Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Uniform and Scalable Sampling of Highly Configurable Systems

Ruben Heradio · David Fernandez-Amoros · José A. Galindo · David Benavides · Don Batory

Received: date / Accepted: date

Abstract Many analyses on configurable software systems are intractable when confronted with colossal and highly-constrained configuration spaces. These analyses could instead use statistical inference, where a tractable sample accurately predicts results for the entire space. To do so, the laws of statistical inference requires each member of the population to be equally likely to be included in the sample, i.e., the sampling process needs to be "uniform".

SAT-samplers have been developed to generate uniform random samples at a reasonable computational cost. However, there is a lack of experimental validation over colossal spaces to show whether the samplers indeed produce uniform samples or not. This paper (i) proposes a new sampler named BDDSampler, (ii) presents a new statistical test to verify sampler uniformity, and (iii) reports the evaluation of BDDSampler and five other state-of-the-art samplers: KUS, QuickSampler, Smarch, Spur, and Unigen2. Our experimental results show only BDDSampler satisfies both scalability and uniformity.

Keywords Uniform sampling · Configurable systems · Software product lines · Binary decision diagrams · SAT-solvers

Ruben Heradio
Universidad Nacional de Educación a Distancia, Madrid, Spain
E-mail: rheradio@issi.uned.es

David Fernandez-Amoros
Universidad Nacional de Educación a Distancia, Madrid, Spain
E-mail: david@issi.uned.es

José A. Galindo
University of Seville, Seville, Spain
E-mail: jagalindo@us.es

David Benavides
University of Seville, Seville, Spain
E-mail: benavides@us.es

Don Batory
University of Texas at Austin, Austin, Texas, U.S.A.
E-mail: batory@cs.utexas.edu

1 Introduction

Generating random SAT-solutions is of critical importance in several domains: ***Software Product Lines* (SPLs)** analysis and configuration [39,48,53], software testing [12,21,57,58], and integrated circuit simulation and verification [31,51,68].

To get a sense of this problem's relevancy and complexity, consider an example taken from the SPL domain. BusyBox[1] is a software tool that replaces many standard GNU/Linux utilities with a single small executable, thus providing an environment customized for a diversity of embedded systems. To achieve size-optimization, BusyBox is remarkably modular, supporting the inclusion/exclusion of 613 features at compile time. These features and their interrelationships are specified with a configuration language named Kconfig[2]. To guarantee that every valid configuration satisfies all dependencies, the Kconfig model of BusyBox is translated into a Boolean formula that is then processed with a logic engine [5,24] (e.g., a SAT solver [8]). A valid configuration corresponds to a satisfiable assignment of the formula, also called, a ***SAT solution*** [57] or a ***witness*** [12].

As a consequence of the inter-feature dependencies, the space of valid configurations ($7.428 \cdot 10^{146}$) is a tiny portion of the whole configuration space (2^{613}): only $2.185 \cdot 10^{-36}$% of the possible configurations are valid [29]. Nevertheless, the population of valid configurations is still colossal. Those SPL analyses that examine every valid configuration are unscalable.

For instance, Halin et al. [28] adopted an exhaustive strategy to test the JHipster[3] system, checking all its valid configurations. JHipster is a code generator for web applications with 45 selectable features that can produce a total of 26,256 valid configurations. Checking this modest configuration space with the INRIA Grid'5000[4] required 4,376 hours of CPU time ($\sim$ 182 days), and 5.2 terabytes of disk space.

Others have advocated approaching this and related problems via statistical inference [3,4,27,35,38,49,53,61,66]; that is, working with a tractable sample that predicts the results for the entire population. An essential requirement is that all samples be genuinely representative of the population [36]. In other words, each member of the population must be equally likely to be included in a sample. Authors often use the term ***uniform random sampling*** [53,57,59] for this idea.

A naive approach to get such a sample would (i) generate a random configuration set without considering feature dependencies, and then (ii) check with a logic engine if each configuration conforms to those dependencies. Unfortunately, and as mentioned above, feature dependencies shrink the configuration space extraordinarily, and so getting a single valid configuration randomly is extremely unlikely. As a result, more advanced algorithms generate valid and uniform random samples at a reasonable computational cost.

Verifying that these algorithms and their tools indeed generate genuine uniform samples is a challenge by itself, because it requires examining the consistency between sample *statistics* and their corresponding population *parameters* (e.g., how

[1] https://busybox.net/

[2] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

[3] https://www.jhipster.tech/

[4] https://www.grid5000.fr/

frequently a feature appears in a sample compared to its probability of being included in every valid configuration [30]). As configuration spaces can be colossal, current procedures that certify a sampler's uniformity has the severe shortcoming of requiring gigantic sample sizes to estimate reliable statistics [21,2,12]. Consequently, sampler uniformity has been checked only on miniature models so far, which is not convincing. Also, most uniformity procedures compute population parameters in a poorly scalable way (e.g., requiring calling a #SAT solver thousands of times [57]).

This paper extends our paper in SPLC'20 [29], where (i) a statistical test is formulated to reduce the sample size required for assessing a samplers' uniformity, and (ii) population parameters are computed with scalable algorithms we proposed in [30]. The additional contributions of this present paper are:

1. A new sampler called BDDSampler, which is built upon a ***Binary Decision Diagram* (BDD)** [10] technology (see Section 3).
2. A new statistical test to validate a samplers' uniformity, reducing the sample size requirements even more than our previous test (see Section 4).
3. An experimental validation with our new test of BDDSampler and other five state-of-the-art samplers (KUS [59], QuickSampler [21], Spur [2], Smarch [54], and Unigen2 [13,11]) on configuration models with up to 18,570 variables (see Section 5).
4. Experimental results show (i) our new statistical test needs the smallest sample size of all existing uniformity validation methods, and (ii) BDDSampler is the only sampler that satisfies both uniformity and scalability. Our software artifacts (BDDSampler, and the data and code scripts for replicating the experiments) are freely available at public repositories (see Section 8).

2 Related Work

Before discussing related work, a terminological clarification is needed. In the machine learning, the term *sample* usually refers to a single data point [15]. However, in *inferential statistics*, a *sample* is typically a collection of *cases*, where the number of cases in the sample is the ***sample size*** [14,36]. This paper adopts this latter terminology, and consequently, a ***sample*** is a set of configurations (i.e., a collection of SAT-solutions), whose cardinal is its sample size.

Here is additional standard statistical terminology that we will use in this paper. Inferential statistics aims to generalize the results obtained from a sample to the entire population. To do so, the most widespread approach, called ***Null Hypothesis Significance Test* (NHST)**, quantifies the probability of obtaining the sample results conditioned on the assumption that a given ***null hypothesis*** (H_0) is true (NHST fundamentals are explained in Chapter 13 of [36] and Chapter 3 of [65]). If such probability (named ***p-value***) is less or equal than an established threshold (called the ***significance level*** (α)) then H_0 is rejected, and thus its alternative hypothesis H_a accepted. Otherwise, H_0 is kept. As Table 1 shows, two mistakes under this framework can be made due to unusual random samples: rejecting a true H_0 (named ***Type 1 error***), and failing to reject a false H_0 (called ***Type 2 error***). The expression $1-\beta$ is known as the test's

power. The experimenter can adjust the Type 1 and 2 error probabilities through the thresholds α and β (see Chapter 4 of [65]).

Table 1 Type 1 and 2 errors under the NHST framework.

	H_0 is true in reality (H_0)	H_0 is false in reality ($\neg H_0$)
The decision inferred from the sample is "reject H_0" (R)	$\Pr(R\|H_0) = \alpha$ ***Type 1 error***	$\Pr(R\|\neg H_0) = 1-\beta$ ***Power***
The decision inferred from the sample is "do not reject H_0" ($\neg R$)	$\Pr(\neg R\|H_0) = 1-\alpha$	$\Pr(\neg R\|\neg H_0) = \beta$ ***Type 2 error***

2.1 Uniform Random Samplers

The following sections summarize some of the most common strategies to generate uniform random samples for a model of a configuration space that is encoded as a Boolean formula φ.

2.1.1 Atomic Mutations (QuickSampler)

QuickSampler[5] [21] uses a heuristic to gain scalability by minimizing the number of calls to a constraint solver. It generates a random configuration without taking into account the formula constraints. This configuration often violates constraints and thus is unsatisfiable. So, QuickSampler calls the Z3 solver [47] to fix the configuration by finding a MAX-SAT-solution. Then, QuickSampler flips the value of each variable and calls again Z3 to get another valid configuration. The differences between the variable values of the original and flipped SAT configurations are called ***atomic mutations***. By combining mutations, QuickSampler quickly generates new configurations without calling the solver as those configurations are usually legal [21].

2.1.2 Hashing-based sampling (Unigen2)

Several techniques divide the space of SAT-solutions into small "cells" of approximately the same size using r independent hash functions. Accordingly, sampling is done by choosing a cell at random, and then getting a satisfying assignment for that cell using a SAT solver. A critical point of these techniques is determining the "right" r value. For instance, Bellare et al. [6] showed that an r equal to the number of formula variables guarantees uniformity. However, Chakraborty et al. [13] reported that

[5] https://github.com/RafaelTupynamba/quicksampler

such r does not scale in practice; in contrast, $r = 3$ scales better and ensures near-uniformity. Unigen2[6] [11] develops these ideas further, giving stronger uniformity guarantees.

2.1.3 Counting-based sampling (KUS, Smarch, and Spur)

In Section 7.1.4 of [37], Knuth showed how to accomplish uniform random sampling by subsequently partitioning the SAT-solution space on variable assignments, and then counting the number of solutions of the resulting parts. Again, φ be a Boolean formula of v variables $x_1, x_2, \ldots, x_v$; let $\#SAT(\varphi)$ denote the number of solutions to φ; and let $r \in [0,1]$ be a random number in the unit interval. Conceptually, the procedure works as follows: The number of solutions where x_1 is true is counted, namely $\#\mathrm{SAT}(\varphi \wedge x_1)$. x_1 follows a Bernoulli distribution with probability $p_1 = \frac{\#\mathrm{SAT}(\varphi \wedge x_1)}{\#\mathrm{SAT}(\varphi)}$. x_1 is assigned false if $r \leq p_1$, true otherwise. Suppose x_1 is assigned false. Then, x_2 follows a Bernoulli distribution with probability $p_2 = \frac{\#\mathrm{SAT}(\varphi \wedge \bar{x}_1 \wedge x_2)}{\#\mathrm{SAT}(\varphi \wedge \bar{x}_1)}$, and it would be randomly assigned. The procedure advances until the last variable x_v is assigned, and thus the random solution is completed.

The original algorithm by Knuth is specified on BDDs, as the probabilities required for all the possible SAT-solutions are computed just once with a single BDD traversal, and then reused every time a random configuration is generated. Oh [53] reinvented Knuth's algorithm and was the first to implement and apply it to SPL analyses. Since then, Knuth's algorithm has been adapted to other ***knowledge compilation*** and ***Davis-Putnam-Logemann-Loveland* (DPLL)** [19] approaches. In particular, (i) the KUS[7] sampler [59] substitutes BDDs with ***deterministic-Decomposable Negation Normal Forms* (d-DNNFs)**, and (ii) Spur[8] [2] and Smarch[9] [54] count SAT solutions with a #SAT-solver named sharpSAT [62].

2.1.4 New: BDDSampler*, a scalable and uniform sampler*

Section 3 describes a new sampler called BDDSampler, which is based on Knuth's algorithm and implemented on top of the CUDD[10] library for BDDs.

According to the experimental results reported in Section 5, the only sampler that satisfies both scalability and uniformity is BDDSampler. More specifically, evidence shows that:

- BDDSampler, KUS, QuickSampler, and Spur are considerably faster than Smarch and Unigen2.
- In terms of uniformity, there are three types of samplers: (i) those that mostly fail to produce uniform samples (QuickSampler), (ii) those that usually work but from time to time generate non-uniform samples (KUS and Spur), and (iii) those that always produce uniform samples (BDDSampler, Smarch, and Unigen2).

[6] https://bitbucket.org/kuldeepmeel/unigen
[7] https://github.com/meelgroup/KUS
[8] https://github.com/ZaydH/spur
[9] https://github.com/jeho-oh/Kclause_Smarch
[10] https://github.com/vscosta/cudd

2.2 Prior Work on Testing Sampler Uniformity

The following sections summarize the methods that have been devised to test the uniformity of a random sampler $\mathbb{S}$.

2.2.1 Method 1: Generate a massive sample with $\mathbb{S}$, and compare it with another one obtained simulating an ideal uniform sampler

This is the most common technique in the literature [2,11,21,57,59]. First, the total number n of SAT-solutions is counted for the Boolean formula φ, typically using a #SAT-solver. Having n, the generation of a uniform sample with size s is simulated as follows: imagine that numbers $1,2,\ldots,n$ are put into a box; then, s numbers are sampled with replacement from the box, guaranteeing that the probability each number has to be extracted is $\frac{1}{n}$.

For example, JHipster encompasses 26,256 valid configurations [28]. Fig. 1 shows the histogram of a sample ten times greater than the number of configurations ($s = 26,256 \cdot 10$), which has been obtained sampling with replacement from the set {1, 2, …, 26256}. The x-axis depicts numbers' occurrences, i.e., there are numbers that appear 0, 1, …, 27 times in the sample; the y-axis shows how frequent are those occurrences in the sample. As expected, most numbers appear ten times (see the red vertical line in Fig. 1), however, and due to randomness, some numbers appear more frequently than others.

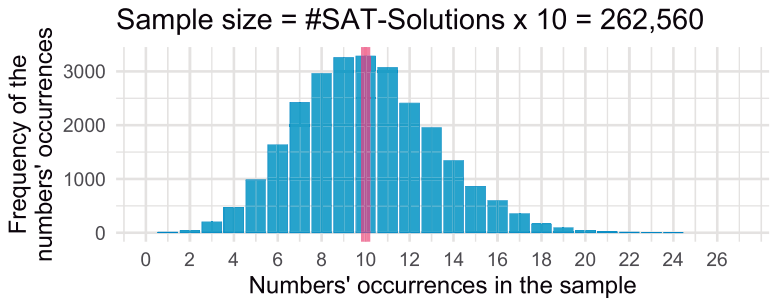

Fig. 1 Simulated uniform random sample of the JHipster configuration model.

Another sample with size s (whose value is quantified shortly), is then generated with sampler $\mathbb{S}$. For this sample, a counterpart histogram to Fig. 1 is obtained, representing how often solutions appear in that sample.

Finally, the uniformity of $\mathbb{S}$ is verified by measuring the distance between both histograms, using, for instance, the Kullback-Leibler divergence [2].

