p value representing the probability that two samples are equal. The significance level is 0.05. That means, if the p value is less than or equal to 0.05, we reject the null hypothesis that the two samples are equal.

Table 4 Fault matrix

Faults	Configurations								
	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
f_1				×	×				
f_2	×	×		×				×	
f_3		×		×			×		
f_4			×		×				
f_5			×			×			
f_6			×	×	×			×	

5.2 Experiment with code base of existing product lines

In this section, we describe our experiment using three product lines in order to address *RQ1.1*.

Subject product lines we selected three product lines that have been previously used to evaluate verification strategies of product lines [7,44]. These product lines have real faults caused by erroneous feature interactions:

- The Elevator system is an Elevator model designed by Plath and Ryan [52].
- The Mine-pump system simulates a water pump in a mining operation [39].
- The E-mail system of Hall [27] provides several features, such as encryption, decryption, and automatic forwarding.

The aforementioned product lines are implemented using feature-oriented programming. Each feature of these product lines has individual specifications. The specifications, in form of assertions, are introduced using AspectJ. Apel et al. [7] adapt the initial specifications that are written by the original authors of these product lines. The three product lines contain faults that are documented by the authors of these product lines. Apel et al. [7] focus on the faults that are caused by the feature interaction. These faults violate at least one specification. Mainly, the specifications concern specific safety properties. For instance, in the Elevator system, if the weight is more than the maximum weight, the elevator should not move. The detected faults are caused by

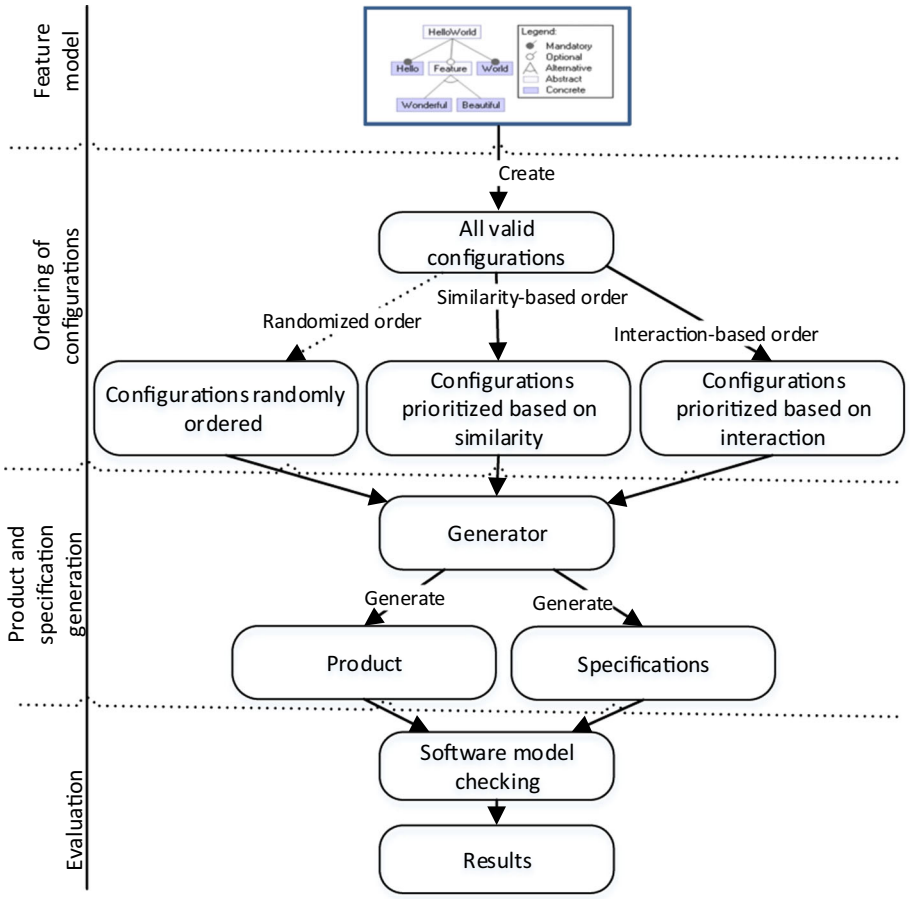

Fig. 3 Steps of the experiment with code *base* of product *lines*

the interaction of up to three features. In the case of E-mail system, an encrypted E-mail must not be transferred in plain text. The detected faults in E-mail system are caused by the interaction of up to six features. For the Mine-pump system, an example of a safety concern is that in case a methane gas is detected, Mine-pump must be deactivated. The detected faults in this system are caused by the interaction of up to four features. We use *Java Pathfinder* (JPF) [61] as a software model checker to verify the code. The information of these product lines is summarized in Table 3. In addition to the product lines and their feature models, the information about how these faults are caused by the feature interaction is publicly available.[1]

Experiment design Figure 3 visualizes the experiment steps. First, we generate all valid configurations for testing, as the subject product lines are small enough. Second, we prioritize these configurations with similarity-based prioritization and measure the potential improvement in effectiveness compared to random orders and to the interaction-based approach [13]. Third, the products and the specifications are generated using FeatureIDE. Finally, we verify whether the products satisfy the specification using JPF. If a product does not satisfy the specification, we consider the violation of specification as a fault and record the interacting features.

Experiment results in Fig. 4, we show APFD value distribution of random orders, our similarity-based prioritization approach, and interaction-based approach for three product lines. From Fig. 4, we showed that the APFD values of our approach are higher than the values of the interaction-based approach and the median APFD values of random orders for all product lines, especially for product lines *Elevator and E-mail*. However, we observed in product line *Mine-pump* some random orders are better than similarity-based prioritization. A possible reason for that could be the limited number of detected faults in the product line *Mine-pump* (only four faults) compared to the product lines *Elevator* (eight faults) and *E-mail* (nine faults). Using the Mann–Whitney *U* test, we found that our approach and random orders are significantly different for the three product lines *Elevator* (p=0), *Mine-pump* (p=0.0004), and *E-mail* (p=0). Comparing our approach to the interaction-based approach (cf. Table 5), we found that the APFD values for our approach are higher than the values of interaction-based approach for the three product lines. The reason for the potential improvement in effec-

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

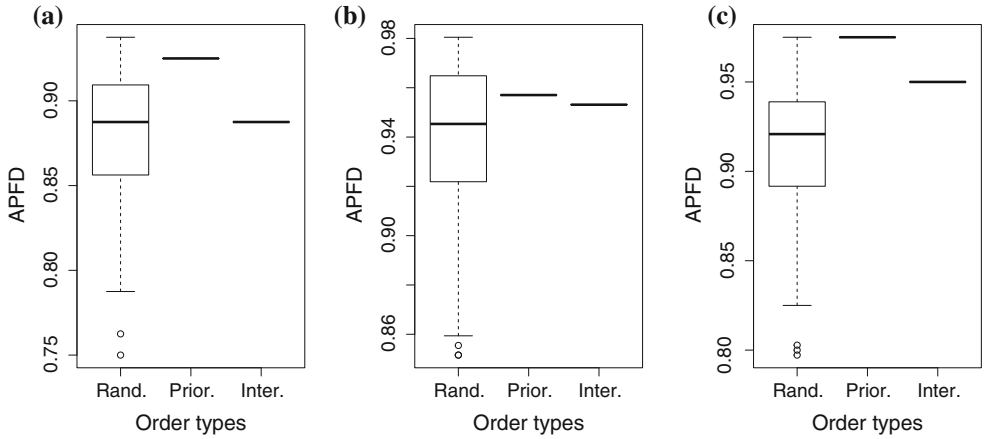

Fig. 4 Delta-oriented prioritization with distance minimum. **a** Elevator. **b** Mine-pump. **c** E-mail

Table 5 Average APFD value over 100 random orders, the APFD value of similarity-based prioritization, and the APFD value of the interaction-based approach

Product line	Random	Prioritization	Interaction-based
Elevator	0.881	0.925	0.888
Mine-pump	0.937	0.957	0.953
E-mail	0.912	0.975	0.950

tiveness by our approach compared to the interaction-based approach is that most of the faults, which caused by the interaction of selected features, are detected in the first product (*all-yes-config*). In the interaction-based approach, the first product is selected randomly, because each product in the first step covers the same number of combinations. Hence, it might be the case that most of the features are not selected. However, the interaction-based approach can be improved by starting with *all-yes-config* as well.

