IncLing: Efficient Product-Line Testing using Incremental Pairwise Sampling

Mustafa Al-Hajjaji,[1] Sebastian Krieter,[1] Thomas Thüm,[2] Malte Lochau,[3] Gunter Saake[1]

[1] University of Magdeburg, Germany, [2] TU Braunschweig, Germany, [3] TU Darmstadt, Germany

Abstract

A software product line comprises a family of software products that share a common set of features. It enables customers to compose software systems from a managed set of features. Testing every product of a product line individually is often infeasible due to the exponential number of possible products in the number of features. Several approaches have been proposed to restrict the number of products to be tested by sampling a subset of products achieving sufficient combinatorial interaction coverage. However, existing sampling algorithms do not scale well to large product lines, as they require a considerable amount of time to generate the samples. Moreover, samples are not available until a sampling algorithm completely terminates. As testing time is usually limited, we propose an incremental approach of product sampling for pairwise interaction testing (called IncLing), which enables developers to generate samples on demand in a step-wise manner. Furthermore, IncLing uses heuristics to efficiently achieve pairwise interaction coverage with a reasonable number of products. We evaluated IncLing by comparing it against existing sampling algorithms using feature models of different sizes. The results of our approach indicate efficiency improvements for product-line testing.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Testing-Reusable Software; D.2.13 [*Software Engineering*]: Reusable Software-Domain Engineering

General Terms Reliability, Verification

Keywords Software product lines, model-based testing, combinatorial interaction testing, sampling

GPCE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4446-3/16/10...
http://dx.doi.org/10.1145/2993236.2993253

1. Introduction

A software product line represents a family of similar software systems in terms of common and variable functionality. Customers are able to automatically generate products from a large space of features organized in a product line. The different products of a product line share common features and differ in others [4, 9, 13, 33]. A feature is defined as an increment in functionality recognizable by customers [5]. Software product lines have benefits such as reduced development cost, increased quality, and reduced time to market [4]. Numerous companies have adapted their development process to product lines [36]. Despite the advantages product lines bring, new challenges concerning the quality assurance arise.

Despite the difficulty to test a single software system, testing a product line is even more challenging due to the potentially exponential number of products. Ideally, every product in a product line should be tested individually, especially in case of safety-critical systems. However, it is often infeasible to test all products in realistic product lines due to the exponential number of products in the number of features. Thus, several approaches have been proposed to reduce the number of products to test [29, 31, 32]. In this regard, combinatorial interaction testing (CIT) is one of the most prominent approaches that has been introduced to select a reduced, yet sufficient subset of products to achieve a certain coverage of feature combinations [29, 31, 32]. For instance, in T-wise CIT, every combination of T features is required to appear in at least one product of a sample.

Several sampling algorithms have been proposed to achieve T-wise coverage, such as ICPL [19], CASA [15], Chvatal [8], MoSoPoLite [30], and IPOG [24]. These algorithms approximate the solutions to this constraint optimization problem. However, those sampling algorithms do not scale well to larger product lines in terms of CPU time and memory consumption [16, 27]. In addition, even for small sets of features these algorithms require a considerable amount of time to compute the result, since they output the entire sample instead of delivering intermediate products one by one until reaching sufficient feature interaction coverage. Johansen et al. [19] report that generating products for the

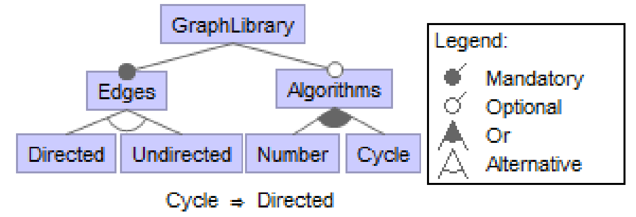

Figure 1. Feature diagram of product line *graph*

Linux kernel (with 6,888 features) requires approximately nine hours for pairwise CIT. Those samples are usually not available until a sampling algorithm is completely terminated.

To tackle the aforementioned problems, we propose **Inc**remental samp**Ling** (IncLing) to generate and test products one at a time to enhance the sampling efficiency, in terms of the required time to generate a sample, as well as testing effectiveness, in terms of the interaction coverage rate. With IncLing, we take already generated and tested products into account while selecting further products into the sample. Our greedy algorithm selects the next product that covers as many of uncovered feature combinations as possible. We increase the diversity among products by covering dissimilar pairwise feature combinations each time a further product is generated to be tested [3]. Increasing the covering rate of feature combinations might lead to a faster fault detection. This way, we dynamically go on generating further products into the sample until testing time is over.

We evaluate IncLing using a corpus of real-world and artificial feature models of different sizes and compare it against four sampling algorithms and random configurations. The results show that, on average, IncLing is capable of reducing computation time required by existing approaches by about 70%. Furthermore, we found that for up-to the first 40% of configurations, IncLing covers more feature interactions than other sampling algorithms. One potential limitation of IncLing is that it may generate more products than other sampling algorithms until achieving pairwise coverage. However, the difference is not significant. In summary, we contribute the following:

- With IncLing, we propose an incremental sampling approach based on the principles of ICPL [19] and dissimilarity [3] that keeps a similar level of effectiveness for product-line testing, but has improved scalability.
- We implement IncLing in FeatureIDE [35].
- We evaluate IncLing against existing sampling algorithms regarding: 1) the required computation time to generate products while achieving a certain degree of coverage, 2) the number of generated products, and 3) feature interaction coverage.

This paper is organized as follows. In Section 2, we briefly introduce the necessary concepts used in this paper relating to feature modeling and product configuration. IncLing is presented in Section 3. In Section 4, we present the results of our experimental evaluation of IncLing. Related work is discussed in Section 5 and we conclude in Section 6.

2. Background

In this section, we present background on feature models, combinatorial interaction testing in product lines, and existing algorithms that have been proposed to generate samples.

2.1 Feature Modeling and Product Configuration

A feature model is a hierarchical structure used to define the variability of a product line [20]. *Figure* 1 shows the feature model of our running example, a *graph* product line. A combination of a set of features is a *valid configuration* if it does not violate the feature dependencies and the constraints defined in the feature model. In particular, the selection of a feature implies the selection of its parent feature. A feature can be mandatory, which means it is required by its parent. Otherwise, they are optional. An example of a mandatory feature in *Figure* 1 is feature *Edges*. Furthermore, features can be grouped into *alternative* or *or* groups. Only one feature of an *alternative* group must be selected in a configuration. For instance, a configuration can have only one feature of *Directed and Undirected*. From features in an *or* group, at least one of them must be involved in a configuration. Features *Number*, *Cycle*, or both can be involved in a configuration.

