A Classification of Product Sampling for Software Product Lines

Mahsa Varshosaz,[1] Mustafa Al-Hajjaji,[2] Thomas Thüm,[3] Tobias Runge,[3]
Mohammad Reza Mousavi,[4,1] and Ina Schaefer[3]
[1] Halmstad University, Sweden [2] Pure-Systems GmbH, Germany
[3] TU Braunschweig, Germany [4] University of Leicester, UK

ABSTRACT

The analysis of software product lines is challenging due to the potentially large number of products, which grow exponentially in terms of the number of features. Product sampling is a technique used to avoid exhaustive testing, which is often infeasible. In this paper, we propose a classification for product sampling techniques and classify the existing literature accordingly. We distinguish the important characteristics of such approaches based on the information used for sampling, the kind of algorithm, and the achieved coverage criteria. Furthermore, we give an overview on existing tools and evaluations of product sampling techniques. We share our insights on the state-of-the-art of product sampling and discuss potential future work.

CCS CONCEPTS

• **Software and its engineering → Software product lines**;

KEYWORDS

Sampling Algorithms, Software Product Lines, Testing, Feature Interaction, Domain Models

ACM Reference Format:
Mahsa Varshosaz,[1] Mustafa Al-Hajjaji,[2] Thomas Thüm,[3] Tobias Runge,[3] Mohammad Reza Mousavi,[4,1] and Ina Schaefer[3]. 2018. A Classification of Product Sampling for Software Product Lines. In *SPLC '18: 22nd International Systems and Software Product Line Conference, September 10–14, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3233027.3233035

1 INTRODUCTION

Software product lines (SPLs) have become common practice for mass production and customization of software systems. In an SPL, products are developed based on a common core. The main goal of using SPLs is to enable systematic reuse in different phases of development by considering the commonalities and variabilities among the products in an SPL [50].

Testing and analysis of software product lines is known to be challenging due to the sheer number of possible products, which makes exhaustive testing and analysis practically impossible. To alleviate this problem, one may resort to product sampling techniques [82] that provide a subset of all valid products. These products are supposed to collectively *cover* the behavior of the product line and hence for example testing them should reveal most faults in all other products.

Several approaches have been proposed for product sampling in the context of software product lines, in order to search the vast space of valid products [28, 56, 66]. For such approaches, a myriad of search algorithms for finding a sample to cover a product line have been proposed, wherethe notion of coverage may also vary from one approach to another. Different algorithms use different types of information sources to find a covering sample. Moreover, the proposed algorithms have typically been evaluated with respect to different criteria and with different degrees of tool support and reproducibility.

We aim for bringing more structure onto the extensive literature on product sampling. In contrast to existing surveys on product sampling [49, 71] or product-line testing [56], we do not follow a systematic process in which all interesting research questions is defined up-front. In contrast, our goal is to provide more insights for readers by means of a detailed classification of existing sampling techniques. We envision that our insights can be used to have a better understanding of such techniques for education and research and also for recognizing their requirements and shortcomings to apply such techniques in practice. To this end, we considered a literature catalog with 48 publications [1–48]. We limited our search to find studies that are focusing on new sampling algorithms [1–38] or evaluations of existing ones [39–48].[1]

Our contributions are threefold. First, we propose a classification for product sampling, involving input data used for sampling, the actual algorithm and achieved coverage, as well as its evaluation (cf. Section 3). Second, we surveyed and classified the literature with respect to the classification (cf. Section 4–6). The list of studies and their classification can be found online.[2] We plan to update this list in the future and welcome any pointers by the community. Third, we identify underrepresented research areas to be addressed by future work.

Our synthesis results indicate that most techniques used problem-space input information, in terms of feature models in generating product samples. Solution space information, such as test artifacts or code coverage, has rarely been used and we think bridging this gap may lead to novel research results. Regarding the developed techniques and algorithms, greedy and evolutionary algorithms have been developed most in this domain. Also, there are no techniques that consider the history of feature models and evolution in software product lines. Regarding evaluation, there are very few

[1]References are sorted by author names but grouped into proposed algorithms, evaluations of sampling algorithms, and other references.
[2]http://thomas.xn--thm-ioa.de/sampling/

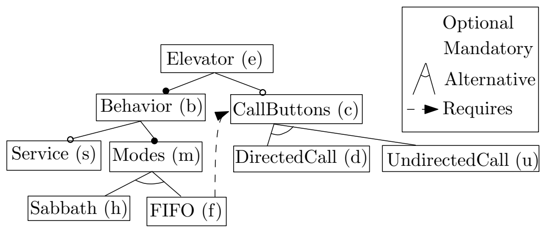

Figure 1: Elevator SPL feature diagram

evaluations on industrial-scale systems and there is a clear need for a benchmark (with different types of information, including feature models, test suites and test results) and agreed-upon metrics for efficiency and effectiveness. We elaborate on these observations in the remainder of the paper.

2 MOTIVATING EXAMPLE

In software product line engineering, the variabilities and commonalities among the products are described in terms of *features.* A feature is defined as a user-visible aspect or characteristic of a software system [62]. Features in a product line have different relations that can be described compactly by means of *feature models* [54], which can be graphically represented as a *feature diagram.* As an example, Fig. 1, represents the feature diagram of an Elevator product line. (This is a simplified version of the example provided by Krieter et al. in [64].) Each node in this diagram represents a feature. There are some features in this diagram such as *Behavior* and *Modes* that are mandatory. This means that any elevator in this product line should include these features. There are also optional features such as the feature *Service*, which is used to facilitate the maintainability of the elevator, that are optional, which means that there can be valid products with or without this feature. Additionally, there are different types of relationships between sub-features. For example, *Sabbath* and *FIFO* that are two possible modes for an elevator have alternative relationship. This means an elevator in this product line can only include one of these modes. Furthermore, a feature can require or exclude other features. For example, in an elevator with *FIFO* mode, there should be a button for the user inside the elevator and in each floor. Hence, there is a requirement relation between the *CallButton* and *FIFO* features. This relation is represented by the dashed arrow in Fig. 1. The products in a product line can be described as subsets of features. As an example the elevator product line with feature diagram represented in Fig. 1, consists of 10 valid products.

In Listing 1, we represent a part of the code related to the control unit of the elevator product line. (The code is a very simplified version of the one given in [72]. The code is in Java and preprocessing directives are used to add variability.) In lines 2–11, a set of variables are defined that are used by methods and based on the set of selected features. In this code there is a main method, called *run()* (lines 12–20), which triggers the execution of the functions that embody the main parts of the functionality of an elevator. Based on the code, in each step the next state of the elevator and its direction is calculated based on the current state and the mode of the elevator. This is done using methods *calculateNextState()* and *setDirection().* As shown in lines 21–31, the calculation of the next state can be done in different ways based on the features included in a product. The preprocessing directives are used to separate the parts of the code related to each feature. The execution of these parts is depending on the features included in a product.

Assume that the goal is to test the behavior of the products in the elevator product line. A subset of these products is selected as a representative to be tested. The products can be selected in a random manner or with regards to a specific criteria. The quality of the selected set can be specified considering different measures, e.g., code coverage or the number of faults that are revealed by testing the sample set. Consider two sample sets $S_1 = \{\{e, b, m, c, s, h, d\}, \{e, b, m, c, s, h, u\}, \{e, b, m, c, s, f, d\}\}$, and $S_2 = \{\{e, b, m, h, s\}, \{e, b, m, f, c, d\}, \{e, b, m, f, c, u\}\}$; the size of the two sets is the same but, testing the the products in S_2 results in revealing a compile error which is resulted from interaction of two features *FIFO* and *UndirectedCall.* This is due to the inclusion of a call in line 29, to a method which is not included in the code in case that feature *UndirectedCall* is selected (the implementation of the method in line 39 for this feature). As another example consider another possible sample set $S_3 = \{\{e, b, m, s, h, \}, \{e, b, m, c, s, h, u\}\}$. Using S_3 for testing will not reveal the existing fault (null pointer reference) in line 8 in the code. In order to reveal such a fault the feature *DirectedCall* should be included at least in one of the products in the sample set.

3 CLASSIFICATION OF PRODUCT SAMPLING

In this section, we give an overview of our classification of product sampling. The set of publications considered in this classification has been reached after several iterations. We first performed a targeted search and produced a sample of the key results in the field and we started with an initial classification given our experience in this area [2, 3, 39, 40, 81, 82] and adapted it in the process of studying the literature. Consequently, we had to repeatedly reclassify surveyed publications. Similarly, we expanded our studied literature using existing surveys [49, 56, 71]. We reached our literature catalog with 48 publications [1–48] by filtering out irrelevant publications and adding others by means of snowballing. In particular, we have limited our search to studies that are focusing on sampling algorithms that incorporate a feature model or evaluations of existing algorithms. In order to give a structure to our classification, we identified three research questions as follows.

(RQ1) What type of data is used as input for product sampling?
(RQ2) How are sample sets selected?
(RQ3) How are the sampling techniques evaluated?

