Stability of Product-Line Sampling in Continuous Integration

Tobias Pett
TU Braunschweig, Germany

Sebastian Krieter
Hochschule Harz, Germany

Tobias Runge
TU Braunschweig, Germany

Thomas Thüm
Universität Ulm, Germany

Malte Lochau
Universität Siegen, Germany

Ina Schaefer
TU Braunschweig, Germany

ABSTRACT

Companies from various industrial branches, such as car manufactures, strive to implement continuous integration into their chain of development to ensure the quality of their systems by testing and re-testing their systems after each update. However, testing all configurations from a highly-configurable software system is not feasible due to the combinatorial-explosion problem. Numerous sampling algorithms have been proposed that aim at computing a considerably smaller yet sufficiently representative set of configurations to be tested. Those algorithms are typically evaluated with regard to efficiency (i.e., number of configurations in a sample and computational effort for generating a sample) and effectiveness (i.e., feature-interaction coverage or number of faults detected). In this paper, we argue that a further crucial characteristic of sampling algorithms is their tendency to produce similar configurations when applied consecutively to an evolving configurable system. We propose *sampling stability* as a new evaluation criterion for sampling algorithms. We present a procedure to compute the sampling stability of sampling algorithms based on the similarity between consecutive samples. In our evaluation, we compare the sampling stability of multiple established t-wise sampling algorithms on large real-world systems.

CCS CONCEPTS

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

KEYWORDS

product lines, sampling, product-line evolution

ACM Reference Format:
Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. 2021. Stability of Product-Line Sampling in Continuous Integration. In *Krems '21: VaMos, February 09–11, 2021, Krems, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

Krems '21, February 01–11, 2021, Krems, Austria

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Configurable systems evolve frequently to adapt to changing customer needs. However, every evolution may potentially introduce new faults or unwanted behavior in existing functionality, which may break parts of the system. To decrease the risk of introducing faults during evolution, continuous integration (CI) is a promising approach. Continuous integration is an iterative procedure that consists of four repeating phases: develop, commit, build, and test [13, 14, 29, 37]. In addition, it makes use of regression testing, which re-uses test assets from each CI-cycle to potentially improve testing efficiency and effectiveness of the next CI-cycle.

Still, effective testing for complex and highly-configurable systems is not trivial due to the well-known combinatorial-explosion problem [5, 6, 8, 28]. Real-world systems often comprise hundreds or even thousands of configuration options (i.e., features), leading to an enormous number of possible configurations, which makes testing every single configuration unfeasible in practice. To cope with the combinatorial-explosion problem, algorithms for configuration sampling have been proposed, which substantially reduce the number of configurations to be tested. They enable effective testing for highly-configurable systems with reasonable effort. Sampling algorithms aim to select a small subset of configurations of a configurable system that is still representative of the variability of the system [12, 23, 27, 40, 41]. Over the past two decades, various sampling algorithms have been proposed [2, 9, 19, 23]. One prominent sampling strategy beyond random sampling is combinatorial interaction testing [33], in which configurations are selected in such a way that they cover interactions between any t features [10, 11, 26].

Typically, the quality of sampling algorithms is assessed by measuring their efficiency (i.e., number of configurations in a sample and computational effort for generating a sample) and effectiveness (i.e., feature-interaction coverage and the number of faults detected) [27, 40]. However, these metrics fail to evaluate a property of sampling algorithms that is highly relevant for regression testing in continuous integration, namely *sampling stability*. After the evolution of a configurable system, a new sample needs to be generated for the CI testing phase. If already tested functionality is tested again, a similar sample should be used. If new functionality is tested, a sample containing still untested configurations is needed. Knowledge of a sampling algorithm's tendency to produces similar samples (i.e., stable behavior), or dissimilar samples (i.e., unstable behavior) increases the confidence in the results of sample-based regression testing.

To this end, we propose the novel metric *sampling stability* for sampling algorithms, which is based on the similarity of samples for consecutive versions of a configurable system over time. According

to this metric, a sampling algorithm A is more *stable* than another sampling algorithm A', if A generates more similar samples for consecutive versions of a configurable system than A'.

We evaluate the *sampling stability* of multiple sampling algorithms for combinatorial interaction testing, namely Chvatal [9], ICPL [18, 19], IncLing [2], and YASA [21] on the full history and monthly snapshots of Busybox. Our evaluation shows that *sampling stability* depends on the algorithms' implementation and the extent of changes between the feature model in the evolution history.

2 SAMPLE-BASED TESTING

In the following, we define fundamental terms for managing the variability of configurable systems.

2.1 Feature Models and Configurations

Users can configure a configurable system by selecting and deselecting features. The resulting set of selected and deselected features is called a *configuration*. Typically, there exist dependencies between features, such as one feature requiring or excluding another feature. The set of features and their dependencies can be represented by a *feature model*. A configuration is called *valid* if it satisfies all dependencies of a feature model and *invalid* otherwise. We formally define feature models and configurations as follows:

Definition II.1. Feature Model and Configuration *Let $\mathcal{F}$ denote the universe of feature names and $\mathcal{D}_{\mathcal{F}}$ denote the set of propositional formulas (dependencies) over the feature set $\mathcal{F}$. A feature model M is a pair $M = (F, D)$, where $F \subset \mathcal{F}$ is a finite set of features and $D \subset \mathcal{D}_F$ is a finite set of dependencies over F. $\mathcal{M}$ denotes the set of all possible feature models $\mathcal{M} = \{(F, D) \mid F \subset \mathcal{F},\ D \subset \mathcal{D}_F\}$.*
A configuration c *is a tuple* $\mathrm{c} = (F_{inc}, F_{exc})$*, where $F_{inc}, F_{exc} \subset \mathcal{F}$ are two finite, disjoint sets of features, with F_{inc} representing the selected (included) features and F_{exc} representing the unselected (excluded) features. $\mathcal{C}$ denotes the set of all possible configurations, $\mathcal{C} = \{(F_{inc}, F_{exc}) \mid F \subset \mathcal{F},\ F_{inc} \subseteq F,\ F_{exc} = F \backslash F_{inc}\}$ and $\mathrm{C}_M \subset \mathcal{C}$ the finite set of all possible configurations with regard to a feature model M, $\mathrm{C}_M = \{(F_{inc}, F_{exc}) \mid M = (F, D),\ F_{inc} \subseteq F,\ F_{exc} = F \setminus F_{inc}\}$.*