Unfortunately, this method has a severe limitation: it does not scale except for formulas with a small number of SAT-solutions because, to produce reliable results, s needs to be much larger than n (see [2,21] for an explanation). For example, Dutra et al. [21] propose $s \geq 5n$. As the number of solutions grows exponentially with the number of variables of φ, the method only works for the simplest models with just a few features.

2.2.2 Method 2: Assume the existence of a uniform sampler $\mathbb{U}$, and compare the samples generated by both $\mathbb{S}$ and $\mathbb{U}$

Chakraborty and Meel [12] proposed this method and implementation called ***barbarik***[11]. The method makes a strong assumption: there is a sampler $\mathbb{U}$ that is known to be uniform. Thus, two samples of the same size s are generated with $\mathbb{S}$ and $\mathbb{U}$ and, depending on the *distance* between the samples, i.e., on how similar they are, barbarik decides if $\mathbb{S}$ is approximately uniform.

The key of the method is how to define "approximately" for reaching a balance between uniformity and sample size, i.e., for avoiding the large s that Method 1 requires. Two parameters, called tolerance ε and intolerance η, adjust the definition of "uniformity" to avoid the above problems. A sampler is uniform whenever the probability $p_1, p_2, \ldots, p_n$ of all n solutions is exactly $\frac{1}{n}$.

Barbarik relaxes this definition, proposing that a sampler is *additive almost-uniform* if $p_1, p_2, \ldots, p_n \in \left[\frac{1-\varepsilon}{n}, \frac{1+\varepsilon}{n}\right]$. Moreover, a sampler is η-*far* from uniformity if

$$\left| \sum_{i=1}^{n} p_i - \frac{1}{n} \right| \geq \eta$$

Chakraborty and Meel claim that s depends on ε and η exclusively, but not on n. In particular, they state that a uniformity test with *significance level* $\alpha = 0.1$ (i.e., 0.9 probability of accepting the uniformity of a sampler when it is genuinely uniform) and *Type 2 error* $\beta = 0.1$ (i.e., 0.9 probability of rejecting the uniformity of a sampler that is not uniform) is accomplished when $\varepsilon = 0.6$ and $\eta = 0.9$, requiring a sample size of $1,729,750$. Unfortunately, they do not provide a detailed formal proof for these settings in [12].

An evident weakness of this method is the necessity of a sampler $\mathbb{U}$ with certified uniformity as a support lever. It is worth noting that, although an algorithm can be proven to generate uniform samples theoretically, some of its implementations may have errors. In other words, every sampling program needs to be tested, and thus Method 2 implicitly assumes the existence of another reliable uniformity testing method.

2.2.3 Method 3: Compare the theoretical variable probabilities in φ with the empirical variable frequencies in a sample generated with $\mathbb{S}$

Plazar et al.'s method [57] begins computing the theoretical probability each variable x has to appear in a SAT-solution. To do so, the procedure introduced in Section 2.1.3 is adopted, calling a #SAT solver repeatedly, one time per variable. $\#\mathrm{SAT}(\varphi)$ gives the total number of SAT-solutions, and $\#\mathrm{SAT}(\varphi \wedge x)$ calculates the number of solutions where x is true. Hence, the probability of x is $p = \frac{\#\mathrm{SAT}(\varphi \wedge x)}{\#\mathrm{SAT}(\varphi)}$. Likewise, if x is true t times in a sample of size s, its empirical frequency is $f = \frac{t}{s}$. Then, the *deviation* between p and f is $d = 100 \cdot \frac{|p-f|}{p}$. Finally, Plazar et al. propose two *thresholds* for d: (i) when $d \leq 10$ for all variables, the deviations are *very low*, and

[11] https://github.com/meelgroup/barbarik

thus sampler uniformity is accepted; (ii) when $d \geq 50$ for some variables, they show *very high* deviations, and so uniformity is rejected. Regarding the sample size, Plazar et al. propose always using $s \sim 10^6$, independently of the number of variables of φ (no formal justification is given for this specific value in [57]).

Regrettably, this method often throws false negatives for variables with low probabilities. Suppose a variable has $p = 0.01$. Then, a genuine uniform sampler might easily generate a sample where f is slightly different just due to randomness, e.g., $f = 0.015$. Therefore, $d = 100 \cdot \frac{|0.01-0.015|}{0.01} = 50$, and thus the sampler uniformity would be rejected. The chances that these types of wrong diagnoses happen increases with the number of low-probability variables, and it is worth noting that real models with numerous low-probability variables are not "corner cases"; for example, in three out of the seven configuration models analyzed in [30], more than 46% of their variables have $p \leq 0.05$: the open-source project *Fiasco v2014092821*, the *Dell* laptop configurator, and the *Automotive 02* system.

2.2.4 Method 4: a statistical goodness-of-fit test that compares theoretical variable probabilities in φ with the empirical variable frequencies in a sample generated with $\mathbb{S}$

In the past [29], we presented a procedure called *Feature Probability* (FP) test, which compares the empirical feature frequencies in a sample with the theoretical feature probabilities in the whole population of SAT-solutions. Instead of using the limited Method 3 *deviation* measure, our FP method (i) has a robust mathematical basis, (ii) estimates the *statistical significance* of the results (i.e., how generalizable they are), and (iii) supports adjusting the sample size according to precise statistical criteria (i.e., *Type 1* and *2 errors*, and *effect size*).

It is worth noting that a major shortcoming of Methods 1, 2, and 3 is the large sample size they need. For instance, in [2] and [59], Method 1 is applied on a model called blasted_case110 with 287 variables, requiring $s = 4 \cdot 10^6$ SAT-solutions. In [12], Method 2 is used on blasted_case110 as well, needing this time $1,729,750$ SAT-solutions to ensure probability errors of Type 1 $\alpha = 0.1$ and Type 2 $\beta = 0.1$. In contrast, our FP test provides stronger test guarantees ($\alpha = 0.01$ and $\beta = 0.01$) for blasted_case110 with a minimal sample size of 13,027 solutions (i.e., a 99.25% sample size reduction with respect to Method 2).

2.2.5 New: an improved goodness-of-fit test

Section 4 presents a new procedure that improves Method 4 by, instead of examining the variable probabilities, analyzing how the number of variables assigned to true distributes along the SAT-solutions. We show in Section 5 that the new method requires even smaller samples, thus widening the support for testing samplers' uniformity on larger models. For example, the sample size our new method requires for blasted_case110 with $\alpha = 0.01$ and $\beta = 0.01$ becomes 6,563 solutions.

2.2.6 Recap

Excluding the methods presented in this article and in our conference paper, there are serious practical problems in applying existing ways to test for sampler uniformity. We provide experimental evidence in Section 5 that our improved goodness-of-fit test is superior to prior work as it requires the smallest sample size of all existing tests, thus enabling the verification of samplers' uniformity in large models. As we will see, this highly increases the test sensitivity to detect samplers' uniformity flaws. Moreover, results show that our test provides (i) *valid* judgements, which are consistent with the verdicts given by the alternative methods proposed in the literature, and (ii) *reliable* judgements, which remain consistent when the test is applied repeatedly to the same model and sampler.

3 The BDDSampler Tool

This section describes BDDSampler: a sampler that uses ***Binary Decision Diagrams (BDDs)***. A practical example how configuration models can be translated into Boolean formulas is presented in Section 3.1. Then, a BDD encoding of a Boolean formula is covered in Section 3.2. Finally, how BDDSampler works is explained in Section 3.3.

3.1 From Configuration Models to Boolean Formulas

Let us start with an example to help to explain BDDSampler and our samplers' uniformity test. As already mentioned in this paper's introduction, BusyBox supports the inclusion/exclusion of a number of features at compile time. These features and their interrelationships are specified with a configuration language named Kconfig which is used in many other relevant open-source projects [7], such as the Linux Kernel, axTLS, EmbToolkit, Freetz, etc.

Fig. 2 shows an excerpt of the Kconfig specification of BusyBox v1.23.2. There are several configs encoding six features and their interdependencies. All features (STATIC, PIE, ..., FEATURE_SHARED_BUSYBOX) are Boolean (see the bool keyword in Lines 2, 4, ..., 15), meaning that they can be either selected or deselected. Configs trigger a prompt to request the user for their Boolean feature value, e.g., Build BusyBox as a static binary (no shared libs) in Line 2. Finally, some dependencies between features are set, e.g., according to the depends sentence in Line 10, BUILD_LIBBUSYBOX can only be selected if none of the following features are selected: FEATURE_PREFER_APPLETS, PIE, neither STATIC.

The graph in Fig. 3 depicts the entire BusyBox configuration model, which includes 613 features and 530 inter-dependencies; nodes represent features, and edges depict dependencies. The Kconfig excerpt in Fig. 2 is zoomed in Fig. 3.

Given the configuration models' complexity, they are usually translated into Boolean formulas that are then processed with logic engines. For instance, Equation 1 is the Boolean encoding of Fig. 2 (a detailed explanation of how to convert Kconfig

```
config STATIC
  bool "Build BusyBox as a static binary (no shared libs)"
config PIE
  bool "Build BusyBox as a position independent executable"
  depends on !STATIC
config FEATURE_PREFER_APPLETS
  bool "exec prefers applets"
config BUILD_LIBBUSYBOX
  bool "Build shared libbusybox"
  depends on !FEATURE_PREFER_APPLETS && !PIE && !STATIC
config FEATURE_INDIVIDUAL
  bool "Produce a binary for each applet, linked against libbusybox"
  depends on BUILD_LIBBUSYBOX
config FEATURE_SHARED_BUSYBOX
  bool "Produce additional busybox binary linked against libbusybox"
  depends on BUILD_LIBBUSYBOX
```

Fig. 2 Excerpt of the BusyBox Kconfig specification.

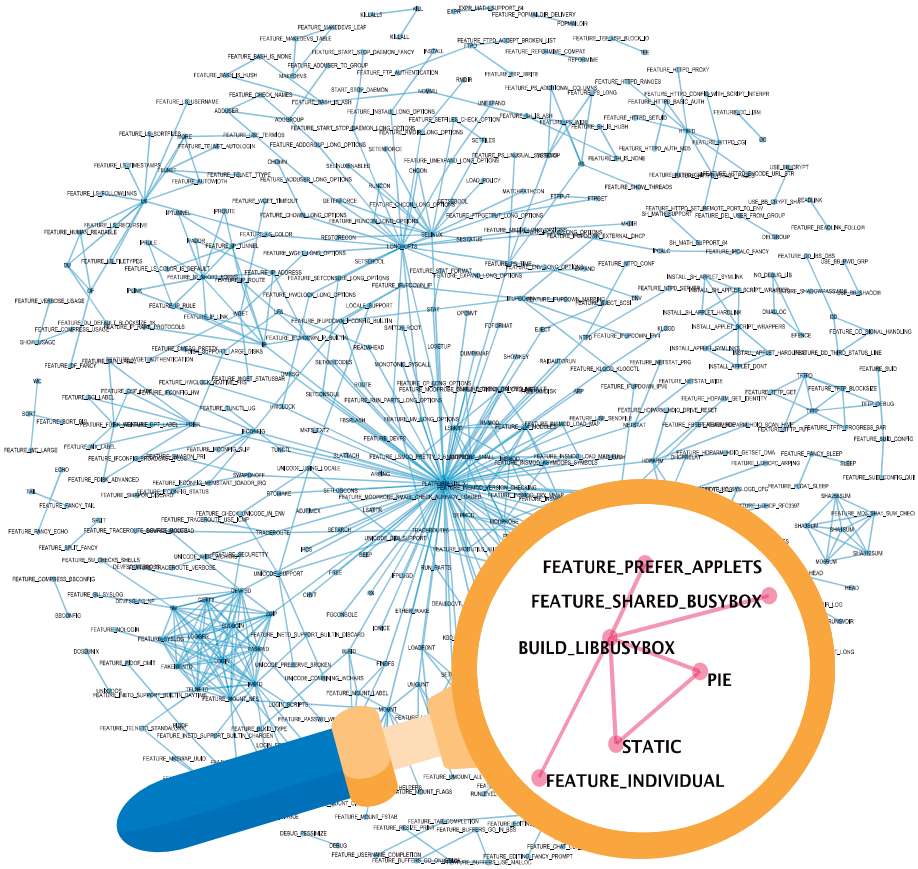

Fig. 3 Graph-representation of the BusyBox Kconfig specification.

specifications into Boolean formulas is given in [24]). In this section and the following one, we explain how to use BDDs for (i) generating random samples from the formulas, and (ii) testing the uniformity of an input sampler.

$$\begin{aligned}\varphi \equiv &(\neg \text{STATIC} \vee \neg \text{PIE}) \wedge \\ &(\neg \text{BUILD\_LIBBUSYBOX} \vee \neg \text{FEATURE\_PREFER\_APPLETS}) \wedge \\ &(\neg \text{BUILD\_LIBBUSYBOX} \vee \neg \text{PIE}) \wedge \\ &(\neg \text{BUILD\_LIBBUSYBOX} \vee \neg \text{STATIC}) \wedge \\ &(\neg \text{FEATURE\_INDIVIDUAL} \vee \text{BUILD\_LIBBUSYBOX}) \wedge \\ &(\neg \text{FEATURE\_SHARED\_BUSYBOX} \vee \text{BUILD\_LIBBUSYBOX})\end{aligned} \tag{1}$$

3.2 A Brief Introduction to BDDs

A BDD [10] encodes a Boolean formula as a rooted directed acyclic graph composed of terminal and non-terminal nodes. Terminal nodes are represented as $\boxed{0}$ and $\boxed{1}$, and non-terminal nodes are labeled with the formula variables. Two edges, named *low* and *high*, come out of every non-terminal node. Low is depicted with a dashed line ($\dashrightarrow$), and high with a solid line ($\rightarrow$). A BDD encodes every possible assignment of the formula variables as a path that descends from the root to the terminal nodes, going through solid lines when the corresponding variables are assigned to true and through dashed lines otherwise. An assignment is satisfiable, i.e., it evaluates the formula to true, whenever the traversed path ends at $\boxed{1}$.

Fig. 4 depicts a BDD that encodes the BusyBox excerpt specified by Equation 1. A configuration whose only activated features are BUILD_LIBBUSYBOX and FEATURE_SHARED_BUSYBOX conforms with the constraints (i.e., it is valid) and so it corresponds to the path BUILD_LIBBUSYBOX $\rightarrow$ FEATURE_INDIVI-DUAL $\dashrightarrow$ FEATURE_SHARED_BUSYBOX $\rightarrow$ FEATURE_PREFER_APP-LETS $\dashrightarrow$ STATIC $\dashrightarrow$ PIE $\dashrightarrow$ $\boxed{1}$. In contrast, as STATIC and PIE are mutually exclusive, no configuration includes them simultaneously. Thus, all paths with solid lines coming out of both STATIC and PIE finish at $\boxed{0}$.

