Uniform Retainment Sampling for Evolving Software Configuration Spaces

Nikolai Käfer[1][0000−0002−0645−1078], Sven Apel[2][0000−0003−3687−2233], Christel Baier[1][0000−0002−5321−9343], Clemens Dubslaff[3][0000−0001−5718−8276], and Holger Hermanns[2][0000−0002−2766−9615]

[1] Dresden University of Technology, Dresden, Germany
{nikolai.kaefer,christel.baier}@tu-dresden.de
[2] Saarland University, Saarbrücken, Germany {apel,hermanns}@cs.uni-saarland.de
[3] Eindhoven University of Technology, Eindhoven, The Netherlands
c.dubslaff@tue.nl

Abstract. Modern software systems tend to have vast configuration spaces with myriads of valid configurations, rendering exhaustive testing and analysis infeasible. A common approach towards collecting a partial but still representative set of test configurations is by sampling the configuration space uniformly at random. However, as systems evolve, their configuration spaces change, requiring new samples to be drawn and tested after each update. We propose *retainment sampling*, a new sampling technique that reuses as many samples as possible from previous versions while still guaranteeing a uniform selection of samples for the evolved system. We formally prove the correctness of the approach and address the key challenge of sampling from negated constraints with two complementary methods. Experiments on five real-world histories of configurable software systems demonstrate the effectiveness of retainment sampling, with substantial savings of up to 73 % fewer configuration samples in total while maintaining equal coverage.

1 Introduction

Most software systems today are highly configurable [3]. *Feature models* are a common approach to specify the space of valid system configurations, where individual features represent units of functionalities that can be included in a configuration, and a set of constraints defines inter-feature dependencies [15]. While constrained, the number of valid configurations typically grows exponentially with the number of features [29]. Since deployment and test execution times for even a single configuration can range from minutes for tiny systems [9,34] to several hours for large-scale systems [8], exhaustive testing of all valid configurations is thus infeasible in practice. Instead, sample-based testing approaches are commonly employed [24,25], where the aim is to test a sample set of representative configurations. One such method, *uniform random sampling* (URS) [30,36,14], ensures that samples are evenly distributed across the configuration space, thereby enabling statistical confidence metrics like t-wise interaction coverage [4,10] or learning performance prediction models [31,29].

The challenge of dealing with vast configuration spaces is amplified as systems and their feature models evolve over time. Changes naturally arise when introducing new features, removing legacy features, or adjusting cross-feature constraints [41]. Even small changes can have a large impact on the configuration space, as for instance adding a single new unconstrained feature doubles the number of valid configurations. Statistical results established on a sample set drawn by URS before a feature model update may therefore no longer be valid, requiring expensive re-sampling and testing to restore coverage.

In this paper, we propose *retainment sampling*, a method that avoids expensive resampling in evolving configurable systems by reusing previously sampled configurations while preserving strict uniformity guarantees. The core idea is that many feature model updates affect only a small portion of the configuration space, so previously valid configurations can often be retained. New configurations are then sampled to complement the retained set such that the combined sample remains uniform with respect to the updated feature model. A key part of the paper is a formal notion of uniformity for samplers that return multiple samples at once, which provides the foundation for reasoning about retainment.

Realizing this idea raises several technical challenges. Feature models are represented as propositional formulas in *conjunctive normal form* (CNF) [37], and sampling from sub-formulas or their negations can trigger an exponential blowup, which we avoid via *rejection sampling* and the *Tseitin transformation* [43]. The paper presents two algorithms for uniform retainment sampling that differ in their *expected retainment*, along with a model-counting heuristic to predict retainment ahead of sampling. We evaluate both algorithms on five real-world feature-model histories, finding that retainment sampling can retain 50–73 % of samples in many cases. Since the retained samples have already been analyzed in the previous update, this translates to a potentially significant reduction in overall testing effort. A replication package with all proofs, the implementation, and evaluation data is provided [22].

2 Related Work

URS for configurable systems is commonly realized via SAT-solution samplers or compiled representations [11], with established benchmarks supporting comparative research [1,12,39]. Exact URS can be costly and may not scale [36], motivating *approximative URS* tools like QuickSampler [6]; however, these can yield distributions that deviate substantially from uniformity in highly configurable systems [36]. Retainment sampling targets strict URS and is solver agnostic, so any existing uniform sampler can be used as backend.

A key application of URS is testing, where it provides representative configurations for fault detection. *Combinatorial interaction testing* (CIT) takes a complementary view, targeting t-wise feature interaction coverage [20,19,34,12], and URS has also been used to enhance CIT [30,7]. The evolution of feature models [45,32,33,28] and the resulting need to adapt tests has been studied for CIT [34,40]; retainment sampling extends this line of work to the URS setting.

Conceptually, retainment sampling is an instance of *stratified sampling* [5], partitioning the configuration space into strata that are sampled independently. Its objective of maximizing overlap between consecutive samples is also related to *"Keyfitzing"* from survey sampling [16].

3 Preliminaries and Formal Definitions

Formally, a *feature model* for a configuration space is a pair $\mathcal{F} = \langle \mathcal{V}, \mathcal{C} \rangle$ with a set of *feature variables* $\mathcal{V}$ and a set $\mathcal{C}$ of propositional *constraints* ϕ over variables $\mathcal{V}$. The conjunction of all constraints is denoted by $\Phi_{\mathcal{F}} \coloneqq \bigwedge_{\phi \in \mathcal{C}} \phi$. As a shorthand, we may identify $\mathcal{F}$ with $\Phi_{\mathcal{F}}$ when clear from context, for example by writing $\neg\mathcal{F}$ to denote $\neg\Phi_{\mathcal{F}}$. A subset $\theta \subseteq \mathcal{V}$ is called a *configuration* of $\mathcal{F}$, where all features $X \in \theta$ are considered *activated*, and all features $Y \in \mathcal{V} \backslash \theta$ are *deactivated*. Consequently, we will also interpret θ as a Boolean variable assignment $\theta \colon \mathcal{V} \to \mathbb{B}$, which assigns all activated variables to `true` and deactivated variables to `false`. A configuration θ is *valid* w.r.t. $\mathcal{F}$ if all constraints $\phi \in \mathcal{C}$ are satisfied by θ, i.e., $\theta \models \phi$. The set of all valid configurations for $\mathcal{F}$ is given by $[\![\mathcal{F}]\!] \coloneqq \{\theta \subseteq \mathcal{V} \colon \theta \models \phi \text{ for all } \phi \in \mathcal{C}\}$. The *model count* of $\mathcal{F}$ is the number of its valid configurations, i.e., $\left|[\![\mathcal{F}]\!]\right|$.

3.1 Feature Model Evolution

Feature models evolve over time: new features may be added, old features discontinued, and constraints changed. A *feature model history* consists of a series of feature models $\mathcal{F}_1, \ldots, \mathcal{F}_k$ that are snapshots of a model $\mathcal{F}$ evolving over time. Every consecutive pair $(\mathcal{F}_i, \mathcal{F}_{i+1})$ for $1 \leqslant i < k$ forms an *update*. Following a method informally introduced by Thüm et al. [41], we consider a joint domain of features that unifies all domains of individual models $\mathcal{F}_i$:

Definition 1 (Unification). *The* unification *of two feature models* $\mathcal{F}_1 = \langle \mathcal{V}_1, \mathcal{C}_1 \rangle$ *and* $\mathcal{F}_2 = \langle \mathcal{V}_2, \mathcal{C}_2 \rangle$ *is given by* $\hat{\mathcal{F}}_1 \coloneqq \langle \hat{\mathcal{V}}, \hat{\mathcal{C}}_1 \rangle$ *and* $\hat{\mathcal{F}}_2 \coloneqq \langle \hat{\mathcal{V}}, \hat{\mathcal{C}}_2 \rangle$ *where* $\hat{\mathcal{V}} \coloneqq \mathcal{V}_1 \cup \mathcal{V}_2$, $\hat{\mathcal{C}}_1 \coloneqq \mathcal{C}_1 \cup \{\neg X \colon X \in \mathcal{V}_2 \backslash \mathcal{V}_1\}$, *and* $\hat{\mathcal{C}}_2 \coloneqq \mathcal{C}_2 \cup \{\neg X \colon X \in \mathcal{V}_1 \backslash \mathcal{V}_2\}$.