Considering the above questions, we describe three main characteristics of product sampling approaches, namely, *input data*, *sampling technique*, and the *evaluation.* We refined these main characteristics iteratively by investigating the more fine-grained characteristics of the studied papers. The details about these three main characteristics and inferred sub-characteristics are explained in the following.

```
class ControlUnit {
  Elevator elevator;
  ElevatorState state, nextState;
  //#if FIFO
  Req req = new Req();
  //#elif DirectedCall
  Req req = new UndReq(this);
  Req dreq = new DirReq(null);
  //#else
  Req req = new Req(this);
  //#endif
  void run() {
    while (true) {
      calculateNextState();
      setDirection(nextState);
      //#if DirectedCall
      sortQueue();
      //#endif
    }
  }
  void calculateNextState() {
    //#if Sabbath
    if (sabbathNextState()) return;
    //#endif
    //#if Service
    if (serviceNextState()) return;
    //#endif
    //#if FIFO
    callButtonsNextState(dreq);
    //#endif
  }
  //#if Service
  boolean serviceNextState() {...}
  //#endif
  //#if Sabbath
  boolean sabbathNextState() {...}
  //#endif
  //#if DirectedCall
  void callButtonsNextState(Req d)
       {...}
  void sortQueue() {...}
  //#endif
}
```

Listing 1: Simplified implementation for Elevator product line given in Fig. 1

3.1 Input Data for Product Sampling

The *input data* is one of the considered characteristics that is specified to address the types of input that sampling approaches exploit.

We consider two main sub-characteristics for *input data*, namely, *problem space* and *solution space*. In general, the problem space concerns the system requirements and specification handled during the domain analysis and the solution space refers to the concrete artifacts that are created through design and implementation process. The problem space includes two sub-characteristics that are *feature model* and *expert knowledge*. We also refine the *solution space* characteristic into two sub-characteristics, namely, *test artifacts* and *implementation artifacts*. These characteristics are represented in Fig. 2 and are explained with more details in the following.

3.1.1 Feature Model. As mentioned in Section 2, feature models represent different relations among features in a product line. Hence, feature models can be used to recognize valid products from invalid ones throughout the sampling process. According to feature diagram in Fig. 1, a sample set that contains a configuration including both features *Sabbath* and *FIFO* is not valid as these two features are in an alternative relation. The feature model can be also considered as a kind of expert knowledge, we separate this sub-characteristic, as many studies use feature models as a part of information that they use.

3.1.2 Expert Knowledge. In general, expert knowledge is the knowledge about the characteristics of the environment that the system operates in. Recognizing important feature interactions and generation of products based on that can be considered as a case for which expert knowledge is used. As an example, consider the elevator product line given in Section 2. An expert might know that the products that include a combination of features *Sabbath*, *Service* and *DirectedCall* are more prevalent and in demand in the market. Then the products containing combinations of these features can be included in the sample set to make sure that they are analyzed.

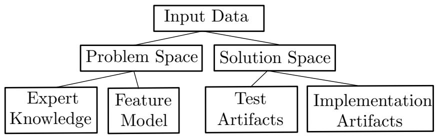

Figure 2: Input data for product sampling

3.1.3 Implementation Artifacts. This is another sub-characteristic of input data used by some of the covered techniques during the sampling process. Implementation artifacts, such as, source code and models, can be used by sampling techniques in order to meet one or more criteria that are considered during the sampling process (e.g., a sample set guarantees a certain code coverage). For example, the code given in Listing 1 is considered as a part of the implementation artifacts for the elevator product line.

3.1.4 Test Artifacts. With *test artifacts*, we refer to all artifacts that are developed and produced during different phases of the testing process. (See the example given at the end of Section 2.)

3.2 Techniques for Product Sampling

While there are different inputs are necessary for sampling, there are also different principal techniques for computing sample sets. we distinguish four main sub-characteristics for the *technique* characteristic, namely, *automatic selection*, *semi-automatic selection*, *manual selection* and *coverage*, which are represented in Fig. 3 and explained in the following.

3.2.1 Automatic Selection. We consider the two following general types of automatic selection approaches:

Greedy: In greedy algorithms, a locally optimal choice is made in each stage of the problem solving [59]. In these algorithms, in each step, a new member is added to the sample set that is the best choice considering the current sample set and the defined criteria for sampling and the process continues with the resulting set. Consider the Elevator product line. Assume that in the current step the sample set is $\{\{e, b, m, h\},\{e, b, m, h, s\}\}$ and that the criterion considered throughout the sampling process is feature interaction (Feature interaction is a software engineering concept which addresses the occurrence of changes in the behavior of the system related to a feature in presence and absence of other features). Then, the solution in the next step after applying a greedy algorithm can be $\{\{e, b, m, h\},\{e, b, m, h, s\},\{e, b, m, f, d, c, s\}\}$ since a new set of interactions among features f and d is covered.

Meta-Heuristic Search: The problem of finding a representative subset of products in a product line can be formulated as an optimization problem. Meta-heuristic algorithms aim at selecting a subset of products as an optimal solution for this problem using computational search in a configuration space. The search space can be potentially large due to high variability and complex feature combinations in an SPL. Many meta-heuristic approaches have been inspired by biological evolution [8]. Based on whether the meta-heuristic searches operate on individual states or population of states [11], the meta-heuristic approaches are divided into the following: *(a) Local search* and *(b) Population-Based Search.*
(a) Local search: Such approaches start with a preliminary set of products as a solution for the optimization problem and the search algorithm will gradually evolve the current set of products to reach a near optimal solutions. Examples of such approaches are simulated annealing and tabu search.
(b) Population-Based Search: Such approaches start with a preliminary set of sample sets (sets of products) and then the algorithm will evolve the current sets of products to reach a final solutions. Examples of such approaches are genetic algorithms and swarm techniques. In such approaches the primary set of solutions are mutated and recombined into new sets in order to find a near optimal solution. A fitness function is usually used as a measure for evolving the set of solutions during the process. As a general example consider the elevator product line and the code given in Listing 1. Assume that in the current step of the sampling process the generated solution consist of two sets, namely S_1 and S_2, each with one member $\{S_1 = \{e, b, m, h\}, S_2 = \{e, b, m, f\}\}$. In the next step the the intermediate solution can be evolved to $\{S_1 = \{\{e, b, m, h\}, \{e, b, m, h, s\}\}, S_2 = \{\{e, b, m, f\}, \{e, b, m, f, c, d\}\}\}$. Considering different criteria one of these sets can be used to continue the sampling based upon.

3.2.2 Semi-Automatic Selection. In semi-automatic selection, different type of data such as the required number of products to be generated, the time for sampling, and a degree for coverage e.g., coverage on feature interactions can be considered.

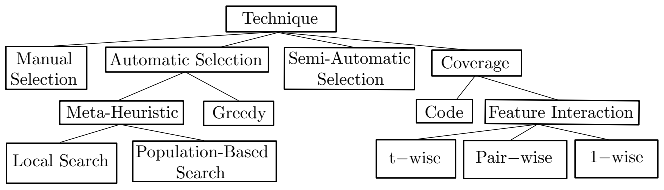

Figure 3: Technique of product sampling

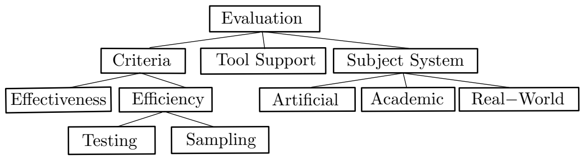

Figure 4: Evaluation of product sampling

Furthermore, in such techniques, the full sample set or a primary set that is generated by other sampling techniques can be used as a starting point for sampling. An expert can provide the set based on information, such as feature model and domain knowledge.

3.2.3 Manual Selection. The set of products can be selected manually. In this approach a set of products are selected by a domain expert and based on the knowledge that they have about the possible and common feature combinations.

3.2.4 Coverage. Coverage criteria are often used to assure the quality of product sampling. One widely used criterion is *feature interaction* coverage. Considering this criterion, the main goal during the sampling process is to provide coverage for different kinds of feature interactions such as feature-wise (aka. 1-wise), pair-wise (aka. 2-wise), and t-wise. As an example, when considering the pair-wise coverage criterion, all the valid/possible pairs of features should be covered by configurations in the sample set. A common mean for extracting the sample set based on the feature interactions is a covering array [19]. A t-wise covering array is a subset of products that covers all the valid t-wise feature combinations in the product line. A covering array is commonly represented using a table where each row represents a feature and each column represents a product. As an example consider the elevator product line given in Section 2. A sample set that fulfills the pair-wise criterion for this example is $\{\{e, b, m, h, s\},\{e, b, m, h, c, d\},\{e, b, m, s, h, c, u\}, \{e, b, m, f, s, c, d\},\{e, b, m, f, c, u\}\}$.

