Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?

Quentin Plazar*, Mathieu Acher*, Gilles Perrouin†, Xavier Devroey‡ and Maxime Cordy§
*Univ Rennes, Inria, CNRS, IRISA, Rennes, France. Emails: quentin.plazar@gmail.com, mathieu.acher@irisa.fr
†PReCISE/NaDI, Faculty of Computer Science, University of Namur, Namur, Belgium. Email: gilles.perrouin@unamur.be
‡Delft University of Technology, Delft, The Netherlands. Email: x.d.m.devroey@tudelft.nl
§SnT, University of Luxembourg, Luxembourg, Luxembourg. Email: maxime.cordy@uni.lu

***Abstract*—Uniform or near-uniform generation of solutions for large satisfiability formulas is a problem of theoretical and practical interest for the testing community. Recent works proposed two algorithms (namely UniGen and QuickSampler) for reaching a good compromise between execution time and uniformity guarantees, with empirical evidence on SAT benchmarks. In the context of highly-configurable software systems (e.g., Linux), it is unclear whether UniGen and QuickSampler can scale and sample uniform software configurations. In this paper, we perform a thorough experiment on 128 real-world feature models. We find that UniGen is unable to produce SAT solutions out of such feature models. Furthermore, we show that QuickSampler does not generate uniform samples and that some features are either never part of the sample or too frequently present. Finally, using a case study, we characterize the impacts of these results on the ability to find bugs in a configurable system. Overall, our results suggest that we are not there: more research is needed to explore the cost-effectiveness of uniform sampling when testing large configurable systems.**

***Index Terms*—configurable systems, software testing, uniform sampling, SAT, variability modeling, software product lines**

I. Introduction

A. Configurable Systems and Feature Models

Configurable systems form a vast and heterogeneous class of software systems that encompasses: *Software Product Lines* [1], [2], operating systems kernels, web development frameworks/stacks, e-commerce configurators, code generators, *Systems of Systems (SoS)*, *software ecosystems* (*e.g.,* Android's "Play Store"), autonomous systems, *etc.* While being very different in their goals and implementations, configurable systems see their behaviour affected by the activation or deactivation of one or more *feature(s)*, i.e. units of variability, configuration *options*. Configurable systems may involve thousands of features with complex dependencies. In order to represent the set of valid combinations of options a configurable system can have – its *variants* (which we also name *configurations*) – we can use a tree-like structure, called *feature model* [3]. Each feature may be decomposed into sub-features and additional constraints may be specified amongst the different features. Within feature models, features can be mandatory (present in every configuration) or selected depending on the groups they belong to (OR, XOR, etc.) and cross-tree constraints (dependence on or exclusion of other feature selections). To support automated reasoning, feature models have been equipped with formal semantics and in particular based on first-order logic [4], [5]. These efforts led to a variety of tool-aided analyses that use constraint solvers [6]. In particular, satisfiability (SAT) solvers have shown to scale for large feature models [7], [8].

This research was partially funded by the EU Project STAMP ICT-16-10 No.731529, the NIRICT 3TU.BSR (Big Software on the Run) project, the FNRS Grant O05518F-RG03, and the ANR-17-CE25-0010-01 VaryVary project. This work was done when Maxime Cordy worked at the University of Namur.

B. Sampling of Configurations

One important issue when testing configurable systems is that the number of variants grows exponentially with the number of features, preventing the exhaustive enumeration of all valid configurations authorized by the feature model in most cases. Therefore, a simple strategy to cope with this issue is to *sample* configurations of interest before testing the corresponding variants. There are many ways of sampling configurations depending on testing objectives. T-wise sampling adopts a Combinatorial Interaction Testing (CIT) approach, which relies on the hypothesis that most faults are caused by undesired interactions of a small number of features [9]. T-wise sampling being an NP-complete problem in the presence of constraints, various heuristics have been proposed [10], from greedy algorithms [11], [12] to meta-heuristics [13], [14]. Meta-heuristics are also at the heart of dissimilarity sampling techniques that maximize distances between configurations [15], [16]. There are also approaches that combines several objectives (coverage, cost of configurations, *etc.*) [17]–[19]. These techniques exhibit different tradeoffs between the need to maximise the distances and preserving the validity of generated solutions, this validity being checked typically using SAT solvers. Evolutionary algorithms generally start from initial random solutions, and the way they are produced matters [20]. For example, Maaranen *et al.* observed an improvement of 10% on average when using uniformly distributed random numbers for initialization [21]. De Perthuis de Laillevault *et al.* theoretically demonstrated the relevance of repeated uniform random sampling to initialize the population of evolutionary algorithms [22]. For configurable systems, Henard *et al.* noticed that SAT solvers' internal order yields non-uniform (and predictable) exploration of the search space potentially biasing results of search-based configuration selec-

© 2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

https://doi.org/10.1109/ICST.2019.00032

tion and prioritization [15]. Other criteria rely on the idea of minimal (as few features selected as possible) and maximal (as many features selected as possible) configurations [23], [24]. Randomness is involved in such criteria too. For example, one-enabled selects one random configuration having only one feature enabled, most-enabled-disabled selects minimal/maximal configurations (several pairs of configurations may satisfy the criterion). In their experiments, Medeiros *et al.* relied on the first configuration returned by the SAT solver [24]. However, Halin *et al.* showed that taking the first configuration did not reveal faults in their studied system, while an approach relying on random sampling was effective [25]. In the absence of constraints between the options, Arcuri *et al.* theoretically demonstrate that a uniform random sampling strategy may outperform t-wise sampling [26]. Random sampling thus typically serves as a baseline to evaluate and compare sampling strategies.

C. Motivation & Problem Statement

The current body of knowledge emphasizes the importance of random uniform sampling and its potential for configurable systems. It can indeed act as a standalone sampling method to select one configuration amongst all the *valid* ones, such that each valid configuration receives an equal probability to be selected. It can also support other techniques (e.g., evolutionary algorithms). To assess its applicability in practice, the first step is to experiment actual state-of-the-art implementations on feature models. To this end, we selected two approaches from the literature, UniGen [27], [28] and QuickSampler [29] because they exhibit interesting trade-offs with respect to uniformity and scalability (e.g., QuickSampler sacrifices some uniformity for a substantial increase in performance). While there exist benchmarks [29] that evaluate and compare these tools, those do not consider feature models and their peculiarities. For example, the `uClinux-config` model, representing the configuration options of an embedded Linux for micro-controllers, has $7.7 * 10^{417}$ possible solutions. In contrast, the largest formula used by UniGen and QuickSampler has $\approx 10^{48}$ solutions [29]. Considering these differences, our objective is to investigate: *(i)* whether UniGen and QuickSampler are efficient enough (in terms of computation time) to be ***applied on such feature models*** and *(ii)* whether they do so while ***guaranteeing to generate a reasonably-uniform sample*** of configurations.

