YASA: Yet Another Sampling Algorithm

Sebastian Krieter
University of Magdeburg
Magdeburg, Germany
Harz University of Applied Sciences
Wernigerode, Germany

Thomas Thüm
University of Ulm
Ulm, Germany

Sandro Schulze
University of Magdeburg
Magdeburg, Germany

Gunter Saake
University of Magdeburg
Magdeburg, Germany

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany

ABSTRACT

Configurable systems allow users to derive customized software variants with behavior and functionalities tailored to individual needs. Developers of these configurable systems need to ensure that each configured software variant works as intended. Thus, software testing becomes highly relevant, but also highly expensive due to large configuration spaces that grow exponentially in the number of features. To this end, sampling techniques, such as t-wise interaction sampling, are used to generate a small yet representative subset of configurations, which can be tested even with a limited amount of resources. However, even state-of-the-art t-wise interaction sampling techniques do not scale well for systems with large configuration spaces. In this paper, we introduce the configurable technique YASA that aims to be more efficient than other existing techniques and enables control over trading-off sampling time and sample size. The general algorithm of YASA is based on the existing technique IPOG, but introduces several improvements and options to adapt the sampling procedure to a given configurable system. We evaluate our approach in terms of sampling time and sample size by comparing it to existing t-wise interaction sampling techniques. We find that YASA performs well even for large-scale system and is also able to produce smaller samples than existing techniques.

KEYWORDS

Configurable System, Software Product Lines, T-Wise Sampling, Product-Based Testing

ACM Reference Format:
Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2019. YASA: Yet Another Sampling Algorithm. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https: //doi.org/10.1145/3377024.3377042

VaMoS '20, February 5–7, 2020, Magdeburg, Germany

ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377042

1 INTRODUCTION

Software testing is an important task in software engineering to increase software quality and to check intended software behavior [5, 38]. However, extensive testing can be quite costly and binds resources that could be used in other phases of development. This is especially an issue for testing of highly-configurable systems, such as Software Product Lines (SPLs), whose functionality is determined by configurations, consisting of a set of options that can be either set to *true* or to *false*. The resulting configuration space typically grows exponentially, regarding the number of configuration options [12, 32].

A straight-forward testing strategy for SPLs is product-based testing, in which the test cases of a system are executed for different configurations [50]. As testing every possible configuration is usually not feasible, sampling strategies, have been developed to generate a small but representative set of products to test [42, 43]. One such sampling strategy is t-wise interaction sampling, which aims to generate a small set of configurations that covers all possible interactions of t configuration options (e.g., all selected, none selected, only one selected, etc.) [10, 36]. Using t-wise interaction sampling, developers can ensure that each possible combination of t configuration options is indeed contained in at least one configuration in the generated sample.

T-wise interaction sampling is a promising approach, because even when using small values for t (i.e., $t \in \{2, 3\}$) it achieves effective results with a relatively small sample [1, 26, 36]. However, even when using small values of t and a state-of-the-art sampling algorithm, sampling can take an infeasible amount of time, especially regarding large-scale systems with thousands of features [44].

In this paper, we want to tackle the scalability problem of t-wise interaction sampling for large-scale systems. To this end, we introduce a new approach for t-wise interaction sampling that aims to be efficient and more scalable and flexible than existing approaches. For this, we focus on three improvements compared to existing approaches. First, we generalize t-wise interaction sampling to not only cover every possible t-wise feature interaction, but a customized set of feature interactions. This allows to add domain knowledge for a system to the sampling process making it more efficient. Second, in our approach we apply heuristics, caching, and precomputed data structures to increase its performance. Third, we introduce additional parameters to adapt some of our used heuristics to enable more control over the trade-off between sampling time and sample size.

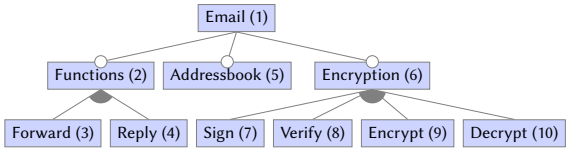

Figure 1: Example feature model of an email client.

We implemented our approach within a prototype called *YASA (Yet Another Sampling Algorithm)* to investigate its sampling time and sampling size compared to other state-of-the-art algorithms for t-wise interaction sampling. In summary, we contribute the following:

- We propose a configurable sampling algorithm for t-wise interaction sampling.
- We provide an open-source implementation as part of FeatureIDE[1].
- We evaluate our approach using feature models from multiple real-world systems.

2 FOUNDATIONS OF CONFIGURABLE SYSTEMS

Our approach takes a feature model as input and generates a list of configurations (i.e., a sample). Thus, in the following, we revisit the basic notion of feature models and configurations. We formally define all notions using propositional formulas and set notation.

2.1 Feature Models

A feature model specifies all features of an SPL and their interdependencies [6, 8, 45]. We formally define a feature model $\mathcal{M} = (\mathcal{F}, \mathcal{D})$ as a tuple, consisting of a set of features $\mathcal{F}$ and a set of dependencies $\mathcal{D}$. As a notational convention, we represent each feature of the feature model as a single integer number ranging from 1 to n (i.e., $\mathcal{F} = \{1, ..., n\}$), where n is the total number of features in the feature model. We represent the dependencies of a feature model as clauses of a propositional formula in conjunctive normal form (CNF). Each dependency in $\mathcal{D}$ represents one such clause (i.e., $\mathcal{D} = \{D_1, ..., D_m\}$, where m is the number of clauses). We denote a clause as a set of literals. In our notation, a literal is either a number from $\mathcal{F}$ (i.e., a positive literal) or a negated number from $\mathcal{F}$ (i.e., a negative literal). We define the function $\mathcal{L}$ that provides the set of literals for a feature set of a feature model, $\mathcal{L}(\mathcal{F}) = \{-n, ..., -1, 1, ..., n\}$.

