Designing Samplers is Easy: The Boon of Testers

Priyanka Golia
Indian Institute of Technology Kanpur
National University of Singapore

Mate Soos
National University of Singapore

Sourav Chakraborty
Indian Statistical Institute, Kolkata

Kuldeep S. Meel
National University of Singapore

***Abstract*—Given a formula φ, the problem of uniform sampling seeks to sample solutions of φ uniformly at random. Uniform sampling is a fundamental problem with a wide variety of applications. The computational intractability of uniform sampling has led to the development of several samplers that heavily rely on heuristics and are not accompanied by theoretical analysis of their distribution. Recently, Chakraborty and Meel (2019) designed the first scalable sampling tester, Barbarik, based on a grey-box sampling technique for testing if the distribution, according to which the given sampler is sampling, is close to the uniform or far from uniform. While the theoretical analysis of Barbarik provides only unconditional soundness guarantees, the empirical evaluation of Barbarik did show its success in determining that some of the off-the-shelf samplers were far from a uniform sampler.**

The availability of Barbarik has the potential to spur development of samplers techniques such that developers can design sampling methods that can be accepted by Barbarik even though these samplers may not be amenable to a detailed mathematical analysis. In this paper, we present the realization of this aforementioned promise. Based on the flexibility offered by CryptoMiniSat, we design a sampler CMSGen that promises the achievement of sweet spot of the quality of distributions and runtime performance. In particular, CMSGen achieves significant runtime performance improvement over the existing samplers. We conduct two case studies, and demonstrate that the usage of CMSGen leads to significant runtime improvements in the context of combinatorial testing and functional synthesis.

A salient strength of our work is the simplicity of CMSGen, which stands in contrast to complicated algorithmic schemes developed in the past that fail to attain the desired quality of distributions with practical runtime performance.

I. Introduction

Given a formula φ, the problem of uniform sampling seeks to sample solutions of φ uniformly at random. Uniform sampling has emerged as an essential technique in the context of constrained-random simulation [33], constraint-based fuzzing [5], [19], [22], configuration testing [13], [23], bug synthesis [36], and the like. For example, in the context of constrained-random simulation, uniform sampling is employed to generate test cases that satisfy the set of constraints encoding domain knowledge from sources such as designers, end-users, and the like.

The widespread applications of uniform sampling have led to several algorithmic proposals over the years with varying theoretical guarantees and empirical scalability. Chakraborty, Meel, and Vardi introduced the first practical almost-uniform sampler, UniGen [11], [12], which has since been improved to UniGen3 [9], [39]. Recently, Sharma et al. proposed a knowledge compilation-based approach [37], called KUS, that can perform uniform sampling. While UniGen3 and KUS can scale to hundreds of thousands of variables for some problems, their performance still falls short of the desired scale for some real-world instances. The need for scalability has led to the development of several tools that seek to achieve scalability at the cost of theoretical guarantees. The underlying techniques for such tools cover a broad spectrum ranging from adapted BDD-based techniques [26], random seeding of DPLL-based SAT solvers [32], Markov Chain Monte Carlo-based (MCMC) methods [24], [43], interval propagation and belief networks-based methods [14], [20], MaxSAT-based techniques [16].

The lack of guarantees for various samplers leads their designers to illustrate the quality of samples generated via computation of statistics for generated distributions over a small set of benchmarks. Such demonstrations, however, do not generalize to many classes of benchmarks, and it is often the case that subsequent studies tend to demonstrate cases where previously proposed samplers generate distributions far away from uniform. While the theoretical guarantees of uniformity can be viewed as a holy grail, much of the software engineering progress owes to the development of testing methodologies. These methodologies employed both to validate the system and find bugs by the developers themselves in the form of test-driven development (TDD) and to build trust with the end-users; all without requiring the developers to supply a formal proof of correctness.

A major contributing factor to the dramatic improvement in the robustness and scalability of SAT solvers has been the development of the DRAT proof format and associated proof checker drat-trim [44]. The availability of drat-trim allows SAT solver developers to find bugs that would be hard to discover owing to the complex architecture of state-of-the-art SAT solvers. While the problem of checking whether a given formula is UNSAT is *merely* Co-NP, the problem of testing whether a sampler is a uniform requires $\Omega(2^n)$ samples given black-box access to the sampler [3], [8], where n is the number of variables.

Recently, Chakraborty and Meel proposed the first scalable sampler test framework, Barbarik [8]. This framework distinguishes whether the distribution generated by the given sampler is ε-close to uniform (Accept) or η-far from uniform (Reject), while the number of samples required depends only

https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_31

This article is licensed under a Creative Commons Attribution 4.0 International License

on ε and η, and is independent of n. The core idea of the Barbarik is to reduce testing of uniformity over the entire solution space of φ to the testing of uniformity over solutions space of another formula, $\hat{\varphi}$ constructed over two randomly chosen solutions of φ (observe that $\hat{\varphi} \to \varphi$). The subroutine to construct $\hat{\varphi}$ is called Kernel. The analysis of Barbarik states that if Barbarik Rejects a sampler, the distribution generated by sampler is indeed (probabilistically) far from uniform, but if Barbarik Accepts a sampler, the sampler's distribution is close to uniform under the assumption of *non-adversality* with respect to Kernel. Informally, the *non-adversality* assumption with respect to Kernel dictates that given φ, the conditional distribution of the sampler over the solutions of $\hat{\varphi}$ is same as the distribution of the sampler with $\hat{\varphi}$ as input. Note that this allows some samplers to behave in an *adversarial* manner, i.e., such samplers may not generate uniform distribution over φ, however may generate uniform distributions for $\hat{\varphi}$. In such a case, causing Barbarik will return Accept for such samplers. At this point, it is worth remarking that given the strong lower bounds on black-box testing, the usage of such an assumption is a *practical* necessity.