D. Contributions

The contributions of this paper are the following:

1) An empirical assessment of UniGen and QuickSampler on boolean formulas stemming from 128 feature models of various sizes and taken from real-world benchmarks [8], [30], [31]. Some of them exhibit more than 10,000 features.
2) Given that some feature models are huge, we show that conventional methods for evaluating the uniformity of a sample distribution are not appropriate. Accordingly, we propose an adapted new method of assessing sample distributions based on the frequency of features compared to the ground truth. This method has the merit of pinpointing some weaknesses (regarding deviation) of sampling techniques while being intuitive to interpret in the context of configurable systems.
3) A presentation of the results showing that while QuickSampler scales on the most significant models, sample distributions deviate from up to 800% from a uniform distribution. Furthermore, UniGen guarantees uniformity to the price of scalability: it cannot process feature model models of more than 1,000 features.
4) A first assessment of the ability of QuickSampler to cover buggy features on JHipster [32], a configurable web development stack. QuickSampler can over-represent a feature (+116%) involved in a rare bug (appearing in only 0.49% of all configurations) but also one (+94%) involved in three of six the JHipster's interaction bugs. We explore these deviations with respect to a uniform distribution (also achieved by UniGen) and provide recommendations for the future.

Section II describes the experimental protocol we followed to asses performance and uniformity of QuickSampler and UniGen and Section III presents the results we obtained. Section IV presents the assessment of generated samples on JHipster. Section V discusses our findings. Section VI presents the threats to our empirical analyses. Section VII discusses additional related work, and Section VIII concludes the paper.

II. Study Design

A. Research Questions

Uniform sampling is an interesting approach to testing configurable systems. However, practitioners and researchers ignore whether state-of-the-art algorithms are applicable over feature models. Specifically, we aim to address three research questions:

- **RQ1** (scalability and execution time): *Are UniGen and QuickSampler able to generate samples out of feature models?* To study scalability and execution time when applied to feature models.
- **RQ2** (uniformity): *Do UniGen and QuickSampler generate uniform configurations out of feature models?* We aim to assess the quality of the sample with respect to uniformity (prior work [29] suggests that QuickSampler is close to uniformity).
- **RQ3** (relevance for testing): *How does QuickSampler's sacrifices on uniformity impact its bug-finding ability in JHipster?* By relating sampled frequencies of features with their associated bugs on the JHipster case [25], we perform an early exploration of how these techniques behave with respect to bug distribution [26].

B. UniGen and QuickSampler

Several SAT samplers exist in the literature [27]–[29], [33]–[36] and achieve varying compromises between performance and theoretical properties of the sampling process (*e.g.,* uniformity, near-uniformity). We will focus our study on two

samplers that achieve state-of-the-art results, UniGen [27], [28], and QuickSampler [29]. A recent ICSE'18 paper [29] compares the two algorithms on large real-world benchmarks (SAT instances), showing that QuickSampler is faster than UniGen, with a distribution reasonably closed to uniform.

On the one hand, UniGen uses a hashing-based algorithm to generate samples in a nearly uniform manner with strong theoretical guarantees: it either produces samples satisfying a nearly uniform distribution or it produces no sample at all. These strong theoretical properties come at a cost: the hashing based approach requires adding large clauses to formulas so they can be sampled. These clauses grow quadratically in size with the number of variables in the formula, which can raise scalability issues.

On the other hand, QuickSampler's algorithm is based on a strong set of heuristics, which are shown to produce samples quickly in practice on a large set of industrial benchmarks [29]. However, the tool offers no guarantee on the distribution of generated samples, or even on the termination of the sampling process and the validity of generated samples (they have to be checked with a SAT solver after the generation phase).

C. Input Feature Models

We use a large number of well-known and publicly available feature models in our study, which are of various difficulty. Specifically, we rely on the benchmarks used in [8], [30], [31]. Specifically, feature models were used to assess the SAT-hardness of feature models, to investigate the properties of real-world feature models [31] and to evaluate a configuration algorithm for propagating decisions [30].

1) Feature model benchmark properties: In total, we use the feature models of 128 real-world configurable systems (Linux, eCos, toybox, JHipster, etc.) with varying sizes and complexity. We first rely on 117 feature models used in [30], [31]. The majority of feature models contain between 1,221 and 1,266 features. Of these 117 models, 107 comprise between 2,968 and 4,138 cross-tree constraints, while one has 14,295 and the other nine have between 49,770 and 50,606 cross-tree constraints [30], [31]. Second, we include 10 additional feature models used in [8] and not in [30], [31]; they also contain a large number of features (e.g., more than 6,000). Third, we also add the JHipster feature model [25], [32] to the study, a realistic but relatively smaller feature model (45 variables, 26,000+ configurations). We later refer to these benchmarks as the feature model benchmarks.

Once put in conjunctive normal form, these instances typically contain between 1 and 15 thousand variables and up to 340 thousand clauses. The hardest of them, modelling the Linux kernel configuration, contains more than 6 thousand variables, 340 thousand clauses, and is generally seen as a milestone in configurable system analysis.

2) Replication of [29]: In addition to these feature models, we have replicated the initial experiments on industrial SAT formulas as conducted in [29]. We use these results as a sanity check, to ensure that we are using the tools with the same configurations that were previously compared. Moreover, since these original formulae are much smaller than the feature models we use (typically a few thousand clauses), they will provide a basis of results for statistical analysis, in case a solver cannot produce enough samples on the harder formulas. We later refer to these benchmarks as the *non-feature model benchmarks*.

D. Experimental Setup

We are interested in several characteristics of the samplers under study, which include scalability (execution time) as well as the quality of generated samples (regarding their statistical distribution). For scalability, we evaluate execution time by running each sampler on every benchmark, until it generates 1 million samples or execution times out after 2 hours. For QuickSampler, the timeout duration is split equally between sample generation and validity check (one hour each). Experiments were run on an Intel(R) Core(TM) i7-5600U (2,6 GHz, 2 cores), 16GB RAM, running Linux Fedora 22.

1) Frequency of features: The quality of a sampler's distribution is very hard to evaluate on large benchmarks, since the huge size of the solution space makes standard statistical testing inapplicable. For example, the `uClinux-config` feature model has $7.7 * 10^{417}$ possible solutions. Instead, we rely on an approximate measure of statistical indicators, namely the frequency of observation of certain features in the samples generated. These indicators are not sufficient to show if the distribution of samples is uniform, but they may show it is not. Importantly, our indicators can pinpoint flaws in the sampling process that are critical for testing purposes, such as a feature never being selected in the produced samples (despite the ground truth states this feature should be selected 80% of the time). We give more details about the computation of this indicator in the next section.