For example, we depict a feature model in form of a feature diagram in Figure 1. The feature model represents a simplified email client and consists of ten features, which we each map to a number from 1 to 10: Email (1), Functions (2), Forward (3), Reply (4), Addressbook (5), Encryption (6), Sign (7), Verify (8), Encrypt (9), Decrypt (10). The dependencies between features are specified via the hierarchy and the edge types in the feature diagram. For instance, the dependency between the features `Functions`, `Reply`, and `Forward` is represented by an OR-group edge type in the diagram and can be written as $(\neg Functions \vee Reply \vee Forward)$ in CNF or $\mathcal{D} = \{\{-2, 3, 4\}, ...\}$ in our notation.

[1] https://featureide.github.io/

2.2 Configurations

A configuration represents a selection of features from a feature model. For a configuration, we can derive the corresponding product using the variability mechanism of the SPL [6, 8, 45]. We define a configuration as a set of literals C, such that $C \subseteq \mathcal{L}(\mathcal{M})$ with $\forall l \in C : -l \notin C$. If a literal is contained in a configuration, the corresponding feature is defined as either selected (positive literal) or deselected (negative literal).

If all features are defined within a configuration, we call it *complete* and otherwise *partial*. We define this with the function:

$$complete(C, \mathcal{M}) = \begin{cases} true & |C| = |\mathcal{F}| \\ false & \text{otherwise} \end{cases}$$

For instance, corresponding to the feature model in Figure 1, a *complete* configuration, which selects all features from the product line, can be written as $C_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. In contrast, the configuration $C_2 = \{1, 2, 3, 4, 5, -6\}$ is *incomplete* as it does not define values for the child features of the feature `Encryption`.

If a configuration contains at least one literal from a clause in $\mathcal{D}$, it satisfies this clause. Contrary, if a configuration contains all complementary literals of a clause, it contradicts this clause and hence the entire feature model. Thus, if a configuration contradicts at least one clause, we call it *invalid*. Note that, a partial configuration may neither satisfy nor contradict a clause. We call a configuration *valid*, if it allows all clauses of a feature model to be satisfied:

$$valid(C, \mathcal{M}) = \begin{cases} true & \exists C' \supseteq C : \forall D \in \mathcal{D} : C' \cap D \neq \emptyset \\ false & \text{otherwise} \end{cases}$$

A valid configuration may be partial and is not required to satisfy all clauses, as long as all clauses can be satisfied by adding more literals to the configuration.

Considering the feature model in Figure 1, an example for a *valid*, partial configuration is $C_3 = \{1, 2, -3, 4, 5, 6\}$. Although it does not contain values for the child features of `Encryption`, it does not contradict any constraint from the model either and can fulfill all constraints by defining the missing values accordingly. On the other hand, the partial configuration $C_4 = \{1, 2, -3, -4, 5, -6\}$ is *invalid* as it contradicts the clause $D_1 = \{-2, 3, 4\}$ from the feature model (i.e., $C_4 \cap D_1 = \emptyset$).

The result of a t-wise interaction sampling is a configuration sample $\mathcal{S}$. This sample is a set that contains valid configurations (i.e., $\mathcal{S} = \{C_1, C_2, ...\}$). In a complete t-wise interaction sample (i.e., 100% t-wise interaction coverage) every valid interaction (i.e., does not contradict the feature model) is a subset of at least one configuration. An interaction I is represented by a set of exactly t literals (e.g., for $t = 3$: $I = \{2, -3, 4\}$) and is considered valid if the partial configuration containing only this set of literals is also valid.

3 T-WISE SAMPLING WITH YASA

In the following, we present our new approach for t-wise interaction sampling. It consists of a basic algorithm that starts with a given empty sample and then iterates over all t-wise interactions one at a time. For each, we either add a new partial configuration with the literals of the interaction to the sample or we add the literals to an existing configuration. We enhance this basic algorithm by applying

different heuristic and caching methods that aim to improve its sampling time. In addition, we add several parameters, compared to existing approaches, that enable users to fine-tune the trade-off between sampling time and sample size.

In the following, we start with describing the input parameters of our approach. Then, we briefly describe its basic algorithm and go into the details about our employed heuristics and caching methods.

3.1 Input and Parameters

Common to other sampling algorithms, our approach requires a value for the parameter t and a set of variables along with their interdependencies, which, in our context, are specified within the feature model. However, as we aim to develop a flexible approach, such that we can fine-tune it for a given system, we included three additional parameters. First, we can divide the set of variables (i.e., features) into smaller subsets to reduce the number of considered interactions. Second, we allow to customize the employed covering strategy. Finally, we include a mechanism in our algorithm for refining the resulting sample by removing and resampling some configurations (cf. Section 3.2.2), which can be controlled with an additional parameter m. In the following, we describe these parameters in more detail.

Feature Model and Interaction Size. Both, the parameter t and a feature model are common to most t-wise sampling algorithms. With the feature model, we specify the set of features and all of their interdependencies, which in turn defines the set of all possible valid configurations. According to Section 2, we assume that every feature is of boolean nature, and thus can be either selected or deselected in a configuration. Regarding dependencies, we allow any kind of dependencies between any number of features that can be written as an expression in propositional logic.

The parameter t defines how many features are considered in an interaction. In theory, our approach allows any integer value greater than or equal to one. Practically the value of t is limited by the sampling time, which increases drastically for higher values of t, and the memory it needs to store the resulting sample.

Feature Subsets. In regular t-wise interaction sampling every possible subset of features of size t is considered. However, in large systems, many of these features may not interact at all, maybe because they belong to different subsystems or maybe because there is no data or control flow between them. In the case that developers are aware of such circumstances, it can be helpful to ignore the combinations between these features for sampling, as it can save sampling time and reduce the sample size. Therefore, instead of combining every feature from $\mathcal{F}$ with each other, we enable users to specify a set $\mathcal{G}$ of (possibly overlapping) subsets from $\mathcal{F}$, which are used for building the interactions (cf. Section 3.2). The actual distribution of features into subsets could for instance be derived from domain knowledge about the system or through source code analyses (e.g., data flow analysis).