We use a modified version of the Body Comfort System (BCS) case study as running example. The case study was originally developed by Müller et al. [30] and frequently used in experiments for configurable systems [25, 32, 34]. In Figure 1a, we show the feature model of our running example, represented as a *feature diagram*. The system is composed of a mandatory wiper functionality (*Wiper*), which can either have premium (*Premium*) or budget (*Budget*) quality. Each configuration of the running example contains a door system (*Door_System*) with optional power windows (*Power_Window*) and heatable mirrors (*Heatable_Mirrors*). Customers can order additional security functionality, such as a central locking system (*CLS*) and remote key control (*RCK*). However, the remote key control feature is only available if the system has the central locking system feature ($RCK \Rightarrow CLS$). In addition, customers can choose to have an infotainment system with Bluetooth (*Bluetooth*) and USB-ports (*USB*).

Using our notation from Def. II.1, the feature model can be represented as:

$$M_{BCS} = (\{BCS, Wiper, \ldots, USB\},\ \{BCS \Leftrightarrow Wiper, Wiper \Rightarrow Budget \oplus Premium,\ \ldots,\ RCK \Rightarrow CLS\}).$$

2.2 Configuration Sampling

In sample-based testing, a configurable system is tested by deriving a subset of configurations from the set of all configurations (i.e., a sample) and then running tests against every configuration one-by-one [38]. Such a sample can either be created manually by a developer or automatically by a sampling algorithm, such as random sampling [31], t-wise sampling [11, 26, 33], and dissimilarity-based sampling [4]. We formally define a sample and sampling algorithm as follows:

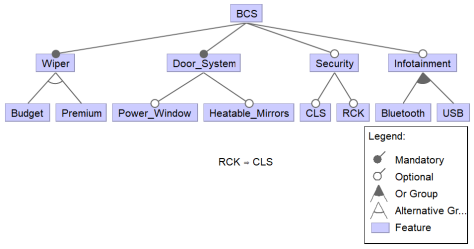
Definition II.2. Sample and Sampling Algorithm *A sample S is a finite set of configurations $S = \{\mathrm{c}_1, \ldots, \mathrm{c}_n\}$ such that every configuration $\mathrm{c} \in S$ corresponds to the same feature model $M \in \mathcal{M}$ (i.e., $\exists M \in \mathcal{M} : S \subseteq \mathrm{C}_M$). $\mathcal{S}$ denotes the set of all possible samples, $\mathcal{S} = \{S \mid M \in \mathcal{M}, S \subseteq \mathrm{C}_M\}$.*
A sampling algorithm a *is a function* $\mathrm{a} : \mathcal{M} \rightarrow \mathcal{S}$ *that maps a feature model $M \in \mathcal{M}$ to a sample $S \in \mathcal{S}$ with $S \subseteq \mathrm{C}_M$ denoted* $\mathrm{a}(M) = S$*. $\mathcal{A}$ denotes the set of all possible sampling algorithms.*

In the remainder of this paper, we assume that all configurations in a sample are valid with regard to the feature model used as input for the sampling algorithm.

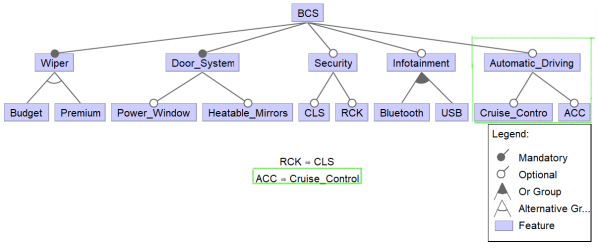
2.3 Sample-based Testing in Continuous Integration

In modern system development, system quality is typically assured by CI. The development follows four phases, *develop*, *commit*, *build*, and *test* [13]. Due to the combinatorial explosion problem, it is not feasible to build and test all products of a configurable system. We extend the typical CI cycle by a *sample* phase in which a sample for testing is created. We exemplify the process of sample-based CI with our running example shown in Figure 1. (1) The developers change the original version of our running example (cf. Figure 1a) and introduces the functionalities for cruise control (marked by the colored rectangle) in the *Development Phase*. (2) They commit the new feature model (cf. Figure 1b) to a version control system, as part of the *Commit Phase*. (3) All possible configurations of the system cannot be built. Therefore, a sampling algorithm is used to select a representative subset of all configurations in the *Sampling Phase*. (4) The selected configurations are built during *Build Phase*. (5) All configurations of the sample are tested in the *Test Phase*.

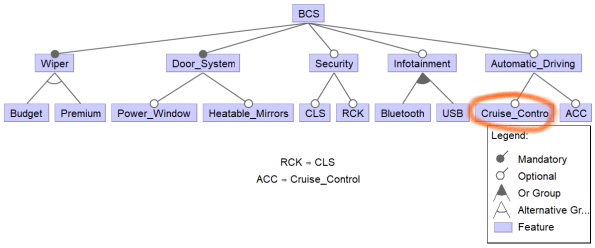
To realize the two other evolution steps of our running example, the CI-cycle starts again with Phase 1. The feature model of our running example does not change in the second evolution step of our running example. Instead, the already existing feature *Cruise_Contro* gets a software update. The update is indicated by an orange circle in Figure 1c. In Figure 1d, we show the fourth version of our running example. During the evolution, the *Voice_Control* feature



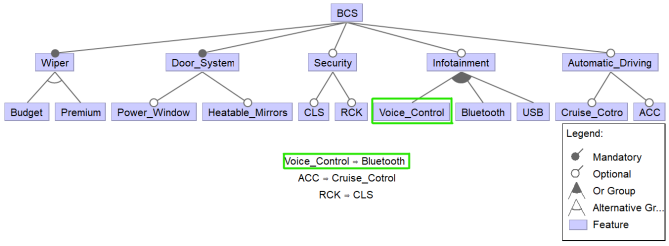
(a) Features of the BCS running example (first version).