2) MIS: As a final note, both QuickSampler and UniGen can benefit from the knowledge of an *independent support* for the formula they sample. An independent support is a subset of the formula's variables which, when assigned a truth value, leaves at most one possible satisfying valuation for the remaining variables. We used some tool[1] to compute Minimal Independent Supports (MIS). We attempted to compute the MIS for all our benchmarks and followed the procedure provided by QuickSampler's authors to detect invalid MIS [29]. Indeed, we ensured that the number of solutions for a given model was not larger than $2^{|S|}$, where $|S|$ represents the number of variables of the independent support. Should it be the case, this means that the MIS cannot be used to enumerate all the valid instances of the model. On benchmarks shared by QuickSampler, we found the same incorrect MIS as a sanity check. For feature models, we found that two Linux kernel variants (2.6.3X) on which MIS computation either ran out of memory or could not be computed within an hour. Additionally, there was one more Linux model (2.6.28) as well as buildroot and freetz for which it was not

[1]We used the implementation https://bitbucket.org/kuldeepmeel/mis based on algorithm presented in [37]

possible to compute the exact number of solutions within 10 minutes with SharpSAT [38]. We nevertheless note that for other feature models both MIS computation and validity check were possible. On some selected feature models, we successfully computed multiple MIS and gave them as input to QuickSampler and UniGen and found no impact on these models. In particular, for UniGen, giving the MIS did not help to compute any sample within one hour on 24 feature models, UniGen was only able to compute a sample for the JHipster feature model (both with and without MIS). This, combined with the fact that the Linux feature models are reference cases in the configurable systems' community, led us to a different choice than our predecessors [29]: instead of removing these intractable MIS cases and missing out on (some) reference models, we chose not to use MIS for feature models in our experiments. This threat to validity is discussed in Section VI.

E. Reproducibility

We provide a Git repository with all feature models of the benchmarks, Python scripts to execute experiments, Python scripts to compute plots, figures, and statistics of this paper as well as additional ones, and instructions to reuse our work:

https://github.com/diverse-project/samplingfm

III. Results

A. RQ1 (scalability and execution time)

1) Reproduction of [29]: Though comprehensive results can be found online in our repository, we do not report here the detailed performance results for the non-feature model benchmarks. They are indeed very similar to the ones published in the previous studies. In particular, we reproduce the fact that UniGen was not able to scale for some SAT instances. Another interesting phenomenon is that UniGen never produces more than a 10th of the samples produced by QuickSampler (we have tried to produce more samples than reported in [29]).

2) Results on feature model benchmarks.: The first noticeable fact is that UniGen was not able to produce samples for any of the feature model benchmarks in the allowed time limit. This can in most cases be explained by the fact that feature model benchmarks are in general much larger than the previous benchmarks used, and therefore imply a substantial overhead during the algorithm initialization and for solving hashing clauses. Interestingly, there are some feature model benchmarks which are comparable in size to their non-feature model counterparts. One of them is the `toybox` feature model, which has 544 variables and 1,020 clauses. Using MIS, we were able to compute a minimal independent support of size 64. Given this independent support, UniGen was not able to compute a single sample in 24 hours. In comparison, the `ProcessBean.sk_8_64` also has an independent support of size 64, 4,768 variables, 14,458 clauses, and UniGen can compute several thousand samples for this benchmark. Of course, size metrics do not provide an accurate account of a formula's intrinsic hardness, but these results clearly show that UniGen does not scale well enough to tackle complex feature models.

On almost all feature model benchmarks, QuickSampler is able to produce 1 million samples in less than one hour. On 4 instances, namely `embtoolkit`, `freetz`, and the two linux kernel representations, QuickSampler runs out of memory. Manual inspection revealed that QuickSampler is in fact able to produce samples for these benchmarks, but crashes before outputing them to a file, making their validation impossible. Table I shows the detailed performance results on a selection of feature model benchmarks. The first column shows the benchmark's name, the next three columns show formula metrics (number of variables, clauses and solution). The number of solutions were computed using SharpSAT [39] with a time limit of 5 minutes, which was not always sufficient to obtain a result. The last 3 columns show QuickSampler's performance measures:

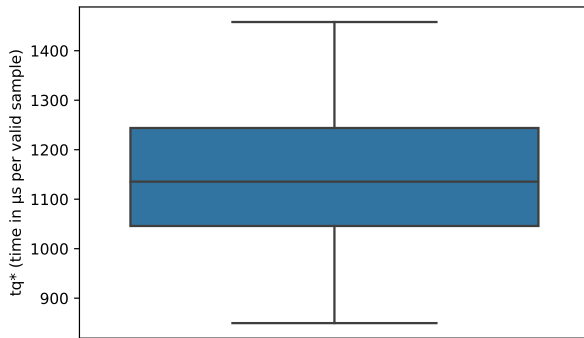
- *Samples* is the number of samples generated;
- *Valid* is the proportion of generated samples which turned out to be correct after validation;
- *Time/sample* ($t_q * (\mu s)$) is the time elapsed per generated valid sample. This time is calculated as the ratio between the number of valid samples and the sum of both the generation time and the validation time.

These numbers were obtained without the use of independent support. However, we expect them to be similar with their use, since most of the time of QuickSampler is spent on sample validation, which is virtually unaffected by the use of independent supports. The results show QuickSampler's good scalability (it is able to produce around one valid sample per millisecond on most feature model benchmarks), which is largely due to its heuristic nature: the overhead of dealing with large benchmarks is low. Figure 1 depicts a boxplot showing this trend: the median time is 1.1 millisecond, closed to the first and third quartile. There are some outliers (see the violin plot of Figure 1) that can be classified in two categories: (1) very low execution time (`JHipster`: 84.4 nanosecond and `toybox2`: 59.2 nanosecond); (2) very high execution time (`Linux 2.6.28.6-icse11`, `buildroot`, and `coreboot` that take around 30 millisecond).

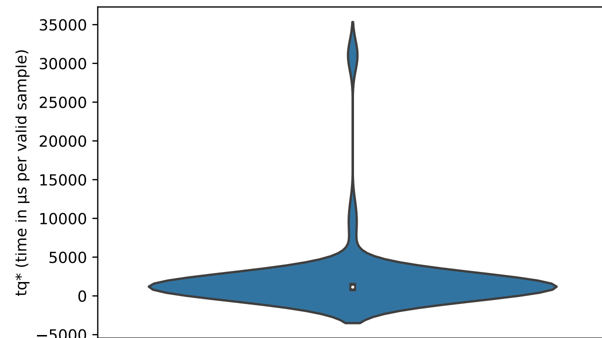
In practice, the majority of the samples produced by QuickSampler turn out to be valid. Figure 2 shows that the median is 0.74, closed to the first quartile (0.68) and third quartile (0.78). There are some outliers: on some benchmarks, such as `buildroot`, the results show a very low proportion of valid samples, but this is in most case due to the validation process timing out, which has the effect of considering every unchecked sample as invalid. Hence such outliers are due to our experimental measurements and can be ignored. However there are two remaining outliers for which the validation timeout did not occur: the `JHipster` and `fiasco` feature models for which QuickSampler has a remarkably low success rate (resp. 0.118 and 0.047). We recall that the Valid ratio has no incidence on the quality of the sample since after the validation step, only valid solutions are kept in the sample. Our results simply show that the heuristic of QuickSampler is effective in general but can also have difficulties for some

TABLE I
PERFORMANCE EVALUATION RESULTS FOR QUICKSAMPLER (WE SHOW ONLY AN EXCERPT OF THE 128 FEATURE MODELS)