In addition to the previous dependencies between features, other dependencies between features that are not representable in the hierarchical structure are defined by so-called *cross-tree constraints*. An example of these constraints in the *graph* product line is that if a graph includes feature *Cycle*, feature *Directed* is required to also be included in the configuration. As a result of feature dependencies and constraints, a feature can be a conditionally *dead* or *core* if it is under certain circumstances (i.e., features that must be selected or deselected given the feature model and already fixed features of the current configuration) [6]. For instance, if feature *Undirected* is selected, feature *Directed* will be a dead feature. The feature *Directed* may also be core, if feature *Cycle* is selected.

In this paper, we use feature models as input to our approach to create valid configurations. As the number of possible configurations can increase exponentially with the number of features, CIT techniques have been proposed to generate a subset of all valid configurations that achieves a certain coverage criteria.

2.2 Combinatorial Interaction Testing

Combinatorial interaction testing (CIT) is an approach used to restrict the number of products to be tested by selecting a subset of valid configurations satisfying certain coverage criteria [7, 29, 32]. Concerning CIT in particular, triggered faults by erroneous interactions between at most T features are likely to be discovered in at least one selected product. A feature interaction occurs when one or more features modify or influence the behavior of other features [17].

With T-wise CIT, a kind of feature-combination coverage needs to be achieved. For instance, in pairwise CIT (i.e.,

T=2), each valid combination of two features is required to appear in at least one configuration of the sample. For example, the valid combinations of feature *Directed* (D) and feature *Undirected* (U), in our running example, are $D \wedge \neg U$ and $\neg D \wedge U$, and the invalid ones are $D \wedge U$ and $\neg D \wedge \neg U$. Each valid combination, is required to appear at least in one of the created configurations of the sample.

Creating these configurations are instances of the *covering array* problem [18]. In particular, the configurations can be represented as covering array. The main challenge of generating covering arrays is to find the minimal number of configurations that covers the T-wise combinations of features, which is an NP-hard problem [14]. In addition, finding a valid configuration in a feature model is an NP-complete Satisfiability Problem (SAT) [12]. Numerous algorithms have been proposed to approximate these minimal covering arrays [8, 15, 18, 19]. In the following, we give an overview of a set of existing sampling algorithms, which have been used to sample configurations using the feature models as an input.

2.3 Covering Array Algorithms

CASA [15] uses simulated annealing to generate T-wise covering arrays of product lines. It is a non-deterministic algorithm where different configurations may be created in different orders when applied multiple times to the same feature model. CASA separates the problems into two iterated steps. The first step minimizes the number of created configurations. The second step ensures that a certain degree of coverage is achieved.

Chvatal is a heuristic algorithm proposed by Chvatal [8] to approximate optimal solutions for the minimal covering array. The basic version of the algorithm does not incorporate feature dependencies. Johansen et al. [18] adapted and improved the algorithm to create samples from feature models. The steps of generating covering arrays with the Chvatal algorithm are as follows. First, all T-wise feature combinations are generated. Second, an empty configuration is created. Third, all feature combinations are iterated to add them to the configuration. Each time a new combination is added to the configuration, the validity of this configuration, with respect to the feature model, is checked using a SAT solver. If the configuration is invalid, the combination is removed from the configuration. The newly created configuration is added to the final set of configurations if it at least contains one uncovered combination. The creation of configurations continues until all valid T-wise feature combinations are covered at least once.

ICPL [19] is based on the Chvatal algorithm with several improvements, such as identifying invalid feature combinations at an early stage. It generates the T-wise covering array more efficiently, because the parallelization of the algorithm shortens the computation time significantly. The goal of ICPL is to cover the T-wise combinations of features as fast as possible by covering the maximum number of uncovered feature combinations, each time a configuration is created.

IPOG [24] is an algorithm proposed to create covering arrays. These arrays are generated from scratch for the first T features, then the arrays grow horizontally and vertically. With the horizontal growth, a feature and its value are added, while in the vertical growth, new combinations of the newly added feature and the old ones are added to achieve the coverage, if needed. In the following, we illustrate IPOG's functionality with our running example. First, the covering array starts with one feature, e.g., *GraphLibrary*, and its values (*True* and *False*). Second, the covering array grows horizontally by adding a new feature, e.g., *Algorithms*, and its values (*True* and *False*). In particular, we have the following combinations of the two features, *True* $\wedge$ *True* and *False* $\wedge$ *False*. Third, to cover all possible combinations of the two features, the covering array grows vertically by adding the following combinations, *True* $\wedge$ *False* and *False* $\wedge$ *True*. The horizontal and vertical growth continues until all features and their combinations are covered.

Although there are promising approaches for generating covering arrays, most of them do not scale well to large feature models [25, 27] and their execution takes a considerable amount of time. As a result, the Linux kernel developers use the built-in facility of the Linux kernel build system `randconfig` to generate random configurations, because none of the existing sampling algorithms scale to the feature model of the Linux kernel with over 15 thousands features [28]. Furthermore, testers cannot start testing until the entire sampling process has terminated, because no intermediate results are reported. In this paper, we propose the IncLing algorithm to incrementally sample products one by one based on a greedy selection heuristics to approximate pairwise coverage.

3. Incremental Sampling: IncLing

We propose IncLing to efficiently select configurations for sample-based product-line testing. In this section, we present the algorithm of IncLing in detail.

3.1 Overview of IncLing

The product-line testing process, as considered in the following, includes creating configurations (e.g., sampling), generating products, and finally testing them. Similar to ICPL [19], IncLing generates new products by sequentially selecting pairs of features that are not already covered by previously selected configurations. Although, our algorithm is based on the general concept of ICPL, we propose several crucial modifications to improve the performance of the algorithm compared to ICPL. In particular, there are four major modifications.

Incremental Approach. IncLing generates products incrementally, whereas in the current implementation of ICPL, the user has to wait until all feature combinations are covered. The incremental nature of our algorithm has the advantage that products can be generated and tested in parallel until testing time is over or the desired coverage is achieved. Our incremental approach enables us to utilize the testing time effectively, because particular products can be tested immediately after being generated. Since particular products can be tested immediately right after being generated, a potential drawback of the incremental approach is that the order of the products cannot be adapted before testing.