Empirically, Barbarik was able to return Reject for all the state of the art samplers without rigorous mathematical analysis certifying (almost)-uniformity of the generated distributions. In particular, Barbarik was demonstrated to Accept UniGen3 while rejecting the state of the art samplers STS [18] and QuickSampler [16]. It is worth noting that the three samplers, UniGen3, QuickSampler, and STS, were found to be statistically indistinguishable by the usage of simple metrics such as KL-divergence [27] after a small number of samples.

The availability of Barbarik, however, has potential to allow development of samplers, whose algorithmic frameworks may not be amenable to mathematical analysis but can be accepted by Barbarik. The primary contribution of this paper is realization of the promise of Barbarik via development of a new state of the art sampler, CMSGen. In particular, we make following contributions:

A. CMSGen: A State of the Art Sampler

1) We design a new sampler, CMSGen, by modifying the existing state-of-the-art Conflict-Driven Clause Learning (CDCL) SAT solver CryptoMiniSat[1] [41].
2) Since understanding the behavior of CDCL itself is an open problem, we can not provide an unconditional analysis of the distribution produced by CMSGen. We rely on the availability of Barbarik, and observe that surprisingly, Barbarik returns Accept for all the benchmarks. Barbarik's failure to Reject CMSGen stands in sharp contrast to its ability to Reject other samplers without guarantees, such as QuickSampler. Furthermore, we perform empirical comparisons of runtime performance via-a-vis UniGen3, the state-of-the-art sampler with theoretical guarantees. We observe that CMSGen significantly improves upon UniGen3 in terms of runtime performance.

[1]Available at https://github.com/msoos/cryptominisat

B. Case Studies: Combinatorial Testing and Functional Synthesis

3) At this point, one may wonder whether there are practical applications of CMSGen. We next focus on applications that are beyond the reach of UniGen3, and for such cases, one has to rely on the heuristics-based samplers. In particular, we perform two case studies: (1) combinatorial testing, and (2) functional synthesis; two problems with a long history of sustained interest in formal methods and software engineering community. For both the case studies, we observe that the usage of CMSGen leads to significant performance improvements in comparison to usage of other competing samplers UniGen3 and QuickSampler.

It is worth remarking that a salient strength of CMSGen is the simplicity of its design. We find it exciting that a sampler with such a simple design could outperform sophisticated state of the art samplers. Based on our empirical analysis, one would remark that CMSGen aims to achieve the sweet spot of scalability and uniformity. In particular, CMSGen is significantly more scalable than samplers with guarantees and, at the time, achieves distributions of higher quality than samplers without guarantees. The runtime performance combined with the quality of distribution as certified by Barbarik makes CMSGen the ideal choice for applications such as combinatorial testing and functional synthesis where scalability and quality of distribution are equally crucial.

The rest of the paper is organized as follows: In Section II, we present the formal definitions and also present a brief description of the sampler verifier Barbarik. In Section III we present the new sampler CMSGen and in Section IV we present the evaluation of CMSGen both by comparing its runtime performance with other samplers and also its performance against Barbarik. Then in Section V we demonstrate the usefulness of CMSGen with two case studies on problems of fundamental importance to formal methods community: functional synthesis and combinatorial testing. Finally, we conclude in Section VI.

II. Notation and Background

A literal is a Boolean variable or its negation. Let φ be a Boolean formula in conjunctive normal form (CNF), and let X be the set of variables appearing in φ. The set X is called the *support* of φ, denoted by $Supp(\varphi)$. Given an array $\mathbf{a}$, $\mathbf{a}[i:j]$ represents the sub-array consists of all the elements of $\mathbf{a}$ between indices i and j. A *satisfying assignment* or *witness*, denoted by σ, is an assignment of truth values to variables in its support such that φ evaluates to true. A satisfying assignment is also represented as a set of literals. For $S \subseteq Supp(\varphi)$, we use $\sigma_{\downarrow S}$ to indicate the projection of σ over the set of variables S. We denote the set of all witnesses of φ as $sol(\varphi)$. For notational convenience, whenever the formula

φ and/or the set $S \subseteq Supp(\varphi)$ is clear from the context, we omit mentioning them.

A. Samplers

Definition 1: Given a Boolean formula φ, a *CNF-sampler* (or simply *sampler*) $\mathcal{G}$ of φ is a probabilistic algorithm that generates a random element in $sol(\varphi)$. We will assume that a sampler takes as input a CNF-formula φ, a set $S \subseteq Supp(\varphi)$ and an integer k. It generates k elements $\sigma_1, \ldots, \sigma_k$ from $sol(\varphi)$ and outputs $\sigma_{1\downarrow S}, \ldots, \sigma_{k\downarrow S}$. When the integer k and the set $S \subseteq Supp(\varphi)$ is clear from the context (or is not important) we will drop them and use $\mathcal{G}(\varphi)$ or $\mathcal{G}(\varphi, S)$ to denote the sampler.

We use $p_{\mathcal{G}}(\varphi, \sigma)$ (or $p_{\mathcal{G}}(\varphi, \sigma, S)$) to denote the probability that $\mathcal{G}(\varphi, \cdot, \cdot)$ (or $\mathcal{G}(\varphi, S, \cdot)$) generates σ (or $\sigma_{\downarrow S}$). And, we use $\mathcal{D}_{\mathcal{G}(\varphi)}$ (and $\mathcal{D}_{\mathcal{G}(\varphi,S)}$) to denote the distribution induced by $\mathcal{G}$ over the set $sol(\varphi)$ (and $sol(\varphi)_{\downarrow S}$). For a set $T \subseteq sol(\varphi)$, we use $\mathcal{D}_{\mathcal{G}(\varphi)}\downarrow T$ to denote the distribution $\mathcal{D}_{\mathcal{G}(\varphi)}$ conditioned on set T.