Benchmark	Vars	Clauses	Solutions	Samples	Valid	$\mathtt{t_q}*$ (μs)
JHipster	45	104	26256	1001212	0.118	84.4
mpc50	1213	3728	3.16818e122	1006608	0.823	982.2
pc_i82544	1259	3179	1.83678e127	1159521	0.872	1014.6
refidt334	1263	3140	2.94575e134	1104609	0.731	1212.7
dreamcast	1252	3168	1.09233e123	1127845	0.775	1117.7
XSEngine	1260	3803	2.04657e133	1174443	0.656	1266
aim711	1264	3873	1.82286e127	1107607	0.771	1103.2
p2106	1249	3824	3.71280e124	1028172	0.763	1096.9
integrator_arm9	1267	50606	4.05835e129	1056238	0.847	1916.6
uClinux-config	11254	31637	7.78028e417	1149017	1	11548.7
toybox	544	1020	1.44991e17	1021556	0.895	59.3
coreboot	12268	47091	1.40174e94	1149017	0.144	30019.9
busybox-1.18	6796	17836	8.49902e216	1149017	0.725	5098.2
axTLS	684	2155	4.28726e20	1098865	0.386	581.9
2.6.28.6-icse11	6888	343944	N.A.	1089245	0.13	31766
fiasco	1638	5228	3.58108e14	1080270	0.047	9715.6
uClinux	1850	2468	1.62962e91	1149017	1	358.9
toybox2	544	1020	1.44991e17	1129007	0.892	59.2
buildroot	14910	45603	N.A.	1085562	0.14	31226.5
ecos-icse11	1244	3146	4.97468e125	1173038	0.816	987.5
freebsd-icse11	1396	62183	8.38866e313	1152001	0.402	8228.4



(a) Time/sample (box plot, without outliers)



(b) Time/sample (violin plot, with outliers)

Fig. 1. Time (in μs) per valid sample for QuickSampler

feature model instances.

> UniGen is not able to produce samples for any of the feature model benchmarks (except the smallest one, JHipster) and thus cannot be used in the context of large configurable systems. QuickSampler does scale and is able to produce one valid sample per millisecond on most feature models. In general, the heuristic of QuickSampler is effective to select valid configurations, but can also exhibit low valid ratios for some feature models.

B. RQ2 (uniformity)

1) Replication of [29] and limitations of uniformity assessment: In previous work [29], the quality of the sample distribution is estimated by generating a large number of samples on a given benchmark, and counting how many times each possible solution was generated. Solutions are grouped according to how many times they are generated, and the size of each obtained group is measured. The results are then compared to those obtained by performing the same operations on the output generated by a uniform random number generator. The main limitation of this estimation method is that it requires the samplers to generate at least 4 times as many samples as the used benchmark has solutions for the results to be statistically significant. For the feature model benchmark, generating as many benchmarks is almost never possible, because the number of solutions routinely reaches 10^{50} and more (see column *Solutions* in Table I). For the same reason, we cannot perform standard statistical tests, such as the Pearson Chi-squared test.

Another drawback of this approach is that in order to compare two samplers, the same number of samples has to be used for both. This is generally achieved by sub-sampling uniformly

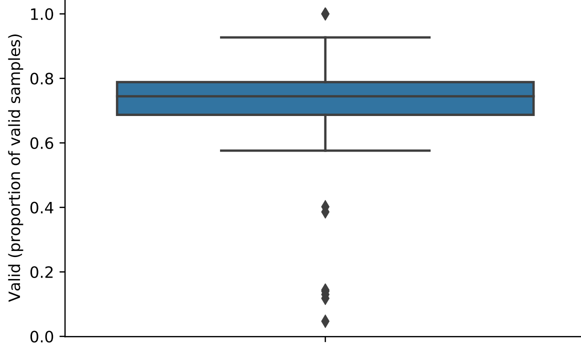


Fig. 2. Proportions of valid samples (QuickSampler)

from the larger of the two set of samples generated, but sub-sampling affects the size of the grouping later calculated and make them look closer to those obtain from a uniform generator. To show this, we had QuickSampler compute 5 million samples on the non-feature model benchmarks. We then successively sub-sampled 1 million and 500 thousand samples from this sample set. For each subset, we draw the following curve (in Figure 3): the x-axis represents the number of times each possible solution was generated and on the y-axis is how many samples were generated x times. On each graph we also show the results obtained from a uniform random number generator. On the first graph we also show the curve obtained from UniGen's samples on the same benchmark. It is clear that sub-sampling has an impact on the shape of the curve, making it closer to uniformity.

Therefore, we have opted for another measure to estimate the quality of sample distribution: the deviation between the frequencies of individual features evaluated to true in the samples. If ϕ is a SAT formula, we denote $\#SAT(\phi)$ the number of solutions to ϕ, and given a solution s, we say that a variable v appears in s if it evaluates to *true* in this solution. We can calculate the frequency of appearance of v in solutions to ϕ as:

$$f_{th}(v) = \frac{\#SAT(\phi \wedge v)}{\#SAT(\phi)}$$

Given a set of sample solutions to ϕ, we can also observe the frequency of v in that set, as being the proportion $f_{obs}(v)$ of solutions in the set in which v evaluates to $true$. We compute the deviation between the theoretical frequency and the observed one as:

$$dev(v) = 100 * \frac{|f_{th}(v) - f_{obs}(v)|}{f_{th}(v)}$$

Figure 4 shows for each feature, the deviation between the observed and theoretical frequency for the non-feature model benchmarks, sorted in ascending order. The lower green horizontal line shows the 10% threshold, below which we consider a deviation as *very low*, and the upper red line shows the 50% threshold, above which we consider a deviation as *very high*. Overall, our results confirm the relatively low deviation between the theoretical and observed frequencies for QuickSampler, as reported in [29]. The frequency deviations of UniGen are in line with the theoretical properties. Moreover, our frequency-based method also reveals insights that could not been visualized with the sub-sampling method. For instance, in Figure 5, it is clear that QuickSampler is not uniform (whereas the original histogram, based on subsampling, suggests QuickSampler is very close to uniform behavior).

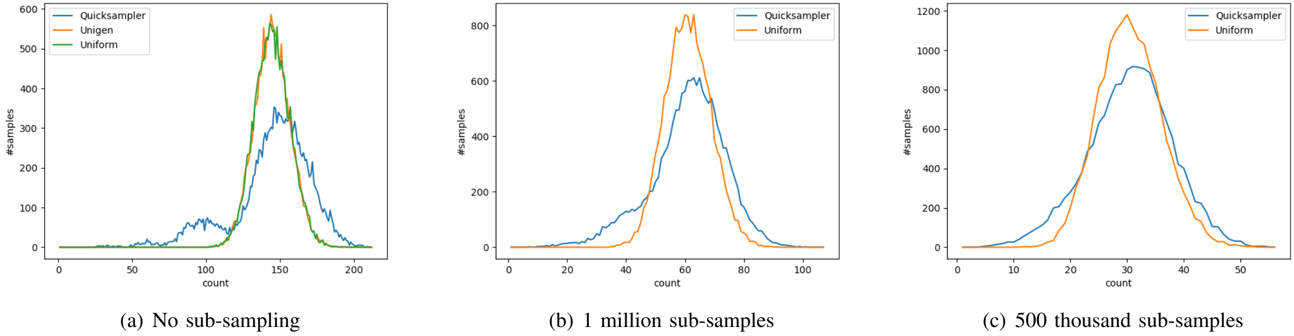
2) Results on feature model benchmarks: We recall that UniGen is not able to produce samples out of feature models (see RQ1) and we can only report frequency results of QuickSampler. Figure 6 presents the deviations for QuickSampler for a subset of our benchmark. We observe that deviations frequently go above the 50% threshold and can be as high as 800%. For `axTLS` (resp. `toybox`, `uClinux`) around 90% (resp. 80%, 95%) of features are above the red line (50% threshold). That is, the vast majority of features have large frequencies deviation w.r.t. uniform distribution. The frequency deviations of `ecos` and `aim711` are less severe compared to `axTLS`, `toybox`, and `uClinux` but remain much more important than with non-feature model instances.