BDDs are typically ordered and reduced. A BDD is ***ordered*** when its variables are in the same position, called ***index***, in every path from the root to the terminal nodes. For example, in Fig. 4, STATIC (whose index is 4) always goes before PIE and after FEATURE_PREFER_APPLETS (whose indices are 5 and 3, respectively). A BDD is ***reduced*** if it is free of redundant information. For instance, every blue/dark-shaded node in Fig. 4 is superfluous because both of its edges point to the same node and thus the formula evaluation is identical whether these variables are assigned to true or false. Consequently, these unnecessary tests are avoided in the reduced BDD in Fig. 5 to save computer memory.

It is worth noting that the variable ordering chosen to build the BDD has a tremendous impact on its size. Whereas a BDD can be reduced optimally (the reduction procedure was presented in the seminal article [10]), obtaining the best variable arrangement that minimizes its size is an NP-problem (Chapters 8 and 9 of [45] provide a comprehensive discussion on this topic). Several variable ordering heuristics [23, 24, 46, 50] have been proposed for the specific case of configuration model formulas. As reported in Section 5, we have been able to synthesize BDDs for large configuration models, with up to 17,000 features, by using these heuristics.

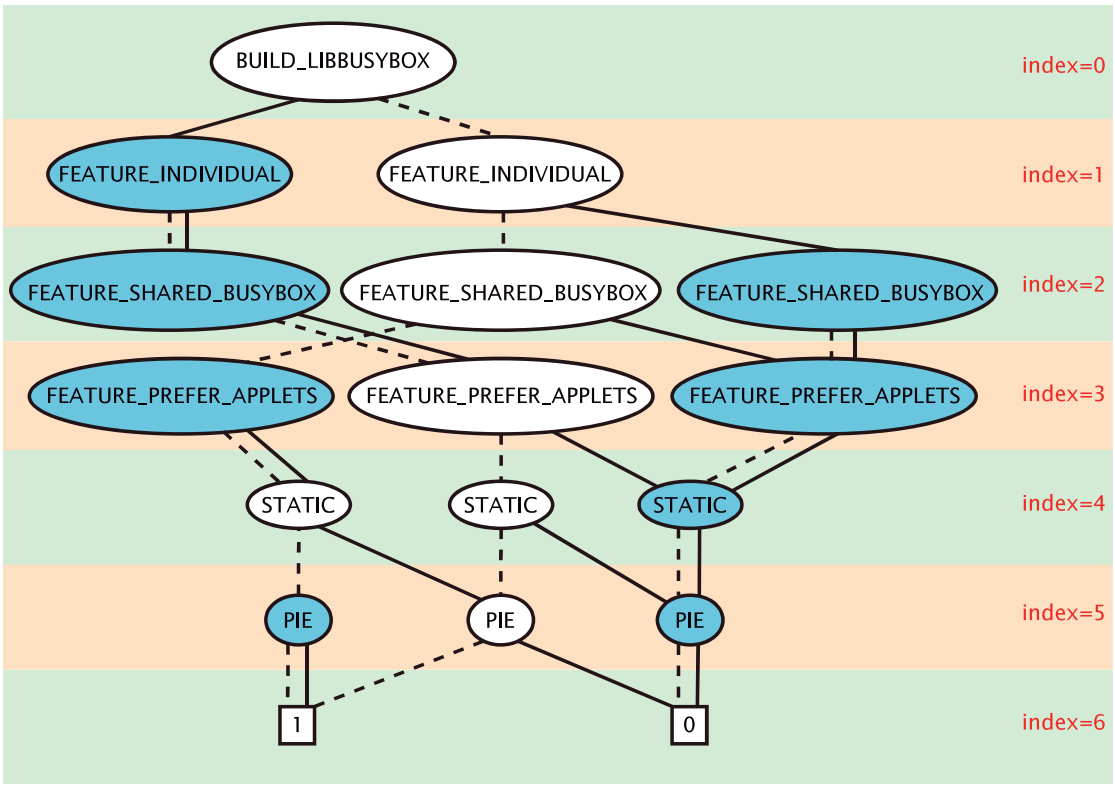

Fig. 4 *Non-reduced* BDD encoding of Equation 1.

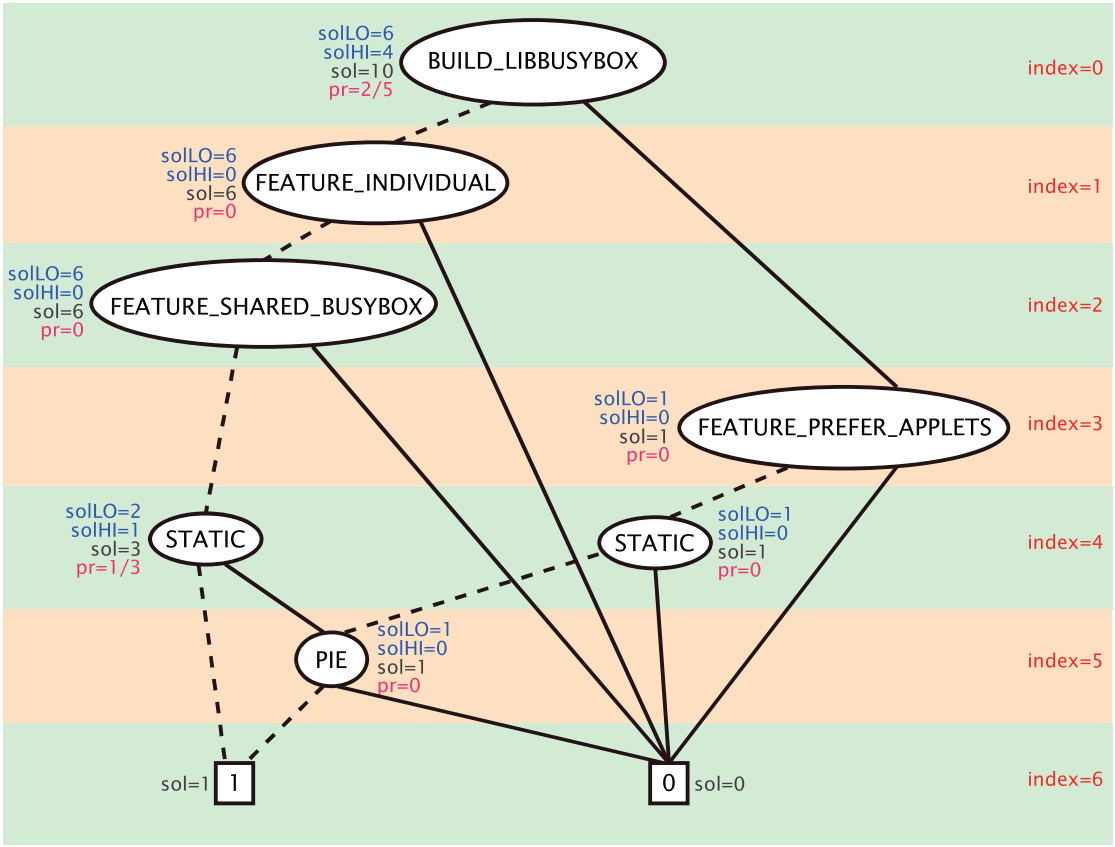

Fig. 5 *Reduced* BDD encoding of Equation 1.

3.3 How BDDSampler Works

BDDSampler takes an ordered and reduced BDD as input and generates random configurations in a two-step process described by Algorithms 1 and 2. Fig. 5 summarizes Algorithm 1 computations for our running example. The algorithm decorates each

non-terminal node with its probability of reaching the terminal $\boxed{1}$ if the associated variable is set to true. Algorithm 1 proceeds in a bottom-up fashion, collecting the number of SAT solutions that can be produced by its low and high children (solLO and solHI in Lines 8-9), adding them up (sol in Line 10), and then computing the ratio corresponding to the high child (pr in Line 11). As the BDD is reduced, Algorithm 1 adjusts the solution counts in Lines 8-9 for the removed nodes with the expression $2^{\text{index}(n_{\text{LO|HI}})-\text{index}(n)-1}$. For traversing efficiently the BDD, Algorithm 1 uses Bryant's method [10] as follows: the algorithm is called in Line 12 with the BDD root as argument and with a Boolean *mark* for every node being either all true or all false; then, it explores all nodes by recursively visiting the low and high children (Lines 6 and 7). Whenever a node is visited, its mark value is complemented (Line 2). Comparing the node with its children's marks, it is decided if the children have already been visited. The method ensures that each node is visited exactly once and that, when the traverse finishes, all node marks have the same value.

Algorithm 1. Get all node probabilities.

```
1  Function getNodePr(n)
2      mark(n) ← ¬ mark(n)
3      if n = 0 then sol[n] ← 0 // [0] is reached
4      else if n = 1 then sol[n] ← 1 // [1] is reached
5      else
           // explore low
6          if mark(n) ≠ mark(n_LO) then getNodePr(n_LO)
           // explore high
7          if mark(n) ≠ mark(n_HI) then getNodePr(n_HI)
           // get node probabilities
8          solLO ← sol[n_LO]·2^(index(n_LO)−index(n)−1)
9          solHI ← sol[n_HI]·2^(index(n_HI)−index(n)−1)
10         sol[n] ← solLO + solHI
11         pr[n] ← solHI / sol[n]
12 getNodePr(ROOT)
```

Whereas Algorithm 1 is run once as an initialization method, Algorithm 2 needs to be run as many times as configurations we want to generate. Algorithm 2 performs a random walk from the root to the terminal $\boxed{1}$. When a non-reduced node is visited, the path is selected randomly according to its probability (Lines 11-16): if the node probability is p, then its low and high edges are chosen with probabilities $1-p$ and p, respectively. Regarding the reduced nodes, the generated configuration will be valid no matter if their variables are set to true or false (that is the reason why these nodes were removed). Thus their value is chosen randomly with a 1/2 probability by taking into account that a reduced node index may be less than the BDD root index (Lines 6-7) or greater (Lines 17-18).

Algorithm 2 is remarkably fast since its time complexity is proportional to the number of indices (i.e., of variables), not the number of nodes in the BDD. Moreover, multiple instances of Algorithm 2 can be run in parallel over the same BDD, as the node probabilities are read but not modified.

Algorithm 2. Generate a random Configuration.

```
1  Function random
2      return a random number ∈ [0, 1)
3  Function fiftyfifty
4      return random () < 0.5
5  Function generateConfiguration()
       // generate random values for variables corresponding to reduced ROOT
          predecessor nodes
6      for (i ← 0; i < index(ROOT); i++) do
7          sample[i] ← fiftyfifty ()
       // generate random values for the remaining variables
8      trav ← ROOT
9      while trav ≠ 1 do // iterate until reaching the 1-terminal node
10         ind ← index(trav)
11         if random () < pr[trav] then
12             trav ← trav_HI
13             sample[ind] ← TRUE
14         else
15             trav ← trav_LO
16             sample[ind] ← FALSE
           // generate random values for variables of reduced intermediate nodes
17         for (i ← index+1; i < index(trav);i++) do
18             sample[i] ← fiftyfifty ()
19     return sample
```

Finally, BDDSampler is built on top of CUDD 3.0[12]. As other modern BDD libraries like Sylvan [20], CUDD uses a technique called *complement edges* [9] to save nodes. With this technique, edges are enriched with a complement attribute that removes the need of having two terminal-nodes (basically, when an edge has the complement attribute enabled, the only terminal node is interpreted as its negation). Accordingly, BDDSampler tweaks Algorithms 1 and 2 to work with complement edges. We have decided to show the algorithms for regular BDDs without complement arcs for simplicity.

The BDDs of 218 models, which will be used in Section 5 to perform our experimental evaluation, are available at https://doi.org/10.5281/zenodo.4514919 in the DDDMP format that CUDD uses for complement edge BDDs.

4 Assessing the uniformity of SAT solution samplers

Fig. 6 sketches our approach to verify that a sampler generates uniform random samples of a model that is encoded as a Boolean formula. The method compares empirical information about a sample with theoretical information about the whole population of SAT-solutions that the model represents.

[12] https://github.com/vscosta/cudd

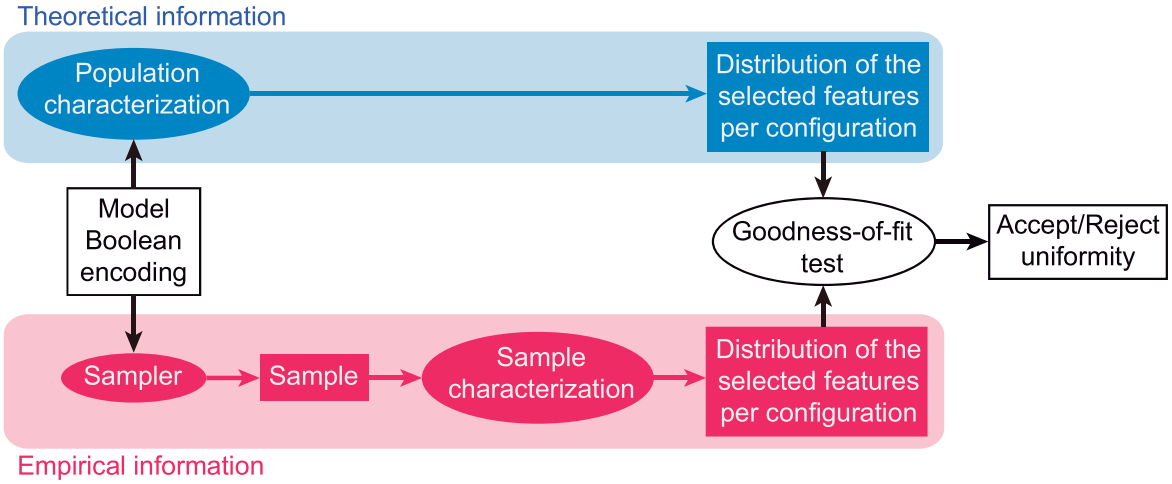

Fig. 6 Proposed method for verifying if a sampler generates uniform samples for a model.

4.1 The SFpC Goodness-Of-Fit Test

In statistics, the procedures for examining how well a sample agrees with the population distribution are known as ***goodness-of-fit*** tests [18]. They require characterizing both the sample and the population in terms of a quantitative measure. In particular, we propose the distribution of the number of variables assigned to true among all SAT-solutions, called the ***Selected Features per Configuration* (SFpC)** test. For instance, Fig. 7 compares the theoretical distribution of all $7.428 \cdot 10^{146}$ SAT-solutions of the BusyBox model with the distribution of 17,738 configurations generated with the samplers BDDSampler and QuickSampler, Figures 7a,b respectively. The justification for this sample size 17,738 is given in Section 4.2.