(b) Features of the BCS running example (second version). Colored rectangle marks added features.



(c) Features of the BCS running example (third version). Colored circle marks updated features.



(d) Features of the BCS running example (fourth version). Colored rectangle marks added features.

Figure 1: Feature model evolution of the BCS running example. From Version 1 (top left) to Version 4 (bottom right)

is added. In addition, the constraint that *Voice_Control* can only be selected if *Bluetooth* is selected is added as well.

In Table 1, we show selected samples to test our BCS example after each evolution step according to different testing objectives. After the second evolution step, already tested configurations containing the *Cruise_Contro* feature must be tested again because *Cruise_Contro* was updated. Thus, Sample S and Sample S' of Table 1 are very similar to each other. Sample S'' contains dissimilar configurations to sample S' because the newly introduced feature *Voice_Control* that must be tested. The relation between the samples S, S', S'' reflects the two use cases of sample-based regression testing. (1) Two samples must be as similar as possible to re-test existing functionality, or (2) they must be as dissimilar as possible to explore still uncovered functionality.

3 PROBLEM STATEMENT

For efficient use of sample-based regression testing in CI, knowledge about the similarity of samples is required. As of now, it is not possible to choose sampling algorithms to support different use cases of sample-based regression testing because of the following challenges:

(1) *There is no established metric to compute the similarity between samples.*
(2) *There are no metrics on how stable, established sampling algorithms are.*

Consequently, testing activities cannot be supported by purposefully choosing a stable or unstable sampling algorithm.

To alleviate this problem, we propose a metric to calculate the *similarity* between two samples, based on the similarity of configurations. In addition, we propose *sampling stability* to discriminate sampling algorithms based on whether they calculate similar samples or not. After updating existing functionality (i.e., no change in the feature model), a sampling algorithm with a high *sampling stability* can be chosen to assure that already tested functionality is tested again. In case new features are introduced to the feature model, a sampling algorithm with lower *sampling stability* can be chosen to ensure that the new functionality of the system is tested.

4 STABILITY OF SAMPLING ALGORITHMS

In the following, we present *sampling similarity* as a metric to measure the similarity between samples, which forms the basis for determining *sampling stability*.

4.1 Formal Definitions

We begin with defining *sampling stability* with regard to a sequence of feature models. A sampling algorithm that reaches the maximal *sampling stability* value calculates an identical sample for each feature model. A sampling algorithm that reaches a minimal *sampling stability* calculates a disjoint sample for each feature model in the sequence. The *sampling stability* value is influenced by how much the feature models in sequence change over time. For instance, if the feature models in the sequence are identical, a sampling algorithm may achieve a *sampling stability* value of 1. However, if the feature models in the sequence change allot, the *sampling stability* value will be low.

Definition IV.1. Sampling Stability *Sampling stability is a function* $\text{stab}_n : \mathcal{A} \times \mathcal{M}^n \rightarrow [0,1]$ *for* $n \in \mathbb{N}, n \geq 2$ *that quantifies the stability of a sampling algorithm* $a \in \mathcal{A}$ *with regard to a sequence of feature models* $\vec{M} = (M_1, \ldots, M_n) \in \mathcal{M}^n$ *with a real value between* 0 *and* 1*, where* 0 *represents minimal and* 1 *maximal stability.*

Table 1: Selected samples of the BCS after each evolution step

Sample S (BCS Version 2)	Sample S' (BCS Version 3)	Sample S'' (BCS Version 4)
c_1 = ({*BCS*, *Wiper*, *Budget*, *Door_System*, *Power_Window*, *Automatic_Driving*, *Cruise_Contro*}, {*Premium*, *Heatable_Mirrors*, *Security*, *CLS*, *RCK*, *Infotainment*, *Bluetooth*, *USB*, *ACC*})	c'_1 = ({*BCS*, *Wiper*, *Budget*, *Door_System*, *Power_Window*, *Automatic_Driving*, *Cruise_Contro*}, {*Premium*, *Heatable_Mirrors*, *Security*, *CLS*, *RCK*, *Infotainment*, *Bluetooth*, *USB*, *ACC*})	c''_1 = ({*BCS*, *Wiper*, *Budget*, *Door_System*, *Power_Window*, *Infotainment*, *Bluetooth*}, {*Premium*, *Heatable_Mirrors*, *Security*, *CLS*, *RCK*, *Voice_Control*, *USB*, *Automatic_Driving*, *Cruise_Contro*, *ACC*})
c_2 = ({*BCS*, *Wiper*, *Budget*, *Door_System*, *Heatable_Mirrors*, *Automatic_Driving*, *Cruise_Contro*, *ACC*}, {*Premium*, *Power_Window*, *Security*, *CLS*, *RCK*, *Infotainment*, *Bluetooth*, *USB*})	c'_2 = ({*BCS*, *Wiper*, *Budget*, *Door_System*, *Power_Window*, *Heatable_Mirrors*, *Automatic_Driving*, *Cruise_Contro*}, {*Premium*, *Security*, *CLS*, *RCK*, *Infotainment*, *Bluetooth*, *USB*, *ACC*})	c''_2 = ({*BCS*, *Wiper*, *Premium*, *Door_System*, *Power_Window*, *Infotainment*, *USB*}, {*Budget*, *Heatable_Mirrors*, *Security*, *CLS*, *RCK*, *Voice_Control*, *Bluetooth*, *Automatic_Driving*, *Cruise_Contro*, *ACC*})
c_3 = ({*BCS*, *Wiper*, *Budget*, *Door_System*, *Power_Window*, *Heatable_Mirrors*, *Automatic_Driving*, *Cruise_Contro*}, {*Premium*, *Security*, *CLS*, *RCK*, *Infotainment*, *Bluetooth*, *USB*, *ACC*})	c'_3 = ({*BCS*, *Wiper*, *Premium*, *Door_System*, *Heatable_Mirrors*, *Automatic_Driving*, *Cruise_Contro*, *ACC*}, {*Budget*, *Power_Window*, *Security*, *CLS*, *RCK*, *Infotainment*, *Bluetooth*, *USB*})	c''_3 = ({*BCS*, *Wiper*, *Budget*, *Door_System*, *Power_Window*, *Heatable_Mirrors*, *Infotainment*, *Bluetooth*, *Voice_Control*, *Automatic_Driving*, *Cruise_Contro*}, {*Premium*, *Security*, *CLS*, *RCK*, *USB*, *ACC*})
c_4 = ({*BCS*, *Wiper*, *Premium*, *Door_System*, *Heatable_Mirrors*, *Automatic_Driving*, *Cruise_Contro*, *ACC*}, {*Budget*, *Power_Window*, *Security*, *CLS*, *RCK*, *Infotainment*, *Bluetooth*, *USB*})		c''_4 = ({*BCS*, *Wiper*, *Premium*, *Door_System*, *Power_Window*, *Infotainment*, *Bluetooth*, *Voice_Control*, *Automatic_Driving*, *Cruise_Contro*, *ACC*}, {*Budget*, *Heatable_Mirrors*, *Security*, *CLS*, *RCK*, *USB*})

