Search-based diverse sampling from real-world software product lines
Xiang, Yi; Huang, Han; Zhou, Yuren; Li, Sizhe; Luo, Chuan; Lin, Qingwei; Li, Miqing; Yang, Xiaowei

DOI: 10.1145/3510003.3510053

License:
Other (please specify with Rights Statement)

Document Version
Peer reviewed version

Citation for published version (Harvard):

Link to publication on Research at Birmingham portal

Publisher Rights Statement:
© 2022 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.

General rights
Unless a licence is specified above, all rights (including copyright and moral rights) in this document are retained by the authors and/or the copyright holders. The express permission of the copyright holder must be obtained for any use of this material other than for purposes permitted by law.

• Users may freely distribute the URL that is used to identify this publication.
• Users may download and/or print one copy of the publication from the University of Birmingham research portal for the purpose of private study or non-commercial research.
• Users may use extracts from the document in line with the concept of 'fair dealing' under the Copyright, Designs and Patents Act 1988 (?)
• Users may not further distribute the material nor use it for the purposes of commercial gain.

Where a licence is displayed above, please note the terms and conditions of the licence govern your use of this document.

When citing, please reference the published version.

Take down policy
While the University of Birmingham exercises care and attention in making items available there are rare occasions when an item has been uploaded in error or has been deemed to be commercially or otherwise sensitive.

If you believe that this is the case for this document, please contact UBIRA@lists.bham.ac.uk providing details and we will remove access to the work immediately and investigate.

Download date: 01. Dec. 2022
Search-based Diverse Sampling from Real-world Software Product Lines

Yi Xiang
xiangyi@scut.edu.cn
South China University of Technology
Guangzhou, China

Han Huang
hhan@scut.edu.cn
South China University of Technology
Guangzhou, China

Yuren Zhou
hhan@scut.edu.cn
School of Computer Science and Engineering, Sun Yat-Sen University
Guangzhou, China

Sizhe Li
South China University of Technology
Guangzhou, China

Chuan Luo
Microsoft Research
Beijing, China

Qingwei Lin
Microsoft Research
Beijing, China

Miqing Li
University of Birmingham
Birmingham, UK

Xiaowei Yang
South China University of Technology
Guangzhou, China

ABSTRACT
Real-world software product lines (SPLs) often encompass enormous valid configurations that are impossible to enumerate. To understand properties of the space formed by all valid configurations, a feasible way is to select a small, valid and representative sample set. Even though a number of sampling strategies have been proposed, they either fail to produce diverse samples with respect to the number of selected features (an important property to characterize behaviors of configurations), or achieve diverse sampling but with limited scalability (the handleable configuration space size is limited to $10^{13}$). To resolve this dilemma, we propose a scalable diverse sampling strategy, which uses a distance metric in combination with the novelty search algorithm to produce diverse samples in an incremental way. The distance metric is carefully designed to measure similarities between configurations, and further diversity of a sample set. The novelty search incrementally improves diversity of samples through the search for novel configurations. We evaluate our sampling algorithm on 39 real-world SPLs. It is able to generate the required number of samples for all the SPLs, including those which can not be counted by sharpSAT, a state-of-the-art model counting solver. Moreover, it performs better than or at least competitively to some state-of-the-art samplers with respect to the diversity of the sample sets. Our results suggest that only the proposed sampler (among all tested ones) achieves scalable diverse sampling.

KEYWORDS
Software product lines, diverse sampling, novelty search, distance metric

1 INTRODUCTION
Software product lines (SPLs) [11], being highly configurable, allow users to derive products by selecting and deselecting features, which are increments of product functionality. That is, a set of features defines a unique product (or configuration) of an SPL. Clearly, as the number of features increases, the number of all possible configurations grows exponentially [44]. A common tool for representing all valid configurations is a tree-like structure, called a feature model (FM) [34], in which features and constraints among them are explicitly specified. The space formed by all valid configurations is called a configuration space, denoted as $\Psi$ henceforth.

In many software engineering tasks, it is important to understand properties of configuration spaces. These tasks include, but not limited to, finding optimal solutions given user-specified objectives and constraints [23, 24, 26, 30, 52, 60], predicting performance of any configuration with a learned model [33, 51, 53], finding bugs caused by feature combinations [13, 25, 40, 41]. Ideally, one would like to investigate every valid configuration, but this is rarely possible in practice due to the sheer size of configuration spaces [33]. For example, large real-world SPLs may encompass (hundreds of) thousands of features, leading to enormous number of valid configurations ($\gg 10^{10}$) [46]. This definitely makes exhaustive enumeration impractical.

An intuitive way of handling the above situation is to select a small, valid and representative sample set from the configuration space. In this paper, a sample set is a collection of samples, i.e., valid configurations of an SPL. Quite often, sample sets must be well-chosen based on domain knowledge. For instance, a sample set should cover all $t$-wise feature combinations in the context of $t$-wise sampling [2, 19, 32]. If no domain knowledge is available, however, sample sets are expected to cover the configuration space as widely and uniformly as possible [33, 46]. There have been several sampling strategies in the literature, e.g., Random sampling [20, 22, 39, 41, 42], Solver-based sampling [10, 18, 26, 28], Coverage-oriented sampling [2, 19, 32] and Uniform sampling [1, 9, 43, 45, 46, 51, 54]. These sampling strategies
focus on different aspects of sampling from SPLs, and come with different strengths and weaknesses (detailed discussions are available in Section 6).

In this paper, we focus on another kind of sampling, known as diverse sampling, which seems to be largely ignored. Diverse sampling brings lots of benefits. For example, it could reduce the risk of missing important configurations with distinct performance behavior when deriving a performance prediction model [33], and forms a scalable and flexible alternative to t-wise sampling [27, 57]. Recently, Kaltenecker et al. [33] proposed a diverse sampling strategy, called diversified distance-based sampling (DDbS). The key idea is to use a distance metric in combination with a discrete probability distribution to spread the sample set across the configuration space as diversely as possible. More specifically, each configuration is assigned with a distance value (i.e., the number of selected features [33]), and DDbS tries to derive samples covering distinct distance values so as to improve diversity.