Definition 2: Given a Boolean formula φ, A *uniform sampler* $\mathcal{G}^u(\varphi)$ is a sampler that given φ guarantees

$$\forall y \in sol(\varphi), \mathsf{Pr}\left[\mathcal{G}^u(\varphi) = y\right] = 1/|sol(\varphi)|, \tag{1}$$

Definition 3: Given a Boolean formula φ and tolerance parameter ε, $\mathcal{G}^{AAU}(\varphi, \varepsilon)$ is an additive *almost-uniform generator* (AAU) if the following holds:

$$\forall y \in sol(\varphi), \frac{1-\varepsilon}{|sol(\varphi)|} \leq \mathsf{Pr}\left[\mathcal{G}^{AAU}(\varphi, \varepsilon) = y\right] \leq \frac{1+\varepsilon}{|sol(\varphi)|} \tag{2}$$

A sampler is allowed to occasionally "fail" in the sense that no element may be returned even if $sol(\varphi)$ is non-empty. The failure probability for such generators must be bounded by a constant strictly less than 1.

Definition 4: Given a Boolean formula φ and an intolerance parameter η an generator $\mathcal{G}(\varphi, .)$ is η-far from uniform generator if the ℓ_1-distance (or, twice the variation distance) of $\mathcal{D}_{\mathcal{G}(\varphi)}$ from uniform is at least η. That is,

$$\sum_{x \in sol(\varphi)} \left| p_{\mathcal{G}(\varphi,x)} - \frac{1}{|sol(\varphi)|} \right| \geq \eta$$

B. Sampler Tester

Given a sampler $\mathcal{G}$, one would like to test if the sampler is indeed correct. Or in other words, one would like to test the following:

1) Does the sampler always output a satisfying assignment?
2) On any CNF-formula φ, is $\mathcal{G}(\varphi)$ an additive almost-uniform generator?

While the first point is very easy to test, testing the second point is quite challenging. Standard verification techniques or black box sampling techniques would need exponential time/samples and thus are very inefficient.

Chakraborty and Meel [8] designed the tester Barbarik that would accept if the sampler is an additive almost-uniform generator on any input and reject if the sampler is far from a uniform generator on some input under certain assumptions discussed below. The idea of Barbarik comes from the world of property testing, where the sample complexity for testing whether a distribution is a uniform is studied. While it was known from classical sample complexity [3] that an exponential number of samples are required to distinguish a uniform distribution from a distribution that is η-from uniform, in [7] it was observed that if given access to conditional samples only a constant number of samples suffice. Conditional samples from a distribution $\mathcal{D}$ means for a subset T of the domain Ω, drawing samples from the conditional distribution $\mathcal{D}_{|T}$. The algorithm for checking whether a given distribution $\mathcal{D}$ over domain Ω is uniform or η-far from uniform, consists of following steps:

1) Draw one sample σ_1 according to the distribution $\mathcal{D}$.
2) Draw one sample σ_2 according to the uniform distribution over Ω.
3) Check if the distribution $\mathcal{D}_{|T}$ is uniform or "far"-from uniform, where $T = \{\sigma_1, \sigma_2\}$.

The last point of the above algorithm can be performed using only a constant number of conditional samples. It can also be shown that the above algorithm, with non-trivial probability, will Accept if $\mathcal{D}$ is uniform and Reject if $\mathcal{D}$ is η-far from uniform, by repeating this algorithm a certain number of times, one can boost the success probability.

While the algorithm is theoretically interesting, applying it to design a sampler test framework required several hurdles to cross. Firstly, for Step 2 of the algorithm, one needs to run a uniform sampler. This is not too much of a hurdle as one can use a non-efficient uniform sampler, since the sampler tester is only to be used a few times to certify if a sampler is good.

The second problem is that the algorithms, as such, could only distinguish between a uniform distribution, and a distribution "far" from a uniform distribution, while a sample tester should also Accept samplers that are "close" to uniform samplers (and not necessarily just uniform samplers).

Finally, the main concern was how to obtain conditional samples. In [8] this was achieved by constructing a new formula $\hat{\varphi}$ on a larger number of variables such that the satisfying assignments of $\hat{\varphi}$ restricted to the original set of variables is either σ_1 and σ_2. In fact if $S = Supp(\varphi)$, then

$$\Pr_{\sigma \sim \mathcal{U}(sol(\hat{\varphi}))}[\sigma_{\downarrow S} = \sigma_1] = \Pr_{\sigma \sim \mathcal{U}(sol(\hat{\varphi}))}[\sigma_{\downarrow S} = \sigma_2] = \frac{1}{2}$$

where $\mathcal{U}(sol(\hat{\varphi})$ denotes uniform distribution over $sol(\hat{\varphi})$ The new formula $\hat{\varphi}$ is obtained from φ by using a subroutine Kernel that uses the chain formula technique from [10].

The goal of the construction of $\hat{\varphi}$ is such that the following two conditions are satisfied:

1) If the sampler $\mathcal{G}(\varphi)$ was ϵ-additive almost-uniform generator then the distribution $\mathcal{D}_{\mathcal{G}(\hat{\varphi},S)}$ is "close" to the uniform distribution on the set $\{\sigma_1, \sigma_2\}$.
2) If the sampler $\mathcal{G}(\varphi)$ was η-far from the uniform sampler in the ℓ_1 distance then the distribution $\mathcal{D}_{\mathcal{G}(\hat{\varphi},S)}$ is "far" from the uniform distribution on the set $\{\sigma_1, \sigma_2\}$.

Now, if the sampler $\mathcal{G}$ is additive almost-uniform generator on any input φ the first condition would be satisfied. But

for the second condition to hold some more assumptions are necessary. This assumption is called the *non-adversarial assumption* in [8].

Definition 5: The **non-adversarial sampler assumption** states that if $(\hat{\varphi}, \hat{S})$ is the output obtained from $\mathsf{Kernel}(\varphi, S, \sigma_1, \sigma_2, N)$ then

- $S \subseteq \hat{S}$
- the output of $\mathcal{G}(\hat{\varphi}, S, N)$ is N independent samples from the conditional distribution $\mathcal{D}_{\mathcal{G}(\varphi,S)|_T}$, where $T = \{\sigma_1, \sigma_2\}$.

Thus Barbarik has the following guarantees.