Code coverage coverage. Considering this criteria, the code should be covered with some percentage using the sample set. As an example, consider the code for the elevator product line given in Listing 1. One notion of code coverage could be that each ifdef block should be included at least once as a part of the code of the products in the sample set. As an example a sample set such as $\{\{e, b, m, f, c, d\}, \{e, b, m, h, s, u\}\}$ will cover all the ifdef blocks in the implementation given in Listing 1 at least once.

Additionally, there are some techniques which do not consider any notion of coverage during the sampling process. Hence, we address this in the classification of the selected studies as well (not as an explicit characteristic).

3.3 Evaluation of Product Sampling

Evaluation is another high-level characteristic in our classification. This characteristic mainly addresses the artifacts and the process taken to evaluate the sampling techniques. Furthermore, we refine this characteristic into three sub-characteristics, namely, *tool support*, *evaluation criteria*, and *subject system*, which are represented in Fig. 4, in addition to a set of sub-characteristics which are explained in detail in the following.

3.3.1 Evaluation Criteria. The evaluation of sampling techniques is performed by considering several criteria. Two major criteria that are recognized in our classification are *efficiency* and *effectiveness*. By considering different experiments we observe that several interpretations of efficiency and effectiveness are provided. As for the efficiency, the criterion can be addressing the efficiency of the sampling technique, which is related to the computation of the sample set or the efficiency of the testing technique that is used combined with the product sampling. An example of sampling efficiency is the time to generate the sample set. Also, the efficiency of the testing technique can be measured in terms of the sample size (the size of the sample set can affect the required time for testing) and the testing execution time. On the other hand, effectiveness addresses the quality of the sample set mainly with regards to a notion of coverage, e.g., fault coverage, feature interaction coverage.

3.3.2 Subject System. In this classification, we consider the subject systems and the case studies that are considered during the evaluation of the sampling technique. The type of subject systems can be an indication of the practicality and the scalability of the technique. In our classification, we classify three types of subject systems, namely artificially generated, which are subject systems generated by random combination of a set of features or using a program with regards to specific rules, academic, which are small subject systems mostly used as examples in academic work and provided by researchers, and real-world, which are systems that are used in practice. As an example the elevator product line example given in Section 2, can be considered as an academic subject system. Furthermore, another factor that we consider about the subject systems is the size of the corresponding feature model.

3.3.3 Tool Support. Another sub-characteristic in our classification is the *tool support*, which we use to indicate whether the sampling technique is supported by an implementation. We also specify if the tool set is open source and/or available for public use. We distinguish these characteristic since this type of information can be useful for users who are interested in application of sampling techniques in practice.

4 CLASSIFICATION BASED ON INPUT DATA

Before discussing how sampling techniques work internally, we answer **RQ1** by giving an overview which input these techniques require to work (cf. Section 3.1). This input typically needs to be available for them to work (e.g., provided by the user).

Feature Model. All sampling techniques that we surveyed take the feature model as input, due to the scope of the survey. Sampling techniques rely on the feature model to distinguish valid from invalid configurations, as only valid configurations are desirable for many applications of sampling. To check validity, feature models are often translated to boolean satisfiability problems (e.g., [2, 21, 46, 78]) or to constraint satisfaction problems (e.g., [29, 46]) to make use of dedicated and efficient solvers. In some cases, the feature model is translated into the set of all valid configurations [5, 6, 26], which does not scale to large product lines. There are also sampling techniques that take advantage of the hierarchy in the feature model (i.e., the feature diagram) [4, 14, 31, 35] or support numerical attributes in feature models [12, 43]. In particular, Reuling et al. and Arcaini et al. apply typical change operations on the hierarchy to immitate faults introduced by domain engineers [4, 31]. If no feature model is available, it is often sufficient to have the list of features and take a tautology as input to the sampling algorithm, which basically means that there are no constraints on the features.

Expert Knowledge. While feature models are a common input to sampling, there are further sources of domain knowledge being used as input. A largely manual, but quite common technique in practice is to let domain experts identify a set of typical products manually. For example, changes to the Linux kernel are typically sent to the mailing list and a continuous integration server applies each patch automatically to compile and test it with a set of ten pre-defined and ten randomly selected products. Besides this continuous integration, Linux developers often only test the kernel with the all-yes-config, a configuration in which almost all features are selected [37, 60]. Many other product lines come with a default configuration being sufficient for many users [24]. Default configuration and all-yes-config are instances of single-configuration heuristics [24]. Multiple pre-defined configurations are supported when compiling or testing products in FeatureIDE [3, 72].

Oster et al. were the first to compute a sample based on a set of pre-defined products [29]. That is, users provide some products as input, which will be in the sample, and the sampling technique extends this set towards a representative set of products. With IncLing, Al-Hajjaji et al. presented a further technique for that purpose [2]. However, those pre-defined products are optional for both sampling techniques. Whereas all those discussed techniques include pre-defined configurations, Johansen et al. allow to define partial configurations (i.e., a subset of the product line) that are used for two purposes [20]. First, to rule out configurations that are valid according to the feature model but not expected for the sample. Second, to assign weights to those partial configurations to distinguish more relevant from less relevant configurations. Ensan et al. enable experts to also rule out certain configurations, but in a much simpler manner. They let experts define a subset of all features defined in the feature model, which is then used for sampling [7]. Similarly, Kowal et al. propose to filter feature models based on priorities assigned to features and known feature interactions, such as shared resources and data [23]. Henard et al. allow experts to specify test costs for each feature, which is then used in the objective function to better estimate the testing effort for specific configurations [17].

Implementation Artifacts. Whereas most sampling techniques are only based on problem space information, there are a few instances that also consider solution space information. Tartler et al. propose a sampling technique for product lines implemented with preprocessors [38]. They analyze the source code to ensure that every lexical code fragment (i.e., #ifdef block) is included in at least one sample product. This technique is also an instance of a code-coverage heuristic [24]. In contrast, Shi et al. use control flow graphs to identify which features can interact with each other [33]. An underlying and quite restrictive assumption of their work is that each feature is implemented by a single method and this method is called only once from a common code base. Whereas both sampling techniques are rather different, both use implementation artifacts as input to reduce the search space defined by the feature model.

Test Artifacts. Similar to implementation artifacts, also test artifacts have been used as input for sampling [21, 22]. Kim et al. proposed to exhaustively consider all configurations, for which a unit test [21] or a runtime check [22] can lead to different results. In particular, they use test and implementation artifacts to detect which features can influence the result of the test. Then they produce a sample for each unit test or runtime check, which covers all combinations of those features, whereas they exclude combinations that are invalid according to the feature model. In contrast to all other sampling techniques, which derive one sample for a product line, Kim et al. derive a separate sample for each test.

Summary and Insights. Without even going into detail how the sampling algorithms work, we have already noticed a large diversity in terms of the input used for sampling. In Table 1, we give an overview all surveyed algorithms and their classification. All surveyed sampling techniques use the feature model to distinguish valid from invalid configurations. Whereas feature models represented as propositional formulas are sufficient to feed them into a SAT solver, there are recent approaches that also incorporate the hierarchy in addition [4, 14, 31, 35]. Other sampling techniques can incorporate further domain knowledge from experts in terms of pre-defined or partial configurations [2, 3, 7, 17, 20, 23, 24, 29, 37, 38]. Besides those inputs, domain knowledge has also been extracted from implementation artifacts [21, 22, 33, 37, 38] and test artifacts [21, 22]. Input besides the feature model is typically used to further restrict the set of valid configurations or derive more realistic samples.

While our survey identified numerous interesting inputs for sampling, the vast majority of techniques only consumes the feature model and cannot incorporate any further input. The advantage of those sampling techniques is the better applicability. There is no need to have further domain knowledge or access to implementation and tests. Also, those techniques are completely independent of the variability mechanism used for domain artifacts and do not require experts. However, from an algorithm point-of-view it is possible that with more input, we can produce better samples with fewer resources.

Here are some future research directions that we identified with respect to further input. First, there is not a single technique incorporating the history of the product line. The history of the feature model may reveal new features or combinations, whereas changes to the pre-defined configurations or domain artifacts indicate what should be covered. While there are numerous techniques for SPL regression testing [56, 66], they typically take a fixed sample for all versions or do not use sampling. There are even applications that require the computation of samples being stable over the history. For example, if we aim to analyze the performance of a product line over time, we should probably not consider a completely different sample for each revision. Second, already known feature interactions derived with static analyses or even defects or vulnerabilities occurred in the past (e.g., documented in issue trackers) may enhance samples even further. Third, we could try to include especially those configurations that were outliers before (e.g., configurations with most defects as well as fastest and slowest configurations). Finally, requirements as well as informal or formal specifications can be used as input for sampling.