Intuitively, the added constraints ensure that features newly introduced in $\mathcal{F}_2$ are always deactivated for $\mathcal{F}_1$, and likewise that discontinued features from $\mathcal{F}_1$ are necessarily deactivated in $\mathcal{F}_2$. Note that this unification step does not change the semantics of both feature models:

Proposition 1. *Unifying feature models* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *preserves valid configurations, i.e.,* $[\![\mathcal{F}_1]\!] = [\![\hat{\mathcal{F}}_1]\!]$ *and* $[\![\mathcal{F}_2]\!] = [\![\hat{\mathcal{F}}_2]\!]$ *holds.*

Figure 1 gives a characterization of updates $(\mathcal{F}, \mathcal{F}')$ in terms of the set relations of their valid configurations [41]. For example, a *refactoring* update does not affect the valid configurations, hence $[\![\mathcal{F}]\!] = [\![\mathcal{F}']\!]$, while an *incomparable* update results in both models having no common valid configurations, i.e., $[\![\mathcal{F}]\!] \cap [\![\mathcal{F}']\!] = \emptyset$.

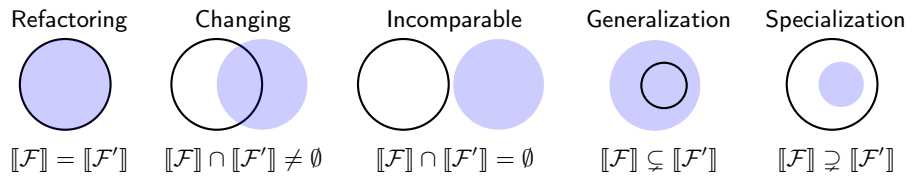


Fig. 1: Five possible types of an update $(\mathcal{F}, \mathcal{F}')$, with $[\![\mathcal{F}]\!]$ given by the black outlined circle and $[\![\mathcal{F}']\!]$ by the shaded circle.

3.2 Uniform Random Sampling

A *single sampler* $\mathcal{U}$ for a finite set Ω is a probabilistic procedure that returns a single *sample* $\theta \in \Omega$. Let $\Pr[\mathcal{U}{=}\theta]$ denote the probability of the event that $\mathcal{U}$ returns the sample θ. We call $\mathcal{U}$ *uniform* if each element in Ω is returned equally likely: $\Pr[\mathcal{U}{=}\theta] = 1/|\Omega|$ for all $\theta \in \Omega$.

Since repeated random sampling may yield the same sample more than once, we need to consider *multisets* of samples in the following. Formally, a multiset S over a finite domain Ω is a function $S\colon \Omega \to \mathbb{N}$ which indicates how often each domain element appears in the multiset. The size of S is $|S| = \sum_{\theta\in\Omega} S(\theta)$, and the intersection with a second multiset S' over Ω is given by $(S \cap S')(\theta) \coloneqq \min\big(S(\theta), S'(\theta)\big)$. The empty multiset $\emptyset$ is the constant function $\Omega \to \{0\}$, while $\mathcal{M}^n_\Omega$ denotes the set of all size-n multisets over Ω. For equally sized multisets $S, S' \in \mathcal{M}^n_\Omega$, we define the *retainment ratio* $\mathrm{retain}(S, S') \coloneqq |S\cap S'|/n$ as the ratio of their intersection size to n.

An *n-sampler* $\mathcal{U}^n$ over Ω is a stochastic process returning a multiset from $\mathcal{M}^n_\Omega$. For $\mathcal{U}$ a single sampler over Ω, we let $\Theta^n_\mathcal{U}$ denote the n-sampler resulting from n independent calls to $\mathcal{U}$. We may treat $\mathcal{U}^n$ as a random variable evaluating to a multiset $S \in \mathcal{M}^n_\Omega$. Thus, $\mathcal{U}^n(\theta) = S(\theta)$ is itself a random variable equivalent to the number of occurrences of θ in the multiset S returned by $\mathcal{U}^n$. Recall that the expected value of a discrete random variable X with domain $dom(X)$ is given by $\mathbf{E}[X] \coloneqq \sum_{k\in dom(X)} k \cdot \Pr[X{=}k]$.

Definition 2 (n-Sampler Uniformity). *An n-sampler $\mathcal{U}^n$ over Ω is* uniform *if there exists a uniform single sampler $\mathcal{U}$ over Ω such that for every multiset $S \in \mathcal{M}^n_\Omega$:*

$$\Pr[\mathcal{U}^n{=}S] \;=\; \Pr[\Theta^n_\mathcal{U}{=}S].$$

Further, $\mathcal{U}^n$ is expectation-uniform *if for every $\theta \in \Omega$, we have*

$$\mathbf{E}[\mathcal{U}^n(\theta)] \;=\; \mathbf{E}[\Theta^n_\mathcal{U}(\theta)].$$

Note that this definition is not dependent on a particular choice for $\mathcal{U}$, as all uniform single samplers induce the same probability distribution over multisets of size n. Intuitively, an n-sampler is uniform if its distribution over multisets is indistinguishable from drawing n-times from a uniform single sampler. Expectation-uniformity is a weaker notion and requires that the expected value of how often each element θ appears in the returned multiset matches the expected value of a uniform sampler, as established in the following proposition:

Proposition 2 (n-Sampler Uniformity).

1. *Every uniform n-sampler is also expectation-uniform.*
2. *For every expectation-uniform n-sampler $\mathcal{U}^n$ over Ω and all $\theta \in \Omega$, we have* $\mathbf{E}[\mathcal{U}^n(\theta)] = {}^{n}/_{|\Omega|}$.

4 Uniform Retainment Sampling

In this section, we formally state the uniform retainment sampling problem. We then introduce two sampling algorithms tackling the problem and provide a method to predict the expected retainment.

4.1 Problem Statement

A *retaining* n-sampler $\mathcal{U}^n[S]$ is a parametrized n-sampler which has access to a sample multiset S (or a sampler providing such a multiset, e.g., $\mathcal{U}^n[\Theta^n_{\mathcal{U}}]$). The *expected retainment ratio ER* of $\mathcal{U}^n[S]$ is given by

$$ER\big(\mathcal{U}^n[S]\big) \;\coloneqq\; \mathbf{E}\big[\mathrm{retain}(S, \mathcal{U}^n[S])\big].$$

This ratio stands for the average fraction of samples that could potentially be reused when $\mathcal{U}^n[S]$ takes advantage of already generated samples S.

Definition 3 (Uniform Retainment Sampling Problem). *Given two unified feature models $\mathcal{F}$ and $\mathcal{F}'$, a sample size $n \in \mathbb{N}$, and a uniform n-sampler $\hat{\mathcal{U}}^n_{[\![\mathcal{F}]\!]}$ for $[\![\mathcal{F}]\!]$, the* (expectation-) uniform retainment sampling problem *asks for an (expectation-) uniform retaining n-sampler $\mathcal{U}^n[\hat{\mathcal{U}}^n_{[\![\mathcal{F}]\!]}]$ for $[\![\mathcal{F}']\!]$ with maximal expected retainment ratio.*

Let us unpack this definition step by step. We assume $\mathcal{F}'$ is the feature model resulting from an update to base feature model $\mathcal{F}$. Innermost is the uniform n-sampler $\hat{\mathcal{U}}^n_{[\![\mathcal{F}]\!]}$, which returns a multiset of n valid configurations for the "old" feature model $\mathcal{F}$. Denoting this multiset with $S_{\mathcal{F}}$, we are looking for a retaining n-sampler $\mathcal{U}^n[S_{\mathcal{F}}]$ for the "new" feature model $\mathcal{F}'$, which has access to the old samples in $S_{\mathcal{F}}$. This retaining sampler needs to be uniform (or expectation-uniform) while achieving a high expected retainment ratio: The multiset $S_{\mathcal{F}'}$ returned by $\mathcal{U}^n[S_{\mathcal{F}}]$ should keep as many configurations from $S_{\mathcal{F}}$ as possible.