The stability of a sampling algorithm is based on the similarity of the samples it produces for subsequent versions of a feature model. We calculate the similarity between two samples as a value between 0 and 1. A maximal *sample similarity* value is achieved if the input samples contain only identical configurations and have the same size. The minimal *sample similarity* value is reached if both samples are disjoint to each other.

Definition IV.2. Sample Similarity *Sample similarity is a function* $\text{ssim} : \mathcal{S} \times \mathcal{S} \to [0,1]$ *that quantifies the similarity between two samples* $S, S' \in \mathcal{S}$ *with a real value between* 0 *and* 1*, where* 0 *represents minimal and* 1 *maximal similarity.*

The sample similarity is based on the similarity between individual configurations of two samples. We calculate the similarity between two configurations by comparing their selected and deselected feature sets. A maximal configuration similarity value of 1 is reached when the sets of selected features and the sets of deselected features of both configurations are identical. A minimal configuration similarity of 0 is calculated when the selected and deselected feature sets of both configurations are disjoint to each other.

Definition IV.3. Configuration Similarity *Configuration similarity is a function* $csim : C \times C \to [0,1]$ *that quantifies the similarity between two configurations* $c, c' \in C$ *with a real value between* 0 *and* 1*, where* 0 *represents minimal and* 1 *maximal similarity.*

Sampling stability depends on the similarity between the configurations of two samples. Here, it is important which configurations are compared to each other and how size differences between samples are handled. In order to compare configurations, we define a configuration matching. The configuration matching is a binary relationship between the configurations of two samples. A configuration is matched to at least one other configuration or a bottom value ($\perp$) if the configuration cannot be matched to another configuration. We define the configuration matching relation as follows:

Definition IV.4. Configuration Matching *Given two samples* $S, S' \in \mathcal{S}$*, a configuration matching is a binary relation* $\text{map} \subseteq S \cup \{\perp\} \times S' \cup \{\perp\} \setminus \{(\perp, \perp)\}$.

4.2 Configuration Similarity

Different metrics for configuration similarity exist that fit our definition in Def. IV.3. Henard et al. [16] use the Jacard Index [17] to compare configurations based on selected features. However, their approach neglects the set of deselected features, impacting the calculation of sample similarity later on. Instead, we use the metric defined by Al-Hajjaji et al. [4], which is an adapted version of the Hamming similarity [15] and considers both, selected and deselected features. In particular, we use the following definition:

$$csim(c, c') = \frac{|F_{inc} \cap F'_{inc}| + |F_{exc} \cap F'_{exc}|}{|F_{inc} \cup F'_{inc} \cup F_{exc} \cup F'_{exc}|}$$

Table 2: Similarities matrix for Samples S and S'.

	c'_1	c'_2	c'_3	$\perp$
c_1	1.0	0.9375	0.8125	0.0
c_2	0.95	0.75	0.81	0.0
c_3	0.75	1.0	0.8	0.0
c_4	0.81	0.7	1.0	0.0
$\perp$	0.0	0.0	0.0	0.0

Originally, the Hamming similarity counts the number of features that are selected and deselected in both configurations and divides the result by the total number of features. This approach is defined for configurations on the same feature set (i.e., $F_{inc} \cup F_{exc} = F'_{inc} \cup F'_{exc}$). However, in the context of feature model evolution, we need to compare configurations of different feature sets. Thus, we use the union of both feature sets from the two input configurations. In detail, we calculate the ratio of the number of common selected (i.e., $|F_{inc} \cap F'_{inc}|$) and common deselected features (i.e., $|F_{exc} \cap F'_{exc}|$) to the number of features in the combined feature set (i.e., $|F_{inc} \cup F'_{inc} \cup F_{exc} \cup F'_{exc}|$). So, features that are added and removed during evolution are considered in the similarity value. A feature that is only contained in one of the configurations automatically counts negatively towards the similarity of both configurations. Therefore, configurations can only reach a similarity of 1 if they consist of the same feature sets.

