EPiC Series in Computing
Volume 57, 2018, Pages 620–636
LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Knowledge Compilation meets Uniform Sampling

Shubham Sharma[1*], Rahul Gupta[1*], Subhajit Roy[1], and Kuldeep S. Meel[2]

[1] Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, India
smsharma@iitk.ac.in, grahul@iitk.ac.in, subhajit@iitk.ac.in

[2] School of Computing,
National University of Singapore, Singapore
meel@comp.nus.edu.sg

Abstract

Uniform sampling has drawn diverse applications in programming languages and software engineering, like in constrained-random verification (CRV), constrained-fuzzing and bug synthesis. The effectiveness of these applications depend on the uniformity of test stimuli generated from a given set of constraints. Despite significant progress over the past few years, the performance of the state of the art techniques still falls short of those of heuristic methods employed in the industry which sacrifice either uniformity or scalability when generating stimuli.

In this paper, we propose a new approach to the uniform generation that builds on recent progress in knowledge compilation. The primary contribution of this paper is marrying knowledge compilation with uniform sampling: our algorithm, KUS, employs the state-of-the-art knowledge compilers to first compile constraints into d-DNNF form, and then, generates samples by making two passes over the compiled representation.

We show that KUS is able to significantly outperform existing state-of-the-art algorithms, SPUR and UniGen2, by up to 3 orders of magnitude in terms of runtime while achieving a geometric speedup of 1.7× and 8.3× over SPUR and UniGen2 respectively. Also, KUS achieves a lower PAR-2[1] score, around 0.82× that of SPUR and 0.38× that of UniGen2. Furthermore, KUS achieves speedups of up to 3 orders of magnitude for incremental sampling. The distribution generated by KUS is statistically indistinguishable from that generated by an ideal uniform sampler. Moreover, KUS is almost oblivious to the number of samples requested.

1 Introduction

Uniform sampling is becoming an important technique in multiple areas in computer science from Bayesian analysis to software engineering and programming languages [35]. Some applications from these domains are as follows:

[*]Part of this work was done during a visit to NUS Singapore
[1]PAR-2 scheme, that is, penalized average runtime, used in SAT-2017 Competition [3], assigns a runtime of two times the time limit (instead of a "not solved" status) for each benchmark not solved by a solver

- Dutra et al. [21] proposed the idea of *constraint-based fuzzing*, where one can fuzz (randomly test) on the prefix of a symbolic path. This can lead to significant speedups by combining the best of symbolic execution [31, 14] and random testing [8, 23, 25, 26].
- Constrained-random verification (CRV) [40] has been extremely successful in testing hardware designs; the constraints are imposed by the hardware designers based on domain-specific knowledge (preconditions, known invariants etc.), and uniform sampling is used to generate tests that satisfy the provided constraints.
- Bug synthesis [42] attempts to create large bug corpora to test and tune the heuristics of bug-finding tools. The algorithm uses a pre-selected path (seed path) to inject a bug, and augments the path by additional guards to make the buggy location challenging to reach. The *quality* of the guard predicates is gauged via uniform sampling: the path condition of the seed path is uniformly sampled to generate tests; the number of these tests reaching the buggy location via a guard is taken as a proxy for the *quality* of the guard predicate and thus, transitively, the *quality of the injected bug*.

The success of application of uniform sampling in testing and debugging can be attributed to the fact that, as the distribution of bugs in a program is not known a priori, a uniform prior tends to be the best possible option.

One of the central problems in the field of artificial intelligence is that of probabilistic inference, wherein we are given a graphical model describing conditional dependencies between variables of interest, and we are required to compute the conditional probability of an event, i.e., valuations of a subset of variables, given some evidence in the form of valuations of another subset of variables. Over the past two decades, Darwiche and Marquis [18] have pioneered the approach of knowledge compilation wherein a logical theory is compiled into a form that allows performing probabilistic inference in polynomial time. It is known that there is a deep connection between probabilistic inference and model counting [15, 41, 13]. Furthermore, Jerrum, Valiant, and Vazirani [30] observed another deep relationship between model counting and uniform sampling. In particular, they showed that given access to an exact model counter, one could design a uniform sampler which requires only polynomially many queries to the exact model counter. In this context, one wonders if the recent advances in knowledge compilation can be harnessed to design a scalable uniform sampler.