4.2 Uniform Retainment Sampling Algorithm

Consider the general setting depicted in Figure 2 where the valid configuration spaces of $\mathcal{F}$ and $\mathcal{F}'$ overlap. Choosing n uniform samples from $[\![\mathcal{F}]\!]$, the size of the overlap $[\![\mathcal{F} \wedge \mathcal{F}']\!]$ in relation to the overall size of the sample space $[\![\mathcal{F}]\!]$ predicts how many of the n samples we expect to fall within the overlap — and thus be eligible for potential re-use for $\mathcal{F}'$. Likewise, for any number of uniform samples for $[\![\mathcal{F}']\!]$, some fraction is expected to be valid also for the old model $\mathcal{F}$, i.e., falling within $[\![\mathcal{F}'_{\text{old}}]\!] \coloneqq [\![\mathcal{F} \wedge \mathcal{F}']\!]$, while the remaining samples are only valid for the updated model and therefore lie in $[\![\mathcal{F}'_{\text{new}}]\!] \coloneqq [\![\neg\mathcal{F} \wedge \mathcal{F}']\!]$.

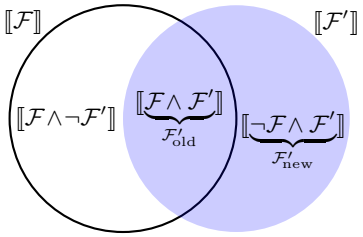


Fig. 2: Space of valid configurations for an update $(\mathcal{F}, \mathcal{F}')$.

The following proposition establishes that we can use a strict partitioning such as $[\![\mathcal{F}']\!] = [\![\mathcal{F}'_{\text{old}}]\!] \cup [\![\mathcal{F}'_{\text{new}}]\!]$ to construct a uniform sampler for the whole sampling space given uniform samplers for each partition:

Proposition 3. *For Ω a sampling space strictly partitioned into subspaces Ω_1 and Ω_2 with respective uniform samplers $\mathcal{U}_1$ and $\mathcal{U}_2$, the following single sampler $\mathcal{U}$ for Ω is uniform:*

- *With probability $|\Omega_1|/|\Omega|$ return a sample from $\mathcal{U}_1$.*
- *With probability $|\Omega_2|/|\Omega| = 1 - |\Omega_1|/|\Omega|$ return a sample from $\mathcal{U}_2$.*

The overall idea towards a uniform retainment sampling algorithm is to follow Proposition 3 and produce the samples for $\mathcal{F}'$ by sampling from the partitions $\mathcal{F}'_{\text{old}}$ and $\mathcal{F}'_{\text{new}}$ with probability matching their respective size ratios. That is, we sample from $\mathcal{F}'_{\text{old}}$ with probability $r := [\![\mathcal{F}'_{\text{old}}]\!]/[\![\mathcal{F}']\!]$, and vice versa from $\mathcal{F}'_{\text{new}}$ with probability $1 - r = [\![\mathcal{F}'_{\text{new}}]\!]/[\![\mathcal{F}']\!]$. Repeating this process n times to yield the requested number of samples forms a Bernoulli process. The random variable k that counts how often we sample from $\mathcal{F}'_{\text{old}}$ thus follows a binomial distribution with parameters n and r. This allows us to sample k times *en bloc*[4] from $\mathcal{F}'_{\text{old}}$, and $n - k$ times from $\mathcal{F}'_{\text{new}}$.

For now, we assume uniform samplers for arbitrary sample sizes are available for each partition: $\mathcal{U}^*_{\text{old}}$ over $[\![\mathcal{F}'_{\text{old}}]\!]$ and $\mathcal{U}^*_{\text{new}}$ with domain $[\![\mathcal{F}'_{\text{new}}]\!]$. Practical implementations for these samplers will be considered in Section 5. To retain as many existing samples as possible, we partition the given sample multiset $S_{\mathcal{F}}$ into the multisets $S_{\mathcal{F}\wedge\mathcal{F}'}$ of samples satisfying the new constraints in $\mathcal{F}'$, and $S_{\mathcal{F}\wedge\neg\mathcal{F}'}$ for those samples that do not satisfy $\mathcal{F}'$ and can thus be discarded. Based on $\mathcal{U}^*_{\text{old}}$, we define the retaining sampler $\tilde{\mathcal{U}}^*_{\text{old}}[S_{\mathcal{F}\wedge\mathcal{F}'}]$: If the number of requested samples is smaller or equal to $|S_{\mathcal{F}\wedge\mathcal{F}'}|$, the respective number of samples is returned from $S_{\mathcal{F}\wedge\mathcal{F}'}$. Otherwise, if more samples are requested, all samples from $S_{\mathcal{F}\wedge\mathcal{F}'}$ and the remaining number of samples from $\mathcal{U}^*_{\text{old}}$ is returned. Algorithm 1 shows the resulting sampling procedure for $\mathcal{F}'$.

Proposition 4. *The sampler implemented by Algorithm 1 is uniform.*

[4] Note that requesting multiple samples in one call can be more efficient than repeated invocations. For compilation-based samplers such as KUS [38], repeated calls either require recompilation or reloading a stored compiled representation, both of which add overhead.

Algorithm 1: (Uniform)

input : Samples $S_{\mathcal{F}\wedge\mathcal{F}'}$
Model counts $|[\![\mathcal{F}'_{\text{old}}]\!]|, |[\![\mathcal{F}']\!]|$

$r \leftarrow \frac{|[\![\mathcal{F}'_{\text{old}}]\!]|}{|[\![\mathcal{F}']\!]|}$
$k \leftarrow \texttt{binomial}(n, r)$
$S'_{\text{old}} \leftarrow \tilde{\mathcal{U}}^k_{\text{old}}[S_{\mathcal{F}\wedge\mathcal{F}'}]$
$S'_{\text{new}} \leftarrow \mathcal{U}^{n-k}_{\text{new}}$
return $S'_{\text{old}} \cup S'_{\text{new}}$

Algorithm 2: (Expect.-uniform)

input : Samples $S_{\mathcal{F}\wedge\mathcal{F}'}$
Model counts $|[\![\mathcal{F}'_{\text{old}}]\!]|, |[\![\mathcal{F}']\!]|$

$r \leftarrow \frac{|[\![\mathcal{F}'_{\text{old}}]\!]|}{|[\![\mathcal{F}']\!]|}$
if $n{\cdot}r \in \mathbb{N}$ **then**
  $k \leftarrow n{\cdot}r$
else
  $x \leftarrow n{\cdot}r - \lfloor n{\cdot}r \rfloor$
  $b \leftarrow \texttt{Bernoulli}(x)$
  $k \leftarrow \lfloor n{\cdot}r \rfloor + b$
$S'_{\text{old}} \leftarrow \tilde{\mathcal{U}}^k_{\text{old}}[S_{\mathcal{F}\wedge\mathcal{F}'}]$
$S'_{\text{new}} \leftarrow \mathcal{U}^{n-k}_{\text{new}}$
return $S'_{\text{old}} \cup S'_{\text{new}}$

Fig. 3: Two algorithms implementing retaining n-samplers for $[\![\mathcal{F}']\!]$.

The number k of samples drawn from $\mathcal{F}'_{\text{old}}$ has a direct influence on the retainment ratio as we will see in Section 4.4. As it is determined by a binomial distribution, the expected value is $\mathbf{E}[k] = n{\cdot}r$ and the variance is $\mathbf{V}(k) = n{\cdot}r{\cdot}(1-r)$. Note that the variance grows linearly with increasing sample size n. In the next section, we will see a variant of Algorithm 1 where we get the same expected value for k, but smaller variance.

4.3 Expectation-Uniform Retainment Sampling Algorithm

The idea of the second algorithm is to choose k — the number of samples drawn from $\mathcal{F}'_{\text{old}}$ — to be as close to $n{\cdot}r$ as possible. In the special case where $n{\cdot}r$ is an integer, we can thus use $k = n{\cdot}r$. Otherwise, for $n{\cdot}r$ a fraction, we could either choose $k = \lfloor n{\cdot}r \rfloor$ or $k = \lfloor n{\cdot}r \rfloor + 1$. We randomize the choice by using the fractional part of $n{\cdot}r$ to weight both options. For example, let $n = 100$ and $r = 0.427$, so $n{\cdot}r = 42.7$ and $x \coloneqq n{\cdot}r - \lfloor n{\cdot}r \rfloor = 0.7$ is the fractional part. Then, with probability $x = 0.7$, we choose $k = \lfloor n{\cdot}r \rfloor + 1 = 43$, and otherwise $k = \lfloor n{\cdot}r \rfloor = 42$. Algorithm 2 implements this approach, with $\texttt{Bernoulli}(x)$ returning 1 with probability x, and 0 with probability $1 - x$.