Although the criteria which we used to prioritize the products are based only on comparing selected and deselected features, they give a clue when we applied our approach on product line with real faults. To validate our approach further, we conducted other experiments with feature models and simulated test execution and fault detection.

5.3 Experiments with feature models

In this section, we present the feature models, the experiment design, and the results of the experiment to address $RQ1.2$, $RQ2$, and $RQ3$.

Subject feature models we use a variety of feature models to evaluate similarity-based prioritization. Besides our running example, we select all feature models in the S.P.L.O.T. repository [45] that have more than 140 features[2] to evaluate the effectiveness of our approach with larger feature models. Furthermore, we consider feature models of very large-scale product lines, such as the Linux kernel in version 2.6.28.6 with 6,888 features. The feature model of the Linux kernel has been used previously to evaluate the scalability of product-line testing [32,35]. Additionally, we have access to a feature model consisting of 2,513 features from an industrial partner in the automotive domain (Automotive1). In addition to the real feature models, we consider 61 artificial feature models generated with S.P.L.O.T. [45] and served previously as a benchmark for evaluation purposes [32]. The feature models that were used in our experiments are listed in Table 6, where we present for each feature model: number of features, number of constraints, ratio of the number of distinct features in cross-tree constraints to the number of features (CTCR), and number of valid configurations using the pairwise sampling algorithm ICPL.

Experiment design in Fig. 5, we show an overview of the experiments with feature models. To evaluate the effectiveness of similarity-based prioritization, we use feature models of different sizes as inputs for the sampling algorithms. The input to similarity-based prioritization is a set of configurations, which are created based on the feature model. Generating all valid configurations as in Sect. 5.2 is not feasible for these feature models. Hence, we create these configurations using the algorithms CASA [26], Chvatal [17], and ICPL [35]. These algorithms are well known in the community, and they have been used before to evaluate product-line testing techniques [2,3,32,54,55]. We compare the effectiveness of similarity-based prioritization to the effectiveness of the default order of each sampling algorithm, but we do not know whether the effectiveness of

[2] Last accessed May 10, 2016.

Table 6 Feature models used in our evaluation (CTCR) stand for cross-tree constraints representative

Feature model	Features	#Constraints	CTCR (%)	#Configurations*
BattleofTanks	144	0	0	459
FM_Test	168	46	28	44
Printers	172	0	0	181
BankingSoftware	176	4	2	42
Electronic shopping	290	21	11	22
DMIS**	366	192	93	29
eCos 3.0 i386pc	1245	2478	99	62
FreeBSD kernel 8.0.0	1369	14,295	93	77
Automotive1	2513	2833	28	913
Linux 2.6.28.6	6888	6847	99	479
10xAFM15	15	2.6	19	13.7
10xAFM50	50	9.7	17	26.4
10xAFM100	100	20	17	56.4
10xAFM200	200	39	17	89.9
10xAFM500	500	100	17	189.6
10xAFM1000	1000	100	14	343.6
1xAFM5K	5542	300	11	685

* Numbers of configurations calculated using pairwise sampling algorithm ICPL
** DMIS stands for A Model for Decision-making for Investments on Enterprise Information Systems. The values next to the artificial feature models 10xAFM15, 10xAFM50, 10xAFM100, 10xAFM200, 10xAFM500, and 10xAFM1000 represent the average values over 10 feature models

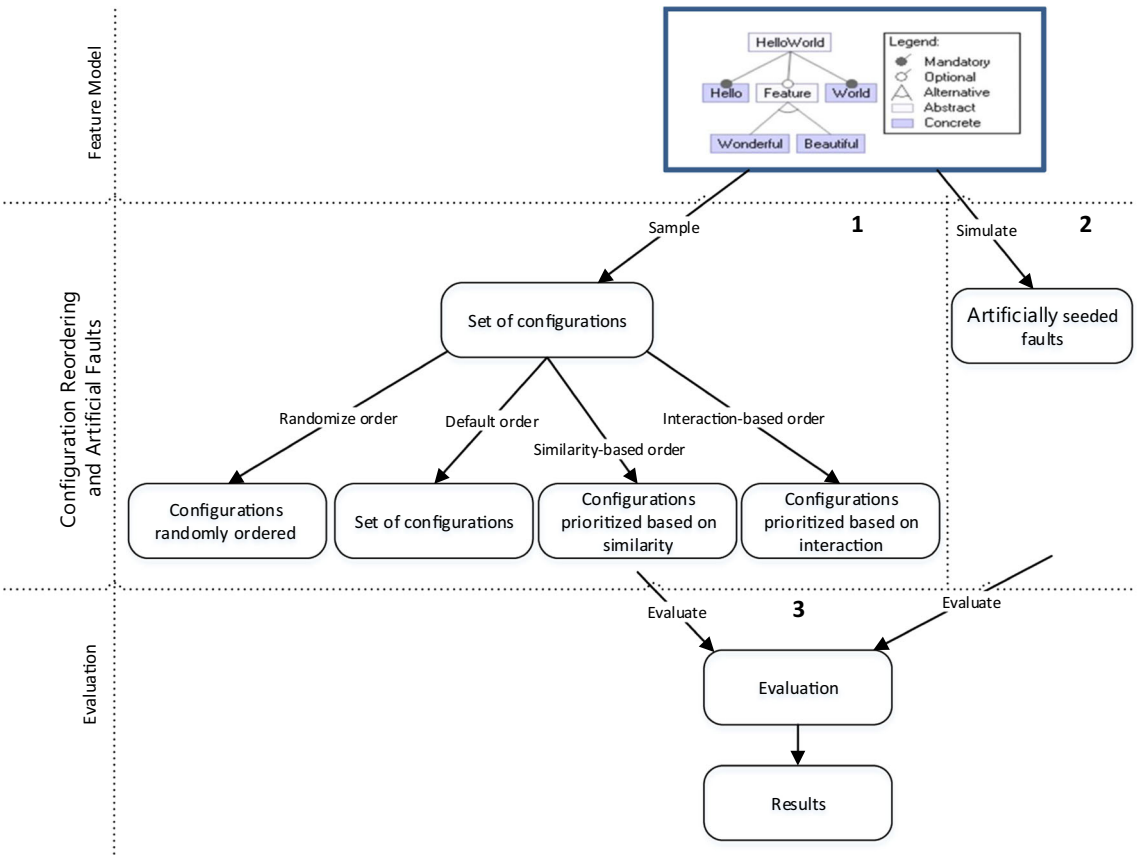

Fig. 5 Steps of the experiment with feature models

default orders of algorithms is effective or not. The order of the created configurations by the sampling algorithms is mainly influenced by the interaction coverage where these algorithms aim to cover the feature combinations as fast as possible. In addition, we compare our approach to random orders and interaction-based orders.

We evaluate similarity-based prioritization by measuring its effectiveness. For this purpose, we implemented a fault distribution simulator based on the fault simulators presented by Bagueri et al. [9,25] and used by Sanchez et al. [54] which has been used in several works to evaluate the fault detection rate of product-line test suites [9,25,54]. Generating these faults is independent of prioritizing products. We assume that the faults are equally distributed over the features in a product line. The issue with this supposition is that faults are regularly discovered where they are not expected. In any case, it appears that assuming equal distribution is better than to build on non-idealized, yet conceivably non-representative distributions. The input of our fault generator is a feature model, and the outputs are valid combinations of features (i.e., partial configurations). We assume that, if the combination of features causing a fault is covered in a configuration, the faults will be detected. The number of simulated faults on each feature model is $n/10$, where n is the number of features. We assume that having a larger feature model leads to potentially more feature interactions. Thus, the number of simulated faults in our experiment is proportional to the feature model size. To assess the effectiveness of our approach, we generate two types of faults from the fault simulator, called exhaustive interaction faults and pattern interaction faults. We use these two types of faults to assess the effectiveness of our approach in terms fault detection ratio with different fault types.