Theorem 1: Given a sampler $\mathcal{G}$, tolerance parameter ϵ, intolerance parameter η and correctness parameter δ,

1) If for all φ, $\mathcal{G}(\varphi)$ is ϵ-additive almost-uniform generator then Barbarik will Accept with probability $(1-\delta)$.
2) If for some φ the sampler $\mathcal{G}(\varphi)$ is η-far from the uniform sampler in the ℓ_1 distance and the sampler satisfies the **non-adversarial sampler assumption** then Barbarik will Reject with probability $(1-\delta)$.

For the implementation, the subroutine Kernel is designed in an attempt to fool the sampler into satisfying the *non-adversarial assumption*. The idea being that the new CNF-formula $\hat{\varphi}$ would be "hard" to distinguish from φ and hence one would expect

$$p_{\mathcal{G}}(\hat{\varphi}, \sigma_1, S) = \frac{p_{\mathcal{G}}(\varphi, \sigma_1, S)}{p_{\mathcal{G}}(\varphi, \sigma_1, S) + p_{\mathcal{G}}(\varphi, \sigma_2, S)}$$

C. Experimental Setup

All our experiments were conducted on a high-performance computer cluster with each node consisting of a $\mathrm{E5-2690\ v3}$ CPU with 24 cores and 96GB of RAM, with a memory limit set to 4GB per core.

III. From CryptoMiniSat to CMSGen

The naive technique to design a sampler is to pick a random assignment of variables, check if it satisfies the CNF formula, and, if so, output the assignment as a witness; otherwise, pick another random assignment and start over again. Using an unbiased random coin for the assignments, it is trivial to see that the technique leads to a uniform sampler. Such a proposal is, however, very inefficient as with a very high probability, every picked assignment is likely not to satisfy the formula.

One way to make such a sampler into an efficient one is by not starting with a complete assignment but build the partial assignment up the variable by variable, set all variables that are implied by the current partial assignment, and if a partial assignment is incorrect, record and learn from the failure. The concept of learning from failure is captured by the well-known conflict-driven clause-learning (CDCL) framework used by most state-of-the-art SAT solvers. We refer the reader to Chapter 4 of [4] for a detailed exposition on CDCL. We present an extension that seeks to combine the CDCL framework with randomization in the choice of partial assignments in Algorithm 1, called UniformLikeWitness. UniformLikeWitness is essentially a randomized variation on the CDCL framework, with a randomized heuristic for what variable to assign next, a randomized heuristic for variable polarities, and without restarts.

Algorithm 1 UniformLikeWitness(F, seed)

```
1:  while true do
2:      x ← pick an unassigned variable at random
3:      assigns[x] ← pick 0 or 1 uniformly at random
4:      conflict, assigns ← perform unit propagation
5:      if assigns is full then return assigns
6:      if conflict is found then
7:          back_lvl, conf_clause ← Conflict-Analysis [32]
8:          if conf_clause is empty then return NULL
9:          Update assigns as per back_lvl
10:         F ← F ∪ conf_clause
11:         if F is too large then
12:             Perform Learnt Clause Deletion [2]
```

One major problem of the above process is that the sampler, just like an SAT solver, may get stuck in the corner of the space where there are no satisfying solutions. Once stuck, it can take much time to record the relevant conflicts before it can escape this part of the search space. In modern SAT solvers, such an escaping is enabled by performing restarts. The idea of a restart is to stop the current search procedure, keeping conflict clause and heuristic data such as polarities, variable activities in the line, but otherwise starting afresh, resetting the assignment state. The idea of performing a restart is to reduce the chance of getting stuck in a non-fruitful part of the search space. Performing regular, frequent restarts is a core component of all state-of-the-art SAT solvers.

CMSGen [2] is a sampler that exploits the flexibility CryptoMiniSat to implement the behaviour of UniformLikeWitness. We use the restart policy based on the number of conflicts, i.e., we perform a restart after the predetermined number of conflicts, which is set to 100. Hence, the final set of options passed to CryptoMiniSat turn off the features unrelated to CDCL (such as bounded variable elimination [17], local search [6], or symmetry breaking [15]), and set the options that control variable branching and polarity picking to match Algorithm 1, and set the restart interval to 100. Note that while it is possible that other CDCL SAT solvers could be adjusted to generate samples as well as CMSGen, the newer and more performant glucose-based SAT solvers [2] tend to be highly tuned without any command-line options to change or turn off heuristics.

We would like to emphasize that we do not claim that CMSGen is expected to generate uniform distributions over all the formulas as it is possible to construct worst case scenarios where CMSGen would not work well. At this point, it is worthwhile to note that, to the best of our knowledge, the current techniques are insufficient to analyse the kind of formulas for which UniformLikeWitness would behave like

[2] CMSGen is available at https://github.com/meelgroup/cmsgen

a uniform sampler given their limitations to understand the behaviour of CDCL itself. Traditionally, the proposal of a new sampler is accompanied by theoretical analysis, but in our case, we seek to rely on the testing framework of Barbarik to analyse the behavior of CMSGen.

IV. The Power of CMSGen

As mentioned above, instead of taking a conventional route focusing on the theoretical analysis of CMSGen, we seek to employ Barbarik to test whether CMSGen is a uniform sampler or not. In addition, we seek to understand the runtime behavior of CMSGen in comparison to other state of the art techniques. We conducted an extensive evaluation of diverse public domain benchmarks employed in prior studies [8], [40].

A comment on the choice of benchmarks for the two studies: For the first study, we selected the same 50 benchmarks that were employed in the evaluation of Barbarik so as to situate the results with prior context [8]. Since Barbarik needs to sample up to 1.835×10^3 solutions, the choice of benchmarks in [8] was restricted to instances for which generating samples is easy. On the other hand, these benchmarks are not meaningful for runtime performance comparison as all the tools finish on them very quickly. To this end, we relied on 70 benchmarks employed in prior sampling studies [38], [39] for runtime performance comparison.

The objective of our evaluation was two-fold:

RQ1 To understand the behavior of Barbarik in terms of the frequency of outputs Accept and Reject with CMSGen as sampler under test.