We also observe that for some feature models, features may have a non-zero theoretical frequency ($f_{th} > 0$) but are never picked in the samples ($f_{obs} = 0$). In theory, QuickSampler always produces at least one sample where each feature is present (and one where it is absent) if such a solution exists. But practically, this relies on the termination of asynchronous solver calls for every variable. When the first solver calls terminate, QuickSampler can begin generating samples. Hence, the required number of samples may be generated before all the solver calls terminate, which explains why some variables are never present in the produced samples. This can be a problem if the samples are used directly because some features will be ignored and the potential bug(s) associated to them may be missed. If used as seeds for evolutionary algorithms, while there remains chances for these features to be finally selected after recombination of solutions, these samples may hamper the initialization phase.

> QuickSampler does not generate uniform samples out of real-world feature models; the difference with a uniform distribution is much more severe than with non-feature models (as in [29]). The majority of features exhibit frequencies that deviate above 50%. Some features have up to 800% frequencies differences or are never part of the sample (despite their theoretical presence).

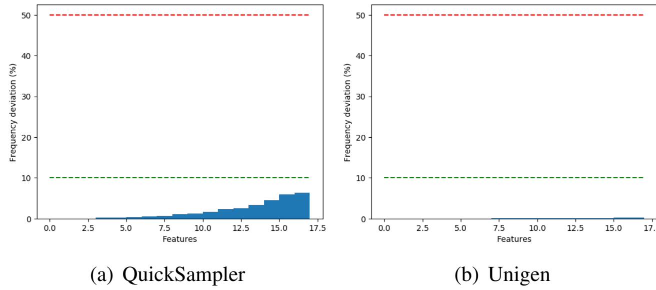
IV. Case study: JHipster

To derive further insights on the relevance of QuickSampler samples for testing, we consider the feature model of JHipster used in [25] (45 features and 26,256 configurations, see Table I). JHipster is an open-source, industrially used generator



(a) No sub-sampling (b) 1 million sub-samples (c) 500 thousand sub-samples

Fig. 3. Effect of sub-sampling on uniformity estimation



(a) QuickSampler (b) Unigen

Fig. 4. Frequency deviations for `case110` (a non-feature model instance) showing the non-uniformity of QuickSampler

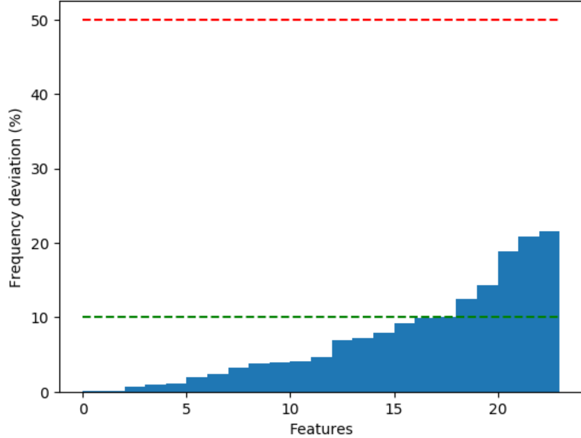


Fig. 5. Frequency deviations for `s820a_15_7_qs` (a non-feature model instance) showing the non-uniformity of QuickSampler. We do not depict UniGen results since it is again closed to perfection with negligible deviations, as in Figure 4

for developing Web applications. Prior work [25] tested all configurations of JHipster and found that 35.70% configurations fail. Importantly, features and feature interactions causing the bugs have been identified. With the JHipster case, we have an interesting opportunity to investigate the quality of the sampling w.r.t. bug-finding ability. This section thus addresses RQ3.

TABLE II
FEATURES INVOLVED IN JHIPSTER BUGS AND THEIR FREQUENCIES DEVIATIONS IN QUICKSAMPLER

feature	f_{obs}	f_{th}	dev	$rdev$
MongoDB	0.039	0.018	116.0	116.0
Uaa	0.248	0.171	45.0	45.0
ElasticSearch	0.408	0.485	16.0	-16.0
Hibernate2ndLvlCache	0.573	0.647	11.0	-11.0
SocialLogin	0.237	0.268	11.0	-11.0
Docker	0.545	0.500	9.0	9.0
MariaDB	0.302	0.324	7.0	-7.0
Gradle	0.518	0.500	4.0	4.0
Monolithic	0.651	0.675	4.0	-4.0
EhCache	0.313	0.324	3.0	-3.0