In fact, the number of selected features for a configuration \( c \in \Psi \), denoted as \( T(c) \), is important to characterize the behavior of this configuration. For example, in the context of configuring SPLs [26, 30, 52, 60], this number (also known as richness of features) is often used as an optimization objective, and it is also highly related to other objectives, e.g., the total cost of a configuration. Moreover, according to Kaltenenecker et al. [33], improving a sample set’s representativeness with respect to the number of selected features is of importance to learn an accurate performance prediction model. Therefore, sampling configurations that are diverse regarding the number of selected features is meaningful, yet poses challenges to state-of-the-art samplers. Figs. 1 (a) and (b) show the distribution of the number of selected features for samples generated by two recent uniform samplers (i.e., Smarch [46] and Unigen3 [54]) on the HiPAcc feature model [33]. As seen, compared with DDbS [33] [see Fig. 1 (c)], both of them are unable to produce diverse samples concerning the number of selected features. Even though DDbS could sample more diverse configurations on this model, it faces the scalability issue. According to the results in [46], DDbS failed to handle configuration space larger than 10^{13}. This is also confirmed by our experiments performed in Section 5.3, and explained later in Section 5.4.

To achieve scalable diverse sampling, this paper provides an alternative perspective, i.e., search-based sampling. The key idea is to generate initial samples using efficient off-the-shelf SAT solvers, and then incrementally improve the diversity of the sample set using a specific search algorithm. This sampling strategy relies on a special distance metric and a search technique, called novelty search (NS) [36, 37]. We name this sampling algorithm NS-based sampling (NSbS for short). To demonstrate merits of NSbS, we compare it with several state-of-the-art sampling algorithms using 39 real-world SPLs adopted by Oh et al. [46]. Experimental results reveal that NSbS indeed enables a scalable diverse sampling from SPLs. In particular, it successfully generates the requested number of configurations (i.e., 100 configurations in our setting) for all 39 SPLs, including the largest ones on which most state-of-the-art samplers fail to generate even one configuration within an hour.

Main contributions of the paper are summarized as follows.

- A tailored distance metric. By using the number of selected features, the configuration space \( \Psi \) is mapped to a small behavior space \( \mathcal{B} = \{ T(c) | c \in \Psi \} \). A distance metric is designed to measure similarities between configurations in both \( \mathcal{B} \) and \( \Psi \). We show, both theoretically and experimentally, that using this distance metric not only improves the coverage in the behavior space, but also promotes diversity in the original configuration space. Considering diversity in both spaces could improve the representativeness of sample sets.

- A befitting search technique. We choose NS as the search engine because of its good theoretical properties [16, 17] that well fit the goal of diverse sampling. Precisely, NS has been shown to tend towards a diverse sampling of the behavior space [16, 17]. The above property could help to improve diversity of the sample set in the behavior space. As shown in Fig. 1 (d), samples generated by NSbS are as diverse as those of DDbS. As mentioned early, DDbS is a tailored diverse sampler.

- Flexibility in the sampling process. Since diversity is improved in an incremental way, it is easy for users to achieve a desired trade-off between diversity and efficiency. If a higher-quality sample set is required, then more execution time can be specified. The flexibility in the sampling process is one of the main advantages of NSbS over other state-of-the-art samplers, most of which are not controllable regarding the execution time.

![Figure 1: Distribution of the number of selected features for samples generated by four samplers on HiPAcc [33]. In this figure, * denotes estimated boundaries of the number of selected features (see Section 2.2). Note that these boundaries may not be reachable.](image-url)
2 PRELIMINARIES

In this section, we provide necessary preliminaries on sampling from configuration spaces, as well as a space mapping strategy. Afterwards, we give a brief introduction to NS.

2.1 Sampling from configuration spaces

Formally, an FM can be seen as a tuple \((\mathcal{F}, \mathcal{C})\), where \(\mathcal{F} = \{f_1, \ldots, f_n\}\) is the set of \(n\) features, and \(\mathcal{C}\) is the set of all constraints among features. A configuration \(c\), represented by \(\{\pm f_1, \ldots, \pm f_n\}\), is defined as a set of selected or deselected features. Precisely, \(+f_i\) and \(-f_i\) indicate that the feature \(f_i\) is selected and deselected, respectively. Certainly, \(c\) can also be represented by a binary string, with 1 indicating a selected feature, and 0 a deselected one. Due to constraints in \(\mathcal{C}\), not all configurations are valid. The configuration that satisfies all the constraints is called a valid configuration, and all valid configurations form the configuration space \(\Psi\).

The size of \(\Psi\) is usually denoted by \(|\Psi|\).

Many software engineering tasks require to derive a small and representative sample set from \(\Psi\). In this context, a sample set, \(S = \{s_1, \ldots, s_N\}\) (where \(N\) denotes the sample size), is a subset of \(\Psi\), i.e., \(S \subseteq \Psi\). Each \(s_i\) \((i = 1, 2, \ldots, N)\) is called a sample, i.e., a valid configuration. Manually deriving a sample is error-prone and time-consuming even for tiny FMs [6]. Therefore, automated solvers, like SAT solvers, have been widely adopted to generate samples from \(\Psi\) [27, 38, 60]. It is well-known that an FM can be easily converted into a propositional formula \(\phi\) [6]. The derived \(\phi\) is then used as the input of automated solvers, which are internally run to find solutions to \(\phi\).

2.2 Mapping to behavior spaces

As mentioned previously, by characterizing the behavior a configuration using the number of selected features, \(c \in \Psi\) is mapped to an integer. Accordingly, the configuration space \(\Psi\) is mapped into the behavior space \(\mathcal{B}\). Let \(\Phi_b\) be the space formed by all configurations with exactly \(b\) \(\in \mathcal{B}\) selected features, then \(\Psi = \cup_{b \in \mathcal{B}} \Phi_b\). That is to say, the whole configuration space \(\Psi\) is decomposed into \(|\mathcal{B}|\) subspaces. We can then sample representative configurations that are diversely distributed among these subspaces.

Understanding \(\mathcal{B}\) is much easier than \(\Psi\) due to that \(|\mathcal{B}|\) is significantly smaller than \(|\Psi|\). In fact, the lower bound and the upper bound for \(\mathcal{B}\) can be approximated by using the number of core features (denoted by \(|\text{core}|\)) and the number of dead features (denoted by \(|\text{dead}|\)), respectively. Notice that core features must be selected in every valid configuration, while dead features must not be selected. To be more specific, \(\min(\mathcal{B}) = |\text{core}|\), and \(\max(\mathcal{B}) = n - |\text{dead}|\), where \(n\) is the total number of features. Therefore, the size of \(\mathcal{B}\) is at most \(n - (|\text{core}| + |\text{dead}|) + 1\). In contrast, \(|\Psi|\) grows exponentially with respect to \(n\). Hence, \(\Psi\) can be astronomically large, especially for large real-world SPLs. For example, \(|\Psi|\) is as large as \(7.78 \times 10^{417}\) for the uClinux-config model [46]. We must mention that knowing exactly \(\mathcal{B}\) is as hard as knowing \(\Psi\) because every configuration should be investigated at the worst case. However, \(\mathcal{B}\) can be well approximated by the following set, \(\mathcal{B}' = \{[\text{core}], [\text{core}] + 1, \ldots, n - |\text{dead}|\}\), which contains all possible integers from \([\text{core}]\) to \(n - |\text{dead}|\). It is possible that there exist some integers to which no configurations are mapped. Therefore, \(\mathcal{B}\) is a subset of \(\mathcal{B}'\). In Section 4.3, \(\mathcal{B}'\) will be used to calculate performance indicators.