	Input Data				Algorithm					& Coverage						Evaluation &						Application				
Publication	feature model	expert knowledge	implementation artifacts	test artifacts	greedy	local search	population-based search	manual selection	semi-automatic selection	feature-wise coverage	pair-wise coverage	t-wise coverage	code coverage	requirements coverage	no coverage guarantee	sampling efficiency	testing efficiency	effectiveness	unavailable tool	available tool	open-source tool	evaluation	testing	type checking	data-flow analysis	non-functional properties
[1]	●				●										●		●	●				●	●			
[2]	●	●			●				●		●					●	●	●			●	●	●			
[3]	●	●						●		●											●		●	●		
[4]	●				●										●	●	●	●	●			●	●	●		
[5]	●				●				●	●				●		●	●	●	●			●	●			
[5]	●					●			●						●	●	●	●	●			●	●			
[6]	●						●								●			●	●			●	●			
[7]	●	●			●				●						●		●					●	●			
[8]	●					●									●		●	●			●	●	●			
[9]	●						●								●	●		●	●			●	●			
[10]	●						●								●		●	●	●			●	●			
[11]	●					●						●				●	●				●	●	●			
[12]	●				●										●		●	●			●	●				●
[13]	●				●										●		●				●	●				●
[14]	●				●							●				●	●		●			●	●			
[15]	●					●									●	●	●	●			●	●	●			
[16]	●					●									●	●	●	●				●	●			
[17]	●	●					●		●						●	●	●	●			●	●	●			
[18]	●				●							●				●					●	●	●			
[19]	●				●							●				●	●				●	●	●			
[20]	●	●			●				●						●		●	●			●	●	●			
[21]	●		●	●	●								●			●	●		●			●	●			
[22]	●		●	●	●								●			●	●		●			●	●			
[23]	●	●			●							●				●	●					●	●			
[24]	●	●						●							●	●	●				●	●		●	●	
[24]	●				●										●	●	●				●	●		●	●	
[25]	●						●								●	●	●	●			●	●	●			
[26]	●					●					●					●	●	●				●	●			
[27]	●						●								●		●	●	●			●	●			
[28]	●				●										●		●	●			●	●	●			
[28]	●				●										●		●	●			●	●	●			
[29]	●	●			●				●		●					●	●		●			●	●			
[30]	●				●							●					●	●	●			●	●			
[31]	●				●										●	●	●	●		●		●	●			
[32]	●				●					●							●	●			●	●				●
[33]	●		●		●							●	●			●			●			●	●			
[34]	●				●										●		●	●			●	●				●
[35]	●				●										●		●	●			●	●				●
[36]	●				●					●							●	●			●	●				●
[36]	●				●						●						●	●			●	●				●
[37]	●		●		●										●			●			●	●		●		
[37]	●	●						●							●			●				●		●		
[38]	●	●						●							●	●		●			●	●		●		
[38]	●		●		●								●			●		●			●	●		●		
[39]																		●				●	●			
[40]																●	●				●	●	●			
[41]																	●	●				●	●			
[42]																●	●	●				●	●			
[43]																	●	●			●	●				●
[44]																	●	●				●	●	●		
[45]																	●	●	●			●	●			
[46]																●	●					●	●			
[47]																	●	●				●				●
[48]																	●	●				●				●

Table 1: Overview on the literature on product sampling, grouped into algorithms [1–38] and evaluations [39–48].

5 CLASSIFICATION BASED ON TECHNIQUE

To answer **RQ2**, we present in this section a classification of the selected studies based on the technique used for product sampling.

Greedy Techniques. Several greedy sampling algorithms have been used to sample products in product lines [1–5, 7, 14, 18–24, 28, 29, 31, 33, 37]. In the following, we explain the main greedy sampling techniques that are addressed in the above studies.

Chvatal is a greedy algorithm that has been proposed to generate a minimal set of configurations [18]. With Chvatal, the combinations of features are generated to be considered during the sampling process. The configurations are added to the sample set in a greedy manner and each newly added configuration should cover at least an uncovered combination. Similarly, the ICPL algorithm is introduced for generating covering arrays for large scale feature models [19]. In fact, the ICPL algorithm is built upon the Chvatal algorithm [18] with additional performance improvements, such as parallelizing the sampling process. Similar to Chvatal, the ICPL algorithm receives a feature model and a coverage strength t as input. Then it generates a covering array that produces the t-wise coverage as output. MoSo-PoLiTe (**Mo**del-based **So**ftware **Pro**duct **Li**ne **Te**sting) is another greedy algorithm that has been proposed to generate a set of products using feature models [29]. In MoSo-PoLiTe, the pair-wise combinations are extracted. The algorithm starts with a pair of features and incrementally adds the rest of pairs by applying forward checking to check whether the selected pair can be combined with the remaining pairs of features to generate valid products. Moreover, MoSo-PoLiTe considers the pre-defined products in the sampling process. Al-Hajjaji et al. [2] propose an algorithm, called IncLing, where the products are generated incrementally one at a time. Similar to MoSo-PoLiTe [29], IncLing considers the set of products that are already selected and tested. This algorithm aims to increase the diversity among selected products by choosing dissimilar pair-wise feature combinations to be covered in the next product in order to increase the possibility of fault detection [2, 3].

A divide and conquer strategy is used in [30] to generate t-wise combinations from a feature model. Using the divide and conquer technique, a set of sub-problems is given to a constraint solver which are then solved automatically instead. In this approach the features in the feature models are translated to Alloy specifications [30]. A set of products then are generated using the resulting model which provide t-wise coverage.

To select optimal products with respect to the non-functional properties of product line, several approaches have been proposed [32, 34, 35, 35, 36]. For instance, Siegmund et al. [35] propose an approach that predicts *footprint* and *main-memory consumption* of products. In their work, they generate a small set of products to approximate the influence of features. In particular, they consider feature-wise sampling, where a product for each feature is generated, to measure the influence of each feature separately, interaction-wise sampling, where a product for each interacting features that are given based on domain knowledge, is generated, to measure the influence of feature interaction, and pair-wise sampling, where a product for each pair of features is generated, to measure the influence of the pair-wise feature interaction. In the same direction, Sarkar et a. [32] predict the performance of configurable systems by adapting sampling strategies that generate samples with the aim of estimating the learning behavior of the prediction model as well as the cost of building, training, and testing it. In particular, they adapt progressive sampling strategy, where in each iteration a fixed set of sample is added to the training set, and projective sampling strategy, where the learning behavior is approximated using a minimal set of initial samples. To do so, they exploit the feature frequencies (i.e., how often the features appear in the current set of samples) to generate the initial set of samples for both sampling strategies.

To reduce the number of products to be tested, several greedy search-based reduction techniques have been proposed [7, 14, 20–23, 33]. For instance, Kim et al. [21] propose a technique, where the features that do not have any impact when the test is performed are removed. Similarly, a compositional symbolic execution technique has been proposed to reduce the number of generated products by identifying the possible interaction between features [33]. Another algorithm is proposed where a product line is divided into sub-product lines with weights given by domain experts [20]. Then, the products that have more weights are selected to be tested first. Similarly, Ensan et al. [7] propose to reduce the size of the sample set by covering the most desirable features, which are usually given by the domain experts. Haslinger et al. [14] propose reduction rules based on the constraints in feature models, which applies during the sampling and results in reducing the number of feature combinations that are used to generate the set of configurations. In the same direction, Kowal et al. [23] propose to cover only features that interact with each other in the generated sample. They propose to model the additional information about the interaction between features into the feature models. Thus, the number of generated products is reduced as a result of reducing the number of combinations of features that are required to be covered.

The aforementioned greedy techniques sample products based on different criteria, such as coverage [2, 19, 29]. However, additional sampling algorithms have been proposed [1, 3, 8, 15, 24, 37]. For instance, random sampling, which can be considered also a greedy technique, generate a set of products randomly [3, 8, 15, 24]. In this technique the configurations that are not valid according to the feature model are discarded. However, with random sampling, no specific coverage criteria through the generation of products can be guaranteed. Most-enabled-disabled (i.e., configurations where most of features are enabled or disabled), all-most-enabled-disabled (i.e., all possible configurations where most of features are enabled or disabled), one-disabled (i.e., configurations where all features are enabled except one), all-one-disabled (i.e., configurations where one feature is disabled and the others are enabled), one-enabled (i.e., configurations where all features are disabled except one), all-one-enabled (i.e., configurations where one feature is enabled and the others are disabled) are sampling techniques in which special combinations of features are obtained by enabling/disabling features [1]. The sample sets can be computed in a greedy manner using these techniques. Several mutation-based approaches have been exploited to sample products [4, 15, 31]. These approaches aim to generate products that have a high probability of containing faults.