For instance, in our example in Figure 1, we might be aware that some features (e.g., `Reply` and `Decrypt`) are not interacting. In this case, we can divide the feature set into smaller subsets; for instance into $\mathcal{G} = \{\{1, 2, 3, 4, 5, 7, 9\}, \{1, 5, 6, 7, 8, 9, 10\}\}$. Note that these subsets do overlap.

Naturally, this parameter increases the risk of missing actual interactions that were not anticipated, and thus are then overlooked during testing. However, when applied carefully using feature subsets has the potential to drastically reduce the amount of interactions that need to be covered, which decreases sampling time as well as sample size. Nevertheless, it is possible to only specify a single set that contains all features instead of multiple subsets, which would serve as regular t-wise interaction sampling.

Covering Strategy. Our algorithm processes all considered interactions sequentially with a user provided covering strategy *CS*. For each given interaction, the applied covering strategy determines which partial configuration in the sample is used to cover it. For this, a covering strategy comprises one function that takes three parameters, the feature model, the current sample and one interaction and tries to cover this interaction within the sample. Employing different covering strategies can affect the sampling time and sample size. Thus, enabling users to customize the sampling strategy can again help them to fine-tune the sampling process to their needs.

In this paper, we focus on our default covering strategy that has been mostly optimized for sampling time (cf. Section 3.2). However, by applying a different strategy it is possible to trade-off sampling time for a smaller sample size or vice-versa.

Removing and Resampling Configurations. As our approach consists of a greedy algorithm and employs heuristics, the resulting sample size is most likely not minimal. To further decrease the sample size, we integrate a mechanism into our algorithm to refine the sample by removing configurations that only cover a small amount of interactions. Afterwards, we reiterate over all interactions that are now uncovered and complete the sample again (cf. Section 3.2.2).

We can repeat the process of removing and resampling several times to improve the overall result. However, increasing the number of iterations also increases the sampling time significantly. During the development and testing of our algorithm, we noticed that increasing the number of iterations above a certain point no longer yields substantial improvements on the sample size. Though, how many iterations are reasonable, depends on the sampled product lines. Thus, we included a parameter m to specify the number of iterations for removing and resampling. Users can choose for themselves by how much they are willing to increase the sampling time in order to reduce the sample size.

3.2 Constructing a Configuration Sample

We depict the basic outline of our algorithm in pseudo code in Algorithm 1. Within this outline, we use all of our three additional parameters (highlighted in blue).

We start by initializing the sample $\mathcal{S}$ with an empty set (Line 1). Then, we iterate over every feature subset $\mathcal{F}_i$ in $\mathcal{G}$ (Line 4). For every feature subset, we iterate over all possible subsets I with size t (Line 6) and call the cover function of the provided covering strategy (Line 7). After every feature subset is processed, all interactions are covered by $\mathcal{S}$. However, we can repeat the covering process to reduce the number of configurations in the sample. For this, we first call the function `trim` that removes configuration that cover only a few interactions (Line 3). Then we reiterate through all subsets

Algorithm 1: Basic sampling algorithm.

Data: Feature Model $\mathcal{M}$, Interaction Size t, Feature Subsets $\mathcal{G}$, Covering Strategy CS, Number of Iterations m
Result: Configuration Sample $\mathcal{S}$

```
1 S ← ∅
2 for 1 to m do
3     S ← trim(S)
4     for F_i ∈ G do
5         L_I ← L(F_i)
6         for I ∈ L_i^t do
7             S ← CS.cover(M, I, S)
8 S ← autocomplete(S)
9 return S
```

and interactions and apply the covering strategy again. In total, we repeat the covering process m times (Line 2). Finally, we complete any partial configuration in the final sample by randomly choosing values for undefined features (Line 8).

3.2.1 Building Interactions from Feature Subsets. The first step for each feature subset is to determine the list of interactions. To this end, we process the set of features, as follows. First, we filter all core and dead features (i.e., features that can only be either selected or deselected according to the feature model), as interactions with these will be covered automatically. Then we create a set of literals L_I for the feature subset $\mathcal{F}_i$ by building a set that contains every feature in the feature subset and its complement (i.e., $L_I = \mathcal{L}(\mathcal{F}_i)$). From theses literal sets, we determine the interactions by generating all possible subsets with size t. In our example, in Figure 1 we create the literal sets $\mathcal{L}_1 = \{-9, -7, -5, -4, -3, -2, 2, 3, 4, 5, 7, 9\}$ and $\mathcal{L}_2 = \{-10, -9, -8, -7, -6, -5, 5, 6, 7, 8, 9, 10\}$. Thus, for $t = 2$ we would iterate over all pairs from $\mathcal{L}_1$ (i.e., $\{-9, -7\}, \{-9, -5\}, \ldots$) and over all pairs from $\mathcal{L}_2$ (i.e., $\{-10, -9\}, \{-10, -8\}, \ldots$).

3.2.2 Trimming the Sample. In order to reduce the sample size, we take a complete sample and remove some configurations before covering any missing interactions again. We determine which configuration to remove by ranking all configurations in the sample by their number of uniquely covered interactions. In detail, we use the following formula for ranking:

$$R(C, \mathcal{S}) = \frac{\sum_{\mathcal{F}_i \in \mathcal{G}} \sum_{I \in \mathcal{L}(\mathcal{F}_i)^t} [I \subseteq C, \nexists C' \in \mathcal{S} : C \neq C', I \subseteq C']}{|C|^t}$$

For each configuration C in the sample $\mathcal{S}$ we count the number of interactions that are only covered by this configuration and none other. As the sample may contain partial configurations, we also factor in the number of defined features for each configuration. A configuration that defines more features has a higher chance of covering an interaction. Thus, for each configuration, we divide its unique interaction count by its number of defined features to the power of t. After calculating a rank for every configuration in $\mathcal{S}$, we compute their arithmetic mean. Then, we remove all configurations that have a rank less than this mean value.