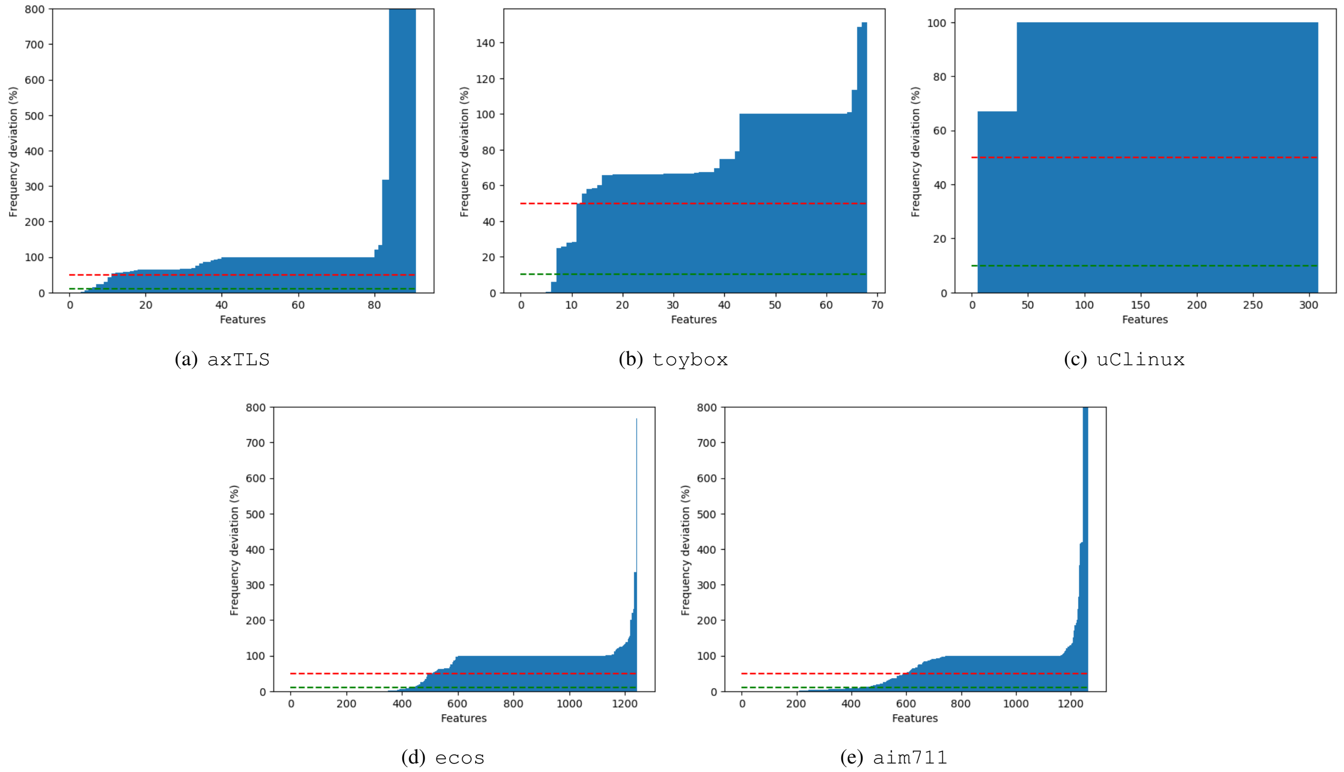
As the JHipster feature model is manageable (only 26,256 configurations), both UniGen and QuickSampler can sample a statistically significant number of samples. Therefore we can plot and exploit the histogram that counts how many times each configuration (SAT solution) has been sampled. Figure 7 shows that UniGen is indistinguishable from uniform, but QuickSampler is not close to uniform behavior. In the following, we consider UniGen as uniform (thus having same bug-finding ability as random uniform sampling study of JHipster [25]) and therefore focus only on QuickSampler ability to find bugs.

To better understand the difference with a uniform distribution, we apply our feature frequency methods (see Figure 8). Beyond the clear difference with UniGen, for QuickSampler, we can notice that 18 features have above 10% frequencies deviations and 5 features deviate above 50%:

- $dev(\texttt{MongoDB}) = 116\%$
- $dev(\texttt{Cassandra}) = 107\%$
- $dev(\texttt{UaaServer}) = 94\%$
- $dev(\texttt{Server}) = 87\%$
- $dev(\texttt{MicroserviceApplication}) = 84\%$

A. RQ3 (relevance for testing)

Table II lists all features involved in the 6 interaction faults that cause 99% of failures. We also report $rdev$ for showing the positive or negative frequencies deviations for



(a) axTLS (b) toybox (c) uClinux

(d) ecos (e) aim711

Fig. 6. QuickSampler frequency deviations on feature model benchmarks

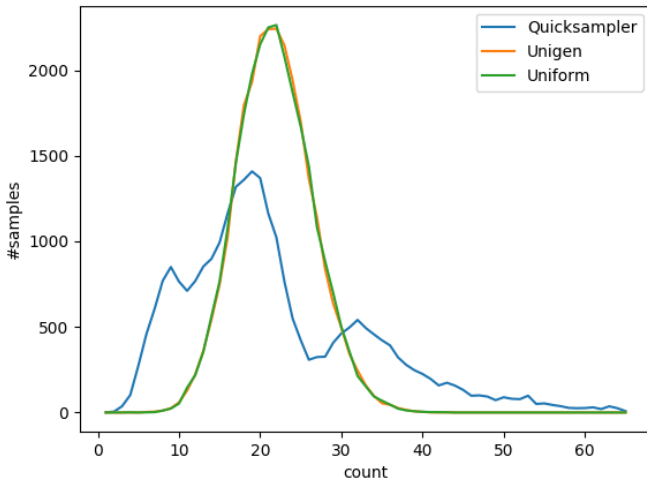
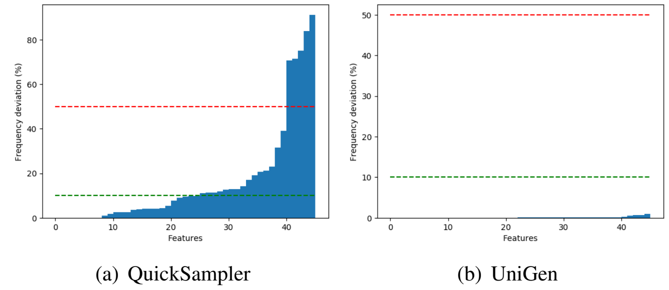


Fig. 7. JHipster feature model: comparison of UniGen, QuickSampler and ground truth (uniform)



(a) QuickSampler (b) UniGen

Fig. 8. Frequency deviations on the JHipster Feature Model

QuickSampler. For instance, the frequency of `Uaa` in QuickSampler samples is greater than the ground truth (+45%) while `MariaDB` is less frequent as it should be (-7%).

Specifically, for the different configuration bug reported by Halin *et al.* [25], we have: **MOSO**, the 2-interaction of `MongoDB` and `SocialLogin` (0.49% out of 35.70% of failures): It is the less important source of bug. `MongoDB` is much more present than it should be (+116%), which has a positive incidence of the finding of this bug. It should be noted that the theoretical frequency of `MongoDB` is very low, since this feature appears in a very few configurations. **MAGR**, the 2-interaction of `MariaDB` and `Gradle` (16.179% out of 35.70% of failures): It is the most severe source of bug. Yet `MariaDB` is under-represented (-7%) in QuickSampler samples while being important and present in 32% of configurations. **UADO**, the 2-interaction of `Docker` and `Uaa` (6.825 % out of 35.70% of failures). Both features are over-represented (resp. +45% and +9%). As a result, there are more chances to find this fault. **OASQL**, the 2-interaction of `Uaa` and `Hibernate2ndLvlCache` (2.438 % out of 35.70%

of failures). Unfortunately `Hibernate2ndLvlCache` is under-represented (-11%) despite a high presence in all configurations (65%). **UAEH**, the 2-interaction of `Uaa` and `EhCache` (2.194% out of 35.70% of failures). `EhCache` is sightly under-represented (-3%). **MADO**, the 4-interaction of `MariaDB`, `Monolithic`, `Docker` and `ElasticSearch` (5.59% out of 35.70% of failures). `ElasticSearch` is under-represented (-16%, only 40% of appearance versus 48% theoretically), as well as `MariaDB` and `Monolithic`.

B. Practical implications

In practice, executing and testing a JHipster configuration has a significant cost in resources and time (10 minutes on average per configuration). The exhaustive testing of all configurations at each commit or release is not an option. Developers and maintainers of JHipster rather have a limited testing budget at their disposal (*i.e.,* a dozen of configurations) [25]. As a result, we cannot take the whole sample of QuickSampler or UniGen and we rather need to take an excerpt of this sample. Various sub-sampling strategies can be considered [24], [40] either based on random, t-wise, *etc.* Without an uniform distribution, the sub-sampling process will operate over a non-representative configuration set, which may incidentally promote or underestimate some features. Overall, it is an open issue how to effectively sub-sample out of UniGen and QuickSampler solutions.