Several meta-heuristic approaches have been proposed to sample product lines [5, 6, 8–10, 15, 17, 25, 27]. In this subset of studies, an initial set of products is selected randomly and then gradually evolves by considering the constraints in the optimization problem. Usually the size of the initial set is given by testers. Based on the search process of finding solutions, we classify these approaches into *local search* and *population-based search* (cf. Section 3)

Local Search Techniques. Several local search techniques have been proposed to sample products [5, 8, 11, 15, 16, 26]. For instance, the 1+1 evolutionary algorithm has been used to sample products [15, 16]. With 1+1 evolutionary algorithm, the main evolution operator is the mutation. In the fitness function, they consider the similarity between products. Their goal is to generate products that are dissimilar to each other with respect to the selected features in each product [15]. CASA is a local search technique that uses simulated annealing to generate a set of products, which achieves a certain degree of t-wise coverage. In this approach, in the first step, the number of configurations are minimized and in the next step it is ensured that a certain degree of coverage is achieved [11]. Ensan et al. [8] propose a meta-heuristic approach to generate optimal products with respect to some criteria, such as feature coverage. While it can be seen as a population-based technique as it considers genetic algorithm, we consider it in this paper as a local search technique, as it operates on individual states. Marijan et al. [26] propose an optimization technique to generate a set of products that achieves pairwise coverage. In particular, the algorithm first, transforms the feature models into an internal constraint model. This model is expressed as a matrix, where columns represent features and rows represent configurations. Given this matrix, an optimization algorithm is proposed to generate a minimal set of products for a given time.

Population-Based Techniques. Several population-based algorithms have been used to sample products [6, 8–10, 17, 25, 27, 41]. Multi-objective evolutionary optimization is recognized among evolutionary approaches for sampling. In such approaches, the product sampling is defined as a multi-objective optimization problem and multi-objective algorithms are used for solving the problem [9]. In such techniques, the objective function is usually defined based on a coverage criteria [25, 27]. Genetic algorithms, which are extensively used, fall into evolutionary techniques category. These algorithms form a correspondence between the genetic structure of chromosomes and the representation of solutions in the optimization problem [8, 17]. Considering this correspondence, the solutions of the optimization problem (i.e., the set of configurations) are represented as chromosomes, and hence by using natural biological evolution operators, such as mutation and crossover, a new near optimal solution can be generated. The Multi-Objective Evolutionary Algorithm based on Decomposition (MOEA/D) [6] is another evolutionary algorithm which decomposes a multi-objective optimization problem into a set of scalable optimization sub-problems and solves them simultaneously [6]. MOEA/D-DRA is an extension of this algorithm that enables dynamic resource allocation [6].

Semi-Automatic Selection Techniques. Oster et al. [29] and Al-Hajjaji et al. [2] consider pre-defined configurations, which can be given by domain experts, in the sampling process. One semi-automatic sampling technique is the single configuration technique where a single configuration is selected as a representative for all the products to be analyzed [24]. This configuration is typically selected by a domain expert, where usually part of the sampling is performed automatically, such as checking the satisfiability. One disadvantage of this technique is that mutually exclusive behavior/code can not be covered by one selected configuration. Note that the other sampling techniques, where the user can be involved in the sampling process, can also be classified as semi-automatic techniques.

Manual Selection Techniques. Several approaches also support manual sampling, where domain experts are usually generate products based on domain knowledge [3, 24, 37]. All-yes-config is a sampling approach, where all the possible features are selected to be included in a product [37, 38]. The main limitation of using this approach is that the selected configuration can include alternative behavior which is not valid according to the constraints between features. Note that the generated all-yes-config may not be the optimal configuration due to the dependencies between features which may affect the sampling process. Another manual selection approach , where a set of configurations in Linux is selected iteratively, is used [38]. In particular, they aim to add more blocks of the code by additionally considering the constraints between the blocks.

Feature Interaction and Coverage. Several greedy algorithms generate a set of products that guarantee a certain degree of coverage. For instance, the following algorithms achieve 1-wise [3, 5, 32, 36]. While some of the algorithms achieve the pair-wise coverage [2, 26, 29, 36], other techniques scale to t-wise interaction coverage [11, 14, 18, 19, 23, 30, 33]. Except for the work of Garvin et al. [11], where they cover up-to 6-wise coverage, most of the sampling techniques that consider meta-heuristic algorithms do not guarantee 100% of a coverage degree [6, 8–10, 15–17, 25, 27, 41]. Note that the random-based techniques [3, 8, 15, 24], semi-automatic selection techniques [2, 24, 72], mutation-based sampling [4, 15, 31], and some of the greedy techniques, such as [7, 20], do not achieve degrees of coverage. Compared to the feature interaction coverage, only a few studies consider the code coverage [21, 22, 33, 38]. With these techniques, a minimal sample set is selected such that every lexical code fragment of the whole system code is covered by at least one product.

Summary and Insights. Based on our classification, we observe that the greedy techniques [1–3, 7, 14, 18–24, 29, 31, 33, 37] and meta-heuristic algorithms [5, 6, 8–10, 15, 17, 25–27, 41] are used more often compared to other techniques (cf. Table 1). However, other semi-automatic selection techniques have been also proposed [2, 24, 29, 72], where the users have the ability to influence the sampling process. Moreover, some of the greedy techniques aim to generate a small set of configurations that achieves a degree of feature interaction coverage (e.g., pair-wise coverage [2, 26, 29, 36] and t-wise [11, 14, 18, 19, 23, 30]). However, the main limitation of most of algorithms is that they do not scale to large product lines [28]. Thus, several approaches have been proposed to improve the scalability by reducing the configuration space [14, 21, 33]. Most of these reduction techniques require more domain knowledge to guide the sampling process.

As an alternative to the t-wise sampling techniques, meta-heuristic algorithms have been proposed to sample products of product lines. These meta-heuristic algorithms aim to handle many objectives during the sampling, which is not the case with most of the greedy algorithms, where only the coverage is often considered in the sampling process. However, the limitation of these meta-heuristic algorithms is that they are not deterministic, which may influence the testing badly, as they often cannot reproduce the same products to check whether detected faults are fixed, especially when feature models of the corresponding product lines are modified. Other greedy techniques (e.g., random sampling) are proposed, which do not guarantee a certain degree of feature interaction coverage. Moreover, we observed that only two greedy algorithms consider the prioritization of the generated products [2, 29], while the order in the other greedy algorithms are often influenced implicitly by the coverage, as they try to cover as many of the uncovered combinations as soon as possible.

In future, we argue that combining different techniques can improve the testing effectiveness and efficiency by avoiding or diminishing the limitations of existing approaches and benefiting from their advantages. For instance, combining the meta-heuristic algorithms with the greedy ones may be helpful to avoid being trapped in the local optima. Furthermore, we noticed that only a few of the existing sampling algorithms consider the code coverage. Thus, sampling techniques that guarantee a degree of code coverage are required, because most of the faults exist in the source code [1]. Moreover, there are no sampling techniques that consider the evolution of product lines, such as the history of existing samples. Taking the history of previous samples into consideration can be used to increase the diversity to cover more interactions or to reduce diversity over time for other applications.

6 CLASSIFICATION BASED ON EVALUATION

The techniques proposed in the literature have been evaluated for different measures of efficiency and effectiveness, against various subject systems, and using various types of tools. With respect to **RQ3**, this section provides a synthesis of these aspects of evaluation for the surveyed techniques.

Efficiency. Most of the analyzed sampling approaches have been evaluated to assess some measure of efficiency. In two cases [3, 7], no evaluation was done. Regarding efficiency, we distinguish between sampling and testing efficiency.
(a) Sampling efficiency. This notion measures the time (and possibly memory and computational resources) to generate the sample. We have identified several studies [2, 4, 5, 9, 11, 14–19, 21–26, 29, 31, 33, 38, 40, 42, 46] that evaluated the sampling efficiency by measuring the execution time of the algorithms to generate the sample. Kowal et al. [23] reduce the feature model to operate on smaller input data. They compare the efficiency of their sampling approach against existing ones by measuring the computation time. Other measures of resource consumption such as (volatile or persistent) memory consumption are also relevant in this respect, but none of these studies evaluated any other measures of efficiency.
(b) Testing efficiency. Testing efficiency focuses on the resulting sample, such as the number- and the size of configurations in the sample set. There are studies evaluating the testing efficiency by counting the number of generated configurations [1, 2, 4, 5, 7, 8, 10–17, 19–32, 34–36, 40–48]. Henard et al. [17] also check the size of the configurations to evaluate the testing efficiency. By the size of a configuration, we mean the number of selected features of the feature model. In all these papers, the problem space was evaluated but not the solution space. Liebig et al. [24] used the solution space, code artifacts besides the feature model, to evaluate the algorithm. Testing efficiency was evaluated by measuring the analysis time, i.e., the time required for type checking and data-flow analysis.