With *Exhaustive Interaction Faults*, we simulate faults up to 6-wise feature interactions. Kuhn et al. [40] mention that 70 and 95% of faults are found with 2-wise and 3-wise coverage, respectively. They mentioned that the more interaction coverage among features is achieved, the higher percentage of faults are found. They indicate that almost all faults are covered in case of 6-wise coverage. Bagueri et al. [9] simulate faults for 2-wise feature interactions, and Sanchez et al. [54] simulate faults for 4-wise feature interactions. We incorporate up to 6-wise feature interactions, because almost all faults are caused by the interaction of up to 6 features [40].

Previous work [9,54] focused on faults caused by the interaction among the selected features as well, and they do not take deselected features into account. However, with our fault generator, we also consider the deselected features, because deselected features may also cause faults in products [1].

The same proportion of each fault type is generated, i.e., 17% for each of the following fault types: single feature, 2-wise, 3-wise, 4-wise, 5-wise, and 6-wise feature interaction. We assume the equality of fault numbers for T-wise interaction faults, because we are not aware of a study that reports what is the percentage of each T-wise fault (i.e., 70% of faults are 1-wise interaction faults). Examples of faults simulated for our running example in Fig. 1 could be given as follows: $\{Camera\}$, $\{MP3, \neg Basic, Screen, HighResolution\}$ represent a fault in feature *Camera* and another fault that applies to all configurations where *MP3*, *Screen*, and *HighResolution* are selected and *Basic* is not selected.

Table 7 Fault patterns [1]

ID	Some selected
1	a
2	$a \wedge b$
3	$a \wedge b \wedge c$
4	$a \wedge b \wedge c \wedge d$
5	$a \wedge b \wedge c \wedge d \wedge e$
ID	**Combination of selected and not selected**
1	$\neg a$
2	$a \wedge \neg b$
3	$a \wedge b \wedge \neg c$
4	$a \wedge b \wedge c \wedge \neg d$
5	$a \wedge b \wedge c \wedge d \wedge \neg e$
6	$\neg a \wedge \neg b$
7	$a \wedge \neg b \wedge \neg c \wedge \neg d \wedge \neg e$

The steps of generating these simulated faults are as follows. We select a value T where $T \in [1:6]$. Then, we select T features randomly. We decide for each feature randomly whether it is selected or deselected. Then, we check whether this combination of features is valid according to the feature model using SAT solver (i.e., whether it occurs in at least one configuration). We accept only valid combinations, because invalid combinations are not relevant. We already showed how the valid combinations of features are created in Sect. 2.

With *Pattern Interaction Faults*, we base the fault generation on a qualitative study [1] for a set of faults that has been collected from the Linux kernel repository. Abal et al. [1] analyze each fault and record their results. They observe that some faults are caused when some features are not selected. They classify their faults as they appear in Table 7. All the reported faults are up to 5-wise feature interactions. We implemented our fault generator to seed faults similar to these faults. We generate the same proportion of faults for each pattern due to aforementioned reasons that other distributions are potentially non-representative.

Experiment results for *RQ1.2*: *Can similarity-based prioritization cover the feature interactions faster than the random orders and the interaction-based approach?* We compare our approach to the average of the results of 100 experiments for

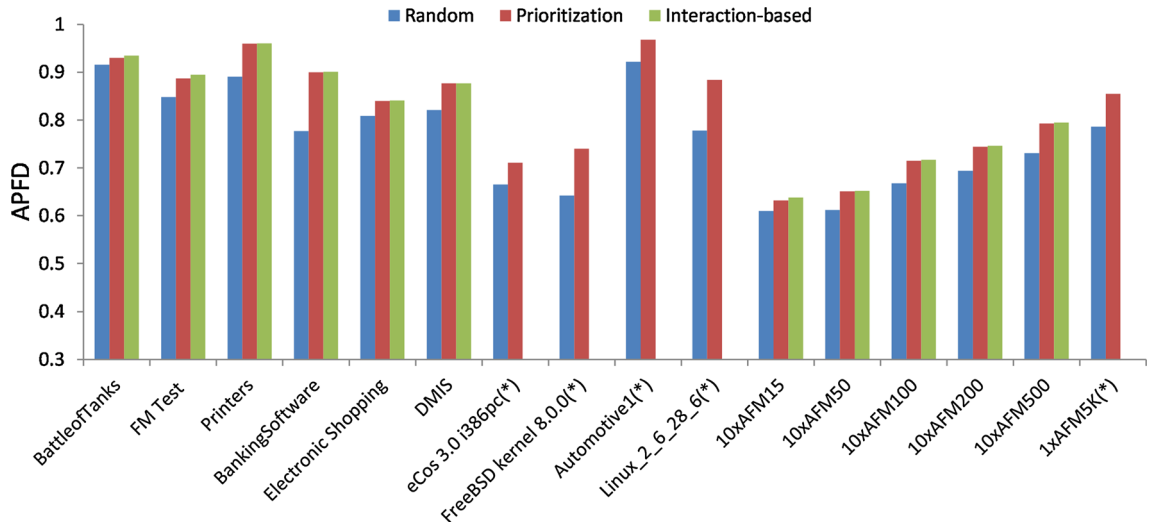

Fig. 6 Average APFD for random orders, similarity-based prioritization, and interaction-based using exhaustive interaction faults. *Asterisk* computation of interaction-based did not finish within 24 h

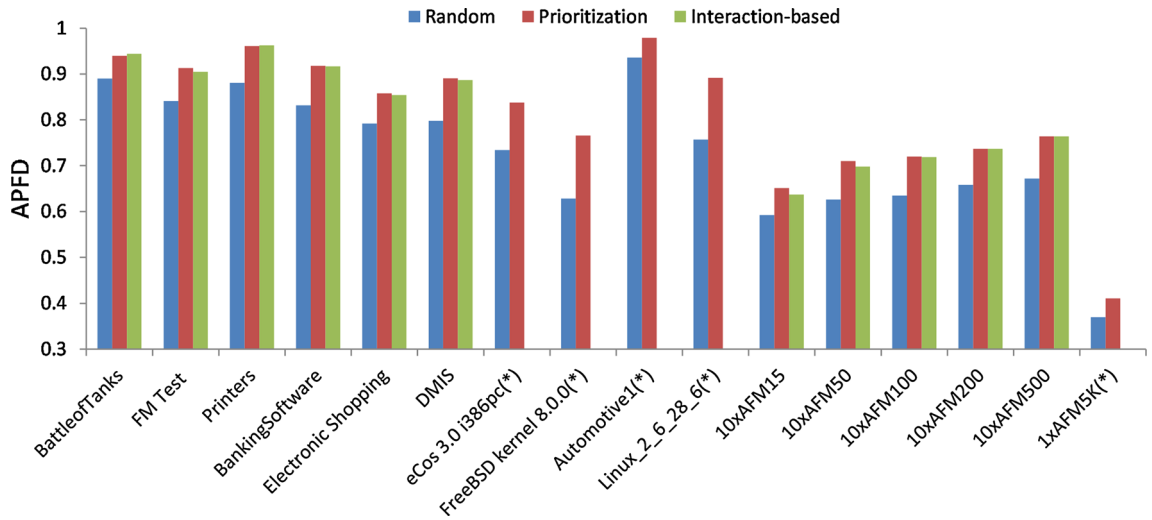

Fig. 7 Average APFD for random orders, similarity-based prioritization, and interaction-based using pattern interaction faults. *Asterisk* computation of interaction-based did not finish within 24 h

random orders and to interaction-based prioritization using aforementioned fault types. In Figs. 6 and 7, we represent the average APFD of random orders, similarity-based prioritization, and interaction-based over 100 sets of exhaustive and pattern interaction faults, respectively. With the Mann–Whitney *U* test, we observed that the difference between similarity-based prioritization and random orders is significant for all feature models for the both fault types. In Fig. 8, we show the distribution of *p* values between APFD values for real and artificial feature models using the exhaustive and pattern interaction faults. We observed that most *p* values are approximately zero for all feature models. Hence, the evaluation results show that similarity-based prioritization improves the effectiveness of product-line testing compared to random orders.