The distribution of the whole population of SAT-solutions of a model can be computed with the ***Product Distribution* (PD)**[13] algorithm we proposed in [30]. PD takes the BDD encoding of a model as input, and as explained in Section 3.D of [30], its time complexity is $O(nv^2)$, where n is the number of BDD nodes and v the number of model variables. Accordingly, PD scales for large models. For instance, on an Intel(R) Core(TM) i7-6700HQ, it took 2.74 minutes to compute the distribution of the Automotive02 model [41], which with 17,365 variables and 321,897 clauses encompasses $5.26 \cdot 10^{1,441}$ SAT-solutions.

As the theoretical histogram shows in Fig. 7a, the smallest and largest BusyBox configurations have 6 and 571 features activated, respectively. 95% of the configurations have between 277 and 327 variables assigned to true.

The BDDSampler histogram (Fig. 7a) agrees with the normally distributed population. However, the QuickSampler histogram (Fig. 7b) is bimodal where most configurations have 100 or 200 features approximately, quite different from the theoretical histogram.

After exploring the sample's goodness-of-fit graphically, it is desirable to advance towards a more formal test that provides an accurate numerical quantification. A good candidate to measure the distance/difference between the sample and population dis-

[13] https://github.com/rheradio/VMStatAnal

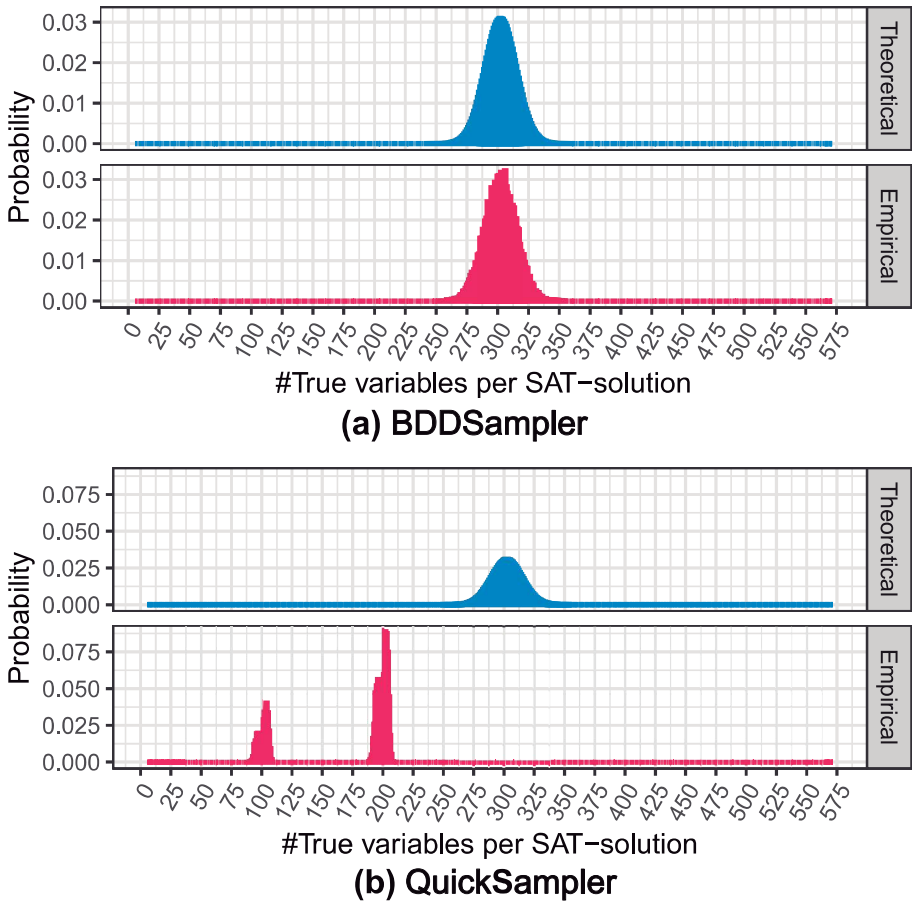

Fig. 7 Distribution of all BusyBox SAT-solutions compared with the distribution of 17,738 configurations generated with BDDSampler and QuickSampler.

tributions is the ***Kullback–Leibler divergence***[14] [17]. For discrete probability distributions P and F specified on the same probability space $\mathbb{X}$, the Kullback–Leibler divergence from F to P is defined as:

$$D_{\mathrm{KL}}(P||F) = \sum_{x \in \mathbb{X}} P(x) \log_2 \left(\frac{P(x)}{F(x)} \right) \tag{2}$$

However, the Kullback–Leibler divergence is not symmetric, and thus it cannot rigorously be considered a ***metric*** [44]. For this reason, we use its symmetrical and normalized version, which is named ***Jensen-Shannon divergence*** [17,44] and defined as:

$$JSD(P||F) = \frac{1}{2} D_{\mathrm{KL}}(P||M) + \frac{1}{2} D_{\mathrm{KL}}(F||M) \tag{3}$$

where $M = \frac{1}{2}(P+F)$.

In our case, vectors F and P are defined as follows:

[14] The Kullback–Leibler divergence and especially one simplified version called *cross-entropy* are widely used as *loss functions* to compare the neural network predicted output with the observations used to train the network (see Chapter 3 of [25] for a summary of the Kullback–Leibler divergence and cross-entropy applications to deep learning).

– $F = [f_0, f_1, \ldots, f_n]$ stores the SAT-solution *frequency* distribution (i.e., the red histograms in Figure 7). That is,

$$f_0 = \frac{\text{\#SAT solutions in the sample with no variables assigned to true}}{\text{sample size}}$$

$$f_1 = \frac{\text{\#SAT solutions in the sample with 1 variable assigned to true}}{\text{sample size}}$$

$$\ldots$$

$$f_n = \frac{\text{\#SAT solutions in the sample with all variables assigned to true}}{\text{sample size}}$$

– $P = [p_0, p_1, \ldots, p_n]$ stores the theoretical SAT-solution *probability* distribution of Fig. 7. That is,

$$p_i = \frac{\text{\#SAT solutions in the population with } i \text{ variables assigned to true}}{\text{population size}}$$

To avoid worthless comparisons, all i-elements with $p_i = 0$ are removed from F and P because, as all solutions in the sample are guaranteed to be valid, the corresponding f_i's are necessarily 0 as well. For instance, the BusyBox model has 613 variables, but all valid configurations have between 6 and 571 variables assigned to true. Therefore, $\{f_0, f_1, \ldots, f_5, f_{572}, f_{573}, \ldots, f_{613}\}$ and $\{p_0, p_1, \ldots, p_5, p_{572}, p_{573}, \ldots, p_{613}\}$ are deleted from F and P, respectively.

The Jensen-Shannon divergence $\text{JSD}(P||F)$ measures to what extent the difference between F and P is greater than expected by chance if F corresponded to a uniform random sample. In the extreme cases, $\text{JSD}(P||F) = 0$ when F totally matches P, and $\text{JSD}(P||F) = 1$ when the F completely disagrees with P.

Nevertheless, JSD is a mere distance/difference metric, i.e., we cannot tell if JSD is *significantly* greater than expected due to randomness. Therefore, a *statistical inference test* is needed to quantify how generalizable the obtained distance is, i.e., a test that estimates the probability of a specific value of $\text{JSD}(P||F)$ assuming that the sampler is genuinely uniform. In the case that the estimated probability is excessively low (below a significance level α), it is unlikely that the disagreement between F and P is due to chance, and so we can conclude that the sampler is not uniform.

Let s be the sample size (whose value we compute in Section 4.2), and m the number of elements in P after having removed those with $p_i = 0$. According to the proof given by Grosse et al. in Section 4.C of [26], $2s(\ln 2)\text{JSD}(P||F)$ has a χ^2 distribution with $m-1$ degrees of freedom. As a result, a *Chi-Squared goodness-of-fit test* built upon the statistic $2s(\ln 2)\text{JSD}(P||F)$ guides us to decide whether the sampler is uniform. In our BusyBox running example, $s = 17,738$ and $m = 613-6-42 = 565$, hence if the sampler is uniform then $2 \cdot 17,738(\ln 2)\text{JSD}(P||F)$ should follow a χ^2_{565} distribution.

In contrast to typical ***Null Hypothesis Significance Tests* (NHSTs)**, where the null hypothesis H_0 states the opposite to what the researcher pursues to demonstrate, goodness-of-fit tests are a special case of NHSTs where H_0 is: "the sample agrees

with the population" (see Chapter 3 of [18] for a detailed description of *Chi-Squared goodness-of-fit tests*). Coming back to our case study, let us set the threshold $\alpha = 0.01$ to test the BusyBox samples generated with:

- BDDSampler:

$$\begin{aligned}
&\text{JSD}(P||F) = 0.001085388\\
&2 \cdot 17,738(\ln 2)\text{JSD}(P||F) = 26.68979\\
&p - \text{value} = \Pr\Big(\text{getting a value } \geq 26.68979 \Big| H_0\Big) \sim 1 > \alpha\\
&\Rightarrow \text{Test result : Do not reject } H_0
\end{aligned}$$

- QuickSampler:

$$\begin{aligned}
&\text{JSD}(P||F) \sim 1\\
&2 \cdot 17,738(\ln 2)\text{JSD}(P||F) = 12,923.04\\
&p - \text{value} = \Pr\Big(\text{getting a value } \geq 12,923.04 \Big| H_0\Big) \sim 0 \leq \alpha\\
&\Rightarrow \text{Test result : Reject } H_0
\end{aligned}$$

To sum up, the test corroborates numerically the histogram comparison in Fig. 7: BDDSampler generated a uniform sample, but QuickSampler did not.

4.2 Sample Size Estimation

The reliability of a Chi-Squared goodness-of-fit test depends on the following parameters (see Table 1):

- The *significance level* α sets the probability of making a *Type 1 error*, i.e., the probability of rejecting H_0 when it is indeed true (false positive). It is worth noting that α is also the threshold for rejecting H_0 (i.e., H_0 is rejected whenever the p-value $\leq \alpha$).
- β sets the probability of making a *Type 2 error*, i.e., the probability of accepting a false H_0 (false negative). The expression $1 - \beta$ is called the test's *power*, i.e., the probability of rejecting a false H_0.

When H_0 is false, *it is false to some degree*. That degree is measured by another parameter called the *effect size* [43]. In particular, Cohen [16] proposes the index w for measuring the effect size in Chi-Squared tests. As a rule of thumb, w values of 0.1, 0.3, and 0.5 correspond to *small*, *medium*, and *large* effect sizes, respectively.

Interestingly, sample size, effect size, α, and β have an intimate relationship in NHSTs: given any three of them, the fourth can be determined. In Section 7.3 of [16], Cohen provides different *power tables* to estimate the minimum sample size required to ensure the reliability of a Chi-Squared test given the values of α, β, w, and χ^2's degrees of freedom. Nowadays, there is available statistical software that provides

those tables, e.g., the R package pwr[15] (see Chapter 10 of [33]) and the G*Power[16] tool [22].

In the previous section, we saw that the goodness-of-fit of any sample from the BusyBox configuration model can be undertaken with a Chi-Squared test with 565 degrees of freedom. Then, according to Cohen's power tables, the required sample size is 17,738 configurations when $\alpha = \beta = 0.01$, and $w = 0.1$.

5 Empirical Evaluation

This section describes the experimental evaluation of our approach using the ***Goal/Question/Metric* (GQM)** method [60]. As Fig. 5 shows, the evaluation pursues two goals (G1 and G2), which are refined into five questions (Q1-Q5) that are answered using different metrics.

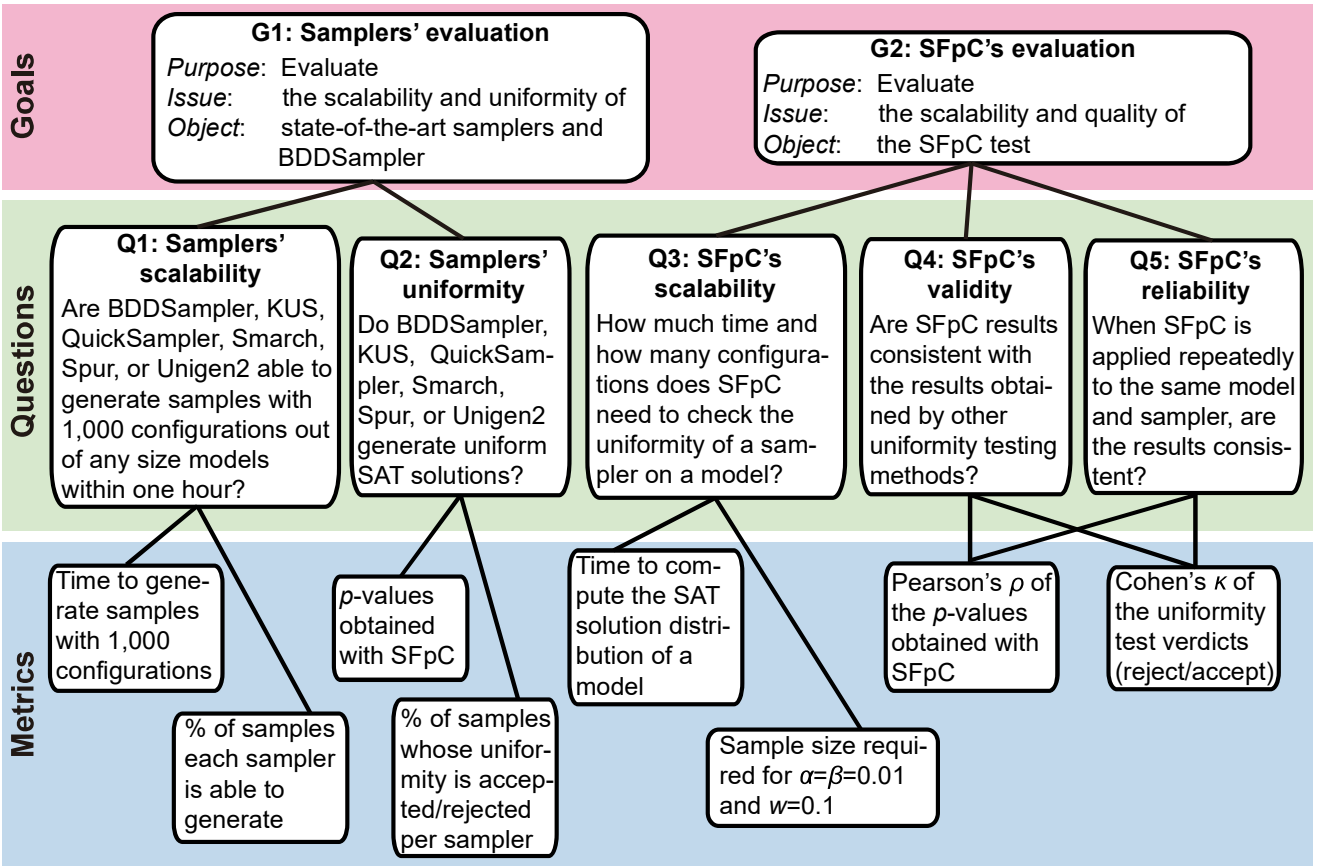

Fig. 8 Overview of the performed empirical evaluation with the GQM method.