Effectiveness. By effectiveness, we mean the quality of the generated sample set and its capability in detecting faults. Effectiveness typically measure some notion of coverage. We subdivided the measurement of effectiveness into three types, such as (a) feature (interaction) coverage, (b) fault coverage (also incarnated in measures such as mutation score), or (c) code coverage.
(a) Feature (interaction) coverage. The feature (interaction) coverage was measured in [2, 6, 8–10, 17, 20, 25, 30, 42, 45] to rate the effectiveness of algorithms. Johansen et al. [20] weighted the feature interactions in the covering array. In a special case [16], the feature coverage of the sample set is measured for a specific amount of time. All of these approaches checked if a pairwise/t-wise feature coverage is achieved. This means, every pair (respectively, every t-wise tuple) of features has to occur in at least one configuration. Here, only the problem space is used in the evaluation. In some cases, no 100% coverage is achieved because evolutionary algorithms are used [6, 8, 42].
(b) Fault coverage. Other studies measured fault coverage [1, 4, 8, 28, 31, 37, 39, 42, 44], i.e., the capability of detecting certain faults. Ensan et al. [8] and Tartler et al. [37] use a program which marks those features or combinations that contain a fault. The evaluation then analyzes if all errors are discovered by the configurations in the sample. To discover features which contain an error, only the specific feature has to be in a configuration, but to discover errors stemming from feature interactions, all features of the corresponding combination have to be in a configuration. Ferreira et al. [9], Filho et al. [27], and Henard et al. [15] evaluate their approach using a mutation score (i.e. the discovered mutants in a feature model). Al-Hajjaji et al. [39, 40] introduce two prioritization approaches but they also measure how effective the default order of sampling algorithms is w.r.t. the fault detection rate. Al-Hajjaji et al. in [39] use delta-modeling for the evaluation. They analyze differences between products and select a sample based on these information. Halin et al. [44] use a real subject system and associate test cases to measure the fault coverage instead of only feature models. Abal et al. [1] also detect real faults. They use the commit history of the Linux kernel and map fixed bugs to features. Then, they evaluate if their sampling approach covers these features to find the bug.
(c) Code coverage. We analyzed the studies based on which input data they used for the evaluation. Most of them only use the feature model, but two studies also use code as input [38, 40]. For example, Tartler et al. [38] use code block coverage as an evaluation metric.

Siegmund et al. [34, 36, 47, 48], Grebhahn et al. [12, 43] and Sarkar et al. [32] evaluate the effectiveness of sampling algorithms in a different way. They use the sampling result to predict the performance of products and compare against real performance measurements. The prediction error for different sampling strategies is evaluated. Siegmund et al. [35, 36] also used the approach to predicted footprints of products.

Subject Systems. The subject systems used for evaluation vary significantly. We distinguish between real [1, 2, 13, 16, 18–22, 24, 28, 30, 32, 34–38, 40, 41, 47, 48, 78], academic [2, 6, 8–10, 16, 17, 17, 19, 23, 26, 27, 32, 34–36, 39, 41, 44, 45, 47, 48, 78] and artificial [2, 4, 16, 29, 31, 40, 46] feature models used as subjects. As mentioned in Section 3, real feature models are models of existing projects used in practice (e.g., in open source or industrial projects), such as the Linux kernel. The difference between academic and artificially-generated feature models is that academic feature models are created by researchers, while artificially-generated feature models are constructed by a program according to some rules; in other words, to generate artificial feature models no specific domain knowledge and manual intervention is necessary. SPLOT [74] is a popular tool for the generation of artificial feature models.

For example, Oster et al. in [29] use artificial feature models to evaluate the algorithms. The feature model of the Linux kernel is also used in many cases [1, 2, 16, 18, 19, 24, 37, 38, 40]. Furthermore, the E-Shop is an example of an academic feature model which is used in some studies for evaluations [6, 9, 10, 23].

The size of the feature models is distinguished in a range from very small ones with fewer than 20 features to huge feature models containing more than 10,000 features. We collected the size of all evaluated feature models if it was specified by the authors. Therefore, we can only present the lower bounds of the tested feature models. More than 1,000 feature models with less than 100 features were evaluated [2, 4–6, 8–23, 26–36, 39–48]. We collected more than 100 feature models with a size between 100 and 500 features [2, 4, 11, 14–16, 18, 19, 23, 28, 29, 31, 35, 36, 40, 46]. Bigger feature models are less evaluated. In the considered studies approximately 15 models with a size bigger than 5,000 [1, 2, 16, 18, 19, 24, 28, 38, 40, 78] are evaluated. The biggest evaluated feature model of the Linux kernel has 26,427 features [1, 28].

Tool support. Considering tool support, we observe that several studies have an implementation [2–6, 8–22, 24–38, 40, 43, 45–48], and some of these implementations are either open source or publicly available [2, 3, 8, 11–13, 15, 17–20, 24, 25, 28, 32, 34–38, 40, 43]. The implementations are based on various programming languages. The most reported programming language for implementations is Java [8, 17, 19, 30]. There are implementations which cover a set of algorithms, e.g., ICPL [19], and IncLing [2] are implemented in the FeatureIDE framework [72].

Summary and Insights. Most papers evaluate one or more algorithms using some measure of efficiency or effectiveness. Efficiency mostly focuses on the time of sample generation or the size of the sample [1, 2, 4, 5, 7–36, 38, 40–48], while effectiveness mostly addresses feature (interaction) coverage [2, 6, 8–10, 17, 20, 25, 30, 42, 45] and less prominently, fault coverage [1, 4, 8, 28, 31, 37, 39, 42, 44] or code coverage [38, 40]. The used subject systems are equally distributed between real and academic feature models. Artificially generated ones are less used. The size of the feature models also ranges from small feature models with only a few features to very large ones with almost 7,000 features to over 26,000 features. Another interesting point is that many authors do not name their tool. Unique names would be helpful to distinguish implementations.

Regarding efficiency, measuring other resources such as memory consumption or amount of required storage are missing in the literature. We also identified lack of evaluations using the solution space. Most papers only use the feature model as input data in the evaluation. For example, the evaluation of real faults in the code are under-studied. More research seems to be necessary to establish a common measure of effectiveness in terms of fault coverage. Comparing the different measures of effectiveness and studying the compromise between efficiency (sampling and testing time and resources) versus effectiveness in different domains seem to be under-studied. Providing a common benchmark of subject systems based on real examples from various domains will facilitate the evaluation of different algorithms on a common ground. Furthermore, evaluation of sampling algorithms, which consider evolution of product lines, could be an interesting topic for future work.

7 RELATED WORK

There are numerous strategies to cope with many software products during analysis and testing of software product lines [56, 81, 82]. Considering all products separately in an unoptimized product-based analysis is typically not feasible [81]. In optimized product-based analyses, the situation is improved by reducing the number of products or by reducing the effort for each product. Product sampling as discussed in this survey aims to reduce the number of products and is also known as sample-based analysis [81] or simply as sampling [82]. The effort for each product can be reduced by applying regression techniques to software product lines [33, 55, 58, 68–70, 79]. As such regression-based analyses already assume that a sufficiently small number of products is given, they are often combined with product sampling.

In contrast to product-based analyses, a software product line can also be analyzed in a feature-based or family-based fashion [81]. Feature-based means that the implementation of each feature is analyzed separately without taking other features into account. However, this way feature interactions are not covered by the analyses [81]. Family-based analyses also consider domain artifacts instead of generated products, but incorporate valid combinations of features as defined in the feature model [56, 81]. Family-based analyses have the inherent problem that they require special tools, whereas product-based analyses can always use tools from single-system engineering. Special tools are needed for the analysis itself or at least to transform the product line into a metaproduct simulating the behavior of all products [63, 75, 81, 83]. While the family-based strategy has been extensively studied for static analyses [81] and is known to outperform sample-based analyses [24, 51], recent applications to testing indicate that product sampling is still necessary to complement family-based testing [63, 73, 76].

In principle, product sampling can have numerous applications, but it is typically proposed in the context of product-line testing. Consequently, existing surveys on product-line testing discuss product sampling [56, 66], but not as detailed as we do. While our focus is on sampling for product lines, the roots of this research area are in combinatorial interaction testing [77]. In contrast to combinatorial interaction testing, product sampling is specific to product lines and is typically based on feature models. Ahmed et al. [49] conducted a systematic literature study on interaction testing supporting constraints. Their scope is different, as they also incorporate constraints on input parameters for testing of single systems and miss applications of product sampling beyond testing. Furthermore, our classification is more detailed and gives more insights. Lopez-Herrejon et al. [71] perform a systematic literature review about the combinatorial interaction testing approaches in product lines. They briefly classify the proposed techniques into different categories.

In our classification, we provide more details about the discussed techniques, such as the required input as well as the criteria that are used to evaluate these techniques.

Johansen et al. [61] present a survey of empirics of product line testing strategies. They report that only a few research conduct an empirical study to evaluate the corresponding strategies. Lamancha et al. [65] and Carmo Machado et al. [56] conducted a systematic literature review, where sampling techniques have been discussed as part of the approaches that have been proposed to select products for testing. In this paper, we distinguish with more details between the proposed approaches that sample products of product lines with respect to many characteristics, such as the required information for sampling, the considered criteria, and the tool support. Medeiros et al. [28] present a comparative study of 10 sampling algorithms for product lines with respect to the size of the generated samples and the fault detection rate. In our work, we classify the state of the art product sampling approaches with respect to many characteristics, including the sample size and the rate of fault detection.