Proposition 5. *The sampler implemented by Algorithm 2 is expectation-uniform.*

Note that the sampler from Algorithm 2 is not uniform: Suppose the two partitions $\mathcal{F}'_{\text{old}}$ and $\mathcal{F}'_{\text{new}}$ are the same size, i.e., $r = 1/2$. A uniform sampler like Algorithm 1 would be expected to draw half its samples from each partition, yet it could still return any other proportion — just with lower probability the further that proportion deviates from $1/2$. Our second algorithm behaves differently: it will only return multisets that respect the fixed ratio r, every other split is impossible, not merely unlikely. This behavior is reflected in the variance: Since $k = \lfloor n{\cdot}r \rfloor + b$ with $b \sim \texttt{Bernoulli}(x)$, we have $\mathbf{V}(k) = \mathbf{V}(b) = x{\cdot}(1-x)$ as $\lfloor n{\cdot}r \rfloor$

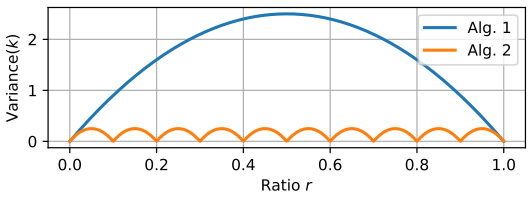


Fig. 4: Variance comparison of Algorithms 1 and 2 for $n = 10$.

is constant. Crucially, the variance no longer grows with increasing sample sizes n. A comparison of the variance for both algorithms and $n = 10$ as a function of r is shown in Figure 4.

4.4 Expected Retainment Prediction

We now take a closer look at the expected retainment ratio ER that lies at the heart of the retainment sampling problem. Intuitively, this ratio states which percentage of old-feature-model-samples we expect to retain for the updated model $\mathcal{F}'$. For a retaining n-sampler $\mathcal{U}^n[\hat{\mathcal{U}}^n_{[\![\mathcal{F}]\!]}]$ (see Definition 3), the ER is

$$ER\big(\mathcal{U}^n[\hat{\mathcal{U}}^n_{[\![\mathcal{F}]\!]}]\big) \;=\; \mathbf{E}\big[\mathrm{retain}\big(\hat{\mathcal{U}}^n_{[\![\mathcal{F}]\!]}, \mathcal{U}^n[\hat{\mathcal{U}}^n_{[\![\mathcal{F}]\!]}]\big)\big].$$

Suppose the inner uniform sampler $\hat{\mathcal{U}}^n_{[\![\mathcal{F}]\!]}$ yields a sample multiset $S_{\mathcal{F}}$, and the outer retaining sampler $\mathcal{U}^n[S_{\mathcal{F}}]$ a set $S_{\mathcal{F}'}$. Then the expected value is

$$\mathbf{E}[\mathrm{retain}(S_{\mathcal{F}}, S_{\mathcal{F}'})] \;=\; \mathbf{E}\left[\frac{|S_{\mathcal{F}} \cap S_{\mathcal{F}'}|}{n}\right] \;=\; \frac{1}{n}\,\mathbf{E}\left[\sum_{\theta \subseteq \mathcal{V}} \min\big(S_{\mathcal{F}}(\theta), S_{\mathcal{F}'}(\theta)\big)\right].$$

While this value can, in principle, be determined exactly, it does not seem to admit a closed form. The computation is further complicated by the fact that $S_{\mathcal{F}}$ and $S_{\mathcal{F}'}$ may share samples purely by chance. For example, consider a toy model in which $[\![\mathcal{F}]\!]$ contains only 10 valid configurations, a refactoring update such that $[\![\mathcal{F}']\!] = [\![\mathcal{F}]\!]$, and $n = 100$ samples are requested. Even without using our retainment-optimizing algorithms, and instead sampling $S_{\mathcal{F}}$ and $S_{\mathcal{F}'}$ independently and uniformly at random, we would still expect a substantial number of configurations to be retained. In practice, however, configuration spaces are typically enormous and exceed n by many orders of magnitude,[5] making accidental retainment extremely unlikely. We therefore argue that, in practice, a simpler prediction is sufficient if we assume that retainment occurs only deliberately.

The ratio of valid samples for $\mathcal{F}$ that remain valid for $\mathcal{F}'$ is given by $v := |[\![\mathcal{F} \wedge \mathcal{F}']\!]| / |[\![\mathcal{F}]\!]|$. For $S_{\mathcal{F}}$ uniformly n-sampled, the expected number of samples that

[5] If the number of valid configurations were smaller than the number of samples we are willing to test, the configurations could simply be tested exhaustively.

are also valid for $\mathcal{F}'$ is therefore $n{\cdot}v$. For both algorithms, the expected number of samples drawn from $\tilde{\mathcal{U}}^*_{\text{old}}$ is $n{\cdot}r$. Thus, for $S_{\mathcal{F}'}$ sampled with either algorithm, we predict the expected number of samples to be close to $\min(n{\cdot}v, n{\cdot}r)$. We therefore define the *predicted expected retainment ratio* ER^* as

$$ER^* \coloneqq \frac{\min(n{\cdot}v, n{\cdot}r)}{n} = \min(v, r).$$

The update types shown in Figure 1 directly influence the expected retainment. For *incomparable* updates, no configuration of $\mathcal{F}$ is valid for $\mathcal{F}'$, so $S_{\mathcal{F}} \cap S_{\mathcal{F}'} = \emptyset$. The exact expected retainment ratio is therefore $ER = 0$, and the predicted ratio ER^* is likewise zero since $v = r = 0$. For *refactoring* updates, we have $[\![\mathcal{F}]\!] = [\![\mathcal{F}']\!] = [\![\mathcal{F} \wedge \mathcal{F}']\!]$. Both Algorithm 1 and 2 deterministically return $S_{\mathcal{F}'} = S_{\mathcal{F}}$ in this case, since $r = 1$ and thus $k = n$. Likewise, $ER^* = 1$ as $v = r = 1$. For the remaining update types the prediction ER^* is no longer guaranteed to match the exact value ER due to the approximation mentioned above. For *generalizing* updates where $[\![\mathcal{F}]\!] \subsetneq [\![\mathcal{F}']\!]$, we have $v = 1$ and thus $ER^* = r$. Vice versa, for *specializing* updates where $[\![\mathcal{F}]\!] \supsetneq [\![\mathcal{F}']\!]$, we have $r = 1$ and $ER^* = v$.

5 Uniform Sampling Methods

The two retainment sampling algorithms described in the previous section can be implemented in different ways, depending on the underlying uniform sampler and data structures that are used. The following operations need to be supported:

- Computing the model counts $|[\![\mathcal{F}']\!]|$ and $|[\![\mathcal{F}'_{\text{old}}]\!]|$ to obtain the ratio r.
- Uniform sampling from $[\![\mathcal{F}'_{\text{old}}]\!]$ and $[\![\mathcal{F}'_{\text{new}}]\!]$, i.e., the samplers $\mathcal{U}^*_{\text{old}}$ and $\mathcal{U}^*_{\text{new}}$.

Feature models are commonly given in *conjunctive normal form* (CNF), which is also the input format expected by many model counters and samplers. For $\mathcal{F}$ and $\mathcal{F}'$ given in CNF, the conjunction $\mathcal{F}'_{\text{old}} = \mathcal{F} \wedge \mathcal{F}'$ is in CNF as well and can thus be handled directly by existing tools. This is not the case for $\mathcal{F}'_{\text{new}} = \neg\mathcal{F} \wedge \mathcal{F}'$, since $\neg\mathcal{F}$ is generally not in CNF. While a transformation to CNF is possible, the straight-forward distributive approach for conversion into CNF can incur an exponential blow-up in the size of the formula.

We provide two implementations which tackle efficient uniform sampling from $\neg\mathcal{F} \wedge \mathcal{F}'$ in different ways. The evaluation in Section 6 will compare the overall efficiency of both implementation.