2.3 Novelty search

As mentioned in Section 1, NS [36, 37] is adopted in our search-based diverse sampling. Therefore, it is necessary to give a brief introduction to this search technique. NS is one of the main divergent search algorithms [36], and its prominent feature is to abandon objectives [36]; it replaces the conventional goal-oriented objective by a criterion measuring novelty of individuals. This criterion is referred to as novelty score, defined as the average distance of an individual to its \(k\) closest neighbors. Formally, \(\rho(x)\), the novelty score of \(x\), is given as follows [16, 36, 37],

\[
\rho(x) = \frac{1}{k} \sum_{j=1}^{k} d(x, x_j) \tag{1}
\]

where \(x_j = \text{the } j\text{-th nearest neighbor of } x\). The \(\rho(x)\) estimates the sparseness of \(x\) in the behavior space. If this score is large, then \(x\) is in a sparse area; in contrast, it is in a dense area in case that the novelty score is small. In general, individuals in sparse regions are preferred to those in dense regions as the exploitation around sparse regions is helpful to perform a diverse exploration of the behavior space [16].

Following the practice in [57], the calculation of the novelty score can be extended from a single configuration to a sample set \(S = \{s_1, \ldots, s_N\}\). Specifically,

\[
\rho(S) = \sum_{i=1}^{N} \rho(s_i) \tag{2}
\]

where \(\rho(s_i)\) is the novelty score of a single configuration, as given by Eq. (1). Clearly, the higher the novelty score, the more diverse the sample set.

Finally, it is worth mentioning that there is an interesting search behavior of NS. That is, the sampling produced by NS covers the whole reachable behavior space [16]. This suggests that NS explores the behavior space diversely. It is a good property, which well matches the goal of diverse sampling from SPLs. Therefore, we choose NS as the search engine.

3 NS-BASED DIVERSE SAMPLING

The NSbS procedure is outlined in Algorithm 1. The key idea is to continuously improve diversity of the initial sample set through the search for novel individuals (or configurations). The algorithm takes the propositional formula \(\phi\) (derived from a given FM) and sample size \(N\) as input, and outputs a set of samples stored in an archive \(A\).
Algorithm 1: NSbS algorithm

Input: $\phi$ (propositional formula), $N$ (sample size)
Output: $A$ (archived samples)
1. Initialize the archive $A$ by generating $N$ solutions to $\phi$ using the randomized SAT4J solver [27];
2. Initialize the distance matrix $D = (d_{ij})_{(N+1) \times (N+1)}$, where $d_{ij}$ ($i, j = 1, \ldots, N$), as given in Eq. (3), is the distance between $x_i \in A$ and $x_j \in A$;
3. For each $x \in A$, calculate its novelty score $\rho(x)$ based on Eq. (1);
4. while the termination conditions is not met do
   5.   $\{p_1, p_2\} \leftarrow$ matingSelection($A$);
   6.   $\{c_1, c_2\} \leftarrow$ crossover($p_1, p_2$);
   7.   for $i \in \{1, 2\}$ do
       8.     $c_i \leftarrow$ mutation($c_i$);
       9.     if $c_i$ is invalid then
           10.   Repair $c_i$ using the probSAT solver [5];
       11. end
   12. $A \leftarrow$ updateAchive($A, c_i$);
13. end
14. end
15. return $A$

3.1 Initialization
As shown in Line 1 of Algorithm 1, $A$ is initialized with $N$ configurations generated by the randomized SAT4J solver [7], in which the order how the logical clauses and the literals are parsed is randomized [26]. According to the implementation in [26, 28], there exist three parsing strategies, i.e., NegativeLiteralSelectionStrategy, PositiveLiteralSelectionStrategy and RandomLiteralSelectionStrategy. Each strategy has an equal chance of being chosen when generating initial configurations. In particular, the first strategy prefers negative assignments to literals, and thus emphasizes configurations with less selected features. Therefore, the lower bound of $B$ can be approximated by using this strategy. Similarly, the second strategy helps to approximate the upper bound of $B$. The third strategy randomly assigning true or false to literals is able to randomize the generated configurations. Using simultaneously three strategies aims at improving diversity of the initial sample set. In particular, bounds of $B$ could be well approximated.

3.2 Distance metric
To measure similarities between two configurations, we define the following distance$^1$:

$$d_{ij} = d(x_i, x_j) = \frac{1}{2} \left( \frac{\text{abs}(T(x_i) - T(x_j))}{n} + \frac{1}{2} \left(1 - \frac{|x_i \cap x_j|}{n}\right) \cdot \delta \right)$$

where $x_i, x_j \in A$ ($x_i \neq x_j$) are two different configurations; $T(x_i)$ denotes the number of selected features in $x_i$; $\text{abs}(\cdot)$ returns the absolute value of a number; and $|\cdot|$ returns the cardinality of a set. The $\delta$ is a constant, as given below.

$$\delta = \begin{cases} \max \left( \frac{T(x_i) + T(x_j)}{n} \right) & \text{if } T(x_i) + T(x_j) \leq n, \\ 0 & \text{otherwise}. \end{cases}$$

As seen, this distance metric consists of two weighted parts. The first part measures the similarity between configurations in the behavior space, while the second part in the original configuration space. In fact, $1 - \frac{|x_i \cap x_j|}{n}$ is the Hamming distance [3] between $x_i$ and $x_j$. Note that using the above two parts is intended to sample configurations covering diversely in the behavior space, and also keeping as dissimilar as possible in the configuration space. In Section 5.1, we will experimentally verify this distance measure.

It is also worth noting that $\delta$ is set based on our theoretical analysis, which is presented in Section S-1 of the online supplement. Our theoretical analysis shows that $\delta$ is needed to mitigate biases towards sampling specific configurations. In other words, Eq. (3) without using $\delta$ can introduce biases in the behavior space, and thus can hamper diversity of the sample set. More detailed discussions can be found in Section S-1 of the supplement. In Section 5.2, we will experimentally investigate $\delta$’s effects. Therein, one will find that using $\delta$ indeed improves diversity of the sample set in the behavior space.