Detecting Invalid Combinations. Invalid combinations are all those feature pairs that are impossible to cover due to feature-model dependencies. IncLing removes invalid combinations at the beginning of the sampling process. In contrast, ICPL removes invalid combinations after it covered a certain number of combinations [19]. The advantage of detecting invalid combinations at the beginning is that the algorithm has to consider only valid combinations and thus saves computation time. The disadvantage of this step is that it needs additional time at the beginning, which is the cost of saving effort during the sampling process.

Feature Ranking Heuristic. IncLing uses a heuristic to rank the list of feature pairs that are potentially being selected in the current product. This heuristic is based on the previously generated products. With each new product, this greedy strategy tries to cover the maximum number of pairwise feature combinations that have not already been covered by the previously selected products. Thus, our algorithm has the potential advantage of covering many feature combinations as fast as possible. However, since it is a greedy strategy, the algorithm might generate more products in total to cover all valid combinations than existing sampling algorithms.

Detecting Conditionally Dead or Core Features. Contrary to ICPL, IncLing does not test whether both features of a feature pair can be selected in the current product simultaneously, but each individually. Like ICPL, IncLing uses a satisfiability solver to test whether it is possible to selected or deselect a feature in the current product. ICPL also uses this method to determine whether a combination of features can be (de)selected simultaneously. However, it can be beneficial to test features individually, as this detects features that are conditionally dead or core [6]. Consequently, combinations that include these features do not need to be considered, since they will be covered automatically. As a result the overall performance of the algorithm is increased.

3.2 Implementation Details of IncLing

We present the main algorithm for IncLing in pseudo-code in Algorithm 1. After the *initialization* phase, we *generate products* one at a time. For each product, we *build a configuration* by consecutively *adding feature combinations* until the configuration is complete.

Initialization. The input of our algorithm consists of the feature model $\mathcal{FM}$, the set of products C_{old} that have been already generated and tested, the desired pairwise $Coverage$, and the testing $Time$ available. The output of the algorithm is a list of generated products to test. At the beginning, we initialize all relevant variables. First, we get all features from the feature model $\mathcal{FM}$ (Line 2). Second, we generate all pairwise combinations $combs_{initial}$ (Line 5). Third, we create a list of all uncovered combinations $combs_{left}$ by removing the invalid combinations and the combinations that are already covered in the set of configurations C_{old} (Line 6).

Algorithm 1 Main algorithm of IncLing.

```
 1: function IncLing($\mathcal{FM}$, $C_{old}$, $Coverage$, $Time$)
 2:     $F \leftarrow$ getFeatures($\mathcal{FM}$)
 3:     $freq \leftarrow$ emptyList
 4:     $signum \leftarrow$ emptyList
 5:     $combs_{initial} \leftarrow$ generateCombinations($\mathcal{FM}$)
 6:     $combs_{left} \leftarrow combs_{initial} \setminus$
            ( generateCombinations($C_{old}$) $\cup$
            generateInvalidCombinations($\mathcal{FM}$) )
 7:     while ( coveredCombinations( ) $< Coverage \wedge$
                passedTime( ) $< Time$ ) do
 8:         forbidConfigurations($\mathcal{FM}$, $C_{old}$)
 9:         updateFrequency($combs_{left}$, $freq$,
                $signum$)
10:         sort($F$, $freq$)
11:         $c_{new} \leftarrow \emptyset$
12:         for $i \leftarrow 1$ to $|F|$ step $1$ do
13:             for $j \leftarrow 0$ to $i$ step $1$ do
14:                 $combs_{test} \leftarrow$ getCombinations(
                        $combs_{left}$, $signum$, $F[i]$, $F[j]$)
15:                 testCombinations($combs_{test}$, $c_{new}$)
16:             end for
17:         end for
18:         AutoComplete($\mathcal{FM}$, $c_{new}$)
19:         $C_{old} \leftarrow C_{old} \cup \{c_{new}\}$
20:         $combs_{left} \leftarrow combs_{left} \setminus combs_{covered}$
21:         generateAndReturnProduct($c_{new}$)
22:     end while
23: end function
```

We demonstrate the single steps of our algorithm with the help of the *graph* product line example (cf. Figure 1). In this example, we get the following list of features $F = (G, E, D, U, A, N, C)$, where each feature is represented by its first letter. The list of all possible pairwise feature combinations consists of 168 elements in total. After the removal of all invalid feature combinations, such as (D, U) (i.e., both features cannot be chosen together) and $(\neg A, N)$ (i.e., contradiction of a parent-child relationship), the list contains 111 valid combinations that need to be covered (e.g., $(D, \neg U)$, $(\neg A, \neg C)$, (C, D), ...).

Generating Products. In the main loop of the algorithm, we build configurations until either running out of testing time or there are no more combinations to cover. We avoid building the same configurations by excluding the already generated ones from the feature model ($\mathcal{FM}$) via adding a blocking clause (Line 8). Before we build a configuration, we calculate the current *signum* and *frequency* of each feature, which we use for our heuristic. Frequency denotes how often each feature occurs in the remaining feature combinations. In Line 10, we rank features in descending order according to their frequencies. The signum of a feature is positive, if the feature is more often selected than deselected in the remaining feature combinations and negative otherwise. We use the signum to increase the diversity among configurations with regard to feature selections (cf. Line 14).

Next, we build a new configuration (cf. Lines 11–18), which we describe in more detail later on. For each new configuration, we exclude all combinations that are covered by the configuration from the list of uncovered combinations $combs_{left}$ (Line 20). Finally, we generate a product from this configuration (Line 21), which may then be tested while the algorithm goes to the next iteration. This process continues until either we reach a certain degree of pairwise feature coverage or we run out of testing time (Line 7). In our example, the initial computation of frequency and signum yields the following results:

Feature	G	E	D	U	A	N	C
Frequency	155	155	161	161	162	163	162
Signum	11	11	1	-1	2	-1	-2

As we excluded all invalid feature combinations in the initialization phase, the numbers for signum and frequency differ from 0 and 168, which is the maximum frequency for 7 features. When we rank the feature list accordingly, we get the list $F = (N, A, C, D, U, G, E)$, where the first feature has the highest frequency.

Building a Configuration. First, we create an empty configuration c_{new} (Line 11). Then, we execute two nested loops over the ranked feature list to generate the remaining combinations for each pair of features (Lines 12–17). For each pair of features, we generate a list of uncovered combinations using the signum for both features by calling the function `getCombinations` (Line 14). We then test whether we can cover one of these combinations within the current configuration c_{new} by executing the function `testCombinations` (Line 15).