5.1 Tseitin-based Sampling

The *Tseitin transformation* is a standard linear-size reduction that converts a Boolean formula Φ into an equisatisfiable CNF $\mathrm{T}(\Phi)$ by introducing auxiliary variables for subformulas [43]. As noted by Kuiter et al. [21], these additional variables do not impede most feature-model analyses, since valid configurations of Φ and $\mathrm{T}(\Phi)$ are in bijection. As an example, consider the negated CNF

$$\Phi \;=\; \neg\big((\neg A \vee B) \wedge C\big) \;=\; \neg(\neg A \vee B) \vee \neg C.$$

We introduce a fresh Tseitin variable to encode each negated clause that contains more than a single literal. For the example, let X encode $\neg(\neg A \vee B)$. Then, by expressing the equivalence as CNF, we yield a representation of Φ in CNF:

$$\begin{aligned}\Phi &= (X \vee \neg C) \wedge \big(X \leftarrow \neg(\neg A \vee B)\big) \wedge \big(X \rightarrow \neg(\neg A \vee B)\big)\\ &= (X \vee \neg C) \wedge (X \vee \neg A \vee B) \wedge (\neg X \vee A) \wedge (\neg X \vee \neg B).\end{aligned}$$

Now, given a valid configuration for Φ, e.g., $\theta \coloneqq \{A{=}\texttt{true}, B{=}\texttt{false}, C{=}\texttt{true}\}$, we get a matching configuration θ' that is valid for $\mathrm{T}(\Phi)$ by extending θ with the assignment for X that follows from the substitution $X \leftrightarrow \neg(\neg A \vee B)$: $\theta' \coloneqq \theta \cup \{X{=}\texttt{true}\}$. Conversely, dropping the auxiliary variables from any valid configuration of $\mathrm{T}(\Phi)$ yields a valid configuration of Φ.

Thus, we can compute $\mathcal{F}'_{\text{new}} = \neg\mathcal{F} \wedge \mathcal{F}'$ via $\mathrm{T}(\neg\mathcal{F}) \wedge \mathcal{F}'$, that is, by negating the CNF for $\mathcal{F}$ and converting the result into a new CNF with help of the Tseitin transform to avoid the exponential blow-up. Our implementation uses SharpSAT-TD [18] as model counter, SPUR [2] as uniform sampler, and the Python library PySAT [13] for the Tseitin transformation.

5.2 Rejection Sampling

The second approach we consider for uniform sampling from $\mathcal{F}'_{\text{new}}$ is *rejection sampling*. This method relies on the observation that a uniform sampler for some sampling space Ω can be used as a uniform sampler for any subspace $\omega \subseteq \Omega$ by keeping samples that fall into ω and rejecting all others [44]. All samples generated for ω in this fashion are then guaranteed to be uniformly random as well. The crucial factor determining feasibility of rejection sampling is the *hit rate* h, defined as the ratio between the solution space and sampling space sizes: $h \coloneqq |\omega|/|\Omega|$. Intuitively, the lower the hit rate, the more samples need to be rejected, requiring more candidate samples and longer runtime overall.

In our setting, we sample configurations from $\mathcal{F}'$ and reject those that are valid for $\mathcal{F}$, thereby obtaining uniform samples for $\mathcal{F}'_{\text{new}} = \neg\mathcal{F} \wedge \mathcal{F}'$. The hit rate is then $h = |[\![\mathcal{F}'_{\text{new}}]\!]|/|[\![\mathcal{F}']\!]| = 1 - r$, with r the ratio defined in Section 4. This is promising, as r directly determines the number of samples needed from $\mathcal{F}'_{\text{new}}$. Thus, we expect that the more samples are needed (small r), the easier it gets to generate them via rejection sampling (high hit rate h). Vice versa, for low hit rates h, the effort to generate valid samples increases, but likewise we expect to require only few of them.

Our implementation again uses SharpSAT-TD for the two required model-counting queries and SPUR for uniform sampling. Validity checking with respect to $\mathcal{F}$ is implemented using PySAT.

6 Empirical Methodology

Our empirical study is designed to answer the following three research questions concerning retainment sampling.

RQ 1 (Expected Retainment) *To what extent do real-world feature model histories admit sample retainment?*

This question examines which update types occur in real-world feature model histories and how they influence opportunities for retainment. To answer RQ 1, we (1) analyze the update types in the evolution histories of our subject systems, (2) compute the predicted expected retainment ratio ER^* for each model, and (3) experimentally evaluate how closely these predictions match the observed retainment rates. This allows us to assess whether retainment sampling can be expected to provide practical benefits.

RQ 2 (Scalability) *Which of the proposed sampling methods achieves better scalability with respect to model size and sampling time?*

This question compares the two implementation methods proposed for handling negated CNF formulas during retainment sampling: the Tseitin-based approach and rejection sampling. To assess scalability, we apply both methods to real-world feature model histories and additionally construct artificial histories for large feature models by reversing random updates (constraint removals, variable removals, and variable renamings). For each update, we sample 1000 configurations[6] and measure execution time. This reveals the scalability limits of both methods and helps determine which is preferable for a given model.

RQ 3 (Sample-and-test Performance) *How do the overall runtimes of sampling and subsequent testing compare between retainment sampling and a baseline without retainment?*

This question addresses whether the additional effort of retainment sampling is offset by savings during testing, especially when testing unique configurations is expensive. To compare end-to-end runtime, we sample 1000 configurations for each update and measure the total number of unique samples across the full history. We use the uniform retainment sampler from Algorithm 1 with the Tseitin-based and rejection-based variants from Section 5. As a baseline without retainment, we sample uniformly with SPUR [2], which also underlies both retainment samplers. This analysis shows whether retainment sampling has the potential to reduce total testing time.

6.1 Subject Systems

To evaluate our research questions, we draw on a recently published comprehensive dataset of real-world feature models [39]. Five models in this dataset include evolution histories preserving feature names.[7] Table 1 summarizes these histories,

[6] See Section 8.2 for a discussion of why we choose $n = 1000$.

[7] For the sixth model, automotive2 [17], feature names are obfuscated to protect corporate secrets, and the history consist of only four snapshots which are all incomparable. We hence excluded this model from our studies.

Table 1: Feature model histories used for evaluation. The lower four histories are artificially generated for existing models.

Model	Snapshots	Features			Constraints	
		Min	Max	Unified	Min	Max
BusyBox [35]	248	439	631	710	463	691
Fiasco [34]	31	213	253	285	1040	1542
FinancialServices [27]	10	557	774	1082	1001	1148
soletta [34]	132	99	430	493	177	1678
uClibc [34]	140	229	256	320	992	1818
adderii [12]	51	1251	1276	1316	13	3206
automotive01 [12]	51	2511	2513	2553	317	10 275
ea2468 [12]	51	1406	1408	1448	22	3470
uclinux-base [12]	51	378	380	430	216	7366

showing the number of snapshots, the minimum and maximum number of features and constraints, and the total number of unified features (see Definition 1). Note that a history with i snapshots amounts to $i-1$ updates. Besides FinancialServices, where the domain is clear from its name, the remaining models stem from the systems software domain.

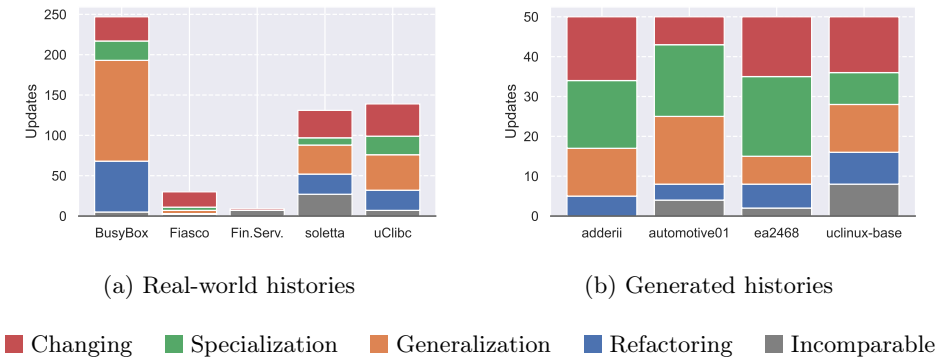
For scalability experiments (RQ 2), we additionally generate artificial histories from larger feature models in the literature. Starting from the final model, we construct histories backwards by applying randomized updates that remove or add constraints, and remove or rename features. Reversing these updates yields a sequence of growing models that evolve into the original. We selected four large models from the UnWise benchmark set [12] that still permit reasonable model counting with SharpSAT-TD, and for each we generated an artificial history of 50 updates (see lower half of Table 1). Parameters guiding the random generation were chosen to achieve a mixture of update types as shown in Figure 5b.