According to Line 2 in Algorithm 1, the distance matrix $D = (d_{ij})_{(N+1) \times (N+1)}$ is initialized by working out the distance between each pair of configurations in $A$. Note that, since $D$ is symmetric, we only need to calculate distances for half of these pairs. We would like to mention that the size of $D$ is $(N + 1) \times (N + 1)$, rather than $N \times N$, because we reserve spaces for storing distances when evaluating a new configuration (see Algorithm 2). After obtaining $D$, as shown in Line 3 of Algorithm 1, the novelty score for each $x \in A$ is calculated based on Eq. (1).

3.3 Genetic operations
Like in genetic algorithms, we perform in order the mating selection, crossover and mutation to generate new individuals. As shown in Line 5 of Algorithm 1, the matingSelection procedure chooses from $A$ two parents $p_1$ and $p_2$ each time. The basic idea of choosing a parent is to select the one with larger novelty score from two different random members in $A$. In case of a tie, a random selection is performed between the two members. Clearly, the above mating selection emphasizes individuals located in sparse regions. Exploration around sparse regions could potentially improve diversity of the samples.

Once two parents $p_1$ and $p_2$ have been selected, the uniform crossover is applied to generate two children, $c_1$ and $c_2$ (Line 6 in Algorithm 1). To be specific, for each index $j \in$...

---

$^1$The online supplement (entitled “OnlineSupplementICSE2022.pdf”) is available at an anonymous website http://doi.org/10.5281/zenodo.4939625
Algorithm 2: $A \leftarrow updateArchive(A, c)$

Input: $A, c$
Output: $A$
1. if $A$ contains $c$ then
2. \hspace{1em} return $A$;
3. end
4. for $i = 1, \ldots, N$ do
5. \hspace{1em} $d_{i,(N+1)} \leftarrow d(x_i, c)$;
6. end
7. $d(N+1,i) \leftarrow d_{i,(N+1)}$;
8. $d_{(N+1),(N+1)} \leftarrow 0$;
9. For each $x \in A \cup c$, calculate its novelty score $\rho(x)$
   based on Eq. (1):
10. \hspace{1em} $x_{\text{worst}} \leftarrow \arg \min_{x \in A} \rho(x) // \text{Find the worst member in } A$;
11. \hspace{1em} if $\rho(c) > \rho(x_{\text{worst}})$ then
12. \hspace{2em} $x_{\text{worst}} \leftarrow c$; \hspace{1em} // Update $D$
13. \hspace{1em} for $j = 1, \ldots, N$ do
14. \hspace{2em} $d_{i,\text{worst}} \leftarrow d_{i,(N+1)}$;
15. \hspace{2em} $d_{\text{worst},j} \leftarrow d_{(N+1),j}$;
16. end
17. \hspace{1em} $d_{\text{worst},\text{worst}} \leftarrow 0$;
18. end
19. return $A$

\{1, \ldots, n\}, we generate a random number $\text{rand}$. If $\text{rand} < 0.5$, then $c_1(j)$ and $c_2(j)$ are set to $p_1(j)$ and $p_2(j)$, respectively. Otherwise, they are set to $p_2(j)$ and $p_1(j)$, respectively. Notice that $c_1(j)$ denotes the value taken in the $j$-th position of $c$. The newly generated individuals are then subjected to bit-wise mutation (Line 8 in Algorithm 1). Specifically, for each bit, the value is changed from 1 (true) to 0 (false), or vice versa. Often, the ratio of bits to be changed is controlled by a parameter $P_n$, called mutation probability. In this work, we set $P_n$ to 0.1, following the common practice in [58].

It is not uncommon that the resulting configurations (after crossover and mutation) are invalid. In this case, as shown in Line 10 of Algorithm 1, the probSAT solver [5], one of the high-performing stochastic local search (SLS) SAT solvers, is adopted to repair invalid configurations. The variables to be flipped by the solver are chosen based on probabilities such that more promising variables are given more chances to be selected. In fact, probSAT [5] has been adopted to repair infeasible configurations in prior work [59, 60] in the context of optimal products selection from SPLs. In particular, the empirical study in [59] suggested that probSAT is more effective than WalkSAT [8], another popular SLS solver, in improving diversity of a configuration set. For more details on probSAT, we direct readers to the original study [5]. Notice that internal parameters of this solver are set following the practice in [5] and [59]. Therefore, a tuning phase is not required in this work.

Configurations operated by probSAT could still be invalid (even though they are valid most of the time)\(^\dagger\), in particular for large-scale FMs. In case of invalidity, we simply request to the randomized SAT solver, as described in Section 3.1, to return a valid configuration.

3.4 Updating archive

The archive $A$ stores novel configurations discovered during the search process. Its update procedure is presented in Algorithm 2. To improve diversity, as shown in Lines 1-3, the producer rejects the entry of any configurations that are identical to already archived ones. When a totally different configuration $c$ is available, we need to fill the distance matrix $D$ by working out distances between $c$ and each $x_i \in A$. These distances are stored in the last row and the last column. In what follows, as indicated in Line 9 of Algorithm 2, the novelty score for each member $x \in A \cup c$ is calculated based on Eq. (1). We should note here that the novelty scores are computed taking into account not only members in $A$ but also the new configuration $c$. This enables an evaluation of the novelty with respect to both previously explored individuals and the current one that represents the most recently visited point [36].

In Line 10 of Algorithm 2, we find the worst member from $A$, and this member is denoted by $x_{\text{worst}}$, where the index $\text{worst}$ is its position in the archive. In case that the novelty score of $c$ is higher than that of $x_{\text{worst}}$, we will replace $x_{\text{worst}}$ by $c$. Subsequently, the distance matrix should be updated. This is achieved by simply copying the last row (column) to the $\text{worst}$-th row (column) (see Lines 13-16). At last, $d_{\text{worst},\text{worst}}$ should be set to 0.

According to the above update procedure, the algorithm consistently looks for novel individuals, pushing individuals to constantly move in the behavior space: in new and unexplored areas first, but also then in already explored areas as their density of individuals is never exactly homogeneous [17]. This way, diversity of the samples can be persistently improved.

3.5 Termination conditions

Termination of NSbS can be flexibly specified by users. We offer the following two termination strategies.

**Strategy 1**: Termination controlled by the maximum running time ($\text{max.t}$). This is a common way of stopping a search algorithm, and the setting of $\text{max.t}$ depends largely on the demands of users.