For every repetition of the sampling process, we use a different random ordering for iterating over the interactions. During our experiments, we found that having a different ordering every time helps in finding a good distribution for the interactions within the configuration sample. However, reexecuting the sampling process means that it is also possible for the sample size to increase instead of decrease. To solve this problem, we always remember the smallest sample achieved so far and return it in the end. In case there are more than one smallest samples, we return the first one we found.

3.3 Covering Strategy

As we mentioned, our algorithm is able to use a customized covering strategy. In this paper, we focus on our own covering strategy that we have optimized to decrease sampling time. To begin, we describe the basic idea of the strategy, which is similar to the established algorithm IPOG [33]. Afterwards, we go into detail about our optimizations.

Algorithm 2: Basic covering strategy.

Data: Interaction I, Feature Model $\mathcal{M}$, Configuration Sample $\mathcal{S}$
Result: Configuration Sample $\mathcal{S}$

```
1  if ∄C ∈ S : I ∈ C then
2      if ¬valid(I, M) then
3          return S
4      for C ∈ S do
5          C' ← C ∪ I
6          if valid(C', M) then
7              S ← (S \ {C}) ∪ {C'}
8              return S
9      S ← S ∪ {I}
10 return S
```

We depict the strategy in pseudo code in Algorithm 2. The algorithm takes as input an interaction I (i.e., a set of literals) that it tries to cover, the feature model $\mathcal{M}$, and the current sample $\mathcal{S}$. First, it checks whether there exists a configuration C in the sample that already covers the given interaction (Line 1). If so, it does not modify the sample. Otherwise, it checks whether the interaction is in contradiction with the feature model (i.e., it cannot be covered by any configuration) using a satisfiability (SAT) solver (Line 2). If the interaction is a contradiction the strategy returns without modifying the sample. In contrast, if the interaction is valid, the strategy iterates through all configurations in $\mathcal{S}$ and checks whether the configuration is still valid when adding the literals from I (Line 6). In that case, it adds the literals to the configuration and returns $\mathcal{S}$. Otherwise, the strategy creates a new configuration containing only the literals from I and adds it to the $\mathcal{S}$.

3.4 Optimized Covering Strategy

The basic covering strategy, we presented above, guarantees to produce a complete sample. However, there is much potential for improving the sampling time, which we address with the following

optimized covering strategy. In Algorithm 3, we show our optimized strategy in pseudo code.

As outlined in Algorithm 2, for every given interaction there are four different possibilities. First, the interaction is already covered. Second, the interaction is invalid, because the feature model does not allow it. Third, the interaction can be covered in an existing configuration. Fourth, the interaction can be covered by creating a new configuration. The most potential for increasing sampling time lies in possibilities two and three, because for both we have to check the validity of partial configurations using a SAT solver, which is an expensive operation.

Therefore, one optimization is to restructure the order of operations. We move the more expensive operations to the end of the algorithm, because we are able to cover some interactions before needing to check their validity using a SAT solver. In addition, we create a filtered sample $\mathcal{S}'$ by filtering all configurations that clearly contradict the literal set, because they contain at least one complementary literal (Line 2). Thus, we do not iterate through every configuration, which reduces the amount of SAT solver calls.

3.4.1 Caching. Another optimization is to employ caching for intermediate results. In particular, we insert two new operations before checking the validity of an interaction that employ caching. For the following operations, we exploit the fact that a SAT solver will provide a satisfying assignment (i.e., a valid configuration) if there is a solution. Whenever we receive a positive result from a SAT solver call, we cache the computed configuration. We use these cached configurations in the function $valid_{nosat}$ (cf. Lines 4 and 7). Instead of calling the SAT solver, this function checks whether the interaction is contained in any of the cached configurations. If this is true, we know that the interaction is valid without calling the SAT solver a second time. Note that, when we are applying $valid_{nosat}$ in Line 4, we have to be aware that all already defined variables in C must also be contained in the cached configurations. Naturally, if the return value of $valid_{nosat}$ is false it does not necessarily mean that the interaction is contradicting. In this case we still have to check with the function $valid_{sat}$, which uses the SAT solver (cf. Lines 8 and 12).

3.4.2 Decision Propagation. Yet another optimization for reducing the numbers of SAT solver calls is to increase the number of already covered interactions. We try to increase this number by applying decision propagation [21, 22, 40] on every configuration, whenever we create it or add new literals to it. That means that we compute the literals that are implied by the literals already contained in a partial configuration. As decision propagation is itself an expensive operation we only apply lightweight decision propagation by using an implication graph instead of SAT solvers. Thus, the decision propagation does not determine every implied literal, but only a subset. Nevertheless, it helps to complete the partial configurations and consequently reduces the number of SAT calls for checking the validity of partial configurations and interactions.

Algorithm 3: Optimized covering strategy.

```
   Data: Interaction I, Feature Model M, Configuration
         Sample S
   Result: Configuration Sample S
1  if ∄C ∈ S : I ∈ C then
2      S' ← {C ∈ S | Ī ∩ C = ∅}
3      for C ∈ S' do
4          if valid_nosat(I, C, M) then
5              S ← (S \ {C}) ∪ {C ∪ I}
6              return S
7      if ¬valid_nosat(I, M) then
8          if ¬valid_sat(I, M) then
9              return S
10     for C ∈ S' do
11         C' ← C ∪ I
12         if valid_sat(C', M) then
13             S ← (S \ {C}) ∪ {C'}
14             return S
15     S ← S ∪ {I}
16 return S
```

4 EVALUATION

With YASA we aim to provide an efficient and adaptable sampling approach that scales even to large-scale systems. Thus, we evaluate YASA by comparing it with other t-wise sampling strategies and different parameter settings with regard to the following metrics:

- Sample size (i.e., the number of configurations in a sample)
- Sampling time (i.e., the time required to compute a sample)

In particular, we aim to answer the following research questions:

RQ_1 Does our approach have the potential to generate samples more efficiently than other approaches?