RQ2 To evaluate the runtime performance of CMSGen vis-a-vis the state of the art sampler with guarantees of almost-uniformity, UniGen3.

In summary, we observe that Barbarik, somewhat surprisingly, returns Accept for CMSGen and UniGen3 on all the 50 instances while returning Reject for all the 50 instances for QuickSampler [16], and for 36 instances for STS [18], the state-of-the-art samplers without guarantees. At the same time, comparison in terms of runtime for over 70 benchmarks arising from different application domains, we observe that CMSGen is significantly faster than UniGen3.

A. *Testing* CMSGen *with* Barbarik

For experimentation evaluations with Barbarik, we used the default parameters suggested by the authors: In particular, we set tolerance parameter ϵ, intolerance parameter η, and confidence δ to be 0.3, 1.8, and 0.1 respectively. For our chosen parameters, the number of samples required to return Accept for a given sampler under test is 1.836×10^3, and to maintain consistency with evaluation setup of Barbarik, we selected benchmarks (50 in total) that were used in evaluation of QuickSampler and UniGen3 for which Barbarik terminates within 2 hours. To test uniformity of distributions generated by CMSGen and other samplers, we employed Barbarik augmented with SPUR [1] as the underlying uniform sampler. We present the results of our evaluation in Table I, where the four columns present results corresponding to QuickSampler, STS, UniGen3, and CMSGen respectively. The first and second rows indicate the number of instances for which Barbarik returned Accept and Reject respectively. We first note that while Barbarik returned Reject for QuickSampler and STS for the 50 and 36 instances respectively, it returned Accept for both CMSGen and UniGen3 for all the instances. It is worth highlighting that UniGen3 provides guarantees of almost-uniformity.

TABLE I: Analysis of different samplers with Barbarik over 50 benchmarks. Parameters $\epsilon : 0.3, \eta : 1.8, \delta : 0.1$, and samples required to return Accept 1.836×10^3.

	QuickSampler	STS	UniGen3	CMSGen
Accept	0	14	50	50
Reject	50	36	0	0

Remark 1: At this point, it is worth highlighting that we arrived at the choice of parameters of CMSGen, such as when to restart via an iterative process where we would run Barbarik for the given choice of parameters and change them based on the number of instances rejected by Barbarik. In this context, it is rather encouraging that such an iterative process led us to design a sampler, CMSGen, which could not be distinguished from UniGen3 by Barbarik while significantly improving upon UniGen3 in terms of runtime performance. This highlights the advantages of a TDD-style design approach.

B. Runtime Comparison

Upon observing that Barbarik returns Accept for all the 50 instances for both CMSGen and UniGen3, a natural question is whether the runtime performance of CMSGen is comparable to that of UniGen3. To this end, we compared CMSGen with UniGen3, STS and QuickSampler on 70 benchmark instances arising from a wide range of application areas of uniform sampling, such as probabilistic reasoning, Bounded Model Checking [37], [40]; these instances had been previously employed in empirical studies focused on the comparison of sampling techniques [38], [39].

For each of the instances, we invoke each of the sampler to generate 1000 solutions within a timeout of 7200 seconds. Figure 1 shows the cactus plot for CMSGen, UniGen3, STS and QuickSampler. We present the number of benchmarks on the x-axis and the time taken on the y-axis. A point (x, y) implies that for a x benchmark, the sampler took less than or equal to y seconds to generate 1000 solutions of x. With a timeout of 7200 seconds, UniGen3 and CMSGen were able to sample 1000 solutions of 51 and 52 benchmarks respectively, whereas STS and QuickSampler generated samples for *merely* 37 and 33 instances respectively. Figure 1 clearly shows that for all the benchmarks that were sampled 1000 times by both UniGen3 and CMSGen, CMSGen outperformed UniGen3 with a geometric speedup of over $420\times$.

Table II represent the runtime performance for QuickSampler, STS, UniGen3 and CMSGen for a representative set of 20 benchmarks. As shown in Table II, there are instances (18 out of 70) for which UniGen3 is able to samples 1000 solutions

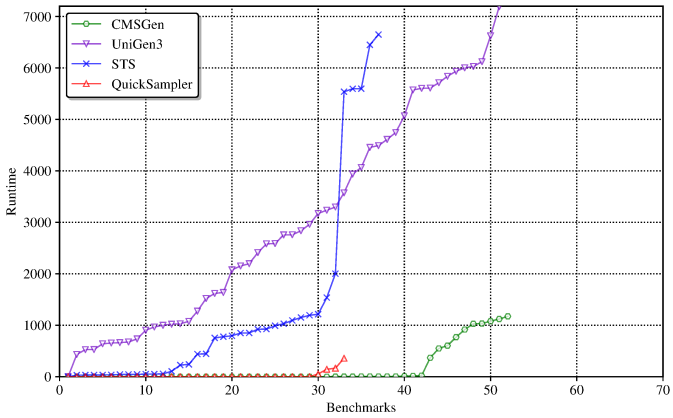

Fig. 1: Cactus plot showing runtime performance of UniGen3, STS, QuickSampler and CMSGen to generate 1000 samples. Timeout: 7200s

TABLE II: Runtime performance of different samplers to generate 1000 solutions for a representative set of benchmarks. Timeout (TO): 7200s.

Benchmarks	QuickSampler	STS	UniGen3	CMSGen
or-70-5-5-UC-20	0.07	36.39	3173.45	0.29
or-60-20-10-UC-20	0.07	43.53	4065.0	0.31
or-100-20-8-UC-40	0.09	51.25	2152.01	0.4
tire-2	1.09	226.01	TO	0.48
or-50-10-7-UC-10	0.06	33.28	2196.98	0.95
b12_2_linear	TO	1214.73	1520.01	2.08
b14_2_linear	TO	926.18	1220.01	2.18
squaring41	TO	5595.0	6002.0	2.8
squaring60	TO	TO	TO	4.52
s15850a_15_7	359.37	TO	675.33	5.58
b12_even2_linear	TO	TO	TO	15.52
isolateRightmost	TO	TO	432.73	21.66
modexp8-5-4	TO	TO	6122.0	550.9
modexp8-6-4	TO	TO	TO	1034.27
modexp8-6-3	TO	TO	6624.0	1079.82
modexp8-6-8	TO	TO	TO	1173.64
prod-20	TO	TO	1274.42	TO
04B-1	TO	5598.0	2410.61	TO
06B-1	TO	6449.0	2835.64	TO
hash-10-7	TO	TO	5610.0	TO

whereas CMSGen could not sample. Similarly, there are 19 instances for which CMSGen is able to samples solutions but UniGen3 could not.