Comparing our approach to the interaction-based approach, the results show that both approaches approximately identical with a slight improvement for one over the other in some feature models. For instance, in Fig. 6, we observe a slight improvement for the interaction-based approach for the some feature models such as *BattleofTank, FM_Test, and BankingSoftware*. In Fig. 7, we notice that similarity-based prioritization is slightly better than interaction-based approach for some feature models, such as *FM_Test, BankingSoftware, DMIS, 10xAFM15, and 10xAFM50.* The reason for the slight improvement of our approach over the interaction-based approach with the pattern interaction faults is that most of the simulated faults are selected features (cf. Table 7). Hence, these faults might be detected with the first product (*all-yes-config*). For the first

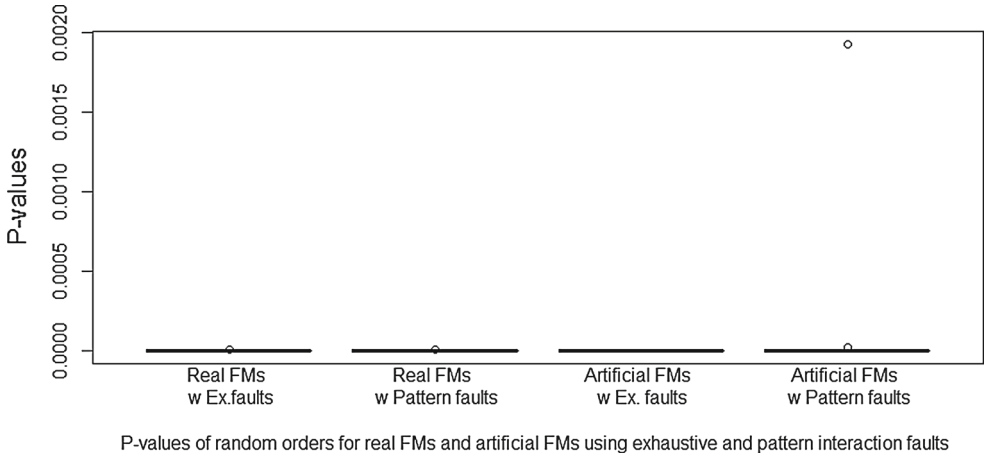

Fig. 8 Distribution of p values from Mann–Whitney U test between random orders and similarity-based prioritization, (Ex.) exhaustive interaction faults

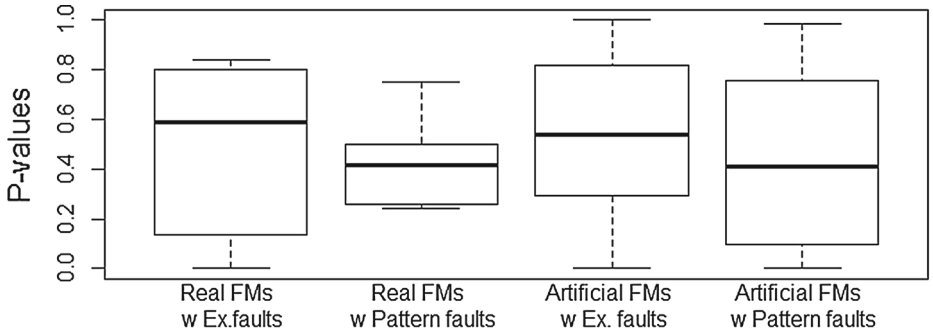

Fig. 9 Distribution of p values from Mann–Whitney U test between interaction-based approach and similarity-based prioritization, (Ex.) exhaustive interaction faults

product with interaction-based approach, it is selected randomly, because any one will cover the same percentage of the feature combinations. Thus, it might be the case that most of the features in the first product are not selected. In Fig. 9, we show the distribution of p values between APFD values for the interaction-based approach and similarity-based prioritization using the exhaustive and pattern interaction faults for real and artificial faults. From Fig. 9, we observe that the median p values of all box-plots are higher than 0.05. Hence, the difference between our approach and the interaction-based approach is not significant.

We conclude from the results that our approach performs well compared to an expensive approach such as the interaction-based approach. The advantage of our approach is that it can scale well to large feature models, while it is not the case with interaction-based where it does not scale in our evaluation to feature models larger than 500 features.

Results for *RQ2*: *Do sampling algorithms produce samples with an acceptable effective order?* For each sampling algorithm, there is an implicit order forced by feature coverage. That is, a sampling tool outputs configurations necessarily in a particular order. We investigate whether these algorithms already have good orders by comparing their orders to similarity-based prioritization.

In Tables 8 and 9, we present the average APFD values over 100 sets of faults for both fault types. We highlight higher average values of APFD with a circle. If the difference between values is significant, the higher values are highlighted with an additional background color. In Table 10, we show the p values resulting of the Mann–Whitney U test between APFD values of our approach and the default orders of sampling algorithms for both fault types.

In Table 8, we show the average APFD values for each feature model using the exhaustive interaction faults. We observed that the average APFD values of similarity-based prioritization for all feature models are higher than the average APFD values of the sampling algorithms. If we look at the results for each feature model separately, we observe that they are varying. For instance, comparing to the default order of sampling algorithm CASA, we find that our approach is better in most cases. In particular, our approach is significantly better for four feature models and significantly worse

Table 8 Average APFD for default order of sampling algorithms and similarity-based prioritization using exhaustive faults

FM	APFD					
	CASA-D	CASA-P	CH-D	CH-P	ICPL-D	ICPL-P
BattleofTanks	0.915	0.910	0.932	0.931	0.930	0.930
FM_Test	**0.845**	0.833	0.866	**0.887**	0.864	**0.887**
Printers	0.926	0.928	0.961	0.961	0.960	0.960
BankingSoftware	0.849	0.851	0.899	0.896	0.901	0.900
Electronic Shopping	0.829	**0.834**	0.847	0.845	0.841	0.840
DMIS	0.823	0.824	0.876	0.874	0.878	0.877
eCos 3.0 i386pc	*	*	0.723	0.717	**0.718**	0.711
FreeBSD kernel 8.0.0	*	*	0.738	0.739	0.740	0.740
Automotive1	*	*	0.967	0.968	0.968	0.968
Linux 2.6.28.6	*	*	*	*	0.884	0.884
10xAFM15	0.595	**0.605**	0.637	0.640	0.633	0.632
10xAFM50	0.608	**0.616**	0.643	0.644	0.646	**0.651**
10xAFM100	0.698	**0.705**	0.713	0.715	0.714	0.715
10xAFM200	*	*	0.744	0.744	0.743	0.744
10xAFM500	*	*	0.791	0.791	0.793	0.793
10xAFM1000	*	*	0.817	**0.818**	0.817	0.818
1xAFM5K	*	*	*	*	0.854	0.855
Average	**0.709**	**0.711**	**0.797**	**0.811**	**0.817**	**0.818**

CH Chvatal algorithm, *D* default order of each algorithm, *P* similarity-based prioritization approach
* No result within a whole day of computation

Table 9 Average APFD for default order of sampling algorithms and similarity-based prioritization using pattern interaction faults

FM	APFD					
	CASA-D	CASA-P	CH-D	CH-P	ICPL-D	ICPL-P
BattleofTanks	0.913	0.918	0.944	0.942	0.942	0.940
FM_Test	0.821	**0.870**	0.852	**0.911**	0.850	**0.913**
Printers	0.923	**0.927**	0.961	0.960	0.962	0.961
BankingSoftware	0.838	**0.867**	0.910	0.911	0.916	0.918
Electronic Shopping	0.833	**0.850**	0.856	0.862	0.854	0.858
DMIS	0.825	**0.828**	0.879	0.884	0.885	0.891
eCos 3.0 i386pc	*	*	0.844	0.846	0.835	0.838
FreeBSD kernel 8.0.0	*	*	0.764	0.767	0.764	0.766
Automotive1	*	*	0.967	0.968	0.968	0.968
Linux 2.6.28.6	*	*	*	*	0.890	0.892
10xAFM15	0.573	**0.612**	0.634	**0.658**	0.632	**0.651**
10xAFM50	0.651	**0.684**	0.692	**0.707**	0.686	**0.710**
10xAFM100	0.689	**0.720**	0.713	**0.717**	0.717	0.720
10xAFM200	*	*	0.735	**0.741**	0.735	0.737
10xAFM500	*	*	0.763	**0.764**	0.762	**0.764**
10xAFM1000	*	*	0.783	**0.784**	0.781	**0.782**
1xAFM5K	*	*	*	*	0.411	0.411
Average	**0.707**	**0.728**	**0.812**	**0.828**	**0.799**	**0.807**

CH Chvatal algorithm, *D* default order of each algorithm, *P* similarity-based prioritization approach
* No result within a whole day of computation

only for feature model *FM_Test*. Comparing our approach to the default order of sampling algorithm Chvatal, we observe that the average APFD values over all feature models of similarity-based prioritization and Chvatal are 0.811, and 0.797, respectively. We notice that the average values of the default order of Chvatal are better but not significantly for five feature models, while the average values of our approach are better but not significantly for five feature models and significantly better for feature models *FM_Test* and *10xAFM1000*.