Calculating the similarity value for every pair of configurations between two samples results in a similarity matrix. In Table 2, we present the similarity matrix for our running example. It contains the calculated similarity values for all configuration pairs between Sample S and Sample S'.

4.3 Configuration Matching

A similarity matrix contains all information to reason about the similarity between the two samples. However, taking the average of all values from the matrix does not lead to a meaningful result. Even if two samples contain two equal configurations, each of these configurations would be compared to every other configuration in the other sample, leading to an average similarity value of less than 1. Hence, we can compute sample similarity by finding a suitable configuration matching map between two samples, which leads to a subset of similarity values from the matrix (cf. Def. IV.4). In general, we try to find the best matching partner for every configuration, and thus maximize the overall similarity score.

Finding a suitable configuration matching can be seen as a graph matching problem in a bipartite graph between two sets of configurations. Every configuration from one sample can be connected to every configuration from the other sample, where each connection between two configurations is weighted with their corresponding similarity value. Configurations may be matched such that each configuration is connected to at most one other configuration (i.e., *exclusive matching*) or that each configuration can be connected to multiple other configurations (i.e., *non-exclusive matching*). A non-exclusive matching allows that every configuration gets its best matching partner. However, size differences between the samples will be hidden. In contrast, an exclusive matching may result in unmatched configurations, which allows us to take size differences into account. To this end, we use an exclusive matching.

Finding the best exclusive matching in a bipartite graph is an NP-hard problem. Therefore, we use the Hungarian algorithm [20, 22], which is a greedy algorithm that computes an exclusive matching, aiming to maximize overall similarity while keeping computational effort feasible.

Applying the Hungarian algorithm to our running example yields the following configuration matching $map = \{(c_1, c'_1), (c_2, \perp), (c_3, c'_2), (c_4, c'_3)\}$. The configuration c_2 stays unmatched, because the samples S and S' have different sizes. In Table 2, we marked the corresponding *sample similarity* values in green.

4.4 Sample Similarity

Using a suitable configuration matching, we can determine a similarity value for two samples by calculating the similarity of all matched configuration pairs and aggregating the results (cf. Def. IV.2). When aggregating the configuration similarity values, we prefer to equally consider all values and take outliers into account as well. A simple and natural way to fulfill these requirements is to compute the arithmetic mean of all values. Other simple aggregations either neglect outliers, such as the geometric mean and median, or do not consider all values, such as minimum and maximum. We compute the *sample similarity* as follows:

$$ssim(S, S') = \frac{\sum_{(c,c') \in map_*} csim(c, c')}{|\, map_\perp \,|}, \; with$$

$$map_* = \{(c, c') \mid c \in S, \; c' \in S', \; (c, c') \in map\} \; and$$

$$map_\perp = \{(c, c') \mid c \in S \cup \{\perp\}, \; c' \in S' \cup \{\perp\}, \; (c, c') \in map\}$$

In this way, we assign a value of 0 to all unmatched configurations. This reflects that a configuration without a match is most dissimilar to all other configurations in the opposite sample. This is in accordance with our decision to use exclusive matching to take size differences of samples into account. Applying this formula to our running example yields the following *sample similarity* values:

$$ssim(S, \; S') = 0.75$$

$$ssim(S', S'') = 0.49$$

4.5 Sampling Stability

As the last step for determining *sampling stability*, we need to aggregate the sample similarity values of a sequence of samples. Given a sampling algorithm and a sequence of feature models, we generate a corresponding sequence of samples. Then, we compute the sample similarity for each two consecutive samples in the sequence. Finally, we aggregate all *sample similarity* values to a *sampling stability* value using the arithmetic mean. Again, we prefer to equally consider all values and take outliers into account, which does not apply to other simple metrics, such as the minimum, maximum, or median. In summary, we compute *sampling stability* as:

$$stab_n(a, (M_1, \ldots, M_n)) = \frac{\sum_{i=1}^{n-1} ssim(a(M_i), a(M_{i+1}))}{n-1}$$

Regarding our example, this results in:

$$stab_3(a, (BCS_2, BCS_3, BCS_4)) = \frac{0.75 + 0.49}{2} = 0.62$$

The *sampling stability* of a given sampling algorithm depends not only on the algorithm itself, but also on the particular feature model sequence. More extensive changes in the feature model sequence make it harder for a sampling algorithm to calculate similar samples. Therefore, *sampling stability* will be generally lower for feature model sequences that include many changes per feature model.

5 EVALUATION

We evaluate our concepts of *sampling stability* and *sample similarity* by comparing various t-wise coverage sampling algorithms and a random sampling algorithm. Our main focus is to investigate the differences in *sampling stability* of those sampling algorithms. We consider the following research questions:

- RQ_1 Is there a difference in *sampling stability* for sampling algorithms achieving t-wise coverage and random sampling?
- RQ_2 Does the extend of changes between consecutive versions of feature models influence *sampling stability*?

5.1 Experiment Design

Our experiment compares the *sampling stability* of t-wise sampling algorithms. We use Chvatal [9], ICPL [18, 19], IncLing [2], and YASA [21]. In addition, we use the random sampling algorithm of FeatureIDE [3]. We apply each sampling algorithm to the history of a subject system, which results in a sequence of samples. For each sampling algorithm, we calculate the similarity values between all consecutive pairs of samples in the sample sequence. Thereafter, we calculate *sampling stability* for each sampling algorithm. To compensate for non-deterministic behavior in any algorithm, we repeat the sample process five times for each algorithm and subject system and aggregate the results respectively using the arithmetic mean.

5.2 Subject System

In our experiment, we use BusyBox[1] as subject system. BusyBox is frequently used as a benchmark to evaluate concepts and tools in the domain of configurable systems [24], [27], [1]. To prepare our evaluation, we analyzed commits in the open-source Git repository of BusyBox in the time from 2007-05-20 to 2010-05-09. We discovered changes to the variability model of BusyBox in 3,714 commits. We extracted those commits from the repository and converted each variability model into a Conjunctive Normal Form (CNF), stored in the DIMACS file format, using KClause[2] [35]. Each of the variability models contains more than 600 features.