V. Case Studies: Functional Synthesis and Combinatorial Testing

Having established that the quality of distribution generated by CMSGen is significantly better than QuickSampler, one wonders about the practical utility of CMSGen. The significant gap between runtime performance of CMSGen and UniGen3 argues for the usage of CMSGen in applications where the quality and runtime performance of samplers are key determining factors.

To this end, we focused on two such application domains: Combinatorial testing and Boolean functional synthesis. The state of the art techniques for each of these domains crucially rely on underlying uniform samplers; in fact the sampler QuickSampler was proposed in the context of combinatorial testing. For each of these case studies, we substitute the three samplers CMSGen, QuickSampler, and UniGen in the state of the art techniques, and analyse their performance on the resulting tool.

A. Combinatorial Testing

Combinatorial testing is considered as a powerful paradigm for testing configurable software. The primary task of a test generator is the generation of a test suite that maximizes t-wise coverage. t-wise coverage is measured as the fraction of feature combinations appearing in the test set out of the possible valid feature combinations. Uniform sampling is considered one of the promising approach to have higher t-wise coverage [31], [34], [35]. Therefore, a natural question is whether CMSGen can serve as a good test suite generator. To this end, we performed a comparative study of CMSGen vis-a-vis UniGen3, STS and QuickSampler on the set of 110 publicly available benchmarks that have been employed in prior comparative studies of sampling techniques in the context of combinatorial testing [25], [29], [35][3].It is worth emphasizing that UniGen3, STS and QuickSampler are viewed as a state of the art test suite generation techniques in the presence of constraints as witnessed by empirical study by Plazar et al. [35].

In our comparative study of sampling techniques of their efficiency in achieving higher t-wise coverage, we focus on the case of $t = 2$ as is standard in the most empirical studies in combinatorial testing. To this end, for every benchmark, we generate 1000 samples from each of the four samplers: CMSGen, STS, QuickSampler, and UniGen3. We used a timeout of 3600 seconds for sampling. UniGen3 is, however, unable to sample for all but six benchmarks. Therefore, we exclude UniGen3 from further analysis.

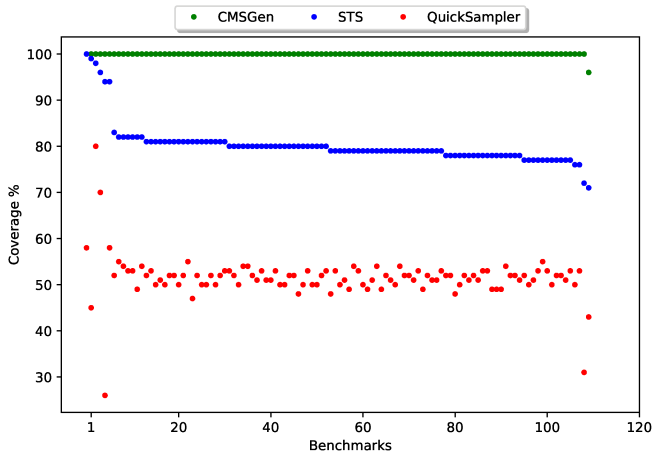

Fig. 2: Plot to show 2-wise coverage% for 110 benchmarks with 1000 samples. Sampling timeout: 3600s.

[3]Benchmarks are available at https://zenodo.org/record/4022395

TABLE III: Analysis for 2-wise coverage with QuickSampler, STS, and CMSGen.

Benchmark	# Feature Combinations	QuickSampler		STS		CMSGen	
		# combination observed	Coverage	# combination observed	Coverage	# combination observed	Coverage
busybox_1_28_0	1965023	513565	0.26	1849127	0.94	1964962	1.0
ecos-icse11	2910229	898195	0.31	2104721	0.72	2910078	1.0
financial	917150	392381	0.43	649279	0.71	876356	0.96
buildroot	621270	278254	0.45	613184	0.99	621252	1.0
vads	2896324	1360422	0.47	2348489	0.81	2895931	1.0
mpc50	2719748	1354164	0.5	2078077	0.76	2719508	1.0
XSEngine	2974825	1498239	0.5	2383688	0.8	2974448	1.0
ocelot	2986129	1519047	0.51	2344079	0.78	2986002	1.0
dreamcast	2908040	1523501	0.52	2253050	0.77	2907734	1.0
refidt334	3022264	1557688	0.52	2356854	0.78	3021978	1.0
integrator_arm7	2957100	1566676	0.53	2275664	0.77	2956958	1.0
pc_i82559	2977432	1582402	0.53	2384286	0.8	2977280	1.0
p2106	2887921	1544728	0.53	2282100	0.79	2887653	1.0
skmb91302	2755776	1451902	0.53	2133950	0.77	2755538	1.0
cma28x	2694432	1419911	0.53	2156230	0.8	2694257	1.0
ipaq	2897450	1576622	0.54	2305020	0.8	2897153	1.0
axtls	16212	9381	0.58	15264	0.94	16212	1.0
uClinux	3013528	1751212	0.58	3013456	1.0	3013528	1.0
toybox	256494	180332	0.7	246484	0.96	256494	1.0
FM-3.6.1-refined	3151	2518	0.8	3075	0.98	3151.0	1.0

Figure 2 shows the experimental results with STS, QuickSampler and CMSGen. We present the number of benchmarks on the x-axis and pair-wise coverage % on the y-axis. A point (x, y) implies that x benchmarks had $y\%$ pair-wise coverage. Benchmarks are ordered in the decreasing order of coverage achieved with the samples produced by STS. Figure 2 shows that almost all the benchmarks had nearly 100% pair-coverage with samples generated by CMSGen, on the other hand, the average pair-wise coverage with samples from QuickSampler and STS is 51.5% and 80.15%. One should view the significant performance improvement due to CMSGen over QuickSampler in light of the fact that the primary motivation behind the proposal of QuickSampler was to achieve higher coverage.