The primary contribution of this paper is marrying knowledge compilation with uniform sampling to design a new algorithm, KUS, that performs uniform sampling, outperforming current state-of-the-art approximately uniform and uniform samplers. The central idea behind KUS is to first employ the state of the art knowledge compilation approaches to compile a given CNF formula into d-DNNF form (formally described in Section 2), and then performing only two passes over the d-DNNF representation to generate as many identically and independently distributed samples as specified by the user. In typical industrial setting, the end user (i.e., verification engineer) typically invokes the underlying uniform generator repeatedly till a bug is triggered or a desired coverage metric is reached. The key advantage of KUS is that a significant amount of work needed for sample generation is performed offline, which is amortized over many invocations made for sample generation. We have built a prototype implementation of KUS, and show by means of extensive experiments that it significantly outperforms existing state-of-the-art algorithms while generating sample distributions that are indistinguishable from those generated by an ideal uniform sampler. KUS performs better than existing start-of-the-art samplers, SPUR and UniGen2, by up to 3 orders of magnitude in terms of runtime while achieving a geometric speedup of 1.7× and 8.3× over SPUR and UniGen2 respectively. Also, KUS achieves a lower PAR-2 score around 0.82× that of SPUR and 0.38× that of UniGen2

(PAR-2 scheme, that is, penalized average runtime, used in SAT-2017 Competition [3], assigns a runtime of two times the time limit, instead of a "not solved" status', for each benchmark not solved by a solver). Furthermore, KUS achieves speedups of up to 3 orders of magnitude for incremental sampling. We believe that the success of KUS will motivate researchers in testing, verification, AI and knowledge compilation communities to investigate a broader set of logical forms favorable for efficient uniform generation.

The rest of the paper is organized as follows. We first introduce notations and preliminaries in Section 2. We discuss related work in Section 3 and present KUS in Section 4. We then describe experimental methodology in Section 5 and discuss results in Section 5. Finally, we conclude in Section 6.

2 Notations and Preliminaries

A literal is a Boolean variable or its negation. Let F be a Boolean formula in conjunctive normal form (CNF) with the set of variables appearing in it denoted by X, referred to as the *support* of F. An assignment of truth values to all the variables in the support of F is referred to as its *satisfying assignment* or *witness* (denoted by σ). All witnesses of F constitute R_F; to avoid clutter, whenever the formula F is clear from the context, we omit mentioning it.

2.1 Uniform Generators

We use $\mathsf{Pr}[X]$ to denote the probability of event X. Given a Boolean formula F, a *probabilistic generator* of witnesses of F is a probabilistic algorithm that generates a random witness in R_F. A *uniform generator* $\mathcal{G}^u(\cdot)$ is a probabilistic generator that guarantees

$$\forall y \in R_F, \mathsf{Pr}[\mathcal{G}^u(F) = y] = \frac{1}{|R_F|}, \tag{1}$$

An *almost-uniform generator* $\mathcal{G}^{au}(\cdot,\cdot)$ ensures that:

$$\forall y \in R_F, \frac{1}{(1+\varepsilon)|R_F|} \leq \mathsf{Pr}[\mathcal{G}^{au}(F,\varepsilon) = y] \leq \frac{1+\varepsilon}{|R_F|} \tag{2}$$

where $\varepsilon > 0$ is the specified *tolerance*. A *near-uniform generator* $\mathcal{G}^{nu}(\cdot)$ further relaxes the guarantee of uniformity, ensuring that $\mathsf{Pr}[\mathcal{G}^{nu}(F) = y] \geq \frac{c}{|R_F|}$ for a constant c, where $0 < c \leq 1$. Probabilistic generators are allowed to "fail" occasionally (with a failure probability bounded by a constant strictly less than 1): no witness may be returned even if $R_F \neq \emptyset$;