RQ_2 Does our approach have the potential to generate smaller samples than other approaches?

We want to investigate, whether our optimizations in YASA affect the sample size compared to other approaches, as our hypothesis is that our approach is at least as effective as other sampling strategies. Even more, we hypothesize that we can achieve a similar sample size as other approaches with shorter sampling time, and thus increasing sampling efficiency.

4.1 Evaluation Setup

Within our experiments, we compute samples for different systems using our prototype YASA and a selection of other state-of-the-art t-wise interaction sampling algorithms. In the following, we describe the setup for our experiments. First, we introduce the algorithms, which we compare against each other. Second, we present the subject systems, for which we generate samples. Finally, we describe our measuring methods for our two evaluation criteria, sampling time and sample size.

4.1.1 Algorithms. We use several state-of-the-art algorithms for t-wise interaction sampling as comparison, which were also used in previous evaluations [2, 4, 24], namely *Chvátal* [7], *ICPL* [23, 24], and *IncLing* [2]. The implementation of these algorithms is provided within the FeatureIDE Java library [3, 39], which we employ

in our evaluation. Consequently, we implemented YASA in Java and integrated it into the FeatureIDE library as well[2]. Within the implementation of YASA, we employ the SAT solver *Sat4J* [31] to check for validity of configurations and interactions.

Not all algorithms support all values for t. IncLing is designed as a strict pair-wise interaction coverage algorithm, and thus only works for $t = 2$. ICPL support values for t up to 3 and Chvátal up to 4. As described in Section 3.2, in theory, we can run our algorithm with any value for t. However, due the restrictions of the other algorithms for the parameter t, in our experiments, we use $t = 2$.

Regarding YASA, we use the following parameters. We test multiple values for m to evaluate its impact. In particular, we choose the values 1, 5, and 10. Further, we choose to not split the feature set into subsets, in order to ensure a fair comparison of the sample size. As covering strategy, we use our optimized covering strategy, including caching and decision propagation (cf. Section 3.4).

In summary, we compare results from the following algorithms:

Name	Symbol	m	Supported t
Chvátal	Chvatal	—	$1 \leq t \leq 4$
ICPL	ICPL	—	$1 \leq t \leq 3$
IncLing	IncLing	—	$t = 2$
YASA	YASA_1	1	$t \geq 1$
YASA	YASA_5	5	$t \geq 1$
YASA	YASA_10	10	$t \geq 1$

4.1.2 Evaluation System. For a fair comparison, we run all algorithms on the same hard- and software:

- *CPU*: Intel(R) Core(TM) i5-8350U
- *Physical Memory*: 16 GB
- *JVM Memory*: Xmx: 14 GB, Xms: 1 GB
- *OS*: Manjaro (Arch Linux)
- *Java*: OpenJDK 1.8.0_222

4.1.3 Subject Systems. We use several subject systems in our evaluation from different sources with varying sizes. In Table 1, we list the system together with their number of features. In detail, we use 11 small to middle-sized system from FeatureIDE, which have between 27 and 366 features. Using the tool Kclause [41], we extracted feature models from 5 real-world systems that employ Kconfig as a configuration tool. These feature models range from 71 to 1,018 features. Furthermore, we use a selection of 39 feature models from subsystem of the eCos system provided by Knüppel et al. [29] ranging from 1,165 to 1,267 features. Finally, we use a feature model for FreeBSD with 1,396 features and for the Linux kernel with 6,888 features provided by She et al. [47].

4.1.4 Measuring Process. Regarding the sample size, we simply count the number of configurations in each sample computed by each algorithm. For measuring sampling time, we take the time that is needed for generating a sample with each algorithm. To be precise, we use the Java method `System.nanoTime()` to get a time stamp directly before calling a sampling algorithm and use the same method to get a time stamp directly after the algorithm returns. We then compute the difference between both time stamps.

[2]https://github.com/FeatureIDE/FeatureIDE/tree/config_generation

Table 1: Subject systems of the evaluation

Source	System Name	#Features
FeatureIDE Example	gpl	27
FeatureIDE Example	dell	46
FeatureIDE Example	berkeleyDB1	53
FeatureIDE Example	SmartHome22	60
FeatureIDE Example	violet	88
FeatureIDE Example	berkeleyDB2	99
FeatureIDE Example	BattleofTanks	144
FeatureIDE Example	BankingSoftware	176
FeatureIDE Example	eShopFIDE	192
FeatureIDE Example	eShopSplot	287
FeatureIDE Example	DMIE	366
Kconfig Project	fiasco	71
Kconfig Project	axtls	95
Kconfig Project	uclibc-ng	270
Kconfig Project	toybox	323
Kconfig Project	busybox-1_29_2	1,018
Knüppel et al.	eCos Subsystems	1,165 – 1,267
She et al.	FreeBSD 8.0.0	1,396
She et al.	Linux Kernel 2.6.28	6,889

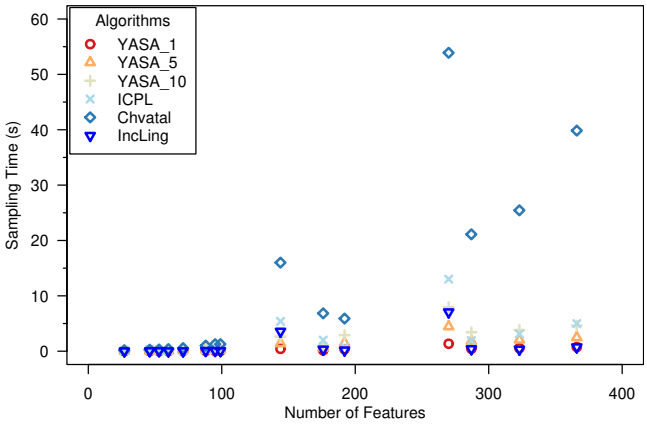

Figure 2: Sampling time for systems with #features < 1,000.