**Strategy 2**: Automatic termination when the algorithm gets relatively steady. This is achieved by adding the following piece of codes after Line 13 in Algorithm 1.

\[
\text{If } \frac{\left| \mu(A) - \mu(A) \right|}{\mu(A)} \leq 0.1\%
\text{ \hspace{1em} counter } + +
\text{ Else \hspace{1em} counter } \leftarrow 0
\]

\(^\dagger\)Different from conflict-driven clause learning (CDCL) solvers [14], such as SAT4J, SLS-type SAT solvers offer no guarantees on finding valid assignments.
where $\mathcal{A}$ is the old archive, while $\mathcal{A}$ is the newly updated one. If the change ratio of novelty scores for the two archives is below 0.1%, then counter is increased by one; otherwise, it is reset to 0. The algorithm will terminate once counter exceeds $R$, a threshold specified by users. In our experiments, we set $R$ to 10. With this setting, it is found that NSbS automatically terminates on almost all of the tested FMs.

We would like to mention that we give users freedom to specify the termination of the sampling process. Most of the state-of-the-art samplers are not controllable with respect to the execution time. Indeed, it may take excessively long before a set of samples is returned [46, 49]. Instead, the proposed NSbS allows users to make a desired trade-off between quality (primarily diversity) and efficiency. If users want a higher-quality sample set, he/she can set $R$ or max$\mathcal{J}$ to a relatively larger value. The flexibility regarding terminations is one of the main advantages of NSbS over other state-of-the-art samplers.

4 EXPERIMENT SETUP

In this section, we start by introducing our research questions (RQs). Then, we give information about FMs used in our empirical study. Subsequently, we describe how the performance of different samplers can be measured using specialized indicators. Finally, detailed implementations are given.

4.1 Research Questions

The distance metric is expected to play an important role in sampling products that are diverse not only in the behavior space, but also in the original configuration space. It is necessary to investigate the effect of each component in Eq. (3). With this regard, we aim at answering the following two RQs.

RQ1: What are the benefits brought by using the two weighted parts in the distance metric?

RQ2: Does the factor $\delta$ matter in the distance metric of NSbS?

To address RQ1, we experimentally compare the distance metric defined in Eq. (3) against a modified one in which the second part is removed. We expect that the adoption of the two parts can promote diversity of the sampled configurations in both spaces. According to our theoretical analysis, the goal of $\delta$ in Eq. (3) is to alleviate biases towards specific configurations in the behavior space. The second research question amounts to experimentally verifying this.

Moreover, we intend to answer one more research question regarding the effectiveness of NSbS in comparison with several state-of-the-art samplers.

RQ3: How effective is NSbS concerning both scalability and diversity in comparisons with state-of-the-art samplers?

To address RQ3, we compare NSbS with SAT-based sampling [26], DDBsS [33], UniGen3 [54] and Smarch [46]. We expect that NSbS performs better than or at least competitively to them with respect to both scalability and diversity.

4.2 Subject Feature Models

In our experiments, we consider 39 FMs that have been carefully selected by Oh et al. [46] in their evaluation of Smarch. Table S-1 in the online supplement gives an overview of the subject FMs, including the number of features ($|\mathcal{F}|$), the number of CNF constraints ($|\mathcal{C}|$), the size of the configuration space ($|\mathcal{S}|$), the number of core, dead and unconstrained features (i.e., $|\text{core}|$, $|\text{dead}|$ and $|\text{uc}|$). Note that $|\mathcal{S}|$ is counted by sharpSAT [56], which fails on the last five largest FMs. All FMs are publicly available in DIMACS format, the standard format for SAT solvers.

4.3 Performance indicators

Performance indicators are required to evaluate the quality of a sample set $S = \{s_1, \ldots, s_N\}$. To measure whether $S$ widely covers the behavior space $B = \{b_1, \ldots, b_{|B|}\}$, the following indicator, which we call Spread, is defined.

$$\text{Spread}(S, B) = \frac{1}{|B|} \sum_{i=1}^{|S|} d_{\text{min}}(b_i, S),$$

where $d_{\text{min}}(b_i, S)$ denotes the minimum distance from $b_i$ to $S$. Mathematically, $d_{\text{min}}(b_i, S)$ is in the following form

$$d_{\text{min}}(b_i, S) = \min_{j=1}^N \text{abs}(b_i - T(s_j)).$$

Regarding $B$, as mentioned in Section 2.1, it is not exactly known, but can be easily approximated by $B'$. In practice, we therefore calculate $\text{Spread}(S, B')$, instead of $\text{Spread}(S, B)$. It is clear that a smaller value of Spread indicates a more diverse distribution in the behavior space.

In addition to Spread, the novelty score of a sample set, $\rho(S)$, as given in Eq. (2), also serves as a performance indicator. It measures the diversity of $S$ in the original configuration space.

4.4 Detailed Implementations

For each FM, we sample 100 configurations and compute the average sampling time per configuration (measured in milliseconds) to compare efficiency. All samplers except NSbS terminate once 100 configurations are sampled, or the sampling time takes more than 3600,000 milliseconds (i.e., one hour). Since NSbS is able to quickly sample 100 configurations, it terminates either automatically based on Strategy 2 as described in Section 3.5, or forcibly when the sampling time reaches a timeout of one hour. Note that, to mitigate random bias, all samplers are independently run 30 times, and we present and analyze experimental results regarding mean values of the performance indicators.

All experiments are performed on a Quad Core@2.20 GHz with 8 GB of RAM running Ubuntu 20.04.2. Source codes of SAT-based sampling [26], DDBsS [33], UniGen3 [54] and Smarch [46] are downloaded from their authors’ repositories, and they are all executed on a single thread (i.e., without

1Unconstrained features here are those that are not involved in the CNF constraints.

2All FMs are downloaded from https://github.com/jeho-oh/Smarch
parallelization), following the practice in [29]. The codes for NSbS can be found in our repository\textsuperscript{6}.

5 RESULTS

In this section, we provide a series of experimental results regarding the research questions.

5.1 RQ1: Benefits brought by using the two weighted parts in the distance metric

To investigate benefits brought by using two weighted parts in Eq.(3), we consider two different distance metrics. The first one, as given in Eq.(3), uses two weighted parts. Hereafter, we call it weighted distance. The second one retains only the first part, and therefore measures only the similarity in the behavior space. This metric is called unweighted distance. Both distance metrics are tested within the same NS framework in which \( k = 15 \). This setting of \( k \) has been widely employed in NS-related literature [21]. Our tuning experiments presented in Section S-2 of the online supplement suggest that \( k = 15 \) is also a good setting in our context. When the weighted distance is used, the sampling algorithm automatically terminates according to Strategy 2. In the case of unweighted distance, the termination is controlled by Strategy 1, in which \( \text{max}_t \) is set to the running time consumed by the corresponding algorithm using the weighted distance. This setting allows us to investigate the benefits while eliminating potential impacts brought by using different running times.