2.2 Knowledge Compilation and d-DNNF representation

To deal with computational intractability of probabilistic reasoning, knowledge compilation seeks to *compile* a knowledge base, often represented as a propositional formula in CNF, to a target language. Thereafter, probabilistic reasoning tasks, which are often expressed as sequence of queries, are performed by querying the knowledge base in the target language [18]. The earliest and perhaps the most well known target language is Ordered Binary Decision Diagrams (OBDDs), which has proven to be effective in circuit analysis and synthesis [9]. Several variants of OBDD have been explored over the past two years. One of the such popular variant is Deterministic Decomposable Negation Normal Form (d-DNNF), which is strict superset of OBDDs. Since several probabilistic reasoning tasks such as probabilistic inference, Maximum

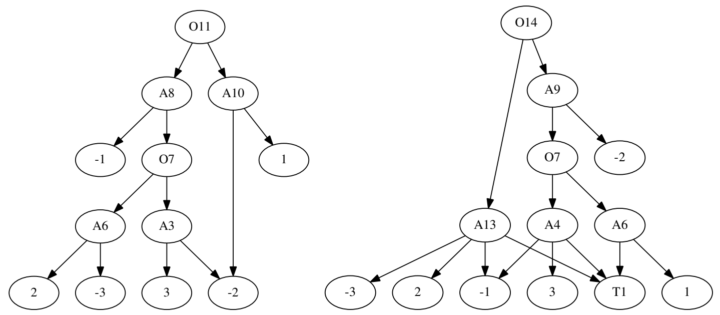

Figure 1: DAGs generated from Dsharp and D4

a Posteriori (MAP) can be answered in polynomial time in the size of d-DNNF, d-DNNF have emerged as a central target language in knowledge compilation community. To formally defined d-DNNF, we first formally define Negation Normal Form (NNF):

Definition 1. *[18] Let X be the set of propositional variables. A sentence in NNF is a rooted, directed acyclic graph (DAG) where each leaf node i is labeled with true, false, y or $\neg y$, $y \in X$; and each internal node is labeled with $\vee$ or $\wedge$ and can have arbitrarily many children.*

The d-DNNF form is a strict subset of this representation that further imposes that the representation is:

- **Deterministic**: We refer to an NNF as deterministic if the operands of $\vee$ in all well-formed Boolean formula in the NNF are mutually inconsistent.
- **Decomposable**: We refer to an NNF as decomposable if the operands of $\wedge$ in all well-formed Boolean formula in the NNF are expressed in a mutually disjoint set of variables.

The d-DNNF formulae can be represented as AND-OR graphs (DAGs) where a node is either an AND node (represented by nodes labeled with "A"), an OR node (represented by nodes labeled with "O") or a literal (the leaves of the DAG). The operands of AND/OR nodes appear as children of the node. Figure 1 shows d-DNNF representations of the same formula, but generated using two different d-DNNF compilers, Dsharp and D4. For every node t, the sub-formula corresponding to t is the formula corresponding to d-DNNF obtained by removing all the nodes u such that there does not exist path from t to u.

The state-of-the-art d-DNNF construction tools like C2D [17], Dsharp [39] and D4 [29], construct the d-DNNF representation where each OR node has exactly two children while an AND node may have multiple children. Since our framework KUS employs modern d-DNNF compiler, we assume that the OR node has exactly two children. This assumption is only for the simplicity of exposition as our algorithms can be trivially adopted to the general d-DNNF representations.

Decision-DNNF [29] is a close cousin of d-DNNF where the deterministic OR nodes are replaced by $ite(u, C_1, C_2)$ for a decision variable u; the semantics of ite is defined to select one of the children, C_1 or C_2, depending on the polarity of u. The children subtrees, C_i, do not contain the variable u. D4 is capable of generating both the d-DNNF and Decision-DNNF representations; we use the d-DNNF representation from D4 for this work.