Table III represents the analysis for 2-wise coverage with CMSGen, STS and QuickSampler for representative 20 benchmarks. In table III, Column 2 present the possible valid feature combinations. Column 3, 5 and 7 present the feature combinations appearing in test set generated by QuickSampler, STS and CMSGen respectively, and Column 4,6 and 8 is for the corresponding coverage. As shown in Table III, the test set generated with CMSGen is able to cover *all* possible feature combinations for all the benchmarks.

B. Boolean Functional Synthesis

Given a formula $\exists Y F(X, Y)$, the problem of Boolean functional synthesis seeks to compute a function φ such that $\exists Y F(X, Y) \equiv F(X, \varphi(X))$. Typically, we view F as a specification and φ as the function that implements the specification φ. Boolean functional synthesis is a fundamental problem with wide variety of applications ranging from logic synthesis [28], cryptography [30], program synthesis [42], and the like. For example, Boolean functional synthesis encompasses program synthesis, where φ can be viewed as the desired program. Consequently, there has been a sustained interest in the design of efficient algorithmic techniques for Boolean functional synthesis. The current state of the art approach, Manthan, was proposed recently and builds on the advances in sampling techniques, automated reasoning, and machine learning [21]. Manthan was demonstrated to solve 70 more benchmarks than the next best technique. In this regard, Manthan serves as a good test-bed to compare different sampling techniques.

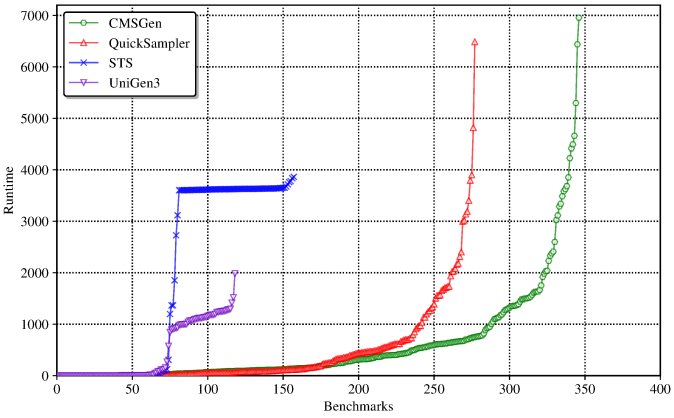

Fig. 3: Cactus plot to show the impact of different sampler on functional synthesis engine, Manthan. Timeout: 7200s

We sought to compare CMSGen vis-a-vis UniGen3, STS and QuickSampler in their impact on the performance of Manthan. We set the timeout of 3600 seconds for the sampling phase of Manthan. To this end, we augment the sampling step of Manthan with the corresponding samplers. We perform the empirical analysis of the same 609 benchmarks[4] that were employed in the analysis of Manthan [21]. We present a

[4]Benchmarks are available at https://zenodo.org/record/3892859

summary of our analysis in the form of cactus plot in Figure 3: the number of instances are shown on the x-axis and the time taken on the y-axis; a point (x, y) implies that Manthan augmented with the corresponding sampler took less than or equal to y seconds to solve x instances.

Table IV shows the time taken to synthesize Boolean functions with samples generated from different samplers for a representative set of 20 benchmarks.

Few observations are in order:

1) Manthan augmented with UniGen3 could solve only 118 instances due to UniGen3's inability to sample for all but 220 instances. Similarly, Manthan with STS could solve only 157 instances.
2) Manthan augmented with CMSGen solves 345 instances while Manthan augmented with QuickSampler could solve only 275 instances.

TABLE IV: Runtime analysis of Manthan with QuickSampler, STS, UniGen3, and CMSGen. Timeout (TO): 7200s.

Benchmarks	QuickSampler	STS	UniGen3	CMSGen
kenflashp02	9.55	1367.12	573.69	26.77
kenoopp1	25.96	1852.07	TO	28.88
bobsynth00neg	114.66	3621.66	TO	74.06
bobtuint04neg	58.62	3636.39	1276.1	109.29
small-swap1-fix-4	TO	TO	TO	148.15
pdtpmsrotate32	TO	TO	TO	279.6
exquery_query42	254.17	TO	TO	281.5
GuidanceService2	529.16	TO	TO	290.71
subtraction256	699.09	3836.48	TO	321.35
IssueServiceImpl	1567.23	TO	TO	424.77
query55_query42	6488.93	TO	TO	766.98
rankfunc48_s_64	TO	TO	TO	775.42
sortnetsort7.006	732.42	TO	TO	785.13
LoginService	TO	TO	TO	1108.0
query30_query42	1134.6	TO	TO	1126.53
ethernet-fixpoint-4	TO	TO	TO	1752.18
query44_query26	TO	TO	TO	2037.54
small-equiv-fix-8	TO	TO	TO	2231.22
pi-fixpoint-2	535.74	3674.9	TO	2373.72
sortnetsort9.010	3795.4	TO	TO	4414.56

Therefore, in conclusion, Manthan augmented with CMSGen solves significantly more instances than Manthan augmented with UniGen3, STS, or QuickSampler.