The following points summarize our evaluation's goals and questions:

G1: Samplers' evaluation. The first goal G1 is to evaluate the scalability and uniformity of BDDSampler and the following state-of-the-art samplers: KUS[17] [59],

[15] https://cran.r-project.org/web/packages/pwr

[16] https://www.psychologie.hhu.de/arbeitsgruppen/allgemeine-psychologie-und-arbeitspsychologie/gpower.html

[17] https://github.com/meelgroup/KUS

QuickSampler[18] [21], Smarch[19] [54], Spur[20] [2], and Unigen2[21] [13,11]. G1 is broken down into Questions Q1 and Q2:

Q1: Samplers' scalability. *Are BDDSampler, KUS, QuickSampler, Smarch Spur, or Unigen2 able to generate samples with 1,000 configurations for models of all sizes within one hour?*

Q2: Samplers' uniformity. *Do BDDSampler, KUS, QuickSampler, Smarch Spur, or Unigen2 always generate uniform samples?*

G2: SFpC's evaluation The second goal G2 is to evaluate the scalability and quality, in terms of validity and reliability, of our SFpC test. G2 is refined into Questions Q3-Q5:

Q3: SFpC's scalability. *How much time and how many configurations does SFpC need to check the uniformity of a sampler on a model?*

Q4: SFpC's validity. *Does SFpC produce results consistent with the results obtained by other uniformity testing methods?*

Q5: SFpC's reliability. *When SFpC is applied repeatedly to the same model and sampler, are the results consistent?*

Section 5.1 presents the experimental setup. As Fig. 9 shows, three experiments E1-E3 were performed to solve the questions (e.g., Experiment E2 supported answering Questions Q2, Q3, and Q5). Sections 5.2-5.6 describe these experiments and the specific metrics used to answer the questions. The detailed results and all the material needed to replicate our experiments are available in the public repositories presented in Section 8.

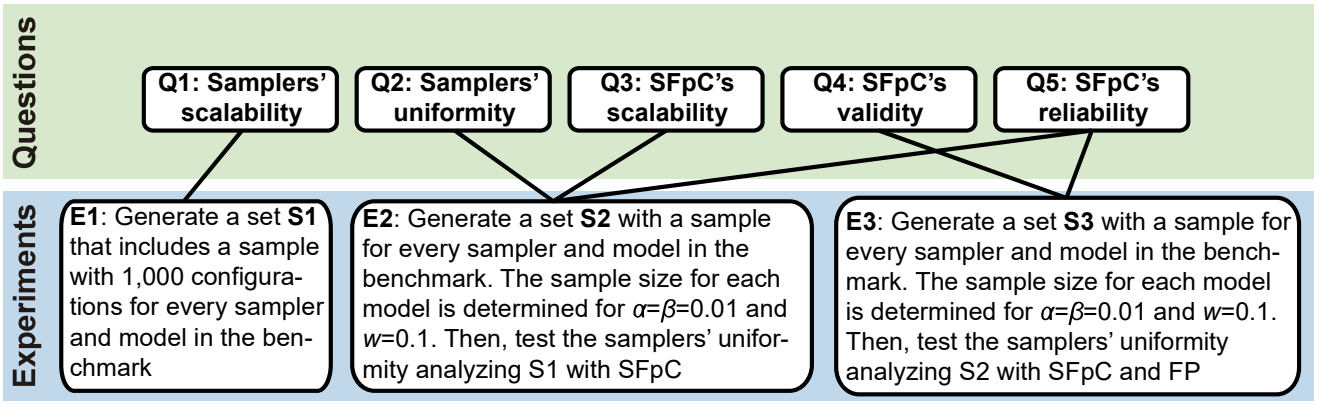

Fig. 9 Relationship between questions and experiments.

5.1 Experimental Setup

The samplers were tested against a suite of 218 models encoded as Boolean formulas in all of the following formats:

[18] https://github.com/RafaelTupynamba/quicksampler
[19] https://github.com/jeho-oh/Smarch
[20] https://github.com/ZaydH/spur
[21] https://bitbucket.org/kuldeepmeel/unigen

1. *DIMACS*, which is the format QuickSampler, Smarch, Spur, and Unigen2 use as input. These samplers rely on SAT technology, and DIMACS is the format for ***Conjunctive Normal Form* (CNF)** formulas that SAT technology uses.
2. *DDDMP*, which is the format BDDSampler and its underlying library CUDD use for BDDs.
3. *NNF*, which is the format KUS uses for d-DNNFs.

In particular,

- The DIMACS files of the industrial SAT formulas and JHipster were retrieved from [57].
- The DIMACS file of LargeAutomotive was gathered from [41].
- The DIMACS file of DellSPLOT was obtained from [52].
- We generated the DIMACS files of axTLS, Fiasco, uClibc, ToyBox, BusyBox, and EmbToolkit by processing their Kconfig specifications with our tool Kconfig2Logic[22] [24].
- We generated all DDDMP files from their corresponding DIMACS files with our tool Logic2BDD[23] [23].
- We generated all NNF files from their respective DIMACS files with the d-DNNF compiler d4 [42] that is embedded in **KUS**.

In total, 209 are industrial SAT formulas (mostly modeling integrated circuits) that are typically used as a benchmark in the SAT-sampling literature [2,11,57]. The remaining nine models represent configurable software systems. Table 2 describes the nine configuration models (the largest model is also referred as Automotive02 in the SPL literature [41]).

The histogram in Fig. 10 shows the model size distribution according to their number of variables. Since there is a wide range from the smallest model in the benchmark to the largest one (from 14 to 18,570 variables), the scale has been logarithmically transformed to shrink the range and thus facilitate the figure interpretation (see Chapter 5 of [67] for an explanation on logarithmic scale transformations). The scatter plot in Fig. 10 represents the model sizes in terms of their variables and clauses. The grey regression line shows that $\text{Log}_2(\#\text{Clauses})$ depends on $1.35 + 1.03 \cdot \text{Log}_2(\#\text{Variables})$. Points corresponding to configuration models are labeled, and models with more and fewer clauses than those predicted by the linear regression are colored red and blue, respectively. Note that in the interval $[9.88, 11.2]$ of $\text{Log}_2(\#\text{Variables})$ there are only 5 models, and all of them have fewer clauses than predicted. As these models are simpler in terms of clauses, processing them requires less time than expected for their variable number, and thus regression curves in Figs. 12, 17, and 16 will show positive convexity in that interval.

The experiments were run on an Intel(R) Core(TM) i7-6700HQ, 2.60GHz, 16GB RAM, operating Linux Ubuntu 19.10. Samplers were executed on a single thread (i.e., with no parallelization), and without considering any Boolean formula preprocessing, such as ***Minimal Independent Support* (MIS)** [32].

[22] https://github.com/davidfa71/Extending-Logic/tree/master/code/Kconfig2Logic
[23] https://github.com/davidfa71/Extending-Logic/tree/master/code/Logic2BDD

Table 2 Software configuration models included in the benchmark.

Model	#Variables	#Clauses	#SAT-Solutions
JHipster [28]	45	104	26,256
axTLS 1.5.3 (http://axtls.sourceforge.net/)	64	96	$3.924 \cdot 10^{12}$
Fiasco 2014092821 (https://os.inf.tu-dresden.de/fiasco/)	113	4,717	$5.144 \cdot 10^{9}$
DellSPLOT [52]	118	2,181	$7.440 \cdot 10^{6}$
uClibc 201 50420 (https://www.uclibc.org/)	298	903	$7.503 \cdot 10^{50}$
ToyBox 0.5.2 (http://landley.net/toybox/)	544	1,020	$1.450 \cdot 10^{17}$
BusyBox 1.23.2 (https://busybox.net/)	613	530	$7.428 \cdot 10^{146}$
EmbToolkit 1.7.0 (https://www.embtoolkit.org/)	2,331	6,437	$3.961 \cdot 10^{334}$
LargeAutomotive [41]	17,365	321,897	$5.260 \cdot 10^{1,441}$

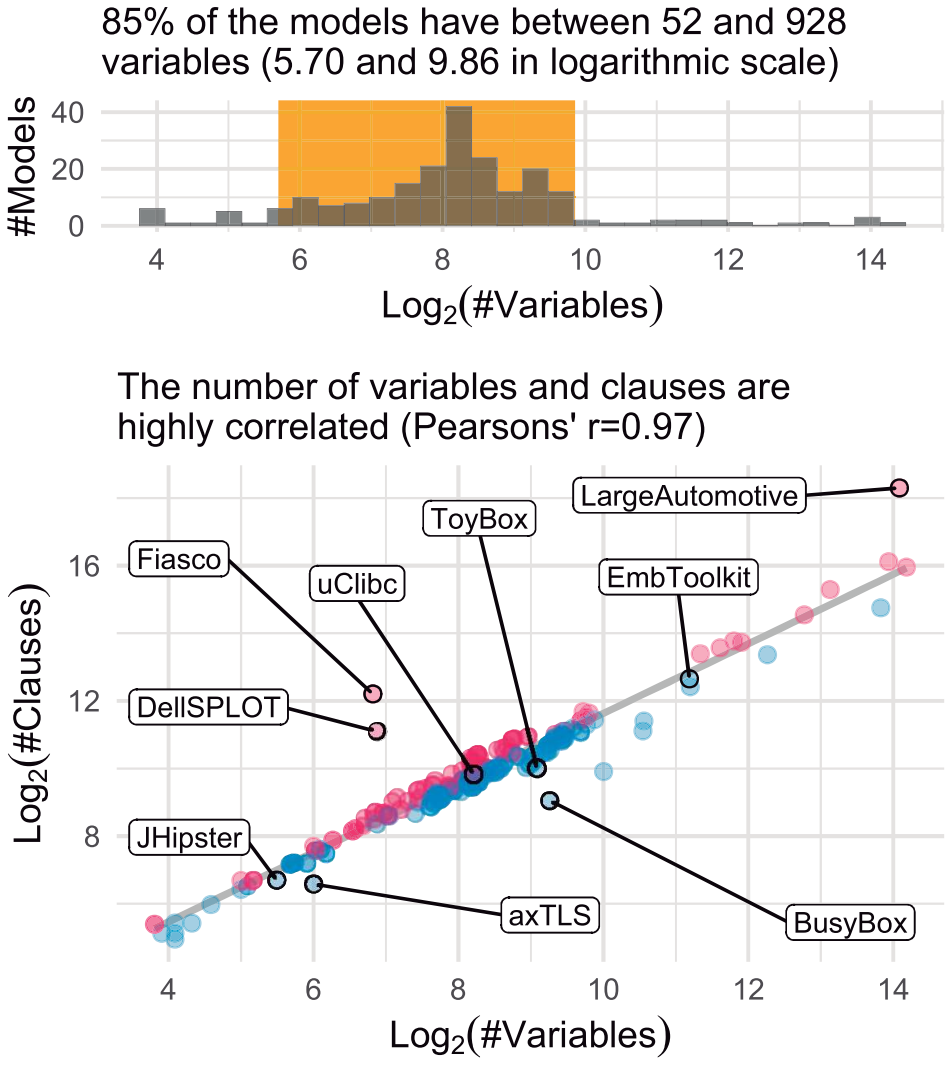

Fig. 10 Size of the benchmark models in terms of the number of variables and clauses.

5.2 Q1: Scalability of Samplers

The following experiment E1 was undertaken to obtain a sample set S1 for answering Q1. Each sampler generated a sample with one thousand configurations for every model in the benchmark. The timeout for each sample generation was set to one hour. The histogram in Fig. 11 shows the percentage of samples that each sampler was able to generate. In total, 257.92 hours (10.75 days) of CPU time were needed for generating the samples (or reaching the timeout).

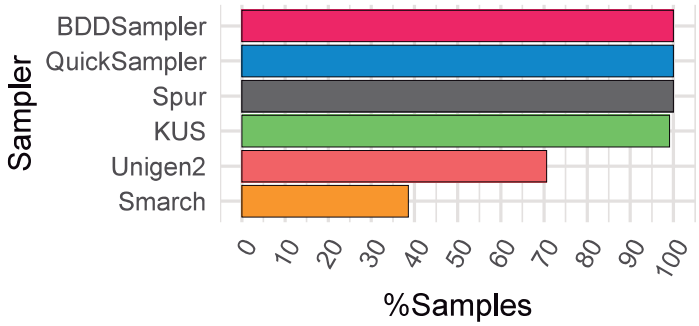

Fig. 11 Percentage of samples that each sampler was able to generate (sample size = 1,000 configurations; timeout = 1 hour).

Table 3 Sample generation time in seconds for the configuration models (sample size = 1,000 configurations; timeout = 1 hour).