3 Related Work

Sampling from solutions of constraint systems is challenged by the dual objectives of providing scalability while ensuring strong guarantees of uniformity. Though it has been shown that, in theory, provably uniform generator of SAT witnesses can be designed to run in probabilistic polynomial time relative to an NP oracle [7], this algorithm does not scale in practice [36]. BDD based techniques [44] to sample for SAT witnesses also suffer from scalability problems [32].

On the other end of the spectrum, adapted BDD based techniques [34] and random seeding of DPLL SAT solvers [38], compromise guarantees of uniformity to achieve scalability [32]. Markov Chain Monte Carlo (MCMC) based methods for model counting and constrained-random verification [43, 32], being heuristic in nature, refrain from providing theoretical guarantees of uniformity. Similarly, sampling techniques based on interval-propagation and belief networks [20, 24, 28] tend to be scalable, but the generated distributions have been shown to deviate significantly from the uniform distribution [33].

Hashing-based sampling [30, 7, 11, 12, 37] operate by partitioning the space of satisfying assignments into small "cells" of roughly equal size using r-wise independent hash functions (for a suitable value of r), and then randomly choosing a solution from a randomly picked cell. Bellare et al. [7] showed that uniform generation can be guaranteed by choosing $r = n$ (with n variables in the propositional constraint), but understandably, it does not scale. Chakraborty et. al [11] design a significantly more scalable near-uniform generator (UniWit) and Ermon et al. [22] suggest further algorithmic improvements to uniform generation of witnesses. UniGen2 [12, 10] exploits a deep connection between approximate counting and almost-uniform sampling [30] to improve upon the ideas of UniWit, thereby providing stronger guarantees of uniformity and scaling to formulae with hundreds of thousands of variables. Even so, the runtime performance of UniGen2 falls short of the performance of heuristic methods commonly employed in industry, for example, to generate stimuli for CRV; a random-constrained test case generator is typically allowed to be $10\times$ slower than a constraint solver [2].

Recently, Dutra et al. [21] proposed a new technique QuickSampler for efficient sampling. QuickSampler uses a small number of constraint solver (Z3 [19]) calls to generate a large number of samples. QuickSampler finds a random satisfying assignment, and then, flips the value of each variable in the assignment; then, it uses constraint solver to check the satisfiability of newly generated assignments. The difference between the original solution and the modified solutions known as *atomic mutations* are then combined and applied to the original solution to generate new solutions. The samples generated by QuickSampler are not guaranteed to satisfy the given constraints. With no theoretical bound on the error, and the empirical evidence showing only about three-fourths of the samples satisfying the constraints, such techniques would not be suitable for a large number of applications. For example, applied to CRV, the false positive rate can grow arbitrarily large due to failure on tests that do not satisfy the preconditions.

Concurrently, Achlioptas, Hammoudeh, and Theodoropoulos have developed another approach, called SPUR [5], to uniform sampling by traversing the search of a component-caching based exact model counters such as sharpSAT. SPUR was shown to be 1-2 orders of magnitude faster than UniGen2 for a large set of benchmarks. Our algorithm, KUS, encompasses SPUR due to a strong connection between search and knowledge compilation demonstrated in [27]; wherein it was shown that the search-based algorithms for model counting such as sharpSAT generate traces belonging to d-DNNF language. As we demonstrate in this paper, this generalization allows us to exploit highly efficient d-DNNF compilers, allowing our algorithm to solve 69 more instances than SPUR out of 1425 benchmarks and gain about $1.7\times$ speedup.