Means of Spread and novelty score obtained by using the two distance metrics are provided in Table S-2 of the online supplement. To determine whether the difference (over all the 30 runs) is significant or not, following guidelines suggested by Arcuri and Briand [4], the Mann-Whitney U test with a 0.05 significance level is performed for each FM. Test results are represented by three symbols: \(*\), \( \dagger\) and \( \circ\) indicating that the weighted distance performs better than, equivalently to and worse than the unweighted distance, respectively. Regarding Spread, the weighted distance performs significantly better than its counterpart on 3 out of all the 39 FMs, but worse on only one FM, i.e., the simplest LLVM. For all the remaining 35 FMs, the two distance metrics have similar performance. The above results suggest that using alone the first part of Eq. (3) (which measures similarities in the behavior space) is enough to obtain good spread in the behavior space in the majority of the cases.

Regarding the novelty score, the weighted distance shows significant improvements over the unweighted one on 26 out of 39 FMs (67%), and degenerations on only lrzip and 2.6.33.3-2var. Clearly, the second part of Eq. (3) is necessary in promoting diversity in the original configuration space.

Therefore, the answer to RQ1 is clear. Using the two weighted parts in Eq. (3) indeed brings benefits: it improves coverage in the behavior space; at the same time, it also promotes diversity in the original configuration space. Boosting diversity in both spaces is beneficial for enhancing representativeness of the sample set.

5.2 RQ2: \( \delta \) matters in the distance metric

According to Section 3.1, \( \delta \) in Eq. (3) is used to mitigate potential biases towards specific configurations. In this section, we are going to experimentally examine the effects of this factor. To this end, we compare NSbS (described in Algorithm 1) against its variant, i.e., NSbS-\( \delta \) in which \( \delta \) is omitted in the distance metric. The only difference between the two algorithms is the presence or absence of \( \delta \).

![Figure 2: Configurations sampled by NSbS widely cover the behavior space, while those sampled by NSbS-\( \delta \) fail. (a) ref4955, (b) 2.6.28.6-icse11](image)

Table S-3 in the supplement gives Spread results of the two algorithms. As shown, NSbS performs better than or at least comparably to NSbS-\( \delta \) on all the 39 FMs. In particular, NSbS significantly outperforms its counterpart on \( 25/39 = 64\% \) of the FMs. Moreover, the improvements are mostly observed on large FMs. Taking ref4955 and 2.6.28.6-icse11 as examples, Fig. 2 graphically shows the distribution of the sampled configurations in the behavior space. As seen, configurations sampled by NSbS are distributed more widely than those sampled by NSbS-\( \delta \) on the two FMs. More specifically, configurations of NSbS-\( \delta \) cover intensively in the middle part, but sparsely at the boundaries.

The above experimental results bring out the following. The \( \delta \) indeed matters: when \( \delta \) is omitted, diversity of the sampled configurations is significantly affected in the behavior space. In particular, boundaries of the behavior space tend not to be sufficiently covered. The above experimental results are in line with our theoretical findings, stating that the adoption of \( \delta \) is able to mitigate bias towards sampling specific configurations.

5.3 RQ3: Effectiveness of NSbS in comparison with state-of-the-art samplers

Table 1 gives the average time (measured in milliseconds) to sample a single configuration for all FMs. If the sampling cannot finish within one hour, then we declare a timeout. According to Table 1, SAT-based simpler scales very well, being able to sample one configuration within 300 milliseconds
even for the largest FM, i.e., 2.6.33.3-2var. Quite often, the sampling takes no more than 1 millisecond. For DDbS, it can only handle 11 small FMs with $|\Phi| \leq 2.16 \times 10^6$. For Unigen3, it succeeds in dealing with 19 FMs with $|\Phi| \leq 1.63 \times 10^{91}$. Regarding Smarch, it scales better than DDbS and Unigen3, but still fails on 11 FMs. For our NSbS, it does not encounter a timeout for all FMs. We would like to mention that even though NSbS runs out of one hour on 2.6.32-2var and 2.6.33.3-2var, it successfully samples 100 configurations as requested. Therefore, we do not declare a timeout. In fact, this is totally different from the timeout of other samplers, which are unable to sample 100 configurations within one hour. Regarding the sampling speed, as shown in Table 1, NSbS is slower than the SAT-based sampler, but much faster than Smarch.

Table S-4 in the online supplement lists Spread results for all samplers, and Table 2 summarizes Wilcoxon’s test results for each pairwise comparison between NSbS and each sampler. In this table, available cases refer to those without a timeout for both samplers. As can be found, NS performs better than or at least competitively to other samplers in almost all the available cases. The only exception is observed on the pairwise comparison between NSbS and DDbS on the simplest LLVM. In this case, NSbS is significantly worse than DDbS. This exceptional case accounts for 9% of all the available 11 cases for the pair NSbS v.s. DDbS. According to the results in Table 2, we can conclude that NSbS is more effective than SAT-based sampler, Unigen3 and Smarch in generating diverse configurations, and that NSbS and DDbS are able to sample configurations covering similarly in the behavior space. However, as discussed previously, DDbS suffers from the scalability issue, being only able to handle very small FMs.

Table 2: Summary of Wilcoxon’s test results (regarding Spread) for pairwise comparisons between NSbS and each sampler