The evolution history of BusyBox with multiple commits each day enables us to execute a thorough evaluation with a large enough dataset to arrive at a valid estimation of *sampling stability* for the used sampling algorithms. Multiple commits per day indicate that the feature model of BusyBox may not have changed much between single commits. Therefore, we evaluate our concept of *sampling stability* also on monthly snapshots of BusyBox in addition to the full BusyBox evolution history.

[1]https://busybox.net/
[2]https://zenodo.org/record/2574218

5.3 Results

Figure 2 shows the *sample similarity* values for each sampling algorithm. Figure 2a shows the *sample similarity* values for the full evolution history of BusyBox, while Figure 2b shows the *sample similarity* values for monthly snapshots. We structured both boxplots in the same way. The x-axis shows the names of all sampling algorithms analyzed in our experiment. The y-axis depicts *sample similarity* values from 0.5 to 1. We reduced the scaling of the y-axis because the relevant *sample similarity* values of all sampling algorithms are in this interval. For each sampling algorithm, a box shows the distribution of *sample similarity* values for the respective evolution history. For the full evolution history of BusyBox, Chvatal and ICPL produce samples with almost the same similarity between a minimum value of about 0.7 and a maximum value of about 0.85. YASA achieves *sample similarity* values in between 0.58 (minimum) and 0.71 (maximum), while IncLing produces samples with a minimum similarity of 0.57 and a maximum similarity of 0.69. The random sampling algorithm calculates samples with a similarity of 0.565 as a minimum and 0.575 as a maximum.

Table 3 shows the *sampling stability* values for random sampling and the sampling algorithms Chvatal, ICPL, IncLing, and YASA. We measure the lowest *sampling stability* values for the random sampling algorithm, with 0.57 on both the full evolution history and on monthly snapshots of BusyBox. IncLing has the second-lowest *sampling stability*, with *sampling stability* values of 0.62 for the full evolution history and 0.57 for monthly snapshots. The YASA sampling algorithm shows medium *sampling stability* with 0.64 (full history) and 0.62 (monthly snapshots). We measure the highest *sampling stability* for Chvatal (0.78 full history and 0.75 monthly snapshots) and ICPL (0.78 full history and 0.74 monthly snapshots).

Table 3: Sampling stability values for the complete history of BusyBox and the monthly snapshots.

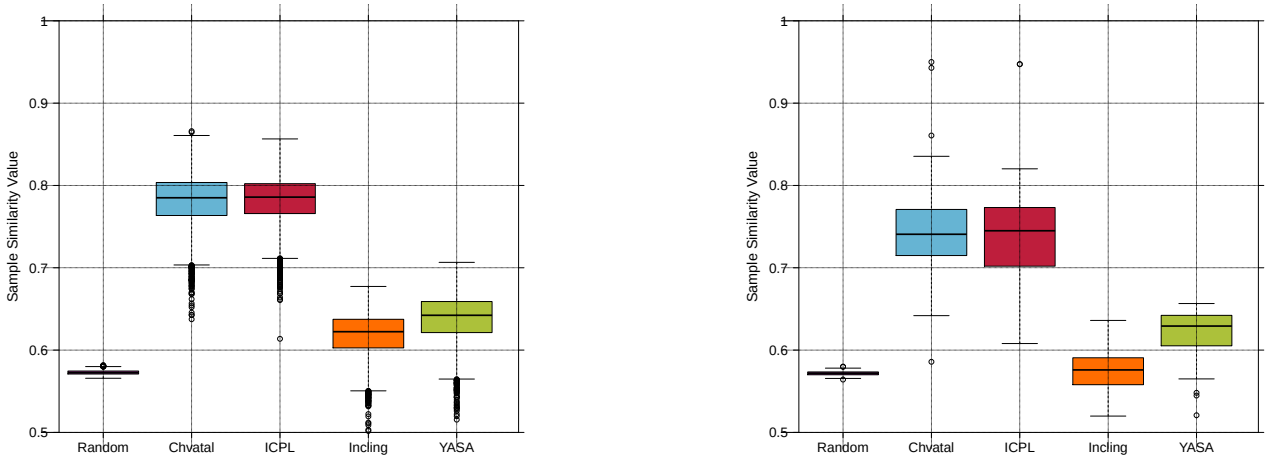
Algorithm	BusyBox Full History	BusyBox Monthly
Random	0.57	0.57
Chvatal	0.78	0.75
ICPL	0.78	0.74
IncLing	0.62	0.57
YASA	0.64	0.62

5.4 Discussion

In this section, we discuss our research questions based on our evaluation results.

RQ1: We evaluate whether there is a difference in *sampling stability* for sampling algorithms achieving t-wise coverage and random sampling. Our results show that we can distinguish between sampling algorithms achieving t-wise coverage and random sampling. Chvatal and ICPL show the highest *sampling stability* of the sampling algorithms that achieve t-wise coverage used in our experiment. Random sampling achieves the lowest *sampling stability*compared between the evaluated algorithms.