[2]Private communication with industry expert W. Hung

Algorithm 1 KUS(F, s)

```
1: dag ← Compile($F$)
2: dag ← Annotate(dag)
3: SampleList ← Sampler(dag.$root$, $s$)
4: SampleList ← RandomAssignment(SampleList)
5: return SampleList
```

We close the review of prior work by a brief exposition to knowledge compilation. As discussed in section 2, knowledge compilation attempts to build representations that aid certain propositional reasoning tasks. Other than the OBDD, d-DNNF and decision-DNNF representations, the Sentential Decision Diagram (SDD) [16] has also become quite popular. SDDs are strictly more general than OBDDs and are canonical up to a given binary tree of variables (referred to as a vtree). Quite a few compilers have been proposed for these forms: the popular BuDDy [1] and MuDDy [2] packages for BDDs, C2D [17], Dsharp [39] and D4 [29] for d-DNNFs and SDD [4] for SDDs. The C2D compiler pioneered efficient d-DNNF compilation: it is based on an exhaustive recursive (rather than iterative) DPLL traversal. Given a CNF, C2D attempts to partition the clauses into two sets that do not share variables; these partitions are then compiled independently and the results are conjoined together. Dsharp improves upon C2D by performing the partitioning dynamically during the DPLL exploration (instead of the static decomposition used by C2D), and further using Implicit Binary Constraint Propagation (IBCP) to discover inconsistencies quicker. D4 is a recent work that uses different decomposition heuristics in an attempt to keep the best of the static and dynamic decomposition strategies from C2D and Dsharp.

4 Algorithm

In this section, we discuss our primary technical contribution: KUS, a uniform sampler that employs knowledge compilation techniques. We close the section with theoretical analysis of KUS.

KUS takes in a CNF formula F and required number of samples s and returns a set of s samples such that each sample is uniformly and independently drawn from the uniform distribution over the set of solutions R_F. KUS is presented in Algorithm 1. KUS first invokes a d-DNNF compiler over the formula F (line 1) to obtain its d-DNNF. Then, the subroutine Annotate is invoked in line 2 that annotates d-DNNF by annotating each node with a tuple consisting of the number of solutions and the set of variables in the node's corresponding sub-formula. Then, the subroutine Sampler is invoked in line 3 that returns s uniformly and independently drawn samples. Finally, KUS gives random assignment to the unassigned variables for each sample in the SampleList to account for unconstrained variables that do not appear in d-DNNF by invoking the subroutine RandomAssignment (line 4). We now describe the two subroutines Annotate and Sampler in detail.

4.1 Annotate

The subroutine Annotate is presented in Algorithm 2. Annotate takes in a d-DNNF as input and returns an annotated d-DNNF where each node t is annotated with a tuple consisting of the number of solutions and the set of variables in the sub-formula rooted at t. Annotate performs

Algorithm 2 Bottom up pass for annotating d-DNNF

```
1: function Annotate(dag)
2:     for $t \in ReverseTopologicalOrder(\mathsf{dag})$ do
3:         if label(t) = Literal then
4:             $count(t) \leftarrow 1$
5:             $set(t) \leftarrow var(t)$
6:         else if label(t) = OR then
7:             $set(t) \leftarrow set(t.left) \cup set(t.right)$
8:             for $c \in \{t.left, t.right\}$ do
9:                 if $set(c) \neq set(t)$ then                ▷ upgrade
10:                    $count(c) \leftarrow count(c) * 2^{|set(t)|-|set(c)|}$
11:                    $set(c) \leftarrow set(t)$
12:            $count(t) \leftarrow count(t.left) + count(t.right)$
13:        else if label(t) = AND then
14:            for $c \in children(t)$ do
15:                $set(t) \leftarrow set(t) \cup set(c)$
16:                $count(t) \leftarrow count(t) * count(c)$
17:    return dag
18: end function
```

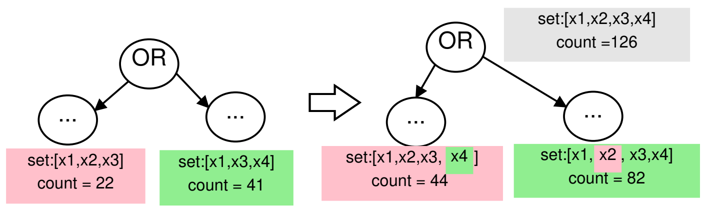

Figure 2: Upgrading nodes

a bottom-up traversal on d-DNNF in a reverse topological order such that a parent node is visited only after all its children nodes have been visited.