Each feature model snapshot is given in conjunctive normal form in DIMACS format. Since feature IDs are not always consistent across snapshots, we unify each history upfront rather than update by update. This ensures that every snapshot ranges over the same feature set and that IDs remain consistent across all snapshots, simplifying analysis and constraint comparison. The unification process is fast, requiring less than four seconds in total for all histories. We also apply the CNF preprocessing tool PMC [23] for equivalence-preserving optimizations, which improve Tseitin-based sampling performance.

All experiments were run on a Ubuntu 24.04.2 LTS system with 64 GB RAM, and an Intel Core i9-10900K CPU.

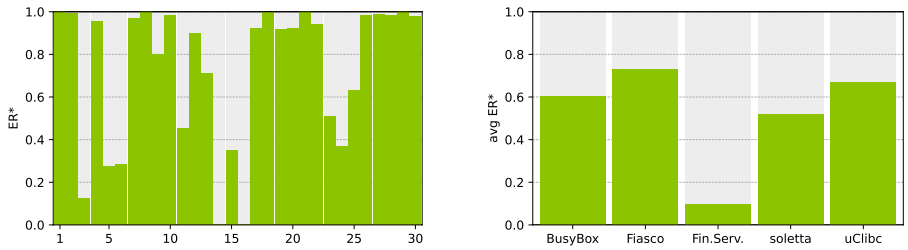
7 Results

RQ 1: Expected Retainment Following the classification of update types from Figure 1, Figure 5a shows their distribution across model histories. Recall from



(a) Real-world histories (b) Generated histories

Fig. 5: Distribution of update types (see Figure 1) for the subject systems.



(a) ER^* for updates in the Fiasco history. (b) Average ER^* across all updates.

Fig. 6: Predicted expected retainment ratios ER^*.

Section 4.4 that refactoring updates allow full retainment, while incomparable updates prevent any retainment. For the remaining update types, at least partial retainment is expected. For the systems software models, the proportion of incomparable updates is generally low, ranging from 2 % in BusyBox to 20 % in soletta. The smaller FinancialServices history contains seven incomparable updates out of nine.[8]

Next, we compute the predicted expected retainment ratio ER^*. As an example, Figure 6a shows the ER^* for each update of the Fiasco model history. For the two incomparable updates, ER^* is zero, for all other updates, moderate to very high retainment is predicted. Figure 6b reports the average ER^* across all updates for the benchmark histories. For the systems software models, predictions are promising, with averages between 50 % and 73 %. Even for FinancialServices, dominated by incomparable updates, the overall average prediction is nearly 10 %.

[8] In practice, incomparable updates can be detected immediately by model counting, allowing a fall-back to standard uniform sampling without additional retainment overhead.

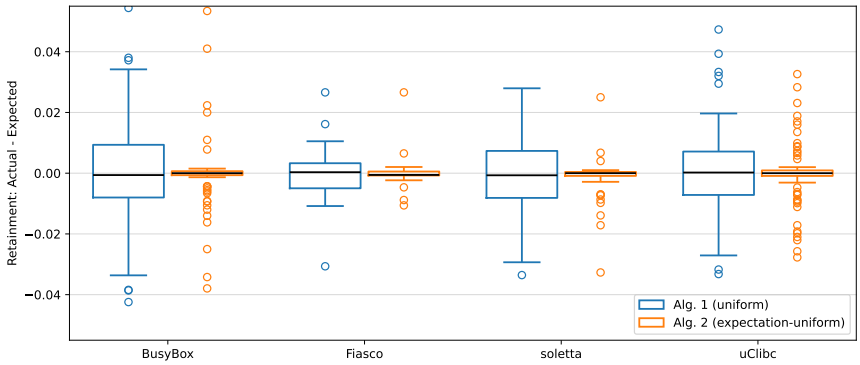


Fig. 7: Prediction accuracy of the expected retainment ratio per update for 1000 samples using the Tseitin-based implementation, excluding incomparable and refactoring updates for which predictions are guaranteed to be exact. Four outliers fall outside the plot range: BusyBox Alg. 1 at −0.067, uClibc Alg. 1 at 0.173, and uClibc Alg. 2 at 0.168 and 0.058.

Finally, we validate predictions against actual retainment. Sampling $n = 1000$ configurations per update with both Algorithms 1 and 2, we compare predicted and observed retainment ratios in the box plot shown in Figure 7. As predictions for incomparable ($ER^* = ER = 0$) and refactoring updates ($ER^* = ER = 1$) are necessarily exact, these update types are excluded to prevent them from skewing the results.[9] For the remaining updates, predictions align closely with observations: median differences (bold line) are essentially zero, with most updates deviating by less than one percentage point.

Answer to RQ 1: Real-world feature model histories provide ample opportunities for sample retainment, with the predicted expected retainment ratio closely matching observed outcomes.

RQ 2: Scalability To compare the performance of the Tseitin and rejection sampling method (Section 5), we sampled 1000 configurations per update and measured the overall execution time for each history.

Table 2 reports sampling times for both retainment methods compared to the baseline (i.e., sampling with SPUR without explicit retainment), along with the achieved retainment rates. The results highlight the scalability limits of the Tseitin-based approach: it runs out of memory for adderii and times out after two hours for automotive01. In contrast, rejection sampling scales in line with the baseline across all cases. While rejection sampling introduces some overhead in medium-sized models, this overhead vanishes for the largest model automotive01,

[9] As FinancialServices contains only two of those updates, there are not enough data points for a meaningful box plot.

Table 2: Sampling times and overall sample retainment for 1000 samples per update using Algorithm 1 (in seconds, timeout after 2 hours).

Model	Sampling Time			Sample Retainment		
	Tseitin	Rejection	Baseline	Tseitin	Rejection	Baseline
BusyBox	162.59	131.76	87.90	60.18 %	60.20 %	0 %
Fiasco	14.58	8.27	3.13	73.07 %	73.02 %	0.01 %
FinancialServices	1478.59	120.10	12.23	9.54 %	9.31 %	6.05 %
soletta	193.21	40.53	32.32	51.92 %	51.92 %	0 %
uClibc	112.64	37.74	26.38	67.03 %	67.16 %	0 %
adderii	memout	123.63	101.74		25.55 %	0 %
automotive01	timeout	3979.31	4039.01		23.24 %	0 %
ea2468	457.18	117.02	101.66	18.57 %	18.49 %	0 %
uclinux-base	82.18	11.99	5.99	36.53 %	36.58 %	0 %

where it even completes about one minute faster than the baseline without retainment.

Overall, these findings suggest that rejection sampling is the more scalable implementation. As discussed in Section 5.2, its applicability depends on sufficiently high hit rates, which our benchmarks confirm.[10] Nevertheless, in situations where hit rates drop too low, the Tseitin-based method remains a viable, though less scalable, fallback.

Answer to RQ 2: Rejection sampling consistently scales better than the Tseitin-based variant, in line with the baseline.

RQ 3: Sample-and-Test Performance We model total runtime as the sum of sampling time t_s and testing time for all unique samples. If u denotes the number of unique samples and x the average test duration per configuration, the total runtime is $f(x) = t_s + u{\cdot}x$. Accordingly, in Figure 8, the slope of each line is determined by u, while the intercept on the y-axis corresponds to t_s. The intersection of a retainment-sampling line with the dashed baseline therefore marks the minimum average test duration at which retainment sampling becomes advantageous.

For BusyBox and FinancialServices, shown exemplarily in Figure 8, retainment sampling adds only modest overhead compared to the baseline. Across all benchmarks (Table 3), the minimum average test duration required to outperform the baseline lies in the millisecond range, which is orders of magnitude below typical configuration testing costs in practice [9,8]. Even for FinancialServices,

[10] For automotive01, on average 78.6 candidate configurations could be generated and checked per second. In that case, to yield one valid sample per second, hit rates above 0.012 are sufficient. For smaller models where checks and candidate generation are faster, hit rates can be correspondingly lower.

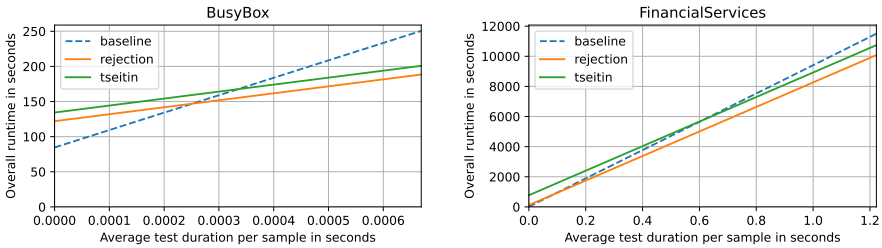


Fig. 8: Runtime projections for sampling 1000 configurations per update and testing all unique ones.