All sampling algorithms in our experiment use a greedy approach to create configurations that cover t-wise feature interactions. They vary in their feature selection to increase sampling efficiency and



(a) Sample Similarity for the Complete Evolution History

(b) Sample Similarity for the Monthly Snapshots

Figure 2: Sample Similarity for BusyBox

sampling effectiveness. Chvatal calculates samples without much regard for optimizing sample size or sampling time. ICPL improves the performance of Chvatal by filtering dead and core features and finding invalid configurations more efficiently. Still, the basic greedy approach of Chvatal remains. IncLing uses a sequential greedy approach to cover pair-wise feature interactions as well. However, it uses a feature ranking to influence how feature interaction pairs are selected for a configuration. This selection strategy changes the order in which feature interactions are covered in comparison to Chvatal and ICPL. YASA improves the sampling efficiency of creating a t-wise interaction sample using an optimized covering strategy to select feature interactions. The covering strategy tries to cover some feature interactions without checking their validity by moving those expensive checks to the end of the algorithm. To this end, YASA changes the order in which feature interactions are covered compared to the simple greedy strategy of ICPL and Chvatal. Furthermore, YASA tries to optimize the sample size by ranking configurations based on uniquely covered feature interactions and swapping those with lower ranks against new configurations. The new configurations fill the missing feature interactions, but the ordering for iterating over the interactions is random. This swapping mechanism introduces a degree of randomness to the sampling procedure, which may decrease the similarity between sample pairs.

Based on the insights about the sampling algorithms, we deduce that improving sampling efficiency and effectiveness by changing the selection strategy of feature interactions influences *sampling stability*. In the case of IncLing and YASA, the chosen selection strategies improve the sampling efficiency but decrease the similarity between samples. The correlation between sampling efficiency and *sampling stability* must be investigated in future work.

Our results show that the random sampling algorithm achieves the lowest *sampling stability* results in our experiment. The *sample similarity* of random sampling is about 0.57. A *sample similarity* of 0.5 could have different meanings. For instance, 50% of the configurations of a sample could have changed completely, the sample size between two consecutive samples could have increased or decreased by 50%, or 50% of the feature selection could have changed in each configuration for consecutive samples. Independent of the cause, a *sample similarity* value of 0.5 between consecutive samples is expected for a random sampling algorithm, which selects configurations randomly. Random sampling shows slightly higher *sample similarity* values, because certain features are required to be always selected or depend on other features to produce a valid product configuration (core and mandatory features). Thus, we deduce that the number of mandatory features and constraints between features also influences the *sampling stability* of sampling algorithms.

RQ2: We evaluate whether the extent of changes between consecutive feature models influences *sampling stability* of sampling algorithms. Based on our result, we argue that *sampling stability* is reduced for a higher extent of change between consecutive feature models in the evolution history. Hence, the extent of changes influences *sampling stability*. The measured *sampling stability* for monthly snapshots of BusyBox is lower than the results for the full evolution history of BusyBox for all sampling algorithms in our experiment. For instance, the *sampling stability* value for IncLing is 0.05 lower for the monthly snapshots of BusyBox compared to the full history. The reduced *sampling stability* for the sampling algorithms results from a higher extent of changes between the feature models of the consecutive system version. Versions of a configurable system from the same day have a greater probability of being similar to each other than system version with a time span of one month in between them. The similarity of the feature models is reflected in the similarity of samples. Thus, we deduce that *sampling stability* depends on the extent of changes between the input feature models. We will investigate the correlation between

edit operations on feature models and *sampling stability* as future work to support sample-based regression testing in CI.

5.5 Threats to Validity

Internal Validity. There is a risk of having faults in our implementation for measuring *sampling stability* and *sample similarity*. However, our implementation of the Hungarian algorithm is based on well-documented sources [20, 22], and we reviewed our entire implementation multiple times. Furthermore, we manually validated the results for *sampling stability* and *sample similarity* on a small but representative evolution history of another configurable system. Both measures address this threat by raising confidence in our implementation.

Potential faults in the sampling algorithms also threaten the internal validity of our results. To this end, we use the open-source tool FeatureIDE, which provides implementations of all the sampling algorithms we use [3]. FeatureIDE is a well-established tool for analyzing configurable systems. We have confidence in the implementations in FeatureIDE so that the threat on hand is addressed.

The base of our experiments are samples produced by different sampling algorithms. We cannot assure that all of those algorithms produce deterministic results at all times. We minimized the impact of this threat by executing the sampling five times for each feature model sequence and sampling algorithm.

External Validity. We consider only sampling algorithms that take the feature model as input. We cannot generalize our results to sampling procedures that use additional implementation artifacts as input for sample generation. We argue that many of the sampling algorithms in literature rely solely on the feature model for sampling [40]. Hence, our results apply to a wide range of algorithms.

Our experiment is based on a single subject system (BusyBox). BusyBox is a medium-sized configurable system, developed by an open-source community, which results in short evolution cycles. We argue that the rapid evolution of BusyBox is representative of how configurable systems are developed with continuous integration. We analyzed a large number of different versions (3,714) of the system. Therefore, our experiment results should reflect the most common change operations of feature models.

6 RELATED WORK

Developers of sampling algorithms evaluate their sampling algorithms with regard to how fast samples can be calculated, how many configurations the sample contains, and which coverage the algorithm achieves [9, 18, 19, 21]. Over the past decades, numerous studies were conducted that elaborate on sample-based testing and the properties of sampling algorithms [12, 23, 26, 27, 40]. Recently, Varshosaz et al. [40] published an extensive literature survey on sample-based product line testing. They analyze specific properties of established sampling procedures such as artifacts used for sampling, the sampling procedure, and the coverage a sampling procedure achieves. Previous work considers sampling efficiency and sampling effectiveness as the two leading metrics to discriminate between sampling algorithms. We complement the extensive work done by other authors before by introducing *sampling stability* as a metric to discriminate sampling algorithms.

We measure the similarity between configurations to discern how similar two consecutive samples are. The concept of calculating configuration similarity was previously used by Al-Hajjaji et al. [4] and Sánches et al. [36] to influence the order in which configurations are tested. Sánches et al. [36] use the Jaccard-Metric as similarity metric while Al-Hajjaji et al. [4] use an adjusted version of the Hamming-Similarity. Both Sánches and Al-Hajjaji apply their similarity calculation to configurations from the same sample. In contrast, we use an adjusted version of the Hamming-Similarity to discern the similarity of configurations between different samples.