Model	BDD Sampler	KUS	Quick Sampler	Smarch	Spur	Unigen2
JHipster	0.04	0.27	0.07	911.08	0.03	3.59
axTLS	0.04	0.34	0.20	1,993.90	0.03	timeout
Fiasco	0.07	0.45	1.47	timeout	0.06	timeout
DellSPLOT	0.08	0.44	0.44	3,278.09	0.07	187.58
uClibc	0.14	0.99	0.50	timeout	0.23	timeout
ToyBox	0.25	1.25	0.78	timeout	0.09	timeout
BusyBox	0.26	1.87	0.67	timeout	0.17	timeout
EmbToolkit	2.61	timeout	4.62	timeout	9.15	timeout
LargeAutomotive	12.07	119.26	77.06	timeout	24.57	timeout

5.3 Q2: Uniformity of Samplers

The following experiment E2 was carried out to obtain a sample set S2 for answering Q2, Q3, and Q5. Each sampler was run to generate a sample for every model in the benchmark. As Section 5.4 will explain in detail, the number of configurations per sample was estimated for $\alpha = 0.01, \beta = 0.01$, and $w = 0.1$. The timeout for each

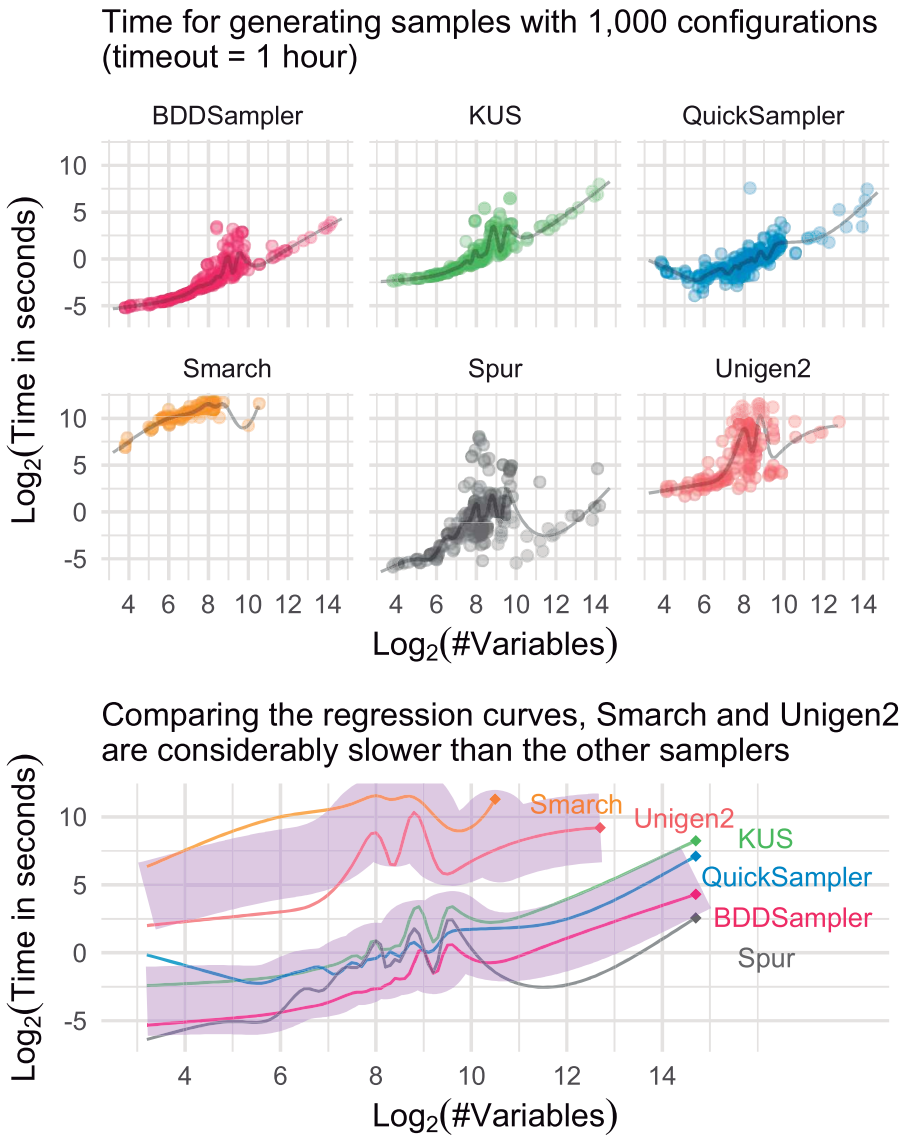

Fig. 12 Time the samplers needed to generate 1,000 configurations for each model in the benchmark.

sample generation was set to one hour. In total, 373.5 hours (15.56 days) of CPU time were needed for generating the samples (or reaching the timeout). The histogram in Fig. 13 summarizes the results. Nearly all samples produced by BDDSampler, Smarch, Spur, and Unigen2 obtained high p-values in the range $(0.9, 1]$. In contrast, KUS and QuickSampler generated many samples with p-values in the interval $[0, 0.1]$. Since α is set to 0.01, remember from Section 4.2 that a p-value less or equal to 0.01 means rejecting the uniformity hypothesis. Likewise, a p-value close to 1 reflects that the sample greatly supports the uniformity hypothesis. Table 4 summarizes the p-values for the configuration models in detail.

KUS and Spur implement Knuth's sampling procedure (see Section 2.1.3). Accordingly, they should be uniform "by design". Moreover, the KUS and Spur empirical validations in [59] and [2], respectively, did not detect any problem (though only small models with a few hundred variables were used). However, our inspection using more varied and larger models revealed the following uniformity flaws:

– As Fig. 13 shows, 16.4% of the KUS samples got a p-value in $[0, 0.1]$. Furthermore, in 15.89% of the cases, the p-values were less than $\alpha = 0.01$, and thus rejected the uniformity hypothesis. Fig. 14 shows four examples were KUS uni-

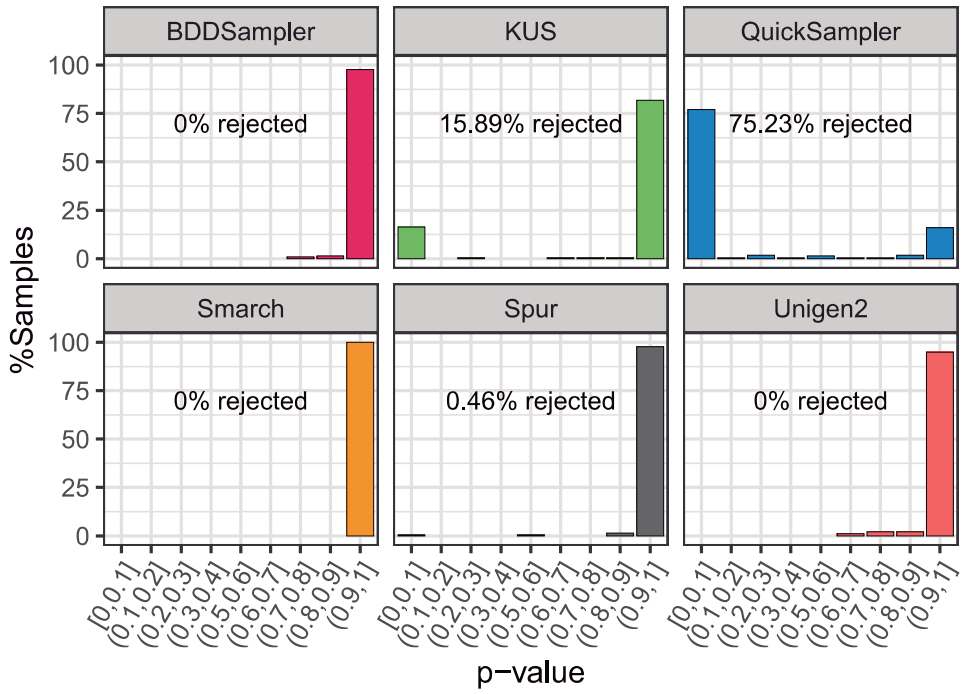

Fig. 13 Goodness-of-fit test results for the whole benchmark ($\alpha = \beta = 0.01$ and $w = 0.1$).

Table 4 Goodness-of-fit p-values for the configuration models ($\alpha = \beta = 0.01$ and $w = 0.1$; timeout = 1 hour).

Model	BDD Sampler	KUS	Quick Sampler	Smarch	Spur	Unigen2
JHipster	~ 1	0.99	~ 0	~ 1	~ 1	~ 1
axTLS	~ 1	~ 1	~ 0	timeout	~ 1	timeout
Fiasco	~ 1	~ 1	0.30	timeout	~ 1	timeout
DellSPLOT	~ 1	0.99	0.85	timeout	~ 1	0.96
uClibc	~ 1	~ 1	~ 1	timeout	~ 1	timeout
ToyBox	~ 1	~ 1	~ 0	timeout	~ 1	timeout
BusyBox	~ 1	~ 1	~ 0	timeout	~ 1	timeout
EmbToolkit	~ 1	timeout	~ 1	timeout	~ 0	timeout
LargeAutomotive	~ 1	timeout	~ 0	timeout	~ 1	timeout

formity was rejected. Each subfigure compares, for a particular model, the histogram of the SAT-solution distribution of the whole population (in blue) with the distribution of the generated sample (in red). Unfortunately, the rejected samples do not show any clear pattern that explains the causes of KUS failures. For instance, KUS exhibits difficulties with small models (blasted_case63) but also with large ones (blasted_squaring26), with normal distributions (blasted_case63 and s1238a_7_4) and non-normal distributions (s526_15_7 and blasted_squaring26), with left-skewed distributions (s1238a_7_4) and right-skewed distributions (blasted_case63), etc.

– In our previous evaluation [29], we detected that Spur generated uniform samples for all models except for EmbToolkit. We thought our test was making a Type 1 error, misjudging the sampler uniformity because an extremely low p-value hap-

pened due to randomness. However, when we checked the samplers' uniformity with our new test, we obtained exactly the same results for this particular model, which raised our suspicions. We repeated the experiment one thousand times and Spur never generated a uniform sample for EmbToolkit. Fig. 15 shows the results for two of those experiment repetitions. In this case, Spur's error always displays the same pattern: the solutions in the sample have more variables assigned to true than in the population.

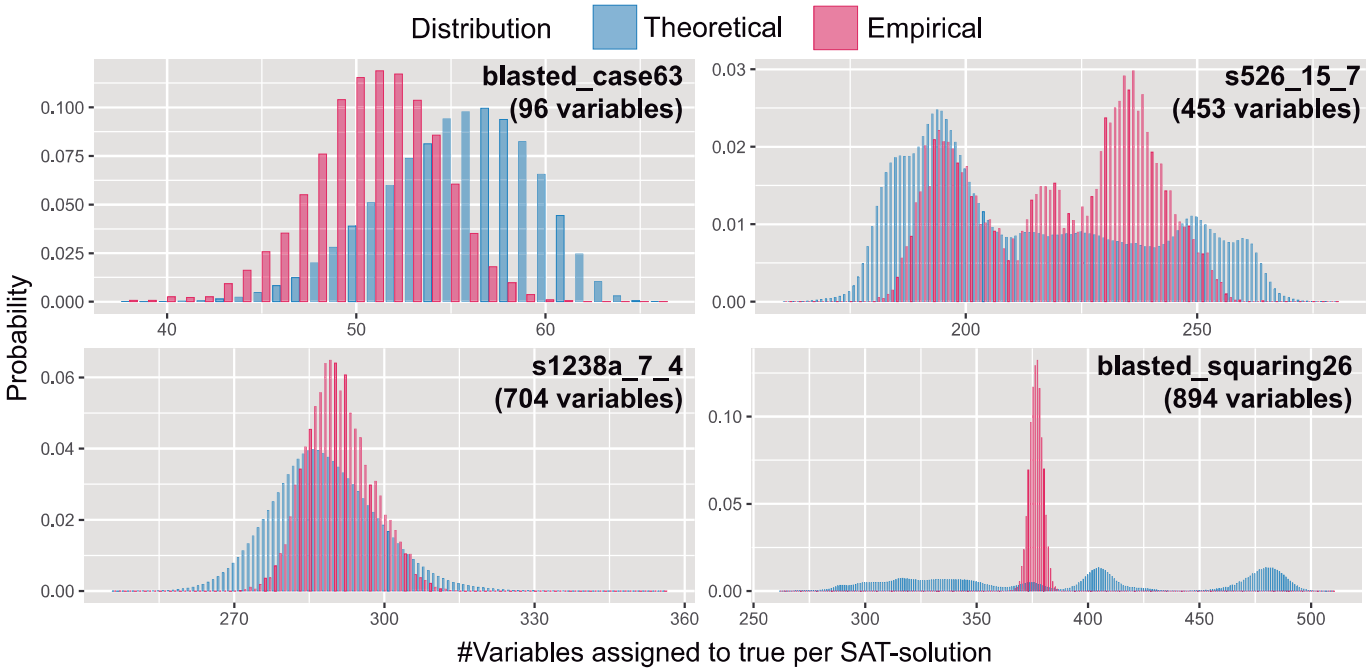

Fig. 14 Example of KUS samples rejected with the goodness-of-fit test.

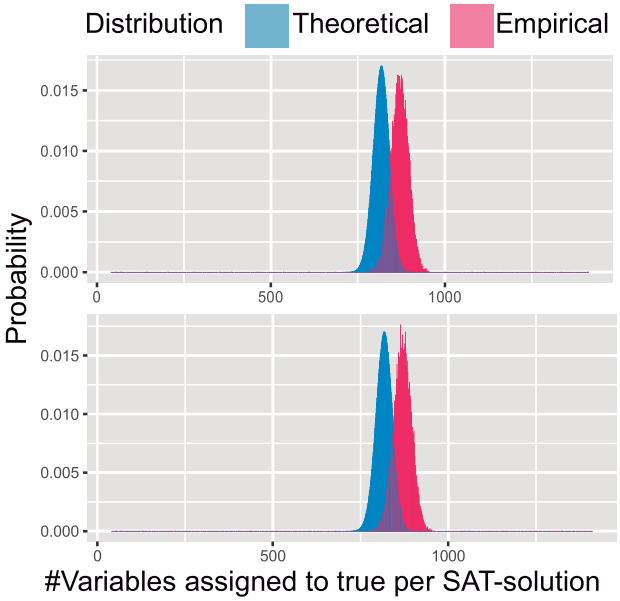

Fig. 15 Two samples Spur generated for EmbToolkit.

5.4 Q3: Scalability of the SFpC test

Two factors influence the scalability of a uniformity test when applied to a particular model and sampler: (i) the number of configurations the test needs to consider, and (ii) the time the test invests in analyzing those configurations.

Concerning the first factor, and as discussed in Section 2, the methods proposed in the literature to verify samplers' uniformity require colossal sample sizes with millions of configurations. Thus uniformity had been tested on trivial models so far, with a few hundred variables. To support evaluating uniformity over more complex models, in [29] we proposed the FP test, which compares the variable frequency distribution of a sample with the variable probability distribution of the entire population. With this test, we could validate samplers' uniformity on models with more than seventeen thousand variables [29]. Fig. 16 compares the sample sizes that the FP test needs (in red) with the sample sizes our new SFpC test requires (in blue), showing that the latter needs fewer configurations in most cases.