> The sample of QuickSampler is not representative of the real features' distribution of JHipster. Yet QuickSampler is fortunate to over-represent `Uaa` (a feature involved in 3 interaction faults) while other large deviations have luckily no incidence on bug finding.

V. Discussion

A. Performance (RQ1)

Our experiments allow us to derive a first result on the applicability of the tested uniform random sampling to feature models. First, it was not possible to obtain any set of samples for our feature model benchmarks (JHipster excepted) with uniformity guarantees through UniGen. The exact impact of minimal independent support on UniGen's ability to produce samples for configurable systems has to be fully explored in future work, while our experiments shown that there are at least some cases in which it was not helpful. We noticed that the size of the independent support is not necessarily a good indicator for the sampling task. In contrast, QuickSampler produces samples quickly on all our models, for the largest "Linux" ones, the tool would need a slight modification to output samples as a stream to ease serialization and SAT validity tasks in order to avoid observed timeouts.

B. Uniformity (RQ2 and RQ3)

On the only feature model on which UniGen was applicable it achieved its uniformity promises. QuickSampler does not offer such promises and it was indeed the case on our feature models. While the deviation remains modest for small models, it raises quickly above 50% for larger ones. Thus, we have to give up completely on uniformity in these cases. Additional research is needed to determine if such samples hinder the performance of additional sampling algorithms. The JHipster case revealed that QuickSampler may over represent some features involved in rare bugs and ignore ones in more frequent ones.

Our benchmarks also challenged the measure of uniformity itself. With such large number of solutions for feature model formulas, we had to define new heuristics to approximate uniformity since sub-sampling was misleading. Ours was based in individual frequencies of enabled feature occurrence, which is both scalable and exploitable by test engineers and uniform tool samplers' developers.

C. Summary: Are We There Yet?

To summarize, there is still some way to go before we can benefit from both fast and uniform random sampling for large feature models. In short: we are not there yet. A theoretical investigation should be conducted on UniGen to understand why it fails on some feature model formulas that have similar characteristics as some non-feature models. We also need to understand the poor validity performance of QuickSampler on some cases. We believe that our experiments form a first step to address such questions both in the configurable systems and SAT solving communities.

VI. Threats to Validity

Internal Validity: A number of issues might threaten the internal validity of our study. First, we did not use minimal independent support (MIS) as inputs to QuickSampler and UniGen. The threat mainly applies to UniGen since QuickSampler is able to compute samples anyway. It would be surprising that MIS has an effect on the sampling distribution, since the algorithm of QuickSampler does not depend on it; MIS is essentially exploited to further enhance execution time. Though MIS can be computed for most of the feature models, it was impossible to compute or to validate it for some of the most important feature models. MIS also did not improve UniGen's scalability on at least 24 feature models. This led us to abandon MIS for feature model benchmarks given the importance of several of these models. MIS is not a strict requirement of UniGen (neither QuickSampler) but we acknowledge that its usage deserves an in-depth and future attention.

Second, bugs or misuse of QuickSampler or UniGen might cause wrong results. We mitigate this issue by first reproducing prior results [29]. We compared the results and found no significant difference w.r.t. execution time, including magnitude order and timeouts. We also successfully reproduced the histogram and properties of the sample distributions as reported in [29]. It allows one to identify some weaknesses of the method used for assessing uniformity. In addition, we contacted the first author of UniGen to verify our settings when processing feature models.

Construct Validity: Our way of assessing uniformity forms a construct validity threat. It has been defined for cases when the distribution of solutions cannot be assessed directly by common statistical methods since the total number of solutions is too large. We believe that this approximation of uniformity distribution assessment is also for interest to study bug finding ability when the samples are used for testing purposes.

External Validity: There are some threats that may affect the generalizability of our study. The benchmarks might not be representative of feature models used in practice. To mitigate this issue, we tested 128 real-world feature models with a varying number of features and constraints that have been used in prior studies [8], [30], [31]. We also include the JHipster feature model that comes with bugs associated to all configurations [25]. Results obtained by Halin *et al.* on the efficiency of an (ideal since *a posteriori*) uniform sampling technique and our preliminary results on the efficiency of QuickSampler need to be confirmed for other studies.

VII. Related Work

To the best of our knowledge, we are the first to empirically assess uniform or quasi-uniform SAT-based sampling techniques to test configurable systems. Our work therefore resides at the intersection of researches on SAT solving and configurable systems testing.

Uniform Sampling of SAT formulas: We selected UniGen [27], [28] and QuickSampler [29] because they implemented different tradeoffs in terms of uniformity and performance, these two objectives being conflicting [41]. Apart from the hashing techniques using in UniGen and by other authors [35], Monte-Carlo methods have been used (e.g., [42]). They do provide guarantees on uniformity but they are sometimes too slow in practice.

Other approaches focus on performance as their top priority while offering some strategies to improve uniformity. For example, Wei *et al.* combined random walks with simulated annealing to "correct" uniformity of the samples [33]. Kitchen *et al.* combines different statistical methods to tend towards uniformity at each sampling step [34]. QuickSampler selects values for variables as uniformly as possible and partially fixes the validity of the solution afterwards.

Configurable Systems Sampling: As outlined by a recent survey by Varshosaz *et al.* [40], many artifacts of configurable systems have been considered for sampling, such as feature models, code, tests, etc. The authors also indicate that feature model remains the *de facto* standard input for many sampling techniques and note a certain prominence of greedy and metaheuristic-based algorithms, which are heavy consumers of random samples. We have discussed these techniques in Section I. Unfortunately the impact of random sampling on resulting samples is not discussed in this survey, the authors just mentioned that the non-determinism inherent to these techniques can be an issue in a regression testing scenario.

Medeiros *et al.* compared 10 sampling algorithms and also found that random sampling is difficult to combine with other techniques because it detects different faults in different runs [24]. Yet, they also found that it was able to find at least 66% of the faults. Halin *et al.* demonstrated that uniform random sampling forms a strong baseline for faults and failure efficiency on the JHipster case [25]. We need additional cases with *enumerable configuration spaces* (like the BLAST model studied by Cashman *et al.* [43]) to confirm or refute this result.

VIII. Conclusion

Can software developers and researchers employ uniform samplers for testing configurable systems? Our empirical study on 128 real-world feature models showed that we are not there yet. State-of-the-art algorithms, namely UniGen and QuickSampler, are respectively either not able to produce any sample (UniGen only succeded on the smallest of our 128 feature models, JHipster) or unable to generate uniform samples (deviations observed up to 800% with QuickSampler). Our empirical results have several consequences.

At the current state, we cannot investigate whether uniform sampling is a cost-effective strategy when testing configurable systems. It remains a striking question whether configuration bugs, due to (faulty interactions between) features, are uniformly distributed in real-world configurable systems. Researchers simply do not have at their disposal a scalable and effective tool for exploring and comparing this solution with other sampling algorithms (based on t-wise or dissimilarity criterion). For instance, without a uniform sampler, a random baseline may be impossible to instrument.