For each node t in the dag, Annotate maintains two attributes, **set** and **count**: the attribute **set** records the set of variables for the sub-formula rooted at t; the attribute **count** records the model count of the respective sub-formula rooted at node t with respect to its *set*. This is done as follows according to the label of the node:

Literal (lines: 3–5) : Add the respective variable to *set* and initialize *count* as 1;

OR (lines: 6–12) : As the children of an OR node represents a disjoint set of solutions of a sub-formula, there are two possibilities depending on whether *set* of variables for the children are identical or different. If the *set* for the two children are not identical, the *set* of variables for both the children are *upgraded* to the union of the *sets* of both the children by adding any missing variable(s). As missing variables imply unconstrained variables, the respective model count on these children are also updated to include all possible valuations on these variables. Figure 2 illustrates this case: to enable the constraint that both the children of an OR node contain the same set of variables as the OR node, the left child was upgraded to contain the variable x_4 and its model count increased to 44 (to allow for the valuations of the unconstrained variable x_4). The right child was also

Algorithm 3 Top Down pass for sampling s samples

```
 1: function Sampler($t, s$)
 2:     if label(t) = OR then
 3:         $p \leftarrow \frac{count(t.left)}{count(t)}$
 4:         $b \sim Binomial(s, p)$
 5:         $\mathsf{SampleList}_0 \leftarrow$ Sampler($t.left, b$)
 6:         $\mathsf{SampleList}_1 \leftarrow$ Sampler($t.right, s - b$)
 7:         SampleList $\leftarrow$ Shuffle($\mathsf{SampleList}_0 \cup \mathsf{SampleList}_1$)
 8:         return SampleList
 9:     else if label(t) = AND then
10:         SampleList = $EmptyList(s)$
11:         for $c \in children(t)$ do
12:             $\mathsf{SampleList}_c \leftarrow$ Sampler($c, s$)
13:             SampleList $\leftarrow$ Stitch(SampleList, $\mathsf{SampleList}_c$)
14:         return SampleList
15:     else                                                    ▷ label(t) == Literal
16:         SampleList $\leftarrow \emptyset$
17:         for i = 1 to $s$ do
18:             Append($val(t)$, SampleList)
19:         return SampleList
20: end function
```

upgraded in a similar manner to include the variable x_2. Following the upgrade process, the *set* is equal for both the children. Finally, the *count* is updated to the sum of count of node's children.

AND (lines: 13–16) : Since d-DNNF has the property of decomposition, it is implied that the *set* for AND node is the union of the (disjoint) *set* on its children and the *count* for an AND node is the product of the *count* of node's children.

Finally, Annotate returns the annotated d-DNNF.

4.2 Sampler

The subroutine Sampler is presented in Algorithm 3. Sampler takes the root node of an annotated d-DNNF, t, and the number of samples required, s, as input and returns a list of s samples. Similar to Annotate, Sampler takes actions based on the label of node (i.e., whether the node is an AND, OR, and a literal). The function EmptyList on line 10 takes a natural number s and returns a list containing s empty lists.

OR (lines 2–8) For an OR node, we first compute the probability p of choosing a sample from the left child by computing the ratio of the count of left child to the count for the node (line 3). Then, we compute the number of models to be sampled from left child, b, via a Binomial distribution with parameters s and p. We, then, extract b and $s - b$ samples from the left and right children. To ensure that every sample is identically and independently chosen, it is important that the union of two lists is randomly permuted, which is achieved by the Shuffle operation in line 7.