<table>
<thead>
<tr>
<th>FM</th>
<th>NSbS v.s.</th>
<th>SAT-based</th>
<th>DDbS</th>
<th>Unigen3</th>
<th>Smarch</th>
</tr>
</thead>
<tbody>
<tr>
<td>leip</td>
<td>2</td>
<td>1</td>
<td>18</td>
<td>&lt;1</td>
<td>185</td>
</tr>
<tr>
<td>LLVM</td>
<td>2</td>
<td>&lt;1</td>
<td>5</td>
<td>&lt;1</td>
<td>110</td>
</tr>
<tr>
<td>X264</td>
<td>2</td>
<td>&lt;1</td>
<td>10</td>
<td>&lt;1</td>
<td>151</td>
</tr>
<tr>
<td>Sun</td>
<td>2</td>
<td>&lt;1</td>
<td>14</td>
<td>&lt;1</td>
<td>165</td>
</tr>
<tr>
<td>BerkeleyDBC</td>
<td>148</td>
<td>&lt;1</td>
<td>14</td>
<td>&lt;1</td>
<td>165</td>
</tr>
<tr>
<td>HiPacc</td>
<td>2</td>
<td>&gt;99</td>
<td>5</td>
<td>329</td>
<td></td>
</tr>
<tr>
<td>JHipster</td>
<td>10</td>
<td>&gt;100</td>
<td>1</td>
<td>161</td>
<td>388</td>
</tr>
<tr>
<td>Polly</td>
<td>92</td>
<td>&lt;1</td>
<td>400</td>
<td>1</td>
<td>373</td>
</tr>
<tr>
<td>7z</td>
<td>120</td>
<td>&lt;1</td>
<td>1299</td>
<td>3</td>
<td>467</td>
</tr>
<tr>
<td>JavaGC</td>
<td>9</td>
<td>&lt;1</td>
<td>780</td>
<td>1</td>
<td>381</td>
</tr>
<tr>
<td>VP9</td>
<td>6</td>
<td>&lt;1</td>
<td>2412</td>
<td>2</td>
<td>396</td>
</tr>
<tr>
<td>fiasco-17_10</td>
<td>12</td>
<td>&lt;1</td>
<td>15</td>
<td>timeout</td>
<td>3260</td>
</tr>
<tr>
<td>axTLS-2.1.4</td>
<td>6</td>
<td>&lt;1</td>
<td>12</td>
<td>timeout</td>
<td>895</td>
</tr>
<tr>
<td>fiasco</td>
<td>93</td>
<td>1</td>
<td>timeout</td>
<td>20</td>
<td>58003</td>
</tr>
<tr>
<td>toybox</td>
<td>13</td>
<td>&lt;1</td>
<td>timeout</td>
<td>7</td>
<td>7123</td>
</tr>
<tr>
<td>axisl</td>
<td>14</td>
<td>1</td>
<td>timeout</td>
<td>34</td>
<td>13515</td>
</tr>
<tr>
<td>uClibc-ng-1.0-29</td>
<td>7</td>
<td>&lt;1</td>
<td>timeout</td>
<td>4681</td>
<td>4239</td>
</tr>
<tr>
<td>toybox-0.7.5</td>
<td>7</td>
<td>&lt;1</td>
<td>timeout</td>
<td>231</td>
<td>3492</td>
</tr>
<tr>
<td>uClimux</td>
<td>61</td>
<td>2</td>
<td>timeout</td>
<td>258</td>
<td>39713</td>
</tr>
<tr>
<td>ref4055</td>
<td>46</td>
<td>1</td>
<td>timeout</td>
<td>37299</td>
<td></td>
</tr>
<tr>
<td>adderH</td>
<td>49</td>
<td>1</td>
<td>timeout</td>
<td>48508</td>
<td></td>
</tr>
<tr>
<td>ecos-ice11</td>
<td>40</td>
<td>1</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>m5272cli</td>
<td>45</td>
<td>1</td>
<td>timeout</td>
<td>43946</td>
<td></td>
</tr>
<tr>
<td>pati</td>
<td>46</td>
<td>1</td>
<td>timeout</td>
<td>38404</td>
<td></td>
</tr>
<tr>
<td>olpce2294</td>
<td>51</td>
<td>1</td>
<td>timeout</td>
<td>53946</td>
<td></td>
</tr>
<tr>
<td>integrator-arm0</td>
<td>65</td>
<td>2</td>
<td>timeout</td>
<td>377666</td>
<td></td>
</tr>
<tr>
<td>at91sam7eck</td>
<td>49</td>
<td>1</td>
<td>timeout</td>
<td>45776</td>
<td></td>
</tr>
<tr>
<td>se77x9</td>
<td>79</td>
<td>2</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>phycore229x</td>
<td>49</td>
<td>1</td>
<td>timeout</td>
<td>56323</td>
<td></td>
</tr>
<tr>
<td>busybox-1.18.0</td>
<td>202</td>
<td>3</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>busybox-1.28.0</td>
<td>37</td>
<td>&lt;1</td>
<td>timeout</td>
<td>18087</td>
<td></td>
</tr>
<tr>
<td>embtoolkit</td>
<td>2306</td>
<td>35</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>freedbsd-ice11</td>
<td>201</td>
<td>7</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>uClimux-config</td>
<td>296</td>
<td>7</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>buildroot</td>
<td>6916</td>
<td>21</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>freetz</td>
<td>16540</td>
<td>35</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>2.6.28.6-ice11</td>
<td>558</td>
<td>29</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>2.6.32-2var</td>
<td>36000</td>
<td>256</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>2.6.33.3-2var</td>
<td>36000</td>
<td>289</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
</tr>
</tbody>
</table>

For configurations generated by the five samplers on JHipster (chosen as an example), we plot in Fig. 3 the number of selected features, and the difference of this number between two successive configurations. Notice that configurations in these sample sets are sorted in increasing order based on the number of selected features. It can be found in Fig. 3 that the number of selected features for NSbS increases more regularly than that for other samplers. To be more specific, the increment of this number for NSbS is steady being always one, while it is either one or two for DDbS and Smarch. In addition, configurations sampled by NSbS can be partitioned into nearly equal-sized subsets based on the number of selected features. For other samplers, however, this partition is less balanced. This can be observed from histograms for the ‘difference’, indicating that some groups have more configurations than others. The above graphical results suggest that NSbS is capable of sampling configurations that are widely and nearly-uniformly distributed in the behavior space.

Experiments performed in this section emphasize the following. First, NSbS and SAT-based sampling are the two best samplers regarding scalability, and both of them can handle all FMs under study. Second, NSbS and DDbS perform best concerning diversity of the samples in the behavior space. Therefore, only NSbS (among all samplers tested in this section) achieves scalable diverse sampling.

5.4 Discussions

It is not surprising that NSbS performs better than SAT-based sampler regarding diversity. In fact, initial population
in NSbS is the outcome of the SAT-based sampler. This initial population is sequently improved by NS in an incremental way. As shown in Fig. 4, novelty scores of the sample sets are persistently improved during the sampling process of NSbS, naturally leading to more and more diverse samples.

It is also easy to explain why DDbS is computationally much more expensive than NSbS. In fact, NSbS uses genetic operations to generate temporary configurations, and then adopts probSAT [5] to repair them if necessary. This is an efficient way of creating new configurations. Instead, each time DDbS requests to the Z3 constraint solver [15] to find a configuration with exactly $d$ selected features, where $d$ is uniformly drawn from the set of all possible distances. However, it is not always easy to find such configurations because they may not exist. Sometimes the constraint solver takes long time to find a feasible configuration, or fails to return any one even after a long time of running. Hence, DDbS suffers from low efficiency.