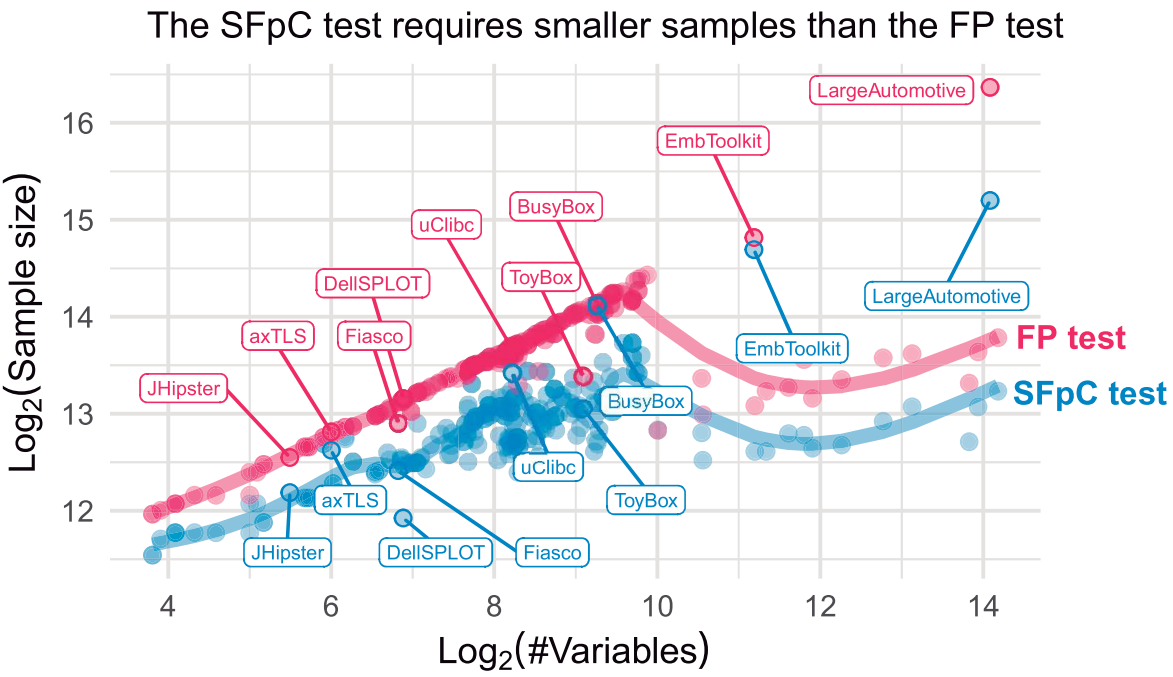

Fig. 16 Comparison of the sample sizes consumed by the FP and SFpC tests ($\alpha = \beta = 0.01$ and $w = 0.1$).

In Fig. 16, each model's sample size was determined with the procedure described in Section 4.2. In particular, the R package pwr[24] [33] was used to perform Cohen's power tables calculations. To ensure the highest reliability of the samplers' uniformity tests (see Section 5.4), we set $\alpha = 0.01, \beta = 0.01$, and $w = 0.1$. That is, the χ^2 test confidence level was fixed to 99%, the power to 99%, and the effect size to *small*. Table 5 compares in detail the samples sizes obtained for the configuration models.

[24] https://cran.r-project.org/web/packages/pwr

Table 5 Sample sizes the SFpC and FP tests required for the configuration models ($\alpha = \beta = 0.01$ and $w = 0.1$).

Model	Sample size for the SFpC test	Sample size for the FP test
JHipster	4,664	5,994
axTLS	6,314	7,198
Fiasco	5,460	7,646
DellSPLOT	3,889	9,131
uClibc	10,987	13,047
ToyBox	8,517	10,739
BusyBox	17,738	18,041
EmbToolkit	26,482	28,866
LargeAutomotive	37,626	84,522

The sample size depends on the model's degrees of freedom in both the FP and the SFpC tests. Nevertheless, each test defines *degrees of freedom* in a different way. The degrees of freedom of the FP test df_{FP} are the number of variables (minus one) whose probability is neither zero nor one (see Section 3 of [29]). The degrees of freedom of the SFpC test df_{SFpC} are the number cases (minus one) for which there is at least one valid configuration with a particular number of variables assigned to true (see Section 4.1). As Fig. 16 shows, in practice, $df_{SFpC} \leq df_{FP}$ and therefore the SFpC test consumes fewer configurations.

Regarding the time SFpC requires to analyze the generated configurations, once the theoretical distribution of SAT solutions is known, the remaining computations can be performed extremely fast (see Section 4.1). So, the SFpC's potential bottleneck is getting such distribution with the algorithm PD. Fig. 17 shows the time it took to compute the theoretical distribution for each model in the benchmark, ranging from 0.02 seconds to 14.14 minutes. Table 6 details the times for the configuration models. It is worth noting that the model which needed the longest time was `s1196a_3_2`, which is an industrial SAT formula (thus not included in Table 2). This illustrates the dependency that BDDs have on variable ordering heuristics. Whereas this model has a medium-size CNF formula (690 variables and 1,805 clauses), the BDD we synthesized was huge (2,284,697 nodes). In contrast, for LargeAutomotive (17,365 variables and 321,897) a more reduced BDD was obtained (30,432 nodes), and hence computing its theoretical SAT-solution distribution just took 2.74 minutes.

5.5 Q4: Validity of SFpC

Two criteria are typically used for assessing measurement quality [63]: *validity* and *reliability*. Since we are interested in the quality of SFpC measurements, *validity* will refer to what extent SFpC actually measures uniformity, and *reliability* will refer to repeatability, i.e., to the consistency of the results obtained when SFpC is applied

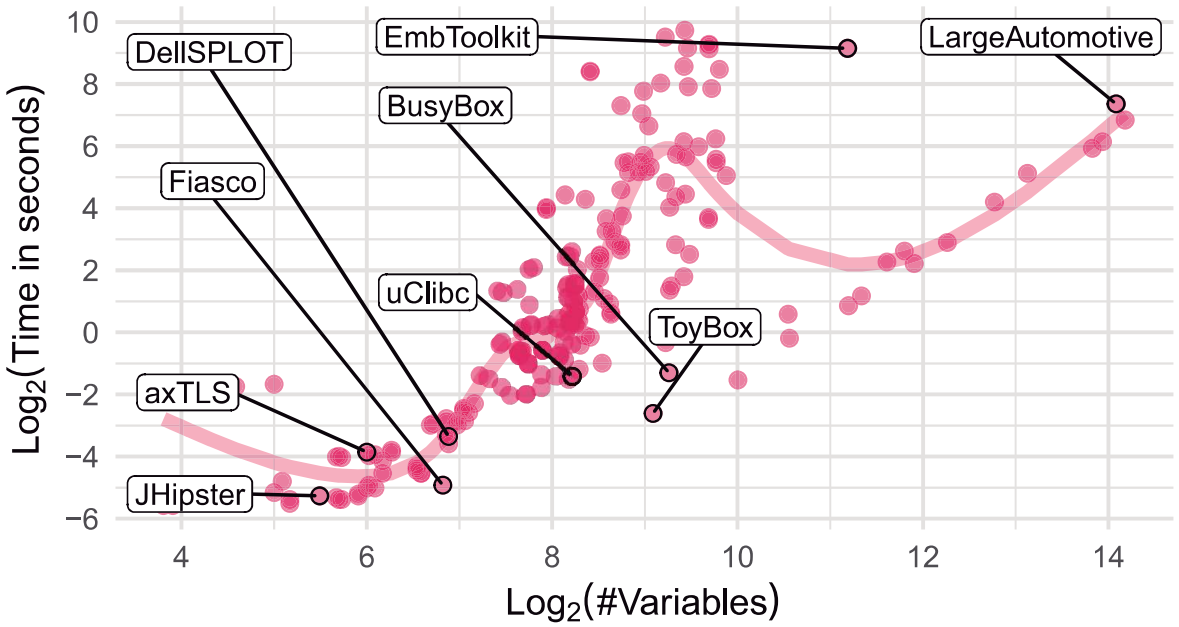

Fig. 17 Time it took to compute the distribution of SAT-solutions for all models in the benchmark.

Table 6 Seconds it took to compute the distribution of SAT-solutions for the configuration models.

Model	Time
JHipster	0.03
axTLS	0.07
Fiasco	0.03
DellSPLOT	0.10
uClibc	0.38
ToyBox	0.16
BusyBox	0.41
EmbToolkit	567.93
LargeAutomotive	164.35

several times to the same sampler and model. This section examines SFpC's validity, and the next section deals with SFpC's reliability.

To evaluate SFpC's validity, we followed a *convergent strategy* [63] by examining the degree to which SFpC results are similar to those obtained by other uniformity tests. Table 7 summarizes the uniformity verdicts reported in the literature. There is a total consensus that Unigen2 is uniform and QuickSampler is not. SFpC results are consistent with this consensus.

As we mentioned in Section 5.4, before the publication of FP in [30], the literature relied on limited tests that only could handle the simplest models with a few hundred variables. As more complex are considered, the chances to detect samplers' additional flaws increases. In other words, the sensitivity of FP and SFpC is higher than their predecessors. Accordingly, we performed a new Experiment E3 focused on checking the convergent validity of FP and SFpC in detail. A new sample set S3 was procured by asking each sampler to generate a sample for every model in the benchmark. Then, the uniformity of the samples was analyzed with both FP and SFpC.

Since FP generally needs larger samples than SFpC (see Section 5.4), the sample sizes were set according to FP requirements.

Pearson's correlation coefficient ρ of the p-values obtained with FP and SFpC was $\rho = 0.953$, and Cohen's kappa κ of the test verdicts (i.e., rejection/acceptance of sampler's uniformity) was $\kappa = 0.942$. As FP and SFpC results were numerically highly correlated, and their final judgments were remarkably consistent, convergent validity was successfully confirmed.

Table 7 Samplers' uniformity judgments reported in the literature. Due to its higher sensitivity compared to prior tests, SFpC detects that KUS and Spur sometimes behave non-uniformly.

Article	Uniform	Non-Uniform
Achlioptas et al. [2]	Spur and Unigen2	-
Chakraborty et al. [11]	Unigen2	-
Chakraborty et al. [12]	Unigen2	QuickSampler
Oh et al. [54]	Smarch and Unigen2	-
Plazar et al. [57]	Unigen	QuickSampler
Sharma et al. [59]	KUS and Spur	-
This present paper (SFpC)	BDDSampler, Smarch and Unigen2	KUS, QuickSampler and Spur

5.6 Q5: Reliability of SFpC

SFpC's reliability was evaluated with a *test-retest* strategy [63] by comparing its results with the sample sets S2 and S3. Pearson's correlation coefficient of the p-values calculated with SFpC in S2 and S3 was $\rho = 0.950$, and Cohen's kappa of the corresponding final judgments (i.e., rejection/acceptance of sampler's uniformity) was $\kappa = 0.939$. As a result, SFpC's reliability was positively evaluated.

6 Discussion

The experimental results indicate that SFpC supports testing samplers' uniformity on complex models with thousands of variables and constraints, providing valid and reliable judgments. The results show that the only sampler that satisfies both scalability and uniformity is BDDSampler. The following points summarize the **key findings** per research question:

Q1: Samplers' scalability. BDDSampler, KUS, QuickSampler, and Spur are by far faster than Smarch and Unigen2. This finding agrees with the prior evaluations reported by Plazar et al. [57] and Heradio et al. [29].

Q2: Samplers' uniformity. Three categories of samplers can be distinguished: (i) those that mostly fail to produce uniform samples (QuickSampler), (ii) those

that usually work but from time to time generate non-uniform samples (KUS and Spur), and (iii) those that always produce uniform samples (BDDSampler, Smarch, and Unigen2). QuickSampler's incapacity to generate uniform samples was previously reported by Chakraborty et al. [12], Plazar et al. [57], and Heradio et al. [29]. However, this paper is the first one that detects problems with KUS and Spur. We think this finding is due to SFpC's ability to test samplers' uniformity on considerably more complex models than previous tests.

Q3: SFpC's scalability. SFpC is the most scalable uniformity test to date. It requires the smallest sample size of all existing tests, enabling the verification of samplers' uniformity in large models even for the most strict quality settings ($\alpha = 0.01, \beta = 0.01$, and $w = 0.1$).

Q4: SFpC's validity. According to the results, SFpC judgments are consistent with the verdicts given by the alternative methods proposed in the literature.

Q5: SFpC's reliability. The results show that SFpC judgments are reliable, i.e., when SFpC is applied repeatedly to the same model and sampler, the reached conclusions are notably consistent.

The **implications** of our research are twofold:

1. As uniform random sampling is a strong requirement for many relevant analyses on configurable systems, BDDSampler's positive impact may be considerable, e.g., to test SPLs [28,57], to support predicting and optimizing the performance of configurable systems [53,34], etc. As an illustrative example of the importance that sampling has to SPL practitioners, in the SPLC 23rd edition, there was a *challenge* dedicated specifically to this topic and entitled "Product Sampling for Product Lines: The Scalability Challenge" [56]. Moreover, different papers have been recently published on uniform random sampling, and other sorts of sampling such as t-wise, in SPLC [64,54,48] and the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS) [40].
   Furthermore, the applicability of BDDSampler goes beyond the SPL domain since sampling is also needed in artificial intelligence [12,21,58], integrated circuit simulation and verification [31,51,68], etc.
2. SFpC can be used to debug and thus improve existing samplers (see Figures 14 and 15), or to validate future samplers. The importance of samplers' validation is well recognized by the SPL community. Recently, in the SPLC 25th edition, there was a session dedicated to "Sampling, variability analysis and visualization", where two tools for samplers' evaluation were presented: BURST [1] and AutoSMP [55]. Those tools could be enhanced by integrating SFpC; e.g., BURST relies on Barbarik, which has inferior performance than SFpC (see Section 2.2.2). Again, the interest in samplers' validation is not restricted to SPLs. In fact, most uniformity tests have been proposed by artificial intelligence researchers, mainly from the SAT community [2,11,21,59,12].

It is worth noting that our work has the following **limitation**: both BDDSampler and SFpC rely on BDD technology. Synthesizing the BDD encoding of a variability model is sometimes unattainable. This is because the variable ordering chosen to build a BDD dramatically impacts its size, and finding the optimal ordering is an NP-

problem. So the search is approached heuristically without guarantees. This problem principally affects BDDSampler, but not much SFpC.

1. BDDSampler receives a model's BDD encoding as input. If the available heuristics fail to find an adequate variable ordering, BDDSampler becomes useless, and an alternative technology (e.g., SAT) must be used.
2. SFpC can evaluate any sampler, independently of the technology in which it is built. Indeed, Section 5 reports the SFpC use for samplers implemented with BDDs (BDDSampler), #SAT-solvers (QuickSampler, Smarch, Spur, and Unigen2), and d-DNNFs (KUS). The impossibility of creating a BDD for a particular model only prevents SFpC from using it as part of the benchmark presented in Section 5.1. Currently, the benchmark includes 218 models with their respective BDDs. In our opinion, the models' great variety in terms of size (from 14 to 18,570 variables) and application domain (automotive industry, embedded systems, a laptop customization system, a web application generator, integrated circuits, etc.) is adequate to ensure samplers' verification to a great degree.