VI. Conclusion

Motivated by the availability of Barbarik, a tester for samplers, we sought to design a sampler for which Barbarik would return Accept. We succeeded in our task by a simple but careful tweaking of the existing state-of-the-art SAT solver, CryptoMiniSat. Our resulting sampler CMSGen is not only accepted by Barbarik but achieves better runtime performance than state-of-the-art samplers with theoretical guarantees. We then show that the resulting sampler, CMSGen, can significantly improve the performance of applications that utilize samplers. It is perhaps worth reiterating that we view the simplicity of CMSGen as its salient strength. The simplicity of CMSGen stands in stark contrast to complicated algorithmic schemes developed in the past that fail to attain the desired quality of distributions with practical runtime performance.

We now turn our attention back to Remark 1; the design of CMSGen was an iterative process with Barbarik in loop. A natural direction of future work would be the development of a tester that provides a quantitative analysis instead of a qualitative answer of Accept or Reject to measure the quality of samplers. The significant runtime improvements in the context of functional synthesis and combinatorial testing due to CMSGen motivate us to study the impact of CMSGen in other application domains; to this end, we will release CMSGen open-source upon publication of our manuscript.

Acknowledgments: This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore: https://www.nscc.sg

References

[1] D. Achlioptas, Z. S. Hammoudeh, and P. Theodoropoulos, "Fast sampling of perfectly uniform satisfying assignments," in *Proc. of SAT*, 2018.
[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proc. of IJCAI*, 2009.
[3] T. Batu, S. Dasgupta, R. Kumar, and R. Rubinfeld, "The complexity of approximating the entropy," *Proc. SIAM Journal on Computing*, 2005.
[4] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.
[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *Proc. of ACM SIGSAC*, 2016.
[6] S. Cai, C. Luo, and K. Su, "CCAnr: A configuration checking based local search solver for non-random satisfiability," in *SAT 2015*, ser. LNCS, M. Heule and S. A. Weaver, Eds., vol. 9340. Springer, 2015, pp. 1–8.
[7] S. Chakraborty, E. Fischer, Y. Goldhirsh, and A. Matsliah, "On the power of conditional samples in distribution testing," *Proc. of SIAM Journal on Computing*, 2016.
[8] S. Chakraborty and K. S. Meel, "On testing of uniform samplers," in *Proc. of AAAI*, 2019.
[9] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform SAT witness generation," in *Proc. of TACAS*, 2015.
[10] S. Chakraborty, D. Fried, K. S. Meel, and M. Y. Vardi, "From weighted to unweighted model counting," in *Proc. of AAAI*, 2015.
[11] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable and nearly uniform generator of sat witnesses," in *Proc. of CAV*, 2013.
[12] ——, "Balancing scalability and uniformity in SAT witness generator," in *Proc. of DAC*, 2014.
[13] L. A. Clarke, "A program testing system," in *Proc. of ACM*, 1976.
[14] R. Dechter, K. Kask, E. Bin, R. Emek *et al.*, "Generating random solutions for constraint satisfaction problems," in *Proc. of AAAI*, 2002.
[15] J. Devriendt and B. Bogaerts, "BreakID: Static symmetry breaking for ASP (system description)," *CoRR*, vol. abs/1608.08447, 2016.
[16] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of SAT solutions for testing," in *Proc. of ICSE*, 2018.
[17] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *SAT 2005*, ser. LNCS, F. Bacchus and T. Walsh, Eds., vol. 3569. Springer, pp. 61–75.
[18] S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman, "Uniform solution sampling using a constraint solver as an oracle," in *Proc. of UAI*, 2012.
[19] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proc. of ESEC/FSE*, 2011.
[20] V. Gogate and R. Dechter, "A new algorithm for sampling CSP solutions uniformly at random," in *Proc. of CP*, 2006.
[21] P. Golia, S. Roy, and K. S. Meel, "Manthan: A data-driven approach for Boolean function synthesis," in *Proc. of CAV*, 2020.
[22] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proc. of USENIX Security 12*, 2012.
[23] J. C. King, "Symbolic execution and program testing," *Comm. ACM*, 1976.

[24] N. Kitchen, "Markov chain monte carlo stimulus generation for constrained random simulation," Ph.D. dissertation, UC Berkeley, 2010.
[25] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer, "Is there a mismatch between real-world feature models and product-line research?" in *Proc. of ESEC/FSE*, 2017.
[26] J. H. Kukula and T. R. Shiple, "Building circuits from relations," in *Proc. of CAV*, 2000.
[27] S. Kullback and R. A. Leibler, "On information and sufficiency," *Proc. of Ann. Math. Statist.*, 1951.
[28] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Complete functional synthesis," 2010.
[29] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman, "Sat-based analysis of large real-world feature models is easy," in *Proc. of sPLC*, 2015.
[30] F. Massacci and L. Marraro, "Logical cryptanalysis as a SAT problem," *Journal of Automated Reasoning*, 2000.
[31] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Prof. of ICSE*, 2016.
[32] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. of DAC*, 2001.
[33] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *Proc. of AI magazine*, 2007.
[34] J. Oh, P. Gazzillo, and D. Batory, "t-wise coverage by uniform sampling," in *Proc. of SPLC*, 2019.
[35] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform sampling of sat solutions for configurable systems: Are we there yet?" in *Proc. of ICST*, 2019.
[36] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proc. of ESEC/FSE*, 2018.
[37] S. Sharma, R. Gupta, S. Roy, and K. S. Meel, "Knowledge compilation meets uniform sampling." in *Proc. of LPAR*, 2018.
[38] ——, "Knowledge compilation meets uniform sampling." in *Proc. of LPAR*, 2018.
[39] M. Soos, S. Gocht, and K. S. Meel, "Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling," in *Proc. of CAV*, 2020.
[40] M. Soos and K. S. Meel, "Bird: Engineering an efficient CNF-XOR sat solver and its applications to approximate model counting," in *Proc. of the AAAI*, 2019.
[41] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Proc. of SAT*, 2009.
[42] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *STTT*, 2013.
[43] W. Wei and B. Selman, "A new approach to model counting," in *Proc. of SAT*, 2005.
[44] N. Wetzler, M. J. H. Heule, and W. A. Hunt, "DRAT-trim: Efficient checking and trimming using expressive clausal proofs," in *Proc. of SAT*, 2014.