Figure 3: For configurations generated by each sampler on JHipster, the number of selected features and the difference of this number between two successive configurations are shown in histograms.

The following is the reason why the two uniform samplers (Unigen3 and Smarch) can not generate diverse samples in the behavior space. In fact, Unigen3 and Smarch aim at deriving uniform samples in the configuration space. The uniformity in this space cannot guarantee diversity in the behavior space. Clearly, this mechanism could hamper diversity in the behavior space.

5.5 Threats to Validity

In this section, we briefly discuss threats to internal validity and external validity, as well as how they could be mitigated.

**Internal validity.** This type of threats can be caused by potential errors in our implementation of NSbS and the samplers used for comparisons. To rule out errors in the implementation, we have thoroughly tested our codes by analyzing the outcomes step by step on small FMs. For samplers used in performance comparisons, they were implemented by codes provided by their authors.

Due to stochastic nature of the samplers under study, outcomes of different runs could be different. To diminish random biases, we independently run the samplers 30 times, and compare them based on mean values of performance indicators. In addition, statistical tests are utilized to make reliable comparisons.

**External validity.** This threat is related to the degree to which we can generalize from the experiments. To increase external validity, we select 39 real-world FMs from different domains, and most of them have been widely used by others to evaluate their sampling algorithms [33, 46, 49]. These FMs are representative with respect to the configuration size, which ranges from $10^2$ to more than $10^{4.17}$. Therefore, we are confident that our results could generalize to many more FMs.

6 RELATED WORK

There are different strategies for sampling configurations from SPLs: random sampling, solver-based sampling, coverage-oriented sampling and uniform sampling.

**Random sampling:** The simplest way to create a sample set is to randomly assign true or false to each feature for each configuration [22, 39, 41, 42]. Due to constraints among features, however, this method is very likely to generate invalid configurations. Instead of randomly selecting features,
there exist sampling approaches randomly selecting configurations either from all the enumerated configuration space [51], or by using the Monte-Carlo method without exhausted enumeration [20]. Nevertheless, these approaches also select invalid configurations, or suffer from low efficiency because of the time-consuming or even impractical enumeration.

**Solver-based sampling:** Off-the-shelf SAT or satisfiability modulo theories constraint solvers have been widely used to derive samples. These solvers include SAT4J [26, 27, 38], PicoSAT [10, 50], Z3 solver [18, 23, 33, 47]) and stochastic local search SAT solvers [57, 59, 60]. This kind of sampling generally scales well to large real-world SPLs, but offers no guarantees about randomness or coverage [33, 49]. In particular, to improve the diversity of configurations, Henard et al. [26] randomized the order how the logical clauses and the literals are parsed. The resulting randomized SAT4J solver, which has been extensively adopted in different contexts [27, 28, 57], is selected as a baseline in this paper. According to our results, this solver cannot give any guarantees about coverage in the behavior space, though.

**Coverage-oriented sampling:** It creates a sample set according to a specific coverage criterion. One of the prominent example is \( t \)-wise sampling in which all possible \( t \) feature combinations must be covered [12]. Nowadays, various \( t \)-wise sampling approaches are available, e.g., Chvatál [31], ICPL [32], IncLing [2], YASA [35] and CASA [19]. Based on the evaluations in [41], however, most of the \( t \)-wise sampling techniques can only deal with small FMs considering often \( t = 1 \) or \( t = 2 \). For large real-world SPLs and/or high \( t \) interaction strengths, they often run out of memory, do not terminate, or take too much running time [48].

**Uniform sampling:** Achieving uniform sampling is important to understand properties of the whole configuration space [46]. Recently, uniform sampling has caught increasing attention from both SAT and SPL communities [1, 29, 49]. UniGen2 [9] partitions the configuration space as evenly as possible using hashing functions. Subsequently, sampling is done by choosing a partition at random, and then generating a valid configuration in that partition using an SAT solver. UniGen2 also supports parallelism on sampling, and its improved version, i.e., UniGen3 [54], is now available. Several strategies perform counting-based uniform sampling. Typically, they subsequently partition the configuration space on variable assignments, and then count the number of configurations of the resulting parts. In [43], the number of valid configurations can be easily counted since an FM is encoded as a binary decision diagram. Both \textit{Spar} [1] and \textit{Smarsh} [46] rely on sharpSAT [56] to count the number of valid configurations. The above samplers guarantee uniform sampling, but may encounter a bottleneck in some cases. According to Sundermann et al. [55], none of their evaluated model counting solvers, including sharpSAT, can count the number of valid configurations for some large industrial SPLs. Alternatively, QuickSampler [18] performs an efficient sampling of configurations using only a small number of MAX-SAT solver calls. This sampler, however, offers no guarantees on uniformity or even validity of the samples [29, 49].

7 CONCLUSIONS

This paper focuses on diverse sampling from SPLs. In practice, the number of selected features for a configuration is important to characterize its behaviors. By using this number, the configuration space is mapped to a small behavior space. Deriving a small set of valid configurations that has a good coverage in the behavior space is required in many software engineering tasks. However, most existing sampling strategies fail to achieve this goal. In this paper, we propose a search-based sampling strategy which adopts an efficient off-the-shelf SAT solver to generate an initial sample set, and then improves its diversity in an incremental way. This is achieved by using a special distance metric in combination with the novelty search algorithm. Experimental results on 39 real-world SPLs demonstrate that our sampling algorithm can not only improve coverage in the behavior space, but also promote diversity in the original configuration space. Moreover, we show, both theoretically and experimentally, that the designed distance metric is able to mitigate bias towards covering specific parts in the behavior space. Finally, our results show that only the proposed sampling algorithm achieves scalable diverse sampling among all the five evaluated samplers. In particular, our sampler scales up well, being able to handle all the SPLs, including those whose configuration space even cannot be counted.

Focusing on sampling diverse configurations from behavioral spaces, this paper provides a search-based sampler. This is a new attempt on diverse sampling from SPLs, and there are some directions for subsequent studies. For example, it is easy to adapt our sampler in different sampling scenarios using other criteria to map configuration spaces into behavioral spaces. For the generated sample sets, it is worthwhile to investigate their \( t \)-wise coverage [57] and performance prediction accuracy [33].

REFERENCES


[38] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-based Analysis of Large Real-world Feature