For the first product in our example, we generate the following combinations. The first feature pair according to the feature ranking is N and A. In this case, the function `getCombinations` returns the combination list $combs_{left} = ((\neg N, A), (N, A), (\neg N, \neg A))$. These combinations are then tried to be added to the current configuration.

Adding a Combination. In Algorithm 2, we give the function `addCombinations`. In this function, we iterate over the given list of combinations $combo_{test}$ and test for each combination whether it is possible to include it in the current configuration c_{new}.

Algorithm 2 Tests whether a combination from the list $combs_{test}$ can be added to the current configuration c_{new}.

```
 1: function ADDCOMBINATIONS($combs_{test}$, $c_{new}$)
 2:     for all $combo \in combs_{test}$ do
 3:         $f_a \leftarrow combo[0]$
 4:         $f_b \leftarrow combo[1]$
 5:         if $(\{f_a, f_b, \neg f_a, \neg f_b\} \cap c_{new} \neq \emptyset) \wedge$
                $( (\neg f_a \in c_{new} \vee \neg f_b \in c_{new}) \vee$
                $( f_a \in c_{new} \wedge f_b \in c_{new}) \vee$
                COVEREDCOMBINATIONS( ) $<$ THRESHOLD( ) )
            then
 6:             continue
 7:         end if
 8:         if $f_b \notin c_{new}$ then
 9:             $c_{new} \leftarrow c_{new} \cup \{f_b\}$
10:             if $\neg$ ISSATISFIABLE($c_{new}$) then
11:                 $c_{new} \leftarrow (c_{new} \setminus \{f_b\}) \cup \{\neg f_b\}$
12:                 return
13:             end if
14:         end if
15:         if $f_a \notin c_{new}$ then
16:             $c_{new} \leftarrow c_{new} \cup \{f_a\}$
17:         end if
18:         if ISSATISFIABLE($c_{new}$) then
19:             return
20:         else
21:             if WASUNDEFINED($f_b$) then
22:                 if WASUNDEFINED($f_a$) then
23:                     $c_{new} \leftarrow c_{new} \setminus \{f_a\}$
24:                 end if
25:                 $c_{new} \leftarrow c_{new} \setminus \{f_b\}$
26:             else
27:                 $c_{new} \leftarrow (c_{new} \setminus \{f_a\}) \cup \{\neg f_a\}$
28:                 return
29:             end if
30:         end if
31:     end for
32: end function
```

First, we check whether a feature from the tested combination is already included in the current configuration (Line 5). In this case, we continue with the next combination in $combo_{test}$ if at least one of the following conditions is true: c_{new} contains the complement of at least one feature in the combination (i.e., the complement of feature A is ¬ A and vice versa), the combination is already contained in c_{new}, or the percentage of covered combinations is below a certain threshold. Otherwise, we try to add the combination using a satisfiability solver. If the current configuration c_{new} is not valid, we remove the combination from the configuration and we move to the next combination.

We use the threshold value as a mechanism to speed up the algorithm. Before reaching a certain amount of covered feature combinations, we exclude every combination with at least one feature in the current configuration (i.e., $(\{f_a, f_b, \neg f_a, \neg f_b\} \cap c_{new} \neq \emptyset)$), which considerably reduces the number of combinations that we have to check. However, excluding feature combinations also decreases the degree of pairwise coverage per configuration. Thus, after reaching the threshold, we consider all remaining feature combinations.

In this function, we do not test the entire combination at once, but try to add the features consecutively. We start by adding the second feature from the combination f_b to c_{new} (Lines 8- 14). If c_{new} is no longer satisfiable, we conclude that f_b is conditionally core or dead and, thus, we add the complement of f_b to c_{new} and return from the function. Next, we add the first feature f_a to c_{new}. If c_{new} is still satisfiable, we are able to cover the given combination and return from the function. Otherwise, we have to consider two cases. First, if f_b was initially not included in c_{new} (i.e., undefined), we have to remove it from c_{new} (Lines 21–25). In this case, the same also applies for f_a (Lines 22–24). We continue to test the next configuration in the list. Second, if f_b was included in c_{new}, we know that the complement of f_a has to be part of c_{new}. Therefore, we add the complement of f_a to c_{new} and return from the function (Lines 27–28).

Regarding our example, the first input of the function is the list of combinations $combs_{test} = ((\neg N, A), (N, A),(\neg N, \neg A))$ and an empty configuration. We start by testing the first configuration $(\neg N, A)$. Since none of both features are included in the current configuration, we add A to c_{new} and find that it is still satisfiable. Next, we add $\neg N$ to c_{new}. As c_{new} is still satisfiable, we successfully added the combination $(\neg N, A)$ to the current configuration and return from the function.

Continuing the execution of our algorithm, the next input consist of the list of combinations $combs_{test} = ((\neg N, \neg C), (N, \neg C), (N, \neg C))$ and the current configuration $c_{new} = \{\neg N, A\}$. The feature N is already contained in the configuration. Thus all combinations containing N or $\neg N$ are ignored. The next relevant feature pair is C and D with the list $combs_{test} = ((\neg C, D), (C, D), (\neg C, \neg D))$. Again, we add the second feature D to c_{new} and find that it is still satisfiable. However, when we add $\neg C$, the configuration becomes unsatisfiable. Thus, we remove both features from c_{new} and test the next combination (i.e., (C, D)). Given the current configuration, it is possible to select both features and, thus, we add them to c_{new} and continue with the next feature pair. Further continuing the process results in the configuration $c_{new} = \{\neg N, A, C, D, \neg U, G, E\}$.

3.3 IncLing in FeatureIDE

FeatureIDE [35] is an open-source framework based on Eclipse. It covers the whole development process and incorporates tools for the implementation of product lines into an integrated development environment. With FeatureIDE, feature models can be constructed by adding, editing, and removing features. In this paper, we focus on the FeatureIDE functionality related to configuration creation. Using FeatureIDE, the user can create configurations manually. However, creating a set of configurations manually is not efficient and should be automated. In previous work, some of us extended FeatureIDE to generate all valid products and prioritize them based on different criteria [1–3]. The limitation of this is that it is not feasible to generate all valid products for large product lines. Hence, we integrated sampling algorithms CASA, Chvatal, and ICPL into FeatureIDE in order to generate a sample of products while achieving a certain degree of coverage [3]. In this work, we extend FeatureIDE by implementing the IncLing algorithm. The new extension allows the user to generate a specific number of products or generate products within a restricted amount of time. The implementation of the algorithm is publicly available to be used for product generation or in upcoming evaluations.[1]

4. Evaluation

We evaluate IncLing against existing sampling algorithms and random configurations with respect to three criteria: sampling efficiency, testing efficiency, and testing effectiveness. With sampling efficiency, we refer to the aggregated computation time of the sampling algorithms to achieve pairwise coverage. Testing efficiency counts the number of products generated by the sampling algorithms to achieve pairwise coverage. For testing effectiveness, we consider the increase of pairwise interaction coverage achieved by the product order generated by our approach compared to the product order returned by the other approaches. In our evaluation, we focus on the following research questions.