Additionally, we set a timeout of 24 hours for every sampling process. If a sampling algorithms fails to return with this period, we cancel the computation.

4.2 Evaluation Results

We structure our findings according to our two research questions, that is sampling time and sample size. Afterwards, we analyze and discuss our results.

4.2.1 Sampling Time. In Figure 2 and 3, we depict the absolute sampling time for all systems for $t = 2$. On the y-axis we show the sampling time in seconds (s). We relate the values to the number of features within each feature model, which is shown at the x-axis. In order to achieve a better scaling, we present the values in two diagrams with different scales for the y-axis. In Figure 2, we show

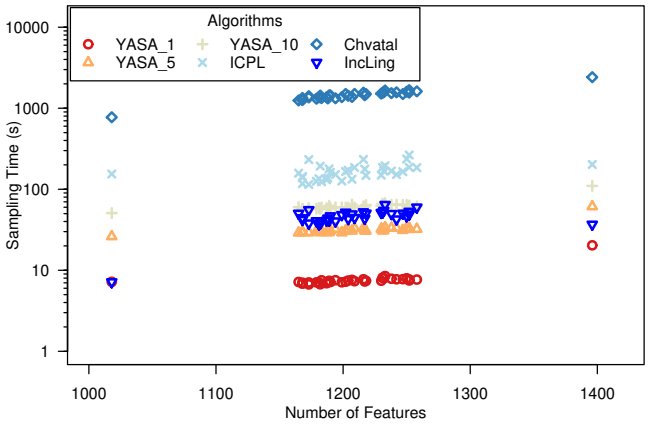

Figure 3: Sampling time for systems with #features ≥ 1,000.

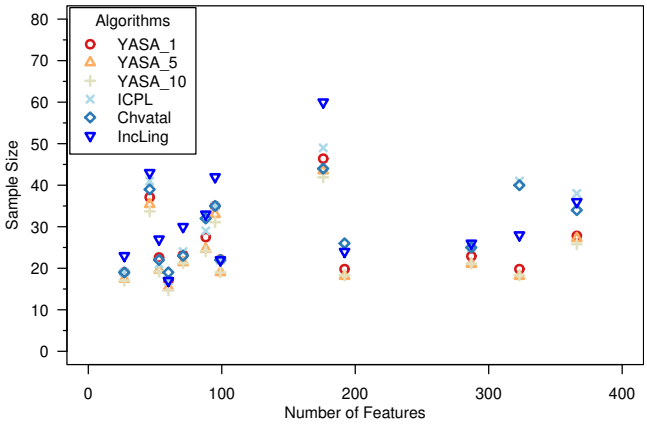

Figure 5: Sample size for systems with #features < 1,000.

Table 2: Results of *Linux 2.6.28*

System Name	Algorithm	Sample Size	Sampling Time
Linux 2.6.28	YASA_1	545	36m 21s
Linux 2.6.28	YASA_5	489	2h 23m 03s
Linux 2.6.28	YASA_10	487	4h 43m 20s
Linux 2.6.28	ICPL	482	4h 00m 14s
Linux 2.6.28	Chvatal	–	*timeout*
Linux 2.6.28	IncLing	781	1h 50m 39s

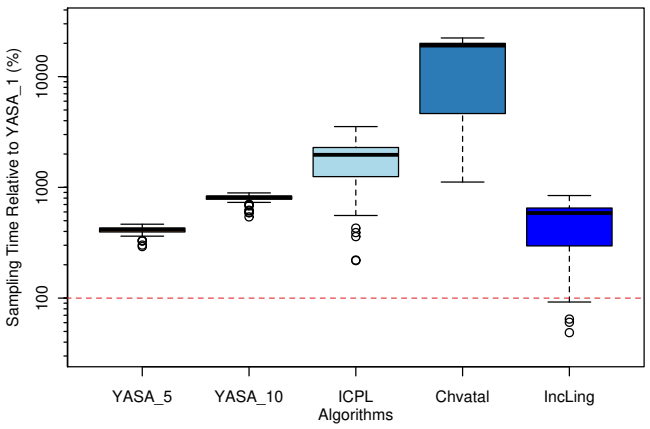

Figure 4: Sampling times relative to YASA ($m = 1$).

all values for feature models with less than 1,000 features and in Figure 3 all values for feature model with more than 1,000 features. Note that, the data in Figure 3 is plotted on a logarithmic scale. Additionally, we show the result for *Linux* in Table 2.

For small systems the sampling time of most algorithms lies below 10 s for our setup. An exception is Chvátal, which takes up to 60 s for some feature models. For larger systems we can see a clearer distinction of the sampling times. Chvátal is the slowest algorithm for all feature models with sampling times between 774 s and 2,418 s. ICPL is the second slowest for almost all models, but is already substantially faster with sampling times ranging from 114 s to 264 s. YASA_10 requires between 50 s and 109 s of sampling time. YASA_5 (26 – 61 s) and IncLing (7 – 64 s) have relatively similar sampling times for most models, while YASA_1 is distinctively faster with sampling times ranging from 6 s to 20 s.

In Figure 4, we show the sampling times for YASA_5, YASA_10, ICPL, Chvátal, and IncLing relative to YASA_1 (i.e., YASA_1 is 100%). With this figure, we visualize the differences in runtime for the different algorithms. Note that, the data is plotted on a logarithmic scale. While there are some cases where YASA_1 is outperformed by IncLing, on average all other algorithms require more sampling time. Using the Mann-Whitney test, we can see that this is significant with p-values all $< 10^{-7}$ (i.e., comparing YASA_1 with all other algorithms). Chvátal requires the most sampling time and, on average, takes more than 200 times longer than YASA_1. ICPL is much faster than Chvátal, but still, on average, takes about 20 times longer than YASA_1. The sampling time of IncLing is close to YASA_5, but is still out performed by YASA_1 by factor 5. We can see that the parameter m has a clear influence on the sampling time. The higher the value of m, the higher the sampling time.