The results of similarity-based prioritization compared to sampling algorithm ICPL are as follows. With our approach, the average APFD values for all feature models are higher than the average APFD values of algorithm ICPL. However, from Table 8, we notice that APFD values of similarity-based prioritization are higher and lower than the APFD values of

Table 10 p values of the Mann–Whitney U test between APFD values of similarity-based prioritization and the default orders of sampling algorithms for the both fault types

FM	Exhaustive faults			Pattern faults		
	CASA	CH	ICPL	CASA	CH	ICPL
BattleofTanks	0.0592	0.3964	0.8460	0.0365	0.1953	0.5368
FM_test	0	0	0	0	0	0
Printers	0.3857	0.8719	0.9377	0.0102	0.9717	0.7806
BankingSoftware	0.4608	0.9562	0.9698	0	0.6868	0.4109
Electronic shopping	0.0273	0.9057	0.8690	0	0.1585	0.1585
DMIS	0.3948	0.1507	0.3139	0.0365	0.1953	0.5368
eCos 3.0 i386pc	*	0.0832	0.0233	*	0.3444	0.4250
FreeBSD kernel 8.0.0	*	0.7966	0.7452	*	0.3824	0.5800
Automotive1	*	0.5787	0.7045	*	0.7394	0.7959
Linux 2.6.28.6	*	*	0.8536	*	*	0.0937
10xAFM15	0.0008	0.4806	0.3982	0	0	0
10xAFM50	0.0004	0.7069	0.0248	0	0	0
10xAFM100	0	0.2197	0.5609	0	0.0274	0.0989
10xAFM200	*	0.8503	0.2297	*	0.0023	0.0950
10xAFM500	*	0.4482	0.8146	*	0.0244	0.0051
10xAFM1000	*	0.0038	0.0540	*	0.03249	0.0009
1xAFM5K	*	*	0.6433	*	*	0.8364

* No result within a whole day of computation

ICPL for six feature models and five feature models, respectively. Except for CASA, the APFD values of our approach are close to the APFD values of sampling algorithms default orders. The reason is that these algorithms are greedy and they cover the feature combinations as fast as possible.

In Table 9, we show the results of comparing our approach to the default order of sampling algorithms using pattern interaction faults. We observed that average APFD values over 100 sets of faults of our approach for all feature models are higher than the average APFD values of sampling algorithms.

Looking closer to the results for each sampling algorithm, we remark that our approach is significantly better than the default order of sampling algorithm CASA for eight feature models. Comparing the default order of sampling algorithm Chvatal to our approach, we observe that our approach is significantly better for seven feature models and, on average, better but not significantly for six feature models. Using sampling algorithm ICPL, the results show that our approach is significantly better for five feature models, and, on average, better but not significantly for eight feature models.

Since our approach is significantly better than random orders and by considering the results of the comparison to the default orders of sampling algorithms (cf. Tables 8, 9), we show that the investigated sampling algorithms have already an acceptable effective order. With *pattern interaction faults*, we notice that the average APFD values of our approach are higher for most feature models than the default orders of sampling algorithms. However, it is not the same situation with *exhaustive interaction faults*. We assume the reason is that the number of deselected features in faults affects the results. The number of deselected features using *exhaustive interaction faults* is more than the number of non-selected features using *pattern interaction faults* (cf. Table 7). Most of these deselected features cannot be detected with the first product, since we select the product with maximum number of features. In response to $RQ2$, although these sampling algorithms have effective orders of products, we show that our approach can be potentially helpful in many cases to increase the rate of early fault detection. In addition, the order of algorithm CASA is non-deterministic for all feature models. For ICPL and Chvatal, we noticed that their orders are non-deterministic for some feature models, such as feature models *BattleofTanks* and *FM_Test*. With our approach, the order is always deterministic.

Results for *RQ3*: *Is the computational overhead required for similarity-based prioritization neglectable compared to the efforts required for sampling?* We computed the average execution time to sample and prioritize products. Then, we compute the percentage of that execution time which is needed to achieve the prioritization process. To accelerate the process, we performed our experiments using two computers. The first one is with an Intel Core i5 CPU @ 3.33 GHz, 16 GB RAM, and Windows 7. On this computer, we run the experiment for the following large feature models: *eCose 3.0 i386pc, FreeBSD kernel 8.0.0, Automotive1, Linux 2.6.28.6, and 1xAFM5K*. The second computer is with an Intel Core i5 CPU @ 3.33 GHz, 8 GB RAM, and Windows 7

Table 11 Percentage of average execution time in second of prioritization to the sampling of each algorithm

Feature model	CASA w / P.	P.%	CH w / P.	P.%	ICPL w / P.	P.%
BattleofTanks	25,493.05	1.16	101.22	83.52	97.65	91.69
FM_Test	9,045.40	0.002	12.63	1.52	4.46	4.20
Printers	680.33	1.02	17.92	34.69	11.06	56.33
BankingSoftware	3,656.32	0.004	15.41	0.97	3.70	4.73
Electronic shopping	3,940.84	0.004	29.30	0.26	2.46	3.01
DMIS**	12,451.14	0.001	49.75	0.21	4.54	2.63
eCos 3.0 i386pc	*	*	1,354.33	0.15	91.16	2.09
FreeBSD kernel 8.0.0	*	*	2,184.73	0.053	110.64	1.09
Automotive1	*	*	66,431.05	4.60	12,225.89	23.68
Linux 2.6.28.6	*	*	*	*	25,102.04	3.94
10xAFM15	1.81	0.08	0.06	5.47	0.10	10.56
10xAFM50	39.31	0.03	0.99	1.58	0.28	5.53
10xAFM100	1,063.02	0.01	6.66	2.01	1.39	10.60
10xAFM200	*	*	31.10	3.19	4.96	21.86
10xAFM500	*	*	1,492.23	1.16	733.31	2.63
10xAFM1000	*	*	4,740.29	7.45	1,196.58	28.66
1xAFM5K	*	*	*	*	45,498.94	7.80

CH Chvatal algorithm, *P* similarity-based prioritization
* No result within a whole day of computation

where we conducted experiments with the remaining feature models.

In Table 11, we show the average execution time over five runs to sample and prioritize products for the listed feature models. In addition, we present the percentage of prioritization time to the computed execution time. In the case of the sampling algorithm CASA, we noticed that the time of prioritization can be neglected since the percentage of prioritization time is less than 1.5% of the computation time to sample and prioritize products. Compared to sampling algorithm Chvatal, the percentages of prioritization time range between 0.053 and 83.52%. For ICPL, the percentages of prioritization time range between 1.09 and 91.69%. From the results, we found only two feature models where the percentage of prioritization is more than 30% of the overall execution time. These feature models are *BattleofTanks* and *Printers*. We investigated these cases, and we found that it occurs when the feature models have no constraints and the number of generated configurations is at least twice as much as the number of features. Thus, the sampling process is computationally cheap. It does not require much time to validate whether these configurations are valid or not. Looking closer to the largest feature models (the Linux kernel and the 1xAFM5K), the prioritization process requires 3.94 and 7.8% of the overall execution time, respectively. In general, as listed in Table 11, the average execution time of similarity-based prioritization for the most cases is small compared to the efforts required for sampling, especially for large feature models.

5.4 Threats to validity

In this section, we discuss the threats to validity of our experiments that may affect our results and explain the steps that we took to mitigate those threats.