Finally, the following **threats to our study's validity** should be taken into account:

1. There is no absolute guarantee that the samplers we have certified as uniform behave non-uniformly in models not included in the benchmark.
2. Our experimental design discards two potential confounders for evaluating the scalability of samplers:
   - *Sampling parallelization*. Although any sampler can be run in a multi-core fashion, thus producing samples concurrently, only Unigen2 and Smarch were specifically designed for that. The focus of our evaluation is on the sampling techniques, not on how those techniques can be parallelized efficiently. Therefore, all samplers were run on a single thread.
   - *Use of preprocessing techniques*. There are some methods to preprocess the model Boolean formulas for speeding up further computations. For example, Ivrri et al. [32] claim that sampling with the formulas' MIS produces 2-3 orders of magnitude performance improvement. Nevertheless, Plazar et al. [57] empirical results contradict that, showing no running time difference between sampling from the whole formula or the MIS. Anyway, we decided to focus on the sampling techniques, not on how any additional preprocessing methods may impact those techniques.

7 Conclusions

The number of SAT solutions that configuration models encompass can be so large that most analyses cannot be performed neither examining every valid configuration, nor calling a SAT solver massively. Statistical inference opens an alternative way to address these problems by working with a tractable sample accurately predicts results for the entire space. However, the laws of statistical inference impose an indispensable requirement that samples must be collected at random, i.e., the configuration space needs to be covered uniformly.

Two major research challenges on SAT-solution random sampling have been addressed in this paper: we (i) developed a new random sampler, called BDDSampler, and (ii) proposed a goodness-of-fit test to verify samplers' uniformity. Our new test requires the least sample size of all existing methods, in the literature, supporting the samplers' uniformity assessment even on colossal models and the most strict reliability arrangements. Using this test, we have undertaken the empirical evaluation of six state-of-the-art samplers, revealing that only BDDSampler satisfies both uniformity and scalability.

It is worth remarking that BDDSampler works with a BDD encoding of a configuration model as input, and synthesizing such BDD is not always feasible as it depends on finding an adequate variable order heuristically. Our work deals with this limitation by exposing uniformity bugs on two scalable samplers based on alternative technologies (KUS on d-DNNFs and Spur on #SAT), thus facilitating their fixing. Having available all these samplers would support coping with the variated difficulties that the Boolean encoding of configuration models poses (e.g., large intractable CNFs, enormous BDDs, etc.).

8 Material

Following *open science*'s good practices, our software artifacts are available publicly.

- BDDSampler is available at https://github.com/davidfa71/BDDSampler
- The code scripts to replicate our experimental validation (i.e., to calculate each model's sample size, run the samplers, and test the scalability/uniformity of the samplers) are available at https://github.com/rheradio/ConfSystSampling
- A detailed report on every research question in Section 5 is available at: https://rheradio.github.io/ConfSystSampling
- The data of Experiments E1 and E2 (including the benchmark models in DIMACS/DDDMP/NNF formats, the generated samples, the goodness-of-fit test results, etc.) are available at https://doi.org/10.5281/zenodo.4514919
- The data of Experiment E3 are available at https://doi.org/10.5281/zenodo.5509947

Acknowledgements This work has been partially funded by the Universidad Nacional de Educacion a Distancia (project OPTIVAC 096-034091 2021V/PUNED/008); the Spanish Ministry of Science, Innovation and Universities (project OPHELIA RTI2018-101204-B-C22 and network TASOVA TIN2017-90644-REDT); the Community of Madrid (network ROBOCITY2030-DIH-CM S2018/NMT-4331); and the Junta de Andalucia (METAMORFOSIS project).

References

1. Acher, M., Perrouin, G., Cordy, M.: BURST: A Benchmarking Platform for Uniform Random Sampling Techniques. In: 25th Systems and Software Product Line Conference (SPLC). Leicester, United Kingdom (2021)
2. Achlioptas, D., Hammoudeh, Z.S., Theodoropoulos, P.: Fast sampling of perfectly uniform satisfying assignments. In: 21st International Conference on Theory and Applications of Satisfiability Testing (SAT), pp. 135–147. Oxford, UK (2018)

3. Alférez, M., Acher, M., Galindo, J., Baudry, B., Benavides, D.: Modeling variability in the video domain: language and experience report. Software Quality Journal **27**(1), 307–347 (2019)
4. Alves Pereira, J., Acher, M., Martin, H., Jézéquel, J.M.: Sampling effect on performance prediction of configurable systems: A case study. In: ACM/SPEC International Conference on Performance Engineering (ICPE), p. 277–288. Edmonton AB, Canada (2020)
5. Batory, D.S.: Feature models, grammars, and propositional formulas. In: 9th Software Product Line Conference (SPLC), pp. 7–20. Rennes, France (2005)
6. Bellare, M., Goldreich, O., Petrank, E.: Uniform Generation of NP-Witnesses Using an NP-Oracle. Information and Computation **163**(2), 510–526 (2000)
7. Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K.: A Study of Variability Models and Languages in the Systems Software Domain. IEEE Transactions on Software Engineering **39**(12), 1611–1640 (2013)
8. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press (2009)
9. Brace, K.S., Rudell, R.L., Bryant, E.: Variability modeling in the real: a perspective from the operating systems domain. In: IEEE/ACM Design Automation Conference (DAC), pp. 40–45. Orlando, Florida, USA (1990)
10. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (1986)
11. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform SAT witness generation. In: 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 304–319. London, UK (2015)
12. Chakraborty, S., Meel, K.S.: On testing of uniform samplers. In: 33rd Conference on Artificial Intelligence (AAAI), pp. 7777–7784. Honolulu, Hawaii, USA (2019)
13. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A Scalable and Nearly Uniform Generator of SAT Witnesses. In: 25th International Conference on Computer Aided Verification (CAV), pp. 608–623. Saint Petersburg, Russia (2013)
14. Chihara, L.M., Hesterberg, T.C.: Mathematical Statistics with Resampling and R. Wiley (2011)
15. Chollet, F., Allaire, J.: Deep Learning with R. Manning Publications (2018)
16. Cohen, J.: Statistical Power Analysis for the Behavioral Sciences. Routledge (1988)
17. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley (2006)
18. D'Agostino, R.B., Stephens, M.A.: Goodness-of-Fit-Techniques. CRC Press (1986)
19. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-proving. Communications of the ACM **5**(7), 394–397 (1962)
20. van Dijk, T.: Sylvan: multi-core decision diagrams. Ph.D. thesis, University of Twente (2016)
21. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient Sampling of SAT Solutions for Testing. In: 40th International Conference on Software Engineering (ICSE), pp. 549–559. ACM, New York, NY, USA (2018)
22. Faul, F., Erdfelder, E., Lang, A.G., Buchner, A.: G*Power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. Behavior Research Methods **2**(39), 175–191 (2007)
23. Fernandez-Amoros, D., Bra, S., Aranda-Escolastico, E., Heradio, R.: Using Extended Logical Primitives for Efficient BDD Building. Mathematics **8**(8), 1–17 (2020)
24. Fernandez-Amoros, D., Heradio, R., Mayr-Dorn, C., Egyed, A.: A Kconfig translation to logic with one-way validation system. In: 23rd International Systems and Software Product Line Conference (SPLC), pp. 303–308. Paris, France (2019)
25. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. The MIT Press (2016)
26. Grosse, I., Bernaola-Galvan, P., Carpena, P., Roman-Roldan, R., Oliver, J., Stanley, H.E.: Analysis of symbolic sequences using the Jensen-Shannon divergence. Physical Review E **65**(2), 041905/1–0419051/16 (2002)
27. Guo, J., Yang, D., Siegmund, N., Apel, S., Sarkar, A., Valov, P., Czarnecki, K., Wasowski, A., Yu, H.: Data-efficient performance learning for configurable systems. Empirical Software Engineering **23**(3), 1826–1867 (2018)
28. Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.: Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. Empirical Software Engineering **24**(2), 674–717 (2019)
29. Heradio, R., Fernandez-Amoros, D., Galindo, J.A., Benavides, D.: Uniform and scalable SAT-sampling for configurable systems. In: 24th Systems and Software Product Line Conference (SPLC), pp. 1–11. Montréal, Canada (2020)

30. Heradio, R., Fernandez-Amoros, D., Mayr-Dorn, C., Egyed, A.: Supporting the statistical analysis of variability models. In: 41st International Conference on Software Engineering (ICSE), pp. 843–853. Montréal, Canada (2019)
31. Hübner, M., Becker, J.: Multiprocessor System-on-Chip: Hardware Design and Tool Integration. Springer (2011)
32. Ivrii, A., Malik, S., Meel, K.S., Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting. Constraints **21**(1), 41–58 (2016)
33. Kabacoff, R.: R in Action: Data analysis and graphics with R. Manning Publications (2011)
34. Kaltenecker, C., Grebhahn, A., Siegmund, N., Apel, S.: The interplay of sampling and machine learning for software performance prediction. IEEE Software **37**(4), 58–66 (2020)
35. Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., Apel, S.: Distance-based sampling of software configuration spaces. In: 41st International Conference on Software Engineering (ICSE), pp. 1084–1094. Montreal, Canada (2019)
36. Kaplan, D.: Statistical Modeling: A Fresh Approach. Project Mosaic (2012)
37. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional (2009)
38. Kolesnikov, S., Siegmund, N., Kästner, C., Grebhahn, A., Apel, S.: Tradeoffs in modeling performance of highly configurable software systems. Software & Systems Modeling **18**(3), 2265–2283 (2019)
39. Krieter, S.: Enabling Efficient Automated Configuration Generation and Management. In: 23rd International Systems and Software Product Line Conference (SPLC), pp. 215–221. Paris, France (2019)
40. Krieter, S., Thüm, T., Schulze, S., Saake, G., Leich, T.: Yasa: Yet another sampling algorithm. In: 14th International Working Conference on Variability Modelling of Software-Intensive System (VaMoS), pp. 1–10. Magdeburg, Germany (2020)
41. Krieter, S., Thüm, T., Schulze, S., Schröter, R., Saake, G.: Propagating configuration decisions with modal implication graphs. In: 40th International Conference on Software Engineering (ICSE), pp. 898–909. Gothenburg, Sweden (2018)
42. Lagniez, J.M., Marquis, P.: An Improved Decision-DNNF Compiler. In: 26th International Joint Conference on Artificial Intelligence (IJCAI). Melbourne, Australia (2017)
43. Lakens, D.: Calculating and reporting effect sizes to facilitate cumulative science: a practical primer for *t*-tests and ANOVAs. Frontiers in Psychology **4**(863), 1–12 (2013)
44. Lin, J.: Divergence measures based on the shannon entropy. IEEE Transactions on Information Theory **37**(1), 145–151 (1991)
45. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design: OBDD - Foundations And Applications. Springer (1998)
46. Mendonça, M.: Efficient Reasoning Techniques for Large Scale Feature Models. Ph.D. thesis, University of Waterloo (2009)
47. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 337–340. Budapest, Hungary (2008)
48. Muñoz, D.J., Oh, J., Pinto, M., Fuentes, L., Batory, D.: Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In: 23rd International Systems and Software Product Line Conference (SPLC), pp. 289–301. Paris, France (2019)
49. Nair, V., Menzies, T., Siegmund, N., Apel, S.: Using Bad Learners to Find Good Configurations. In: 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 257–267. Paderborn, Germany (2017)
50. Narodytska, N., Walsh, T.: Constraint and variable ordering heuristics for compiling configuration problems. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 149–154. Hyderabad, India (2007)
51. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-Based Random Stimuli Generation for Hardware Verification. In: 18th Innovative Applications of Artificial Intelligence Conference (IAAI), pp. 1720–1727. Boston, Massachusetts, USA (2006)
52. Nöhrer, A., Egyed, A.: C2O configurator: a tool for guided decision-making. Automated Software Engineering **20**(2), 265–296 (2013)
53. Oh, J., Batory, D., Myers, M., Siegmund, N.: Finding Near-optimal Configurations in Product Lines by Random Sampling. In: 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 61–71. New York, NY, USA (2017)
54. Oh, J., Gazzillo, P., Batory, D.S.: *t*-wise coverage by uniform sampling. In: 23rd International Systems and Software Product Line Conference (SPLC), pp. 84–87. Paris, France (2019)

55. Pett, T., Krieter, S., Thüm, T., Lochau, M., Schaefer, I.: AutoSMP: An Evaluation Platform for Sampling Algorithms. In: 25th Systems and Software Product Line Conference (SPLC). Leicester, United Kingdom (2021)
56. Pett, T., Thüm, T., Runge, T., Krieter, S., Lochau, M., Schaefer, I.: Product sampling for product lines: The scalability challenge. In: 23rd International Systems and Software Product Line Conference (SPLC), pp. 78–83. Paris, France (2019)
57. Plazar, Q., Acher, M., Perrouin, G., Devroey, X., Cordy, M.: Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet? In: 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 240–251. Xian, China, China (2019)
58. Roy, S., Pandey, A., Dolan-Gavitt, B., Hu, Y.: Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In: 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 224–234. Lake Buena Vista, Florida, USA (2018)
59. Sharma, S., Gupta, R., Roy, S., Meel, K.S.: Knowledge Compilation meets Uniform Sampling. In: 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), pp. 620–636. Awassa, Ethiopia (2018)
60. van Solingen, R., Berghout, E.: The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development. McGraw-Hill (1999)
61. Temple, P., Galindo, J.A., Acher, M., Jézéquel, J.M.: Using machine learning to infer constraints for product lines. In: 20th International Systems and Software Product Line Conference (SPLC), pp. 209–218. Beijing, China (2016)
62. Thurley, M.: sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In: 9th International Conference on Theory and Applications of Satisfiability Testing (SAT), pp. 424–429. Seattle, WA, USA (2006)
63. Trochim, W.M., Donnelly, J.P., Arora, K.: Research Methods: The Essential Knowledge Base. Wadsworth Publishing (2015)
64. Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I.: A classification of product sampling for software product lines. In: 22md International Systems and Software Product Line Conference (SPLC), pp. 1–13. Gothenburg, Sweden (2018)
65. Vasishth, S., Broe, M.: The Foundations of Statistics: A Simulation-based Approach. Springer (2011)
66. Weckesser, M., Kluge, R., Pfannemüller, M., Matthé, M., Schürr, A., Becker, C.: Optimal reconfiguration of dynamic software product lines based on performance-influence models. In: 22nd International Systems and Software Product Line Conference (SPLC), pp. 98–109. Gothenburg, Sweden (2018)
67. Winter, B.: Statistics for Linguists: An Introduction Using R. Routledge (2020)
68. Yuan, J., Albin, K., Aziz, A., Pixley, C.: Simplifying Constraint Solving in Random Simulation Generation. In: 11th IEEE/ACM International Workshop on Logic & Synthesis (IWLS), pp. 185–190. New Orleans, Louisiana, USA (2002)