4.2.2 Sample Size. We depict the sample size for most systems in Figure 5 and 6. On the y-axis we show the sample size. Again, we show the number of features of each feature model on the x-axis. Analogous to the sampling time, in Figure 2 we show all values for feature models with less than 1,000 features and in Figure 3 all values for feature models with more than 1,000 features. We depict the sample size for *Linux* in Table 2.

From both diagrams we can see that the sample size varies greatly for every system and also for the different algorithms. However, for larger systems we can also see a correlation between sample size and the number of features throughout all algorithms. Surprisingly, YASA_10, and even YASA_5, produce mostly smaller samples than all other algorithms. When applying the Mann-Whitney test, we find that this is significant with p-values all $< 10^{-8}$ (i.e., comparing YASA_5 and YASA_10 with all other algorithms).

We visualize the sample size of all algorithms relative to YASA_10 in Figure 7. In the diagram, we can see by how much the samples of YASA_10 are smaller compared to all other algorithms. Both, ICPL

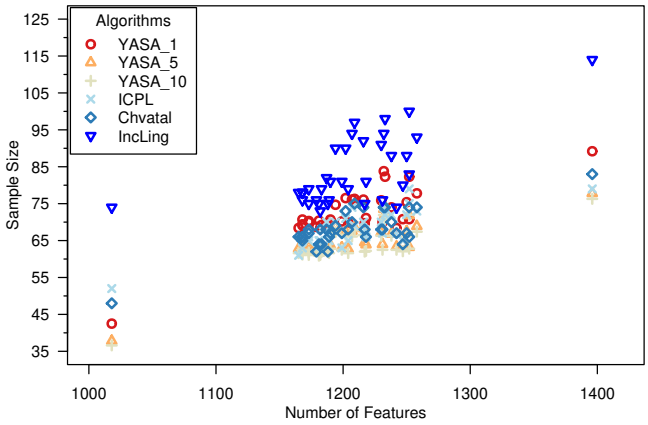

Figure 6: Sample size for systems with #features ≥ 1,000.

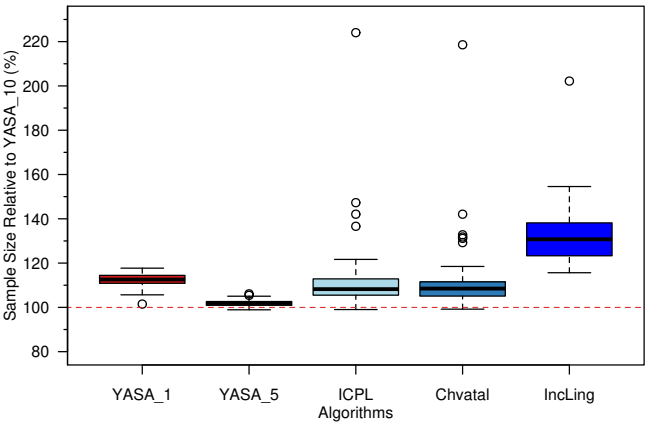

Figure 7: Sample size relative to YASA ($m = 10$).

and Chvátal generate samples that are on average about 10% larger than the samples from YASA_10. We can see a larger difference for IncLing, which generates samples that are on average about 30% larger. As expected, an increase of the value for parameter m decreases the sample size of YASA. However, the average sample size of YASA_5 is close to the average sample size of YASA_10, confirming that a further increase of m only yields little benefit.

4.3 Interpretation

While with YASA we mostly aimed for a more efficient sampling, we also were able to produce relatively small samples for most systems. Thus, we can answer our research questions, as follows. Regarding RQ_1, we found out that our approach is at least as efficient as other state-of-the-art algorithms and can even outperform them. This however, depends on the value of parameter m. We are able to lower the sampling time significantly by decreasing the parameter m, although this has a negative effect on the sample size. In theory, by dividing the feature set into smaller subsets we could decreases the sampling time even further. Nevertheless, we showed that our approach has indeed the potential to efficiently generate samples.

Regarding RQ_2, we found out that we are able to produce smaller samples compared to other tested algorithms. Again, this depends on the value of parameter m. Having a low value for m increases the size of the sample, but increasing m up to a certain point can decrease the sample size by at least 10% compared to $m = 1$.

4.4 Threats to Validity

We are aware that our evaluation might suffer from some biases that may threaten its validity. First, we are using only a relatively small number of feature models in our evaluation. Thus, the results might behave differently for other systems. However, we relied on real-world feature models and tried to included diverse systems differing in origin and number of features. Second, due to the long execution times of some algorithms, we were only able to conduct our experiments once without repetitions. This might threaten the generalizability of our results. Finally, there is the chance of implementation bugs that may bias the result. To this end, we use automated tests to ensure that all computed samples reach a t-wise coverage of 100% and contain only valid configurations.

5 RELATED WORK

Our t-wise sampling interaction sampling algorithm uses a greedy strategy to compute a minimal sample. This strategy is applied by many other sampling algorithms as well [1, 2, 13, 18, 23–25, 27, 28, 30, 34, 42, 46, 48, 49]. In fact, our sampling algorithm is similar to the algorithm IPOG [33], which also starts with an empty sample and iteratively adds partial configurations. However, our sampling algorithm is more flexible and integrates many mechanisms to increase its efficiency.

Furthermore, there also exist many t-wise sampling interaction algorithms that employ meta-heuristic strategies [9, 11, 14–17, 19, 20, 35–37]. These approaches can be seen as alternative or even complementary to our concept, as they could be used to refine the resulting sample.