Experiment with code base of product lines our implementation of similarity-based prioritization could contain faults itself, which may affect the *internal validity*. To overcome this threat, we applied our algorithm on small product lines and analyzed the results manually. Our implementation and experiment data are publicly available for inspection and reproduction purposes.[3]

A potential external threat is that we acquired an intimate knowledge of the subjected product lines for reproduction purposes. However, as similarity-based prioritization is a push-button approach and only based on selection and deselection of features, this threat cannot affect the reported results. A further potential threat that may affect the *external validity* is related to the nature of the subject product lines. We are not able to generalize that the proposed approach will provide the same results for other product lines. We were restricted to these three product lines, because we are not aware of other product lines publicly available with their source code and faults in their code (i.e., interaction faults) and at the same time, test cases to detect these faults. In particular, results may depend on fault distribution, test coverage,

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

size of the code base, number of features, and number of products. However, all these product lines served as benchmarks previously to evaluate verification strategies of product lines, and they were chosen carefully [7,44].

To reduce these threats, we conducted another experiment with feature models and simulated test execution and fault detection.

Experiments with feature models there is a potential threat that may affect the *internal validity* related to the distribution of artificial faults in the experiments with feature models. We assume that the faults are equally distributed over the features in a product line. This is rather problematic in testing as faults are often found where they are not expected. Still, assuming equally distributed faults seems to be better than to build on non-idealized, but potentially also non-representative distributions.

Another *internal validity* threat regarding the artificial faults is that we defined these faults as valid combinations of features. We assume that, if a combination of features is covered in a configuration, the faults will be detected. However, in practice, faults may be exist, but they cannot be found. With *exhaustive interaction faults*, we assume the equality of proportion of faults for T-wise interaction faults, because we are not aware of a study that reports what is the percentage of each T-wise fault. Although Abal [1] list the number of faults for each pattern, we assume the equality of fault numbers with *pattern interaction faults*, for the aforementioned reasons that other distributions are likely non-idealized distributions. In addition, these pattern faults are based on an analysis of real interaction faults in the Linux kernel, which may raise threats related to the applicability of these patterns to all product lines. Although the distribution of the artificial faults in the experiment with feature models is not realistic, we are now more confident because we got the same results for realistic fault distribution in the experiment with code base of product lines and artificial faults in the experiment with feature model.

Another *internal validity* threat related to the default orders of sampling algorithms is that they are non-deterministic. For instance, different runs of sampling algorithm CASA with the same input often lead to different orders and even different numbers of configurations. In contrast, for Chvatal and ICPL, the results for the same input differ only in slight changes to the order. To reduce these random effects, we conducted those experiments five times and we showed average results of these experiments. Since we compare similarity-based prioritization to random orders, there are also random effects, which we mitigate by 100 repetitions for each run.

A potential *external validity* threat related to the nature of feature models is that similarity-based prioritization may not provide similar results for different feature models. To alleviate this threat, in addition to our running example, we use all feature models in the S.P.L.O.T. repository that have more than 140 features. In addition to 61 artificial feature models of different sizes, we include large feature models consisting of up to 6,888 features. The number of configurations for some feature models is large, for example, using ICPL, the numbers of configurations for *BattleofTanks* and *Automotive1* by ICPL are 459 and 913, respectively.

Another external threat related to the nature of the features in the subject product lines and feature models is that the results cannot be generalized to the non-functional features, since our focus was on functional testing. We plan to further investigate as a future work non-functional features in prioritizing products as it may require an additional effort to model and compute these features.

6 Related work

Sampling testing product lines is a challenging task because a large number of products can be derived. Hence, several approaches have been proposed to select a set of products to be tested [15,34,35]. One of these approaches that have been used to select a subset of products is combinatorial interaction testing [40]. Oster et al. [49] present pairwise testing incorporating feature models. They consider binary constraint solving and forward checking. In addition, they take preselected products (i.e., current products) into account. Perrouin et al. [50] use alloy to generate T-wise products. Shi et al. [56] introduce a compositional symbolic execution technique to analyze product lines. The aforementioned sampling techniques reduce the number of products to test, but they do not systematically consider the prioritization of these products. That is, these sampling strategies may all be combined with similarity-based prioritization to increase the effectiveness of product-line testing.

Product prioritization based on feature selection several approaches have been proposed to prioritize products based on different criteria. Using common feature model metrics, Sánchez et al. [54] propose five prioritization criteria. They compare their effectiveness and observe that different orderings of the same product line may lead to a significant difference in the rate of early fault detection. In their criteria, they do not take the deselected features into account, which, however, is crucial according to our experience [4,5]. In addition, in their evaluation part, they generate T-wise faults where $T \in [1 : 4]$. We propose here two types of faults, one type up to 6-wise faults (i.e., exhaustive faults). The other type of faults (i.e., pattern interaction faults) is based on an empirical study of the Linux kernel [1].

Henard et al. [32] sample and prioritize products at the same time by employing a search-based approach to gen-

erate products based on similarity among them. Moreover, they also propose to prioritize a given set of existing configurations, which can be obtained elsewhere, such as sampling algorithms [31]. Similarity-based prioritization differs from previous work [32], as follows: a) the selection of the initial (all-yes-config) configuration, which is the one with the maximum number of selected features, b) the choice of the distance function (Hamming, an edit-based distance vs Jaccard a token-based one), and c) the way prioritization is computed (e.g., taking the maximum of the minimum distances instead of taking sums into account for a more fine-grained evaluation with respect to a coarse-grained approach adopted by Henard et al. [32]).

Product prioritization using domain knowledge several approaches have been proposed to prioritize products based on additional domain knowledge. Johansen et al. [36] prioritize the product by giving a weight on T-wise interactions based on market knowledge. Having such market knowledge of product lines is often not available. Baller et al. [11] introduce a framework to prioritize products under test based on the selection of adequate test suites with regard to cost and profit objectives. The limitation of this approach is that it requires in advance the set of all products and its relation to test cases and test goals. To tackle this limitation, Baller et al. [10] propose an incremental test suite optimization approach for product-line testing that uses a symbolic representation in terms of feature constraints. However, further experiments are required to evaluate the effectiveness of their approach using real-world product lines. In our approach, we use the similarity among configurations in terms of features as criteria to prioritize them. Sánchez et al. [55] prioritize products based on their non-functional attributes, such as feature size and the number of changes in a feature. These informations are often not given, especially in black-box testing.

Devroey et al. [19] perform statistical analysis of a usage model in order to select the products with high probability to be executed. Ensan et al. [24] prioritize products based on the expectations of the domain experts. Depending on the expectation of the experts, they decide which features are desirable, and as a result, they give a priority to the products that have these desirable features over other products. A drawback of these approaches is that they require analysis of a usage model and expectations of the domain experts to be given a priori. Similarity-based prioritization does not require more domain knowledge than the selected and deselected features of each configuration. That is, we follow a purely model-based black-box approach solely requiring feature selection as a basis. Furthermore, similarity-based prioritization can be applied to other product-line analysis strategies beyond testing, such as statistical analysis, type checking, and theorem proving.

Test-case prioritization in product-line testing, test-case prioritization is used to reschedule test-case executions in order to increase the rate of fault detection. Baller et al. [11] present a multi-objective approach to optimize the test suites by considering certain objectives such as cost of test cases, cost of test requirements, and products. Lachmann et al. [41] propose an integration testing approach for product lines based on delta-oriented test specifications. They reduce the number of executed test cases by selecting the most important test cases for each product. Since these approaches mainly focus on test-case prioritization, combining similarity-based prioritization with them may enhance effectiveness of product-line testing.

In single-system testing, as surveyed by Yoo et al. [64], efforts have been made to prioritize test cases. For instance, Rothermel et al. [53] describe several techniques for using test execution information to prioritize test cases in regression testing. This information includes code coverage and an estimation on the ability of test cases to detect faults. Walcott et al. [62] use a genetic algorithm to prioritize test cases, which can be executed within a given time budget. With respect to T-wise testing, Bryce and Colbourn [12] propose a one-test at-a-time approach to cover more T-way interactions in earlier tests. Yoo et al. [65] prioritize test cases by classifying the test cases into clusters. Then, they prioritize the clusters by utilizing domain expert judgment. In the context of product-line testing, it might be that some of these approaches can be applied to prioritize either the products or the test cases of a product line.