- **RQ1:** Does IncLing increase the average sampling efficiency for achieving pairwise coverage compared to existing sampling algorithms?
- **RQ2:** Does IncLing decrease product-line testing efficiency compared to existing sampling algorithms?
- **RQ3:** Does IncLing decrease product-line testing effectiveness compared to random configurations and existing sampling algorithms?

4.1 Experiment Settings

In the experiments, we consider real and artificial feature models of different sizes. The real feature models consist of up-to 6,888 features. The size of the artificial feature models, which we use in our experiment, ranges between 15 and 5,542 features. In *Table* 1, we summarize the properties of each feature model. We report for each feature model the number of features, number of constraints, and ratio of the number of distinct features in cross-tree constraints to the number of features (CTCR). To evaluate IncLing, we generate configurations that are required to achieve the pairwise coverage criterion. For the threshold mentioned in Section 3.2, we use 99%. We repeated the experiment for each setting five times and calculated the average in order to reduce the impact of outliers in our measurements. As exception for random configurations, we conducted the experiment 100 times to additionally mitigate the effect of random impacts. For a fair comparison, we set the number

[1] `http://www.tinyurl.com/IncrementalSampling`

Feature Model	Features	#Constraints	CTCR
Email	10	3	50%
Violet	88	27	66%
BerkeleyDB1	53	20	42%
BerkeleyDB2	99	68	82%
Dell	46	110	80%
EShopFIDE	192	21	10%
EShopSplot	287	21	12%
GPL	27	16	63%
SmartHome22	60	2	6%
BattleofTanks	144	0	0%
FM_Test	168	46	28%
BankingSoftware	176	4	2%
Electronic Shopping	290	21	11%
DMIS	366	192	93%
eCos 3.0 i386pc	1,245	2,478	99%
FreeBSD kernel 8.0.0	1,369	14,295	93%
Automotive1	2,513	2,833	28%
Linux_2_6_28_6	6,888	6,847	99%
10xAFM15	15	2.6	19%
10xAFM50	50	9.7	17%
10xAFM100	100	20	17%
10xAFM200	200	39	17%
10xAFM500	500	100	17%
10xAFM1K	1,000	100	14%
1xAFM5K	5,542	300	11%

Table 1. Feature models used in our evaluation, (CTCR) stands for cross-tree constraints ratio; The values next to the artificial feature models 10xAFM15, 10xAFM50, 10xAFM100, 10xAFM200, 10xAFM500, and 10xAFM1000 represent the average values over 10 feature models.

of created configurations for random configurations to the same number of configurations as created with IncLing for the respective feature model.

In order to answer the aforementioned questions, we conducted experiments using feature models with different sizes in terms of number of features and different complexity in terms of CTCR. We performed the experiments using a PC with an Intel Core i5-4670 CPU @ 3.40 GHz, 16 GB RAM, and Windows 7. In the next section, we discuss the experiment settings.

4.2 Results and Discussion

Sampling Efficiency (RQ1). To answer RQ1, we compare our approach, with respect to the required time to achieve the pairwise coverage, against existing sampling algorithms, namely CASA [15], Chvatal [8], ICPL [19], and IPOG [24].

To measure whether IncLing improves sampling time, we calculate the relative time decrease compared to existing algorithms with the following formula:

$$TimeDecrease = (1 - \frac{Time_{IncLing}}{Time_{Existing}}) \cdot 100\% \qquad (1)$$

where $Time_{IncLing}$ is the sampling time of IncLing and $Time_{Existing}$ is the time of an existing sampling algorithm.

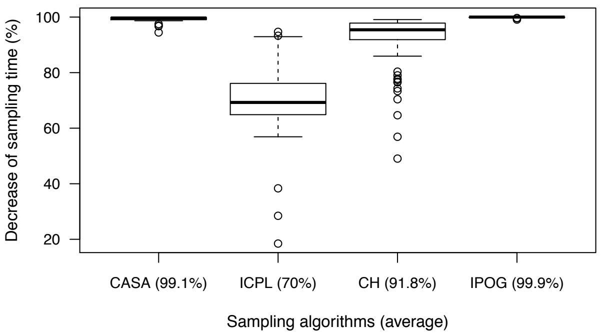

Figure 2. The distribution and average of time decrease percentage for our approach to sampling algorithms over all feature models. (CH: Chvatal algorithm)

In *Table* 2, we show the time and the number of configurations that are required to achieve pairwise coverage using IncLing compared to existing sampling algorithms. We observed that some of these sampling algorithms do not scale well. We stopped computation that required more than 24 hours. We refer to these cases in *Table* 2 as (*).

Examining the values of the required time to achieve the pairwise coverage for each feature model, we observe that IncLing outperforms the other sampling algorithms for all feature models, except for feature model *E-mail* (with 10 features), where the computation sampling time of IncLing is equal to the computation sampling time of Chvatal. For instance, for feature model *GPL* (27 features), we notice that the required time to cover all pairwise combinations for IncLing is 0.06 second, while the sampling algorithms CASA, Chvatal, ICPL, and IPOG require 14.58, 0.24, 0.15, and 173.65 seconds. That is, with IncLing, we can save 99.6%, 75%, 60%, and 99.9% of CASA's, Chvatal's, ICPL's, and IPOG's sampling time, respectively. For feature model *Automotive1*, the required time for IncLing is 8.6 minutes. Sampling algorithms ICPL and Chvatal require 126.72 minutes and 939.08 minutes with 93% and 99.1% decrease of its sampling time, respectively. For the feature model of the *Linux kernel*, only IncLing and ICPL scale to this large feature model. The times required for IncLing and ICPL to achieve the pairwise coverage are 0.92 hours and 5.2 hours, respectively. In particular, 82.5% sampling time can be saved when IncLing is used instead of ICPL.