6 CONCLUSION & FUTURE WORK

In this paper, we presented our new sampling algorithm YASA that aims to be more scalable and flexible than existing algorithms. YASA enables users to adapt the sampling to their system in order to decrease sampling time and sample size. We evaluated our approach by comparing it to existing t-wise interaction sampling algorithms with regard to sampling time and sample size and found that YASA is able to outperform all other tested algorithms in both metrics.

Regarding future work, there are several research aspects, we would like to investigate further. In our current evaluation, we only tested our optimized covering strategy and applied no feature subsets. For further evaluation, we also want to test different covering strategies and different division of the feature set. In addition, we also want to evaluate the impact of different parameter settings on testing effectiveness and efficiency. To this end, we require some kind of measurement for testing effectiveness in order to determine whether a sample performs better or worse than another one.

ACKNOWLEDGMENTS We thank Malte Lochau and Sebastian Ruland for their valuable input and comments. This research is funded by the German Research Foundation (DFG) under Grant LE 3382/2-3 and SA 465/49-3.

REFERENCES

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

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

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

[4] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective Product-Line Testing Using Similarity-Based Product Prioritization. *Software and System Modeling (SoSyM)*, 2019. To appear.

[5] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[6] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

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

[8] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[9] Anastasia Cmyrev and Ralf Reissing. Efficient and Effective Testing of Automotive Software Product Lines. *Int'l J. Applied Science and Technology (IJAST)*, 7(2), 2014.

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

[11] Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Covering SPL Behaviour with Sampled Configurations: An Initial Assessment. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 59:59–59:66. ACM, 2015.

[12] Emelie Engström and Per Runeson. Software Product Line Testing - A Systematic Mapping Study. *J. Information and Software Technology (IST)*, 53:2–13, 2011.

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

[14] Faezeh Ensan, Ebrahim Bagheri, and Dragan Gasevic. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*, volume 7328, pages 613–628. Springer, 2012.

[15] Thiago N. Ferreira, Josiel Neumann Kuk, Aurora Pozo, and Silvia Regina Vergilio. Product Selection Based on Upper Confidence Bound MOEA/D-DRA for Testing Software Product Lines. In *Proc. Congress Evolutionary Computation (CEC)*, pages 4135–4142. IEEE, 2016.

[16] Thiago N. Ferreira, Jackson A. Prado Lima, Andrei Strickler, Josiel N. Kuk, Silvia R. Vergilio, and Aurora Pozo. Hyper-Heuristic Based Product Selection for Software Product Line Testing. *Comp. Intell. Mag. (CIM)*, 12(2):34–45, 2017.

[17] Helson L. Jakubovski Filho, Jackson A. Prado Lima, and Silvia R. Vergilio. Automatic Generation of Search-Based Algorithms Applied to the Feature Testing of Software Product Lines. In *Proc. Brazilian Symposium on Software Engineering (SBES)*, pages 114–123. ACM, 2017.

[18] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 16:1–16:6. ACM, 2013.

[19] Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-Based Generation of Software Product Line Test Configurations. In Claire Le Goues and Shin Yoo, editors, *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer International Publishing, 2014.

[20] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-Objective Test Generation for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 62–71. ACM, 2013.

[21] Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A User Survey of Configuration Challenges in Linux and eCos. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 149–155. ACM, 2012.

[22] Mikolas Janota. Do SAT Solvers Make Good Configurators? pages 191–195, 2008.

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

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

[25] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS)*, pages 269–284. Springer, 2012.

[26] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 181–190. Software Engineering Institute, 2009.

[27] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD)*, pages 57—68. ACM, 2011.

[28] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing Configurations to Monitor in a Software Product Line. In *Proc. Int'l Conf. on Runtime Verification (RV)*, pages 285–299. Springer, 2010.

[29] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is There a Mismatch between Real-World Feature Models and Product-Line Research? In Matthias Tichy, Eric Bodden, Marco Kuhrmann, Stefan Wagner, and Jan-Philipp Steghöfer, editors, *Proc. Software Engineering (SE)*, pages 53–54. Gesellschaft für Informatik, 2018.

[30] Matthias Kowal, Sandro Schulze, and Ina Schaefer. Towards Efficient SPL Testing by Variant Reduction. In *Proc. Int'l Workshop on Variability and Composition (VariComp)*, pages 1–6. ACM, 2013.

[31] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[32] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A Survey on Software Product Line Testing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 31–40. ACM, 2012.

[33] Yu Lei, Raghu N. Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG: A General Strategy for T-Way Software Testing. In *Proc. Int'l Conf. on Engineering of Computer-Based Systems (ECBS)*, pages 549–556. IEEE, 2007.

[34] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM, 2013.

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

[36] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical Pairwise Testing for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 227–235. ACM, 2013.

[37] Rui Angelo Matnei Filho and Silvia Regina Vergilio. A Multi-Objective Test Data Generation Approach for Mutation Testing of Feature Models. 4(1), 2016.

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

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

[40] Marcilio Mendonca and Donald Cowan. Decision-Making Coordination and Efficient Reasoning Techniques for Feature-Based Configuration. *Science of Computer Programming (SCP)*, 75(5):311–332, 2010.

[41] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. Uniform sampling from kconfig feature models. *The University of Texas at Austin, Department of Computer Science, Tech. Rep. TR-19-02*, 2019.

[42] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 196–210. Springer, 2010.

[43] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*, pages 459–468. IEEE, 2010.

[44] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. Product sampling for product lines: The scalability challenge. 2019.

[45] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[46] Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, pages 131–140. ACM, 2015.

[47] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse Engineering Feature Models. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 461–470. ACM, 2011.

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

[49] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. In *Proc. USENIX Annual Technical Conference (ATC)*, pages

421–432. USENIX Association, 2014.

[50] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, 2014.