Numerous of the distinguished sampling approaches in this paper have been used in the evaluation of the existing product prioritization approaches [39, 67]. For instance, Al-Hajjaji et al. [39] aim to find faults faster by prioritizing products based on their dissimilarity with respect to solution artifacts. They evaluate their work against the default order of the sampling algorithm MoSoPolite [29]. On the contrary, Lity et al. [67] prioritize products based on their similarity to reduce the effort during the incremental analysis by minimizing the differences between the consecutive analyzed products.

Baller et al. [52] propose a multi-objective approach that considers test cases, test requirements, products, and their interrelations with respect to the corresponding cost and profit of selecting them. In particular, they use a mapping from requirements (i.e., test goals) to test cases and a mapping from test cases to products as input. They evaluate the effectiveness of their approach by measuring the accuracy of selecting a set of products and test cases to test a product line, with respect to certain restrictions, such as reducing the cost as well as maximizing the profit of testing. While this approach is not mainly about sampling products, it involves selecting products to be tested from a large set of products. In single-system engineering, Yoo et al. [84] surveyed the efforts have been made to select test cases. NIE et al. [77] discuss the combinatorial interaction testing techniques with respect to their important issues, methods, and applications. In this paper, most of the classified sample approaches are considered as instances of combinatorial interaction testing approach.

8 CONCLUSION

In this paper, we presented an overview of the literature regarding product sampling for efficient analysis of software product lines. To this end, we classified the literature in terms of the type of information used in the technique, the algorithms employed, and the methods and measures used to evaluate them. In each characteristic, we identified the areas of strength and the understudied areas, which can help researchers and practitioners to choose an appropriate technique based on their constraints.

We gained numerous insights by means of this survey. A vast majority of techniques only use feature models as input. However, there are other types of input data that can be incorporated in sampling which need more investigation in the future, such as the product-line evolution, different forms of specification, known feature interactions extracted using static analyses, as well as test artifacts and implementation artifacts. Greedy and meta-heuristic techniques are more commonly used among different techniques. However, as scalability is the main issue in many sampling approaches, using semi-automatic selection techniques could help with reducing the configuration space by incorporating the expert knowledge. Moreover, techniques that consider code coverage or notions of diversity seem to be understudied. Considering evaluation, the efficiency, mostly measured by time of sample generation, sample size, and the effectiveness, mostly measured by feature interaction, are most commonly used. Further efficiency measures, e.g., memory or storage consumption, as well as incorporating solution space in evaluation need more investigation. Furthermore, providing a benchmark of subject systems based on real examples facilitates future evaluations.

ACKNOWLEDGMENTS

We gratefully acknowledge constructive discussions on the classification at the FOSD meeting in Gothenburg. In particular, we thank Myra Cohen for her suggestions with respect to the classification of meta-heuristic search techniques. This work has been partially supported by the European project ITEA 3 15010 *REVaMP*^{2} and by the German Research Foundation within the project IMoTEP (grant agreement LO 2198/2-1).

REFERENCES

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Trans. Software Engineering and Methodology (TOSEM)* 26, 3, Article 10 (Jan. 2018), 10:1–10:34 pages. https://doi.org/10.1145/3149119

[2] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*. ACM, New York, NY, USA, 144–155. https://doi.org/10.1145/2993236.2993253

[3] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Tool Demo: Testing Configurable Systems with FeatureIDE. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*. ACM, New York, NY, USA, 173–177. https://doi.org/10.1145/2993236.2993254

[4] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2015. Generating Tests for Detecting Faults in Feature Models. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*. IEEE, 1–10. https://doi.org/10.1109/ICST.2015.7102591

[5] Anastasia Cmyrev and Ralf Reissing. 2014. Efficient and Effective Testing of Automotive Software Product Lines. 7, 2 (2014). https://doi.org/10.14416/j.ijast.2014.05.001

[6] Thiago N. Ferreira, Josiel Neumann Kuk, Aurora Pozo, and Silvia Regina Vergilio. 2016. Product Selection Based on Upper Confidence Bound MOEA/D-DRA for Testing Software Product Lines. In *Proc. Congress Evolutionary Computation (CEC)*. IEEE, Piscataway, NJ, USA, 4135–4142. https://doi.org/10.1109/CEC.2016.7744315

[7] Alireza Ensan, Ebrahim Bagheri, Mohsen Asadi, Dragan Gasevic, and Yevgen Biletskiy. 2011. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *Proc. Inte'l Conf. Information Technology: New Generations (ITNG)*. IEEE, 291–298. https://doi.org/10.1109/ITNG.2011.58

[8] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gasevic. 2012. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Proc. Int'l Conf. Advanced Information Systems Engineering (CAiSE)*. Vol. 7328. Springer, 613–628. https://doi.org/10.1007/978-3-642-31095-9_40

[9] Thiago N. Ferreira, Jackson A. Prado Lima, Andrei Strickler, Josiel N. Kuk, Silvia R. Vergilio, and Aurora Pozo. 2017. Hyper-Heuristic Based Product Selection for Software Product Line Testing. 12, 2 (May 2017), 34–45. https://doi.org/10.1109/MCI.2017.2670461

[10] Helson L. Jakubovski Filho, Jackson A. Prado Lima, and Silvia R. Vergilio. 2017. Automatic Generation of Search-Based Algorithms Applied to the Feature Testing

of Software Product Lines. In *Proc. Brazilian Symposium Software Engineering (SBES)*. ACM, New York, NY, USA, 114–123. https://doi.org/10.1145/3131151.3131152

[11] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing. *Empirical Software Engineering (EMSE)* 16, 1 (2011), 61–102. https://doi.org/10.1007/s10664-010-9135-7

[12] Alexander Grebhahn, Sebastian Kuckuk, Christian Schmitt, Harald Köstler, Norbert Siegmund, Sven Apel, Frank Hannig, and Jürgen Teich. 2014. Experiments on Optimizing the Performance of Stencil Codes with SPL Conqueror. *Parallel Processing Letters* 24, 3 (2014). https://doi.org/10.1142/S0129626414410011

[13] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wąsowski, and Huiqun Yu. 2018. Data-Efficient Performance Learning for Configurable Systems. *Empirical Software Engineering (EMSE)* 23, 3 (June 2018), 1826–1867. https://doi.org/10.1007/s10664-017-9573-6

[14] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2013. Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, New York, NY, USA, Article 16, 16:1–16:6 pages. https://doi.org/10.1145/2430502.2430524

[15] Christopher Henard, Mike Papadakis, and Yves Le Traon. 2014. Mutation-Based Generation of Software Product Line Test Configurations. In *Search-Based Software Engineering*, Claire Le Goues and Shin Yoo (Eds.). Lecture Notes in Computer Science, Vol. 8636. Springer International Publishing, 92–106. https://doi.org/10.1007/978-3-319-09940-8_7

[16] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Trans. Software Engineering (TSE)* 40, 7 (July 2014), 650–670. https://doi.org/10.1109/TSE.2014.2327020

[17] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 2013. Multi-Objective Test Generation for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/2491627.2491635

[18] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*. Springer, Berlin, Heidelberg, 638–652. https://doi.org/10.1007/978-3-642-24485-8_47

[19] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 46–55. https://doi.org/10.1145/2362536.2362547

[20] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. 2012. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*. Springer, Berlin, Heidelberg, 269–284. https://doi.org/10.1007/978-3-642-33666-9_18

[21] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/1960275.1960284

[22] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. 2010. Reducing Configurations to Monitor in a Software Product Line. In *Proc. Int'l Conf. Runtime Verification (RV)*. Springer, Berlin, Heidelberg, 285–299. https://doi.org/10.1007/978-3-642-16612-9_22

[23] Matthias Kowal, Sandro Schulze, and Ina Schaefer. 2013. Towards Efficient SPL Testing by Variant Reduction. In *Proce. Int'l workshop on Variability & composition (VariComp)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2451617.2451619

[24] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, New York, NY, USA, 81–91. https://doi.org/10.1145/2491411.2491437

[25] Roberto Erick Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. 2014. Comparative Analysis of Classical Multi-Objective Evolutionary Algorithms and Seeding Strategies for Pairwise Testing of Software Product Lines. In *Proc. Congress Evolutionary Computation (CEC)*. IEEE, 387–396. https://doi.org/10.1109/CEC.2014.6900473

[26] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. 2013. Practical Pairwise Testing for Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 227–235. https://doi.org/10.1145/2491627.2491646

[27] Rui Angelo Matnei Filho and Silvia Regina Vergilio. 2016. A Multi-Objective Test Data Generation Approach for Mutation Testing of Feature Models. 4, 1 (July 2016). https://doi.org/10.1186/s40411-016-0030-9

[28] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, New York, NY, USA, 643–654. https://doi.org/10.1145/2884781.2884793

[29] Sebastian Oster, Florian Markert, and Philipp Ritter. 2010. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*. Springer, Berlin, Heidelberg, 196–210. https://doi.org/10.1007/978-3-642-15579-6_14

[30] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2010. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*. IEEE, Washington, DC, USA, 459–468. https://doi.org/10.1109/ICST.2010.43

[31] Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. 2015. Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/2791060.2791074

[32] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. 2015. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, Washington, DC, USA, 342–352. https://doi.org/10.1109/ASE.2015.45

[33] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. 2012. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*. Springer, Berlin, Heidelberg, 270–284. https://doi.org/10.1007/978-3-642-28872-2_19

[34] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting Performance via Automated Feature-Interaction Detection. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, Piscataway, NJ, USA, 167–177.