Table 3: Minimal average test durations in seconds for which Algorithm 1 with the respective sampling method outperforms the baseline.

History	Tseitin	Rejection
BusyBox	0.000 33	0.000 25
Fiasco	0.000 31	0.000 19
FinancialServices	0.611 28	0.085 64
soletta	0.000 33	0.000 02
uClibc	0.000 19	0.000 11

where overall retainment is only about 10 %, the reduction in unique samples is sufficient to offset the sampling overhead once individual tests last longer than a fraction of a second.

Answer to RQ 3: Retainment sampling outperforms the baseline in sample-and-test scenarios even when average configuration testing is extremely fast.

8 Discussion

Our results highlight both the potential and the boundaries of retainment sampling. We next discuss its implications, limitations, and threats to validity.

8.1 Implications and Limitations

Retainment sampling is applicable to arbitrary feature models. As such it can be easily integrated into existing tools and workflows (e.g., in FeatureIDE [42]), allowing developers to benefit directly from its advantages. At the same time, achieving testing-time savings in scenarios such as RQ 3 requires that testing-relevant changes be reflected in the feature model. Changes that manifest only at the source code level are therefore out of scope. Nevertheless, evidence from practice suggests that code and feature models often co-evolve [32], and in such settings retainment sampling can be expected to provide tangible benefits.

Our evaluation shows that the model-count-based heuristic predicts expected retainment with high precision, since by-chance retainment occurs only rarely in realistic systems. This opens the door to adaptive sampling strategies: if an update is predicted to provide little or no retainment, one can easily fall back to regular uniform sampling to avoid overhead.

With respect to implementation strategies, our results clearly favor rejection sampling over the Tseitin-based method, since the latter runs into scalability limits. Importantly, hit rates were sufficiently high in all our benchmarks to make rejection sampling practical. Because hit rates can be derived from model counts, the decision of which strategy to apply can be automated: when hit rates are too low, one can still resort to the Tseitin-based approach. Our prototype tool already implements such automatic fall-backs, ensuring that retainment sampling can be applied robustly across diverse scenarios.

Regarding the choice between the uniform (Algorithm 1) and the expectation-uniform (Algorithm 2) retainment samplers, Figure 7 shows that the lower variance in retainment of the second algorithm is also observable in practice. This increases confidence in the expected retainment predictions, which could be beneficial for scenarios where resources for testing are pre-allocated based on this prediction. In terms of runtime, both algorithms behave similarly, and we observed no meaningful performance difference. While the strict n-sampler uniformity ensured by Algorithm 1 is interesting from a mathematical perspective and important for critical applications that require certification, the expectation-uniformity of Algorithm 2 might be enough for most practical purposes.

Finally, the sample-and-test scenario (see RQ 3) illustrates the practical potential of retainment sampling. Retainment rates were found to be high in most of our subject systems, but even in FinancialServices, with only around 10 % retainment, the approach becomes worthwhile as soon as average test durations for a single configuration exceed one second. Conversely, given that test runs in practice frequently take minutes or even hours, our results suggest that retainment sampling can deliver significant savings even under pessimistic assumptions about retainment rates.

8.2 Threats to Validity

Several design decisions in our experiments may affect the validity of our findings. First, the choice of uniform sampler is a potential threat to internal validity. We consider this threat to be low, since we employed SPUR, a state-of-the-art sampler with formal uniformity guarantees [2], which showed the best performance on feature models in recent evaluations [14].

Second, the decision to draw 1000 samples for every update represents a trade-off between runtime and statistical power, and is established in variability studies [26]. We regard this choice as reasonable, since it provides a reliable estimate of performance while keeping experimental costs manageable. Smaller values of n would increase the likelihood of spurious results, while conversely, larger values begin to stress the implementation, resulting in larger sample

files and a greater influence of I/O efficiency, which is beyond the scope of our proof-of-concept implementation.

Third, external validity is threatened by the choice of feature model histories. Our set of histories may not cover all situations encountered in practice, and we cannot rule out that there are systems with very different characteristics. However, we argue that the selected histories are diverse enough to provide meaningful insights into the performance of the samplers across different scenarios. In fact, they have been used extensively in previous research [39] and thus facilitate comparability. Another external validity threat is in the naming of features in feature model histories. We identify features by their names which usually are unchanged across updates. This might be not the case for industrial feature models from public sources, where feature names might appear obfuscated and hence may differ across updates. In the latter case, matching obfuscated feature names to the features they represent would enable improvements in effectiveness of retainment sampling.

9 Conclusion

Testing of configurable systems is a challenge, rooted in the combinatorial blowup of configurations in need for testing. This is especially pronounced in contexts where already the deployment and testing of a single configuration can take hours, thereby limiting the number of configurations that can be tested in reasonable time. Up to now, existing literature mainly focused on generating suitable but small sets of samples that provide good coverage of the overall configuration space. Uniform random sampling (URS) of software configuration spaces has shown to provide a suitable basis for generating such sample sets based on specifications in the form of feature models.

This paper has opened a new dimension in the quest for reducing testing effort through URS, exploring the potential of reusing configurations from already tested versions of an evolving configurable system. Our method, called *retainment sampling*, is based on the observation that updates to configuration spaces often do not invalidate all prior configurations. Exploiting this fact, retainment sampling reuses as many samples as possible from previous versions while guaranteeing uniformity in the target system version. Besides formally proving correctness of our new sampling method in terms of uniformity guarantees, we provide empirical evidence of its effectiveness. By means of experiments on real-world evolving software configuration spaces, we show that retainment rates of up to 73 % can be exploited. In practice, this renders our method beneficial already for systems where deployment and testing is expected to take not even a second. Furthermore, we provide a model-count-based heuristics that accurately predicts which retainment rates are to be expected. Due to formal correctness guarantees and sampling quality we proved in this paper, retainment sampling may serve as a foundation for URS of feature models.

Acknowledgements. This work was partially supported by the DFG under the projects TRR 248 (see https://perspicuous-computing.science, project ID 389792660) and EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy) and the NWO through Veni grant VI.Veni.222.431.

References

1. Acher, M., Perrouin, G., Cordy, M.: BURST: A benchmarking platform for uniform random sampling techniques. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B. pp. 36–40. ACM, Leicester United Kindom (Sep 2021). https://doi.org/10.1145/3461002.3473070
2. Achlioptas, D., Hammoudeh, Z.S., Theodoropoulos, P.: Fast Sampling of Perfectly Uniform Satisfying Assignments. In: Theory and Applications of Satisfiability Testing – SAT 2018, vol. 10929, pp. 135–147. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_9
3. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37521-7
4. Arcuri, A., Briand, L.: Formal analysis of the probability of interaction fault detection using random testing. IEEE Transactions on Software Engineering **38**(5), 1088–1099 (2012). https://doi.org/10.1109/TSE.2011.85
5. Cohen, M.P.: Stratified Sampling. In: International Encyclopedia of Statistical Science, pp. 2704–2708. Springer, Berlin, Heidelberg (2025). https://doi.org/10.1007/978-3-662-69359-9_660
6. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: Proceedings of the 40th International Conference on Software Engineering. pp. 549–559. ACM, Gothenburg Sweden (May 2018). https://doi.org/10.1145/3180155.3180248
7. Golia, P., Soos, M., Chakraborty, S., Meel, K.S.: Designing Samplers is Easy: The Boon of Testers. In: Proceedings of Formal Methods in Computer-Aided Design (FMCAD). TU Wien (Aug 2021). https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_31
8. Haas, R., Nömmer, R., Juergens, E., Apel, S.: Optimization of Automated and Manual Software Tests in Industrial Practice: A Survey and Historical Analysis. IEEE Transactions on Software Engineering **50**(8), 2005–2020 (Aug 2024). https://doi.org/10.1109/TSE.2024.3418191
9. Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.: Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A. SPLC '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3382025.3414985, https://doi.org/10.1145/3382025.3414985
10. Haslinger, E.N., Lopez-Herrejon, R.E., Egyed, A.: Using feature model knowledge to speed up the generation of covering arrays. In: Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2430502.2430524, https://doi.org/10.1145/2430502.2430524
11. Heradio, R., Fernandez-Amoros, D., Galindo, J.A., Benavides, D., Batory, D.: Uniform and scalable sampling of highly configurable systems. Empirical Software Engineering **27**(2), 44 (Mar 2022). https://doi.org/10.1007/s10664-021-10102-5