We contribute to researching the impact of changes to configurable systems during their evolution [7, 39]. Bürdek et al. [7] conduct a case study on real-world configurable systems to analyze the impact of change operations during the evolution of configurable systems. Thüm et al. [39] identify four general categories of feature model changes (refactoring, generalization, specialization, and arbitrary edit). They reason about how each category of change operation influences the configuration space. Moreover, they present a tool to reason about the impact of change operations to the feature model for a specific configuration. Instead, our approach focuses on the properties of sampling procedures. We use the evolution history of existing samples to measure the stability. We assume that changes to the configurable system happened during its evolution, but we do not identify those changes. Therefore, our approach differentiates from those described above.

7 CONCLUSION

In this paper, we presented *sampling stability* as a novel criterion to measure whether a sampling algorithm calculates similar samples for an evolution history of a configurable system. In our evaluation, we measured and compared the *sampling stability* of t-wise sampling algorithms, such as *Chvatal*, *ICPL*, *IncLing*, and *YASA*. Our results show differences in the *sampling stability* of the algorithms. Therefore, we argue that practitioners can use *sampling stability* to decide whether a sampling algorithm is best suited to re-test already tested functionality or to test new functionality. Based on our results, we show a connection between improved sampling efficiency and reduced *sampling stability*. Furthermore, we discovered that a higher extent of changes between feature model version results in a reduced *sampling stability* of a sampling algorithm. We will investigate this correlation in future work by building a controlled experiment based on feature model edit operations presented by Thüm et al. [39]. In this context, we will also investigate the influence of mandatory features on calculating *sampling stability*. Furthermore, we will measure *sampling stability* for more sampling algorithms to investigate the correlation between sampling efficiency and *sampling stability*.

ACKNOWLEDGMENTS

Funded by DFG Grant LE 3382/2-3, Futurlaboritory Mobility Lower Saxony. We thank Hendrik Jüchter for the implementation of the hungarian algorithm.

REFERENCES

[1] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability bugs in highly configurable systems: A

qualitative analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 3 (2018), 1–34.

[2] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: efficient product-line testing using incremental pairwise sampling. *ACM SIGPLAN Notices* 52, 3 (2016), 144–155.

[3] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Tool demo: testing configurable systems with FeatureIDE. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 173–177.

[4] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective product-line testing using similarity-based product prioritization. *Software & Systems Modeling* 18, 1 (2019), 499–521.

[5] Sven Apel, Alexander Von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for product-line verification: case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 482–491.

[6] Jan Bosch. 2004. Software variability management. In *Proceedings. 26th International Conference on Software Engineering*. IEEE, 720–721.

[7] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. 2016. Reasoning about product-line evolution using complex feature model differences. *Automated Software Engineering* 23, 4 (2016), 687–733.

[8] Lianping Chen, Muhammad Ali Babar, and Nour Ali. 2009. Variability management in software product lines: a systematic review. In *13th International Software Product Line Conference*. ACM.

[9] Vasek Chvatal. 1979. A greedy heuristic for the set-covering problem. *Mathematics of operations research* 4, 3 (1979), 233–235.

[10] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2007. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 129–139.

[11] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering* 34, 5 (2008), 633–650.

[12] Ivan do Carmo Machado, John D McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. 2014. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology* 56, 10 (2014), 1183–1199.

[13] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[14] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.

[15] Richard W Hamming. 1950. Error detecting and error correcting codes. *The Bell system technical journal* 29, 2 (1950), 147–160.

[16] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering* 40, 7 (2014), 650–670.

[17] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.

[18] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 638–652.

[19] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 46–55.

[20] Roy Jonker and Ton Volgenant. 1986. Improving the Hungarian assignment algorithm. *Operations Research Letters* 5, 4 (1986), 171–175.

[21] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: yet another sampling algorithm. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, 1–10.

[22] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.

[23] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A survey on software product line testing. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 31–40.

[24] Jörg Liebig, Alexander Von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 81–91.

[25] Sascha Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. 2013. *Delta-oriented software product line test models-the body comfort system case study*. Technical Report. TU Braunschweig, Germany.

[26] Roberto E Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Alexander Egyed. 2015. A first systematic mapping study on combinatorial interaction testing for software product lines. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–10.

[27] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 643–654.

[28] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 495–518.

[29] Mathias Meyer. 2014. Continuous integration and its tools. *IEEE Software* 31, 3 (2014), 14–16.

[30] Tobias Müller, Malte Lochau, Stefan Detering, Falko Saust, Henning Garbers, Lukas Märtin, Thomas Form, and Ursula Goltz. 2009. *A comprehensive Description of a Model-based, continuous Development Process for AUTOSAR Systems with integrated Quality Assurance*. Technical Report. TU Braunschweig, Germany.

[31] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform random sampling product configurations of feature models that have numerical features. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A*. ACM, 289–301.

[32] Sophia Nahrendorf, Sascha Lity, and Ina Schaefer. 2018. *Applying Higher-Order Delta Modeling for the Evolution of Delta-Oriented Software Product Lines*. Technical Report. TU Braunschweig-ISF.

[33] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)* 43, 2 (2011), 1–29.

[34] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly analyses for feature-model evolution. *ACM SIGPLAN Notices* 53, 9 (2018), 188–201.

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

[36] Ana B Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. 2014. A comparison of test case prioritization criteria for software product lines. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 41–50.

[37] Sean Stolberg. 2009. Enabling agile testing through continuous integration. In *2009 agile conference*. IEEE, 369–374.

[38] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 1–45.

[39] Thomas Thum, Don Batory, and Christian Kastner. 2009. Reasoning about edits to feature models. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 254–264.

[40] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A classification of product sampling for software product lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*. ACM, 1–13.

[41] Alexander Von Rhein, Sven Apel, Christian Kästner, Thomas Thüm, and Ina Schaefer. 2013. The PLA model: on the combination of product-line analyses. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, NY, 1–8.