In *Figure* 2, we show the distribution of relative time decrease of existing sampling algorithms when using IncLing instead (cf. Equation 1). We observe an improvement of our approach compared to the other sampling algorithms. If IncLing is used, the median values of the decrease of time for existing sampling algorithms range between 65% and 99.9%. The average time decrease over all feature models shows that 99.1%, 70%, 91.8%, and 99.9% of CASA's, ICPL's, Chvatal's, and IPOG's sampling time can be saved using IncLing.

Feature Model	CASA		Chvatal		ICPL		IPOG		IncLing	
	Conf.	Time	Conf.	Time	Conf.	Time	Conf.	Time	Conf.	Time
Email	**6.0**	0.13	7.0	**0.01**	7.0	0.03	8.0	0.46	9.0	**0.01**
Violet	**22.0**	59.75	27.0	1.38	28.0	0.31	28.0	4841.02	25.0	**0.12**
BerkeleyDB1	**19.6**	47.60	25.0	0.59	24.0	0.08	22.0	664.22	24.0	**0.03**
BerkeleyDB2	**19.0**	5240.35	24.0	1.61	21.0	0.21	22.0	(*)	(*)	**0.05**
Dell	**33.0**	54.95	38.0	0.18	37.0	0.10	43.0	2562.60	43.0	**0.04**
EShopFIDE	24.0	4789.30	22.0	23.16	**21.0**	1.20	(*)	(*)	23.0	**0.30**
EShopSplot	(*)	(*)	23.0	15.57	**21.0**	0.72	(*)	(*)	23.0	**0.22**
GPL	**15.0**	14.58	20.0	0.24	19.0	0.15	18.0	173.65	22.0	**0.06**
SmartHome22	**14.8**	14.40	18.0	0.32	17.0	0.05	17.0	169.56	17.0	**0.03**
BattleofTanks	664.0	16955.84	**448.0**	11.72	460.0	3.38	(*)	(*)	650.0	**2.75**
FM_Test	**40.0**	11945.02	47.0	4.05	45.0	0.62	51.0	16997.68	42.0	**0.20**
BankingSoftware	**37.0**	1423.98	41.0	4.13	42.0	0.48	49.0	19187.96	47.0	**0.15**
ElectronicShoping	31.4	12512.83	23.0	16.08	24.0	0.84	26.0	88680.47	**22.0**	**0.22**
DMIS	27.0	6969.84	27.0	26.19	29.0	1.66	(*)	(*)	**26.0**	**0.47**
eCos 3.0 i386pc	(*)	(*)	66.0	1334.02	**63.0**	78.92	(*)	(*)	64.0	**22.90**
FreeBSD kernel 8.0.0	(*)	(*)	77.0	2155.77	77.0	105.28	(*)	(*)	**75.0**	**32.68**
Automotive1	(*)	(*)	**913.0**	56344.51	**913.0**	7602.94	(*)	(*)	946.0	**510.91**
Linux_2_6_28_6	(*)	(*)	(*)	(*)	479.0	18805.14	(*)	(*)	**450.0**	**3296.41**
10xAFM15	**11.0**	0.77	12.0	0.02	12.0	0.02	13.2	3.20	16.7	**0.01**
10xAFM50	**24.0**	30.84	26.9	0.27	26.6	0.06	28.0	643.10	28.0	**0.02**
10xAFM100	**49.4**	3066.64	56.0	1.45	56.6	0.27	63.5	11991.00	58.5	**0.09**
10xAFM200	(*)	(*)	**88.1**	9.64	89	1.19	(*)	(*)	91.7	**0.43**
10xAFM500	(*)	(*)	185.2	189.4	189.1	14.97	(*)	(*)	**186.4**	**4.76**
10xAFM1K	(*)	(*)	346.7	1744.55	346.7	228.98	(*)	(*)	**340.2**	**35.66**
1xAFM5K	(*)	(*)	(*)	(*)	685.0	35172.81	(*)	(*)	**607.0**	**1870.72**

Table 2. Comparison of sampling with CASA, Chvatal, ICPL, and IPOG using feature models of different sizes. The values next to the artificial feature models 10xAFM15, 10xAFM50, 10xAFM100, 10xAFM200, 10xAFM500, and 10xAFM1000 represent the average values over 10 feature models. Conf. stands for configurations. time is measured in second, and * indicates that computation did not finish within 24 hours

We conclude from the results in *Table* 2 and *Figure* 2 that IncLing is more efficient than the existing sampling algorithms with respect to the required time to achieve the pairwise coverage. In future work, we plan to parallelize the IncLing algorithm, which may improve the performance of the algorithm even more.

Testing Efficiency (RQ2). Regarding RQ2, we expect that we need more products to achieve pairwise coverage compared to existing sampling algorithms. We collected the number of generated products for each feature model to achieve pairwise coverage. In *Table* 2, where we report the number of products and the required time to generate them for each feature model, we highlight the minimum ones in bold font. As illustrated in *Table* 2, CASA generates the least number of products for most feature models that it was able to sample (12 out of 16). In addition, we observe that IPOG does not generate the minimum number of products for any feature model. In the case of Chvatal, ICPL, and IncLing, they generate the minimum number of products for 3, 4, and 7 feature models, respectively.

Moreover, we conducted a Mann-Whitney U Test to investigate whether the differences between IncLing and the existing sampling algorithms are significant regarding the number of generated products for each feature model. The Mann-Whitney U Test is a non-parametric statistical test for assessing whether two independent samples are larger than the other. From this test, we obtain a probability value called *p-value*, which represents the probability that both samples are equal. The difference is significant if the p-value is lower than 0.05. In our results, we observe that the difference is not significant between IncLing and CASA, Chvatal, ICPL, and IPOG with p-values 0.12, 0.70, 0.65, and 0.71, respectively. While it is not our primary goal to generate the minimum number of products, p-values above indicate that all sampling algorithms have the same testing efficiency, because the differences are not significant.

Testing Effectiveness (RQ3). To answer RQ3, we measure the potential loss of testing effectiveness of IncLing with respect to the increase of interaction coverage achieved by the product order compared to random configurations and the existing sampling algorithms. In *Figure* 3, we show the percentage of covered combinations to the percentage of configurations for the 38 feature models that could be sampled by all evaluated algorithms. From *Figure* 3, we observe that on average for the first 40% of the generated configurations, IncLing covers more combinations than the other sampling algorithms.