12. Heß, T., Schmidt, T.J., Ostheimer, L., Krieter, S., Thüm, T.: UnWise: High T-Wise Coverage from Uniform Sampling. In: Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems. pp. 37–45. ACM, Bern Switzerland (Feb 2024). https://doi.org/10.1145/3634713.3634716
13. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: A Python Toolkit for Prototyping with SAT Oracles. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Theory and Applications of Satisfiability Testing – SAT 2018, vol. 10929, pp. 428–437. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_26
14. Käfer, N., Apel, S., Baier, C., Dubslaff, C., Hermanns, H.: When to Sample from Feature Diagrams? In: Proceedings of the 19th International Working Conference on Variability Modelling of Software-Intensive Systems. pp. 11–20. VaMoS '25, Association for Computing Machinery, New York, NY, USA (May 2025). https://doi.org/10.1145/3715340.3715442
15. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (1990)
16. Keyfitz, N.: Sampling with Probabilities Proportional to Size: Adjustment for Changes in the Probabilities. Journal of the American Statistical Association **46**(253), 105–109 (1951)
17. Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., Schaefer, I.: Is there a mismatch between real-world feature models and product-line research? In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 291–302. ACM, Paderborn Germany (Aug 2017). https://doi.org/10.1145/3106237.3106252
18. Korhonen, T., Järvisalo, M.: SharpSAT-TD in Model Counting Competitions 2021-2023 (2023). https://doi.org/10.48550/ARXIV.2308.15819
19. Krieter, S., Thüm, T., Schulze, S., Saake, G., Leich, T.: Yasa: yet another sampling algorithm. In: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems. VaMoS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377024.3377042, https://doi.org/10.1145/3377024.3377042
20. Kuhn, D., Wallace, D., Gallo, A.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering **30**(6), 418–421 (2004). https://doi.org/10.1109/TSE.2004.24
21. Kuiter, E., Krieter, S., Sundermann, C., Thüm, T., Saake, G.: Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–13. ACM, Rochester MI USA (Oct 2022). https://doi.org/10.1145/3551349.3556938
22. Käfer, N.: Artifact for the paper "Uniform Retainment Sampling for Evolving Software Configuration Spaces". Zenodo (Jul 2026). https://doi.org/10.5281/zenodo.21280998
23. Lagniez, J.M., Marquis, P.: Preprocessing for Propositional Model Counting. Proceedings of the AAAI Conference on Artificial Intelligence **28**(1) (Jun 2014). https://doi.org/10.1609/aaai.v28i1.9116
24. Lee, J., Kang, S., Lee, D.: A survey on software product line testing. In: Proceedings of the 16th International Software Product Line Conference - Volume 1. pp. 31–40. SPLC '12, Association for Computing Machinery, New York, NY, USA (Sep 2012). https://doi.org/10.1145/2362536.2362545

25. Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., Apel, S.: A comparison of 10 sampling algorithms for configurable systems. In: Proceedings of the 38th International Conference on Software Engineering. pp. 643–654. ICSE '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2884781.2884793, https://doi.org/10.1145/2884781.2884793
26. Mordahl, A., Oh, J., Koc, U., Wei, S., Gazzillo, P.: An empirical study of real-world variability bugs detected by variability-oblivious tools. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 50–61. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3338906.3338967, https://doi.org/10.1145/3338906.3338967
27. Nieke, M., Mauro, J., Seidl, C., Thüm, T., Yu, I.C., Franzke, F.: Anomaly analyses for feature-model evolution. ACM SIGPLAN Notices **53**(9), 188–201 (Apr 2020). https://doi.org/10.1145/3393934.3278123
28. Nieke, M., Sampaio, G., Thüm, T., Seidl, C., Teixeira, L., Schaefer, I.: Guiding the evolution of product-line configurations. Software and Systems Modeling **21**(1), 225–247 (Feb 2022). https://doi.org/10.1007/s10270-021-00906-w
29. Oh, J., Batory, D., Heradio, R.: Finding Near-optimal Configurations in Colossal Spaces with Statistical Guarantees. ACM Trans. Softw. Eng. Methodol. **33**(1), 7:1–7:36 (Nov 2023). https://doi.org/10.1145/3611663
30. Oh, J., Batory, D., Heule, M., Myers, M., Gazzillo, P.: Uniform Sampling from Kconfig Feature Models. Tech. Rep. TR-19-02, University of Texas at Austin, Department of Computer Science (Mar 2019)
31. Oh, J., Batory, D., Myers, M., Siegmund, N.: Finding near-optimal configurations in product lines by random sampling. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 61–71. ACM, Paderborn Germany (Aug 2017). https://doi.org/10.1145/3106237.3106273
32. Passos, L., Teixeira, L., Dintzner, N., Apel, S., Wasowski, A., Czarnecki, K., Borba, P., Guo, J.: Coevolution of variability models and related software artifacts. Empirical Software Engineering **21**(4), 1744–1793 (Aug 2016). https://doi.org/10.1007/s10664-015-9364-x
33. Passos, L.T., Queiroz, R., Mukelabai, M., Berger, T., Apel, S., Czarnecki, K., Padilla, J.A.: A study of feature scattering in the linux kernel. IEEE Trans. Software Eng. **47**(1), 146–164 (2021)
34. Pett, T., Heß, T., Krieter, S., Thüm, T., Schaefer, I.: Continuous T-Wise Coverage. In: Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A. SPLC '23, vol. A, pp. 87–98. Association for Computing Machinery, New York, NY, USA (Aug 2023). https://doi.org/10.1145/3579027.3608980
35. Pett, T., Krieter, S., Runge, T., Thüm, T., Lochau, M., Schaefer, I.: Stability of Product-Line Sampling in Continuous Integration. In: Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems. pp. 1–9. ACM, Krems Austria (Feb 2021). https://doi.org/10.1145/3442391.3442410
36. Plazar, Q., Acher, M., Perrouin, G., Devroey, X., Cordy, M.: Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet? In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 240–251. IEEE, Xi'an, China (Apr 2019). https://doi.org/10.1109/ICST.2019.00032
37. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: 14th IEEE International Requirements Engineering Conference (RE'06). pp. 139–148 (2006). https://doi.org/10.1109/RE.2006.23

38. Sharma, S., Gupta, R., Roy, S., Meel, K.S.: Knowledge Compilation meets Uniform Sampling. In: LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. pp. 620–602 (2018). https://doi.org/10.29007/h4p9
39. Sundermann, C., Brancaccio, V.F., Kuiter, E., Krieter, S., Heß, T., Thüm, T.: Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking. In: 28th ACM International Systems and Software Product Line Conference. pp. 54–65. ACM, Dommeldange Luxembourg (Sep 2024). https://doi.org/10.1145/3646548.3672590
40. Sundermann, R., Böhm, S., Krieter, S., Lochau, M., Thüm, T.: Covering T-Wise Interactions of Deployed Configurations. In: Proceedings of the 19th International Working Conference on Variability Modelling of Software-Intensive Systems. pp. 21–29. VaMoS '25, Association for Computing Machinery, New York, NY, USA (May 2025). https://doi.org/10.1145/3715340.3715432
41. Thüm, T., Batory, D., Kastner, C.: Reasoning about edits to feature models. In: 2009 IEEE 31st International Conference on Software Engineering. pp. 254–264 (May 2009). https://doi.org/10.1109/ICSE.2009.5070526
42. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming **79**, 70–85 (Jan 2014). https://doi.org/10.1016/j.scico.2012.06.002
43. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_28, https://doi.org/10.1007/978-3-642-81955-1_28
44. von Neumann, J.: Various techniques used in connection with random digits. In: Monte Carlo Method, pp. 36–38. National Bureau of Standards Applied Mathematics Series, 12, Washington, D.C.: U.S. Government Printing Office (1951)
45. White, J., Galindo, J.A., Saxena, T., Dougherty, B., Benavides, D., Schmidt, D.C.: Evolving feature model configurations in software product lines. Journal of Systems and Software **87**, 119–136 (Jan 2014). https://doi.org/10.1016/j.jss.2013.10.010