In the quest for an ideal uniform random sampler, the first step is to understand the reasons for the poor performance of the existing tools. Minimal independent supports and structural metrics of the formulas do not appear to be very good indicators of sampling hardness. Other metrics should be provided. Then, a first direction is to scale up UniGen that has the merit of providing strong theoretical guarantees about the distribution. Another one is to have distributions closer to uniformity with QuickSampler. The heuristic used by QuickSampler suggests a tradeoff between uniformity and diversity. There remains the question of how uniformity and diversity correlate in practice, and what are their respective benefits as coverage criteria to test configurable systems. In particular, it would be interesting to characterize this tradeoff with respect to diversity-based approaches (e.g., [15], [16]).

In practice, we cannot test billions of configurations since the cost is too prohibitive; developers rather have a budget of a dozen (or a hundred) of configurations. Another research direction is thus to efficiently reduce the huge sample of QuickSampler or UniGen. Several sub-sampling strategies can be applied over the original sample. Yet a non-uniform sample can have undesirable effects and ruin the sub-sampling, since some features are never or infrequently included. More research is definitely needed to improve the current situation.

References

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

[2] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, 2013.
[3] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA)," SEI, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.
[4] D. Batory, D. Benavides, and A. Ruiz-Cortés, "Automated analysis of feature models: Challenges ahead," *Communications of the ACM*, December 2006.
[5] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux, "Feature diagrams: A survey and a formal semantics," in *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 136–145.
[6] D. Benavides, S. Segura, and A. Ruiz-Cortes, "Automated analysis of feature models 20 years later: a literature review," *Information Systems*, vol. 35, no. 6, 2010.
[7] M. Mendonca, A. Wasowski, and K. Czarnecki, "Sat-based analysis of feature models is easy," in *Proceedings of the 13th International Software Product Line Conference*, ser. SPLC '09. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 231–240. [Online]. Available: http://dl.acm.org/citation.cfm?id=1753235.1753267
[8] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman, "Sat-based analysis of large real-world feature models is easy," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15. New York, NY, USA: ACM, 2015, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/2791060.2791070
[9] D. Kuhn, D. Wallace, and A. Gallo, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, jun 2004.
[10] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed, "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, 2015, pp. 1–10.
[11] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1*, vol. 1. ACM, 2012, p. 46.
[12] M. Cohen, M. Dwyer, and Jiangfan Shi, "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.
[13] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.
[14] F. Ensan, E. Bagheri, and D. Gašević, "Evolutionary Search-Based Test Generation for Software Product Line Feature Models," in *Advanced Information Systems Engineering: 24th International Conference, CAiSE '12*, J. Ralyté, X. Franch, S. Brinkkemper, and S. Wrycza, Eds. Springer, 2012, pp. 613–628.
[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Trans. Software Eng.*, 2014.
[16] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake, "Similarity-based prioritization in software product-line testing," in *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, 2014, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/2648511.2648532
[17] J. Guo, J. H. Liang, K. Shi, D. Yang, J. Zhang, K. Czarnecki, V. Ganesh, and H. Yu, "SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines," *Software & Systems Modeling*, Jul 2017. [Online]. Available: https://doi.org/10.1007/s10270-017-0610-0
[18] A. S. Sayyad, T. Menzies, and H. Ammar, "On the value of user preferences in search-based software engineering: a case study in software product lines," in *ICSE'13*, 2013, pp. 492–501.
[19] C. Henard, M. Papadakis, M. Harman, and Y. Le Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 517–528. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818819
[20] E. Cantú-Paz, "On random numbers and the performance of genetic algorithms," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 311–318. [Online]. Available: http://dl.acm.org/citation.cfm?id=2955491.2955546
[21] H. Maaranen, K. Miettinen, and M. M. Mäkelä, "Quasi-random initial population for genetic algorithms," *Comput. Math. Appl.*, vol. 47, no. 12, pp. 1885–1895, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.camwa.2003.07.011
[22] A. de Perthuis de Laillevault, B. Doerr, and C. Doerr, "Money for nothing: Speeding up evolutionary algorithms through better initialization," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '15. New York, NY, USA: ACM, 2015, pp. 815–822. [Online]. Available: http://doi.acm.org/10.1145/2739480.2754760
[23] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 421–432.
[24] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *ICSE'16*.
[25] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack," *Empirical Software Engineering*, Jul 2018. [Online]. Available: https://doi.org/10.1007/s10664-018-9635-4
[26] A. Arcuri and L. Briand, "Formal analysis of the probability of interaction fault detection using random testing," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1088–1099, Sept 2012.
[27] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable and nearly uniform generator of sat witnesses," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 608–623.
[28] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform SAT witness generation," in *Tools and Algorithms for the Construction and Analysis of Systems TACAS'15 2015, London, UK, April 11-18, 2015. Proceedings*, 2015, pp. 304–319.
[29] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of SAT solutions for testing," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 549–559. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180248
[30] S. Krieter, T. Thüm, S. Schulze, R. Schröter, and G. Saake, "Propagating configuration decisions with modal implication graphs," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 898–909. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180159
[31] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer, "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, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 291–302. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106252
[32] M. Raible, *The JHipster mini-book*. C4Media, 2015.
[33] W. Wei, J. Erenrich, and B. Selman, "Towards efficient sampling: Exploiting random walk strategies," in *AAAI*, vol. 4, 2004, pp. 670–676.
[34] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 2007, pp. 258–265.
[35] S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman, "Embed and project: Discrete sampling with universal hashing," in *Advances in Neural Information Processing Systems*, 2013, pp. 2085–2093.
[36] S. Ermon, C. P. Gomes, and B. Selman, "Uniform solution sampling using a constraint solver as an oracle," *arXiv preprint arXiv:1210.4861*, 2012.
[37] A. Ivrii, S. Malik, K. S. Meel, and M. Y. Vardi, "On computing minimal independent support and its applications to sampling and counting," *Constraints*, vol. 21, no. 1, pp. 41–58, 2016.
[38] M. Thurley, "sharpsat–counting models with advanced component caching and implicit bcp," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2006, pp. 424–429.
[39] ——, "sharpsat – counting models with advanced component caching and implicit bcp," in *Theory and Applications of Satisfiability Testing - SAT 2006*, A. Biere and C. P. Gomes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 424–429.

[40] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer, "A classification of product sampling for software product lines," in *Proceeedings of the 22nd International Conference on Systems and Software Product Line - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, 2018, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/3233027.3233035

[41] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Balancing scalability and uniformity in sat witness generator," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 60:1–60:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593097

[42] V. Gogate and R. Dechter, "A new algorithm for sampling csp solutions uniformly at random," in *Principles and Practice of Constraint Programming - CP 2006*, F. Benhamou, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 711–715.

[43] M. Cashman, M. B. Cohen, P. Ranjan, and R. W. Cottingham, "Navigating the maze: the impact of configurability in bioinformatics software," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 757–767. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240466