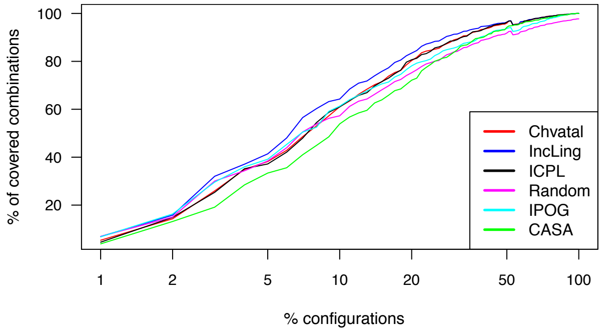

Figure 3. Average percentage of covered combinations over all feature models, with all existing sampling algorithms, in addition to random configurations.

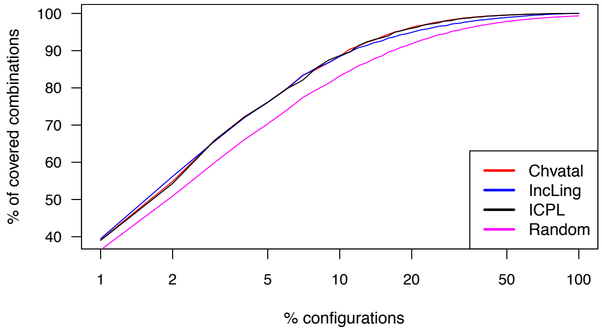

Figure 4. Average percentage of covered combinations for feature models between 200 and 3000 features using IncLing, ICPL, Chvatal, and random configurations.

IPOG covers more combinations than CASA, Chvatal, ICPL, and random configurations until approximately 8%. However, with random, we do not achieve 100% coverage. Surprisingly, CASA performs worse than all other sampling algorithms and even than random configurations.

In *Figure* 4, we show the average percentage of covered feature combinations for the relative number of configurations generated by Chvatal, IncLing, ICPL, and random configurations considering only feature models between 200 and 3,000 features. We find that IncLing covers more feature combinations for the first 3% of configurations than Chvatal, ICPL, and random. For Chvatal and ICPL, we notice that they behave almost similar. In case of random configurations, it does not compete with the other sampling algorithms. From *Figures* 3 and 4, we observe a reduction of the covered combinations by IncLing for the first generated configurations. However, 3% of the configurations is still an acceptable rate. For instance, for product line *Automotive1*, 3% means that IncLing covers more feature combinations for the first 27 configurations (the total number is 913) than the other sampling algorithms.

In *Figures* 5 and 6, we show the aggregated number of covered combinations of product lines *AFM5K* and *the Linux kernel* for IncLing, ICPL, and random configurations. Note

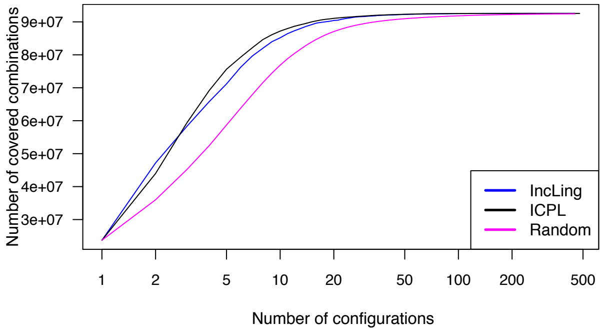

Figure 5. Number of covered combinations for the feature model of Linux kernel (6,888 features) using IncLing, ICPL, and random configurations.

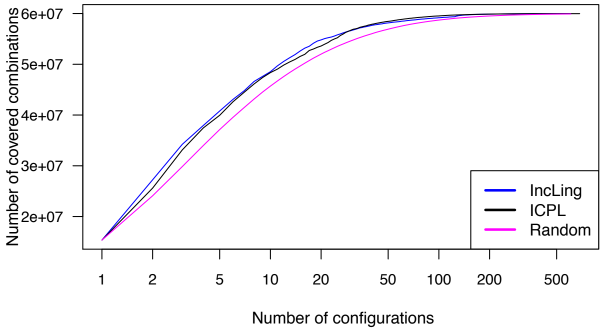

Figure 6. Number of covered combinations for an artificial feature model (5,542 features) using IncLing, ICPL, and random configurations.

that IncLing and ICPL are the only sampling algorithms scaling to these large feature models. The results show that IncLing covers more feature combinations in the first 26 and three configurations for *AFM5K* and *Linux*, respectively. We argue that increasing the diversity among configurations, which is intended in IncLing, yields to cover more feature combinations as early as possible. As a result, many faults may be detected more quickly, which will enhance the testing effectiveness. To answer RQ3, we show that there is no loss in testing effectiveness with respect to the interaction coverage of IncLing. In fact, we found that IncLing increases the testing effectiveness regarding the interaction coverage, because up to the first 40% of the generated configurations by IncLing cover more feature combinations than all existing sampling algorithms.

4.3 Threats to Validity

Internal Validity There is a potential threat that may affect the results with random configurations. To mitigate random effects, we conducted the experiments 100 times. Another internal threat is that we compared IncLing to the existing sampling algorithms. Some of these sampling algorithms, such as CASA, are known to be non-deterministic. Moreover, in some cases we also observed a non-deterministic behavior for Chvatal and ICPL. To minimize the impact of this threat, we repeated all experiments five times.

External Validity We cannot guaranty that the artificial feature models we used in our evaluation are able to simulate realistic real-world product lines. What mitigates this threat is that we evaluate IncLing using artificial feature models of different sizes and complexity, which already served as a benchmark to evaluate product-line testing [16, 18, 19]. In addition, we used real-world feature models, which represent the variability of complex product lines, such as the Linux kernel with 6,888 features. Another threat relates to interaction coverage, which is considered, in this paper, as a testing effectiveness measure.

5. Related Work

Several approaches have been proposed to sample sets of products [7, 10, 11, 18, 19, 21, 26, 31, 34]. Combinatorial interaction testing (CIT) is a promising approach that has been used to select a subset of products [23]. Oster et al. [29] present pairwise testing to generate products from the feature models. Moreover, they take pre-selected products (i.e., current products) into account. Similar to their approach, we consider pairwise testing to generate products based on feature models. However, we generate the products incrementally and efficiently. Perrouin et al. [32] propose a non-deterministic approach to generate T-wise products using Alloy. However, their approach does not scale to large feature models. With our approach, we did not face scalability problems for large feature models.