7 Conclusion

Testing product lines takes a considerable amount of time. Testers wish to increase the probability of detecting faults as soon as possible for the product line under test. Hence, several approaches have been proposed to prioritize products, such that earlier products have a higher probability to contain faults. With similarity-based prioritization, we prioritize products based on similarity of their feature selection and deselection. Products are incrementally prioritized among the already tested and the remaining products from a given product set.

We evaluate similarity-based prioritization using three product lines with real faults in their source code. The results show that our approach can potentially accelerate the detection of faults. In addition, we use several sampling algorithms to generate a subset of all valid configurations. We evaluated similarity-based prioritization against random orders, interaction-based order, and default orders of the sampling algorithms CASA [26], Chvatal [17], and ICPL [35]. We performed our experiments on product lines of different size. The results showed that the rate of early fault detec-

tion of similarity-based prioritization is significantly better than random orders. The default orders of existing sampling algorithms already show promising results, which can even be improved in many cases using similarity-based prioritization. Comparing our approach to the interaction-based approach, the results show that the difference between the effectiveness of both approaches, with respect to the APFD, is not significant. However, our evaluation indicates that the interaction-based approach did not scale to models larger than 500 features (already for $T = 2$). That is, we stopped computation after 24 h, whereas similarity-based prioritization finished in all cases within 48 min.

In future work, we plan to investigate the impact of using global optimization algorithms, such as genetic algorithm, in order to minimize the distances. Since similarity-based prioritization is a greedy approach, applying optimization algorithms might be helpful to avoid being trapped in local optima. In the current work, we consider the functional features as defined in the feature model. Hence, we plan to consider the non-functional properties in prioritizing products as it might require an additional effort for computing these properties.

8 Material

The subject product lines and feature models, our implementation of similarity-based prioritization and of a fault generator, as well as the results are available at http://wwwiti.cs.uni-magdeburg.de/iti_db/research/spl-testing/. In addition, similarity-based prioritization and the other mentioned functions of FeatureIDE are part of FeatureIDE, which is available online at http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/.

Acknowledgements We are grateful to anonymous reviewers for their useful comments. We also thank Martin Fagereng Johansen for his support in integrating the SPLCATool into FeatureIDE. We are grateful to Remo Lachmann, Reimar Schröter, Fabian Benduhn, and Hauke Baller for helpful comments on a draft and for interesting discussions. This work has been supported by the German Research Foundation (DFG) in the Priority Programme SPP 1593: Design For Future Managed Software Evolution (LO 2198/2-1).

References

1. Abal, I., Brabrand, C., Wasowski, A.: 42 variability bugs in the linux kernel: a qualitative analysis. In: Proc. Int'l Conf. Automated Software Engineering (ASE), pp. 421–432. ACM (2014)
2. Al-Hajjaji, M.: Scalable sampling and prioritization for product-line testing. In: Proc. Software Engineering (SE), pp. 295–298. Gesellschaft für Informatik (GI) (2015)
3. Al-Hajjaji, M., Thüm, T., Meinicke, J., Lochau, M., Saake, G.: Similarity-based prioritization in software product-line testing. In: Proc. Int'l Software Product Line Conf. (SPLC), pp. 197–206. ACM (2014)
4. Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., Saake, G.: IncLing: efficient product-line testing using incremental pairwise sampling. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE), pp. 144–155. ACM (2016)
5. Al-Hajjaji, M., Meinicke, J., Krieter, S., Schröter, R., Thüm, T., Leich, T., Saake, G.: Tool demo: testing configurable systems with featureIDE. In: Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE), pp. 173–177. ACM (2016)
6. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, Berlin (2013)
7. Apel, S., von Rhein, A., Wendler, P., Größlinger, A., Beyer, D.: Strategies for product-line verification: case studies and experiments. In: Proc. Int'l Conf. Software Engineering (ICSE), pp. 482–491. IEEE (2013)
8. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proc. Int'l Conf. Software Engineering (ICSE), pp. 1–10. ACM (2011)
9. Bagheri, E., Ensan, F., Gasevic, D.: Grammar-based test generation for software product line feature models. In: Proc. Conf. Centre for Advanced Studies on Collaborative Research (CASCON), pp. 87–101. IBM Corp. (2012)
10. Baller, H., Lochau, M.: Towards incremental test suite optimization for software product lines. In: Proc. Int'l Workshop Feature-Oriented Software Development (FOSD), pp. 30–36. ACM (2014)
11. Baller, H., Lity, S., Lochau, M., Schaefer, I.: Multi-objective test suite optimization for incremental product family testing. In: Proc. Int'l Conf. Software Testing, Verification and Validation (ICST), pp. 303–312. IEEE (2014)
12. Bryce, R.C., Colbourn, C.J.: One-test-at-a-time heuristic search for interaction test suites. In: Proc. Int'l Conf. on Genetic and Evolutionary Computation (GECCO), pp. 1082–1089. ACM (2007)
13. Bryce, R.C., Memon, A.M.: Test suite prioritization by interaction coverage. In: Workshop on Domain Specific Approaches to Software Test Automation: Conjunction with the 6th ESEC/FSE Joint Meeting, DOSTA '07, pp. 1–7. ACM (2007)
14. Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., von Rhein, A., Apel, S., Beyer, D.: Facilitating reuse in multi-goal test-suite generation for software product lines. In: Egyed, A., Schaefer, I. (eds.) Fundamental Approaches to Software Engineering, volume 9033 of Lecture Notes in Computer Science, pp. 84–99. Springer (2015)
15. Carmo Machado, I.D., McGregor, J.D., Cavalcanti, YaC, De Almeida, E.S.: On strategies for testing software product lines: a systematic literature review. J. Inf. Softw. Technol. (IST) **56**(10), 1183–1199 (2014)
16. Cartaxo, E.G., Machado, P.D., Neto, F.G.O.: On the use of a similarity function for test case selection in the context of model-based testing. Softw. Test. Verif. Reliab. (STVR) **21**(2), 75–100 (2011)
17. Chvatal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**(3), 233–235 (1979)
18. Cohen, M.B., Dwyer, M.B., Shi, J.: Interaction testing of highly-configurable systems in the presence of constraints. In: Proc. Int'l Symposium in Software Testing and Analysis (ISSTA), pp. 129–139. ACM (2007)
19. Devroey, X., Perrouin, G., Cordy, M., Schobbens, P.-Y., Legay, A., Heymans, P.: Towards statistical prioritization for software product lines testing. In: Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS), pp. 10:1–10:7. ACM (2014)
20. Devroey, X., Perrouin, G., Legay, A., Schobbens, P.-Y., Heymans, P.: Search-based similarity-driven behavioural SPL testing. In: Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS), pp. 89–96. ACM (2016)
21. Dietrich, C., Tartler, R., Schröder-Preikschat, W., Lohmann, D.: Understanding linux feature distribution. In: Proc. of Workshop on Modularity in Systems Software (MISS), pp. 15–20. ACM (2012)

22. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Prioritizing test cases for regression testing. SIGSOFT Softw. Eng. Notes **25**(5), 102–112 (2000)
23. Engström, E., Runeson, P.: Software product line testing—a systematic mapping study. J. Inf. Softw. Technol. (IST) **53**(1), 2–13 (2011)
24. Ensan, A., Bagheri, E., Asadi, M., Gasevic, D., Biletskiy, Y.: Goal-oriented test case selection and prioritization for product line feature models. In: Proc. Int'l Conf. on Information Technology: New Generations (ITNG), pp. 291–298. IEEE (2011)
25. Ensan, F., Bagheri, E., Gasevic, D.: Evolutionary search-based test generation for software product line feature models. In: Proc. Int'l Conf. Advanced Information Systems Engineering (CAiSE), vol. 7328, pp. 613–628. Springer (2012)
26. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir. Softw. Eng. (EMSE) **16**(1), 61–102 (2011)
27. Hall, R.J.: Fundamental nonmodularity in electronic mail. Autom. Softw. Eng. **12**(1), 41–79 (2005)
28. Hamming, R.W.: Error detecting and error correcting codes. Bell Syst. Tech. J. **29**(2), 147–160 (1950)
29. Harrold, M.J.: Testing: a roadmap. In: Proc. of the Conf. on The Future of Software Engineering (FSE), pp. 61–72. ACM (2000)
30. Hemmati, H., Briand, L.: An industrial investigation of similarity measures for model-based test case selection. In: Proc. Int'l Symposium Software Reliability Engineering (ISSRE), pp. 141–150. IEEE (2010)
31. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Traon, Y.L.: PLEDGE: a product line editor and test generation tool. In: Proc. Int'l Software Product Line Conf. (SPLC), pp. 126–129. ACM (2013)
32. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y.L.: Bypassing the combinatorial explosion: using similarity to generate and prioritize T-wise test configurations for software product lines. IEEE Trans. Softw. Eng. (TSE) **40**(7), 650–670 (2014)
33. Jackson, M., Zave, P.: Distributed feature composition: a virtual architecture for telecommunications services. IEEE Trans. Softw. Eng. (TSE) **24**(10), 831–847 (1998)
34. Johansen, M.F., Haugen, Ø., Fleurey, F.: Properties of realistic feature models make combinatorial testing of product lines feasible. In: Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS), pp. 638–652. Springer (2011)
35. Johansen, M.F., Haugen, Ø., Fleurey, F.: An algorithm for generating T-wise covering arrays from large feature models. In: Proc. Int'l Software Product Line Conf. (SPLC), pp. 46–55. ACM (2012)
36. Johansen, M.F., Haugen, Ø., Fleurey, F., Eldegard, A.G., Syversen, T.: Generating better partial covering arrays by modeling weights on sub-product lines. In: Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS), pp. 269–284. Springer (2012)
37. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (1990)
38. Kästner, C., Apel, S., ur Rahman, S.S., Rosenmüller, M., Batory, D., Saake, G.: On the impact of the optional feature problem: analysis and case studies. In: Proc. Int'l Software Product Line Conf. (SPLC), pp. 181–190. Software Engineering Institute (2009)
39. Kramer, J., Magee, J., Sloman, M., Lister, A.: CONIC: an integrated approach to distributed computer control systems. IEE Proc. Comput. Dig. Tech. **130**(1), 1 (1983)
40. Kuhn, D.R., Wallace, D.R., Gallo Jr., A.M.: Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. (TSE) **30**(6), 418–421 (2004)
41. Lachmann, R., Lity, S., Lischke, S., Beddig, S., Schulze, S., Schaefer, I.: Delta-oriented test case prioritization for integration testing of software product lines. In: Proc. Int'l Software Product Line Conf. (SPLC), pp. 81–90. ACM (2015)
42. Lei, Y., Kacker, R.N., Kuhn, D.R., Okun, V., Lawrence, J.: IPOG: a general strategy for T-way software testing. In: Proc. Int'l Conf. Engineering of Computer-Based Systems (ECBS), pp. 549–556. IEEE (2007)
43. McGregor, J.: Testing a software product line. In: Testing Techniques in Software Engineering, pp. 104–140. Springer, Berlin (2010)
44. Meinicke, J., Wong, C.-P., Kästner, C., Thüm, T., Saake, G.: On essential configuration complexity: measuring interactions in highly-configurable systems. In: Proc. Int'l Conf. Automated Software Engineering (ASE), pp. 483–494. ACM (2016)
45. Mendonça, M., Branco, M., Cowan, D.: SPLOT: software product lines online tools. In: Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), pp. 761–762. ACM (2009)
46. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing, 3rd edn. Wiley, New York (2011)
47. Nguyen, H.V., Kästner, C., Nguyen, T.N.: Exploring variability-aware execution for testing plugin-based web applications. In: Proc. Int'l Conf. Software Engineering (ICSE), pp. 907–918. ACM (2014)
48. Northrop, L., Clements, P.: A framework for software product line practice, version 5.0. SEI (2007)
49. Oster, S., Markert, F., Ritter, P.: Automated incremental pairwise testing of software product lines. In: Proc. Int'l Software Product Line Conf. (SPLC), pp. 196–210. Springer (2010)
50. Perrouin, G., Sen, S., Klein, J., Baudry, B., Le Traon, Y.: Automated and scalable T-wise test case generation strategies for software product lines. In: Proc. Int'l Conf. Software Testing, Verification and Validation (ICST), pp. 459–468. IEEE (2010)
51. Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., Le Traon, Y.: Pairwise testing for software product lines: comparison of two approaches. Softw. Qual. J. (SQJ) **20**(3–4), 605–643 (2012)
52. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. (SCP) **41**(1), 53–84 (2001)
53. Rothermel, G., Untch, R., Chu, C., Harrold, M.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. (TSE) **27**(10), 929–948 (2001)
54. Sánchez, A.B., Segura, S., Ruiz-Cortés, A.: A comparison of test case prioritization criteria for software product lines. In: Proc. Int'l Conf. Software Testing, Verification and Validation (ICST), pp. 41–50. IEEE (2014)
55. Sánchez, A., Segura, S., Parejo, J., Ruiz-Cortés, A.: Variability testing in the wild: the drupal case study. Softw. Syst. Model. 1–22 (2015)
56. Shi, J., Cohen, M.B., Dwyer, M.B.: Integration testing of software product lines using compositional symbolic execution. In: Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE), pp. 270–284. Springer (2012)
57. Thüm, T., Kästner, C., Erdweg, S., Siegmund, N.: Abstract features in feature modeling. In: Proc. Int'l Software Product Line Conf. (SPLC), pp. 191–200. IEEE (2011)
58. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Comput. Surv. **47**(1), 6:1–6:45 (2014)
59. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: an extensible framework for feature-oriented software development. Sci. Comput. Program. (SCP) **79**, 70–85 (2014)
60. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., Burlington (2007)

61. Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. In: Proc. Int'l Conf. Automated Software Engineering (ASE), pp. 3–12. Springer (2000)
62. Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.: Time-aware test suite prioritization. In: Proc. Int'l Symposium in Software Testing and Analysis (ISSTA), pp. 1–12. ACM (2006)
63. Weiss, D.M.: The product line hall of fame. In: Proc. Int'l Software Product Line Conf. (SPLC), p. 395. IEEE (2008)
64. Yoo, S., Harman, N.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. (STVR) **22**(2), 67–120 (2012)
65. Yoo, S., Harman, M., Tonella, P., Susi, A.: Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: Proc. Int'l Symposium in Software Testing and Analysis (ISSTA), ISSTA'09, pp. 201–212. ACM (2009)

Mustafa Al-Hajjaji is a Ph.D. student at the Database and Software Engineering Systems Group of Prof. Gunter Saake at the University of Magdeburg, Germany. His research interests are software product-line testing, model-based testing, and mutation testing.

Thomas Thüm is a postdoctoral researcher at TU Braunschweig in Germany. He received his Ph.D. in Software Engineering in 2015 from the University of Magdeburg under the supervision of Prof. Gunter Saake. His Ph.D. thesis received the Dissertation Award 2015 of the University of Magdeburg and his master's thesis the Software Engineering Award 2011 of the Ernst Denert Foundation. He coauthored more than 50 peer-reviewed scientific publications and is a main contributor of the well-known open-source software called FeatureIDE.

Malte Lochau postdoctoral researcher at Real-Time Systems Lab of Prof. Andy Schürr at TU Darmstadt. His research interests are software product-line engineering, model-based testing and formal semantics. His research is part of DFG SPP 1593 part project IMoTEP and DFG SFB 1053 MAKI.

Jens Meinicke is a Ph.D. student at the Otto-von-Guericke-University in Magdeburg, Germany. Since 2011, he develops FeatureIDE as programmer and researcher. In 2014, he received his M.Sc. in Engineering Informatics with distinction at Magdeburg with his thesis written at Carnegie Mellon University in Pittsburgh. Since fall 2016, he works again at CMU as visiting researcher.

Gunter Saake is a full professor of Computer Science. He is the head of the Databases and Software Engineering Group at the University of Magdeburg, Germany. His research interests include database integration, tailor-made data management, database management on new hardware, and feature-oriented software product lines.