[35] Norbert Siegmund, Marko RosenmüLler, Christian KäStner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. 2013. Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption. *J. Information and Software Technology (IST)* 55, 3 (March 2013), 491–507. https://doi.org/10.1016/j.infsof.2012.07.020

[36] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal (SQJ)* 20, 3-4 (Sept. 2012), 487–517. https://doi.org/10.1007/s11219-011-9152-9

[37] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. In *Proc. USENIX Annual Technical Conference (ATC)*. USENIX Association, Berkeley, CA, USA, 421–432.

[38] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2012. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review* 45, 3 (Jan. 2012), 10–14. https://doi.org/10.1145/2039239.2039242

[39] Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. 2017. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. In *Proc. Int'l Workshop Variability and Complexity in Software Design (VACE)*. IEEE, Piscataway, NJ, USA, 34–40. https://doi.org/10.1109/VACE.2017..8

[40] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2018. Effective Product-Line Testing Using Similarity-Based Product Prioritization. *Software and System Modeling* (2018). To appear.

[41] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2015. Covering SPL Behaviour with Sampled Configurations: An Initial Assessment. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, New York, NY, USA, Article 59, 59:59–59:66 pages. https://doi.org/10.1145/2701319.2701325

[42] Stefan Fischer, Roberto E. Lopez-Herrejon, Rudolf Ramler, and Alexander Egyed. 2016. A Preliminary Empirical Assessment of Similarity for Combinatorial Interaction Testing of Software Product Lines. In *Proc. Int'l Workshop on Search-Based Software Testing (SBST)*. ACM, New York, NY, USA, 15–18. https://doi.org/10.1145/2897010.2897011

[43] Alexander Grebhahn, Carmen Rodrigo, Norbert Siegmund, Francisco José Gaspar, and Sven Apel. 2017. Performance-Influence Models of Multigrid Methods: A Case Study on Triangular Grids. *Concurrency and Computation: Practice and Experience* 29, 17 (2017). https://doi.org/10.1002/cpe.4057

[44] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2017. *Test Them All, Is It worth It? A Ground Truth Comparison of Configuration Sampling Strategies.* Technical Report arXiv:1710.07980. Cornell University Library.

[45] Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. 2011. Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, Article 6, 6:1–6:8 pages. https://doi.org/10.1145/2019136.2019143

[46] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2012. Pairwise Testing for Software Product Lines: Comparison

of Two Approaches. *Software Quality Journal (SQJ)* 20, 3-4 (2012), 605–643. https://doi.org/10.1007/s11219-011-9160-9

[47] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-Influence Models for Highly Configurable Systems. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, New York, NY, USA, 284–294. https://doi.org/10.1145/2786805.2786845

[48] Norbert Siegmund, Alexander von Rhein, and Sven Apel. 2013. Family-Based Performance Measurement. In *Proc. Int'l Conf. Generative Programming: Concepts & Experiences (GPCE)*. ACM, New York, NY, USA, 95–104. https://doi.org/10.1145/2517208.2517209

[49] Bestoun S. Ahmed, Kamal Z. Zamli, Wasif Afzal, and Miroslav Bures. 2017. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access* 5 (2017), 25706–25730. https://doi.org/10.1109/ACCESS.2017.2771562

[50] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg.

[51] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, Piscataway, NJ, USA, 482–491. https://doi.org/10.1109/ICSE.2013.6606594

[52] Hauke Baller, Sascha Lity, Malte Lochau, and Ina Schaefer. 2014. Multi-Objective Test Suite Optimization for Incremental Product Family Testing. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*. IEEE, Washington, DC, USA, 303–312. https://doi.org/10.1109/ICST.2014.43

[53] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conf. (SPLC)*. Springer, Berlin, Heidelberg, 7–20.

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

[55] Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. 2011. Verification of Software Product Lines with Delta-Oriented Slicing. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*. Springer, Berlin, Heidelberg, 61–75.

[56] Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)* 56, 10 (2014), 1183–1199. https://doi.org/10.1016/j.infsof.2014.04.002

[57] Ivan Do Carmo Machado, John D. McGregor, and Eduardo Santana De Almeida. 2012. Strategies for Testing Products in Software Product Lines. *SIGSOFT Software Engineering Notes* 37, 6 (Nov. 2012), 1–8. https://doi.org/10.1145/2382756.2382783

[58] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2012. Towards an Incremental Automata-Based Approach for Software Product-Line Model Checking. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*. ACM, New York, NY, USA, 74–81. https://doi.org/10.1145/2364412.2364425

[59] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[60] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Understanding Linux Feature Distribution. In *Proc. of Workshop on Modularity in Systems Software (MISS)*. ACM, NY, USA, 15–20.

[61] Martin Fagereng Johansen, Oystein Haugen, and Franck Fleurey. 2011. A Survey of Empirics of Strategies for Software Product Line Testing. In *ICST Workshop Proceedings*. IEEE, Washington, DC, USA, 266–269.

[62] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[63] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. 2012. Toward Variability-Aware Testing. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2377816.2377817

[64] Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. 2017. FeatureIDE: Empowering Third-Party Developers. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 42–45. https://doi.org/10.1145/3109729.3109751

[65] Beatriz Pérez Lamancha, Macario Polo, and Mario Piattini. 2013. *Systematic Review on Software Product Line Testing*. Springer, Berlin, Heidelberg, 58–71.

[66] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/2362536.2362545

[67] Sascha Lity, Mustafa Al-Hajjaji, Thomas Thüm, and Ina Schaefer. 2017. Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-line Analysis. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, New York, NY, USA, 60–67. https://doi.org/10.1145/3023956.3023961

[68] Sascha Lity, Thomas Morbach, Thomas Thüm, and Ina Schaefer. 2016. Applying Incremental Model Slicing to Product-Line Regression Testing. In *Proc. Int'l Conf. Software Reuse (ICSR)*. Springer, Berlin, Heidelberg, 3–19. https://doi.org/10.1007/978-3-319-35122-3_1

[69] Malte Lochau, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. 2014. DeltaCCS: A Core Calculus for Behavioral Change. In *Proc. Int'l Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, New York, NY, USA, 320–335. https://doi.org/10.1007/978-3-662-45234-9_23

[70] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. 2012. Incremental Model-Based Testing of Delta-oriented Software Product Lines. In *Proc. Int'l Conf. Tests and Proofs (TAP)*. Springer, Berlin, Heidelberg, 67–82.

[71] Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Aalexander Egyed. 2015. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In *Proc. Int'l Workshop Combinatorial Testing (IWCT)*. IEEE, Washington, DC, USA, 1–10. https://doi.org/10.1109/ICSTW.2015.7107435

[72] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer, Berlin, Heidelberg.

[73] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions In Highly-Configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/2970276.2970322

[74] Marcílio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, New York, NY, USA, 761–762.

[75] Jan Midtgaard, Claus Brabrand, and Andrzej Wąsowski. 2014. Systematic Derivation of Static Analyses for Software Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. ACM, New York, NY, USA, 181–192. https://doi.org/10.1145/2577080.2577091

[76] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, New York, NY, USA, 907–918. https://doi.org/10.1145/2568225.2568300

[77] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *Comput. Surveys* 43, 2, Article 11 (Feb. 2011), 11:1–11:29 pages. https://doi.org/10.1145/1883612.1883618

[78] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. 2013. Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, Piscataway, NJ, USA, 465–474. https://doi.org/10.1109/ASE.2013.6693104

[79] Christoph Seidl, Ina Schaefer, and Uwe Assmann. 2013. Variability-Aware Safety Analysis Using Delta Component Fault Diagrams. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*. ACM, New York, NY, USA, 2–9.

[80] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy Kolesnikov. 2011. Scalable Prediction of Non-Functional Properties in Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*. IEEE, Washington, DC, USA, 160–169.

[81] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (June 2014), 6:1–6:45. https://doi.org/10.1145/2580950

[82] Alexander von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. 2013. The PLA Model: On the Combination of Product-Line Analyses. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, New York, NY, USA, 14:1–14:8. https://doi.org/10.1145/2430502.2430515

[83] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability Encoding: From Compile-Time to Load-Time Variability. *J. Logic and Algebraic Methods in Programming (JLAMP)* 85, 1, Part 2 (Jan. 2016), 125–145.

[84] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability (STVR)* 22, 2 (2012), 67–120.