Johansen et al. [18] adopt the Chvatal algorithm [8] to generate covering arrays from feature models. They propose ICPL based on Chvatal with several enhancements, such as parallelizing the process of generating covering arrays, which reduces the computation time significantly [19]. ICPL tries to cover as many uncovered feature combinations as possible with each configuration added to the covering array. The configurations are included to the covering arrays until all valid combinations of features are covered. Although the process of generating the covering arrays is incremental, with the current implementation of the algorithm, the users have to wait until all valid combinations are covered to generate and test products. With IncLing, we generate the configurations incrementally. Despite we are considering the pairwise coverage, we can also generate configurations within a given testing time. Furthermore, with our approach, we are trying to increase the diversity among the created configurations. In addition, the results show that the performance of our approach potentially outperforms the performance of ICPL, especially for large feature models.

Garvin et al. [15] generate covering arrays using simulated annealing. The algorithm does not scale to large feature models and it takes a long time to generate the covering arrays. With our approach, the performance outperforms CASA. However, considering a search-based algorithm may help our approach to avoid being trapped in local optima. Hence, in future, we plan to investigate how we can get benefit from search-based techniques. Henard et al. [16] propose an alternative to CIT by employing a search-based approach to sample products. They propose the similarity notion to increase the interaction coverage for the generated products. Similar to their approach, with IncLing, we can test a fixed number of products or test as many product as possible in a fixed time. In addition, with our approach, we consider the pairwise CIT to sample products. Moreover, our approach is deterministic, which is not the case with this search-based approach, where different products might be generated in each run.

Kowal et al. [22] propose to provide an additional information to feature model about the actual source of feature interaction. They use this information to reduce the number of products that need to be tested. However, the additional information is typically not available. With IncLing, we are trying not only to reduce the number of products, but also to generate them incrementally and efficiently.

Medeiros et al. [27] compare ten sampling algorithms regarding the size of the samples and their capability of fault detection. They report that existing sampling algorithms do not scale well, as they require a considerable amount of time. With IncLing, we show that its efficiency outperforms all sampling algorithms for pairwise coverage.

6. Conclusion and Future Work

Several approaches have been proposed to sample a set of products as representatives while achieving a certain degree of coverage. However, existing sampling algorithms require a considerable amount of time to sample products, which may not all be tested due to the limitation of testing time. Thus, we propose IncLing to sample products one at a time. With our approach, there is no need to wait for a long time to have the first samples. Beside generating a fixed number of products withing a given time, we can achieve a pairwise coverage efficiently. In particular, the results show improvements compared to existing sampling algorithms with respect to the required time to compute samples. Regarding the number of generated products, the differences between IncLing and existing sampling algorithms are not significant. Moreover, IncLing covers as many feature interactions as soon as possible by increasing the diversity among the created configurations, which is likely to enhance product-line testing effectiveness.

In future work, we plan to consider search-based approaches to avoid being trapped in local optima. In addition, we plan to parallelize IncLing in order to enhance its efficiency even further.

Acknowledgments

We are grateful to Jens Meinicke for interesting discussions about the algorithm. This work has been supported by the German Research Foundation (DFG) in the Priority Programme SPP 1593: Design For Future Managed Software Evolution (LO 2198)

References

[1] M. Al-Hajjaji. Scalable Sampling and Prioritization for Product-Line Testing. In *Proc. Software Engineering (SE)*, pages 295–298. Gesellschaft für Informatik (GI), 2015.

[2] M. Al-Hajjaji, J. Meinicke, S. Krieter, R. Schröter, T. Thüm, T. Leich, and G. Saake. Tool Demo: Testing Configurable Systems with FeatureIDE. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, 2016. To appear.

[3] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-Based Prioritization in Software Product-Line Testing. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 197–206. ACM, 2014.

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

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Engineering (TSE)*, 30(6):355–371, 2004.

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

[7] I. D. Carmo Machado, J. D. McGregor, Y. a. C. Cavalcanti, and E. S. De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)*, 56(10):1183–1199, 2014.

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

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[10] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and Adequacy in Software Product Line Testing. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 53–63. ACM, 2006.

[11] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proc. Int'l Symposium in Software Testing and Analysis (ISSTA)*, pages 129–139. ACM, 2007.

[12] S. A. Cook. The Complexity of Theorem-proving Procedures. In *Proc. of ACM Symposium on Theory of Computing (STOC)*, pages 151–158. ACM, 1971.

[13] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, 2000.

[14] L. Engebretsen. The Nonapproximability of Non-Boolean Predicates. *SIAM Journal on Discrete Mathematics (JDM)*, 18(1):114–129, 2005.

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

[16] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Le Traon. 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):650–670, 2014.

[17] M. Jackson and P. Zave. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *IEEE Trans. Software Engineering (TSE)*, 24(10):831–847, 1998.

[18] M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 638–652. Springer, 2011.

[19] M. F. Johansen, Ø. Haugen, and F. Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 46–55. ACM, 2012.

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

[21] C. H. P. Kim, D. Batory, and S. Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 57—68. ACM, 2011.

[22] M. Kowal, S. Schulze, and I. Schaefer. Towards Efficient SPL Testing by Variant Reduction. In *Proce. Int'l workshop on Variability & composition (VariComp)*, pages 1–6. ACM, 2013.

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

[24] Y. Lei, R. N. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG: A General Strategy for T-Way Software Testing. In *Proc. Int'l Conf. Engineering of Computer-Based Systems (ECBS)*, pages 549–556. IEEE, 2007.

[25] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM, 2013.

[26] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental Model-Based Testing of Delta-oriented Software Product Lines. In *Proc. of Int'l Conf. on Tests and Proofs (TAP)*, pages 67–82. Springer, 2012.

[27] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, ICSE '16, pages 643–654. ACM, 2016.

[28] J. Melo, E. Flesborg, C. Brabrand, and A. Wasowski. A Quantitative Analysis of Variability Warnings in Linux. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, VaMoS '16, pages 3–8. ACM, 2016.

[29] S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int'l Software Product Line Conf. (SPLC)*, pages 196–210. Springer, 2010.

[30] S. Oster, I. Zorcic, F. Markert, and M. Lochau. MoSoPoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing. In *Proc. Int'l Workshop Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 79–82. ACM, 2011.

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

[32] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 459–468. IEEE, 2010.

[33] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[34] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*, pages 270–284. Springer, 2012.

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

[36] D. M. Weiss. The Product Line Hall of Fame. In *Proc. Int'l Software Product Line Conf. (SPLC)*, page 395. IEEE, 2008.