AND (lines 9–14) When an AND node is encountered, Sampler is invoked recursively on each of the children nodes. The disjoint *chunks* of solutions from each of these nodes are then concatenated (represented by the subroutine Stitch in line 13).

Literal (lines 15–19) We return the list with s copies of the literal. $val(t)$ refers to the value of the literal t.

Next, we present the theoretical analysis of our algorithm.

4.3 Theoretical Analysis

Lemma 1. *Every node t in the* dag *returned by* Annotate *is annotated with the count of the solutions and the support of the sub-formula corresponding to node t.*

Proof. The proof proceeds by induction on the height of the dag. The base case is trivial as for dag of height one, i.e. literal, the number of models allowed is 1. As OR node accumulates mutually disjoint solutions of the same constraint (via the property of determinism) from its children, the *support* for both the children should be same. Hence, the children of the OR node are upgraded to contain all possible valuations of the "missing" variables (those available in their sibling) to account for unconstrained variables; the model count on the OR node is then the sum of the (disjoint) solutions from both its children (the count on the children on the OR node is correct by the inductive hypothesis). For the AND node, the property of decomposability implies that all the solution of an AND node can compose with each other; hence, the count on the AND node is the product of the counts on all its children. □

Theorem 1. *For a given F and s, let L be the list of samples generated using* KUS. *Let $L[i]$ indicate the sample at the ith index of the list. Then for each $y \in R_F$, $\forall i \in [n]$ where n is the total number of samples, we have*

$$\Pr[y = L[i]] = \frac{1}{|R_F|}$$

Proof. Lemma 1 shows that dag returned by Annotate is annotated with the count of the solutions for the sub-formula corresponding to node t. To complete the proof, we focus on the subroutine Sampler. The proof proceeds by induction on the height of dag. The base case is trivial as for dag of height one, i.e. literal, there is only one solution and therefore, we apply our induction hypothesis to assume that our algorithm provides a uniformly random sample for all dag of height h. For a dag of height $h+1$, we have the following cases: if the root node is an AND node, the property of decomposability allows us to construct a sample selected uniformly at random by uniformly sampling a solution *segment* from each of the children and appending them. For an OR node, the property of determinism ensures disjoint solutions; hence, it boils down to the problem of selecting a model uniformly at random from two buckets (children nodes) having c_l and c_r number of solutions. To do the same, one performs a Bernoulli trial with $p = \frac{c_l}{c_l+c_r}$, and selecting a solution from the respective bucket of solutions. To draw s samples, s Bernoulli trials can be simulated by Binomial distribution and randomly permuting the union of the solution lists returned from the respective children. □

5 Evaluation

To evaluate runtime performance and quality of samples returned by KUS, we implemented a prototype in Python. KUS uses off the shelf d-DNNF compilers and in our experimental results, we used the tool D4 [29]. We conducted experiments on a wide range of publicly available benchmarks. In all, our benchmark suite consisted of 1425 benchmarks arising from wide range of application areas of uniform sampling such as probabilistic reasoning, Bounded Model

Checking, circuit, product configuration, SMTLib benchmarks, planning, quantified information flow and bug synthesis [6, 10, 11, 12, 29, 42]. We compared KUS with SPUR and UniGen2. For SPUR, we used the default parameters and set the cache size to 2 GB, which was empirically chosen to minimize the number of memory-outs on our experimental setup. UniGen2 was run with default parameters. The experiments were conducted on high performance computer cluster, where each node consists of E5-2690 v3 CPU with 24 cores and 96GB of RAM. All individual instances for each tool was executed on a single core.

The objective of our experimental evaluation was to answer the following questions:

1. How does KUS performs in terms of runtime in comparison to SPUR and UniGen2, the current state-of-the-art samplers for uniform sampling?
2. How does KUS performs in case of *incremental sampling*?
3. How does the distribution of samples generated by KUS compare with the ideal distribution?
4. How does the runtime of KUS scales with the number of samples drawn from it?
5. How does the performance of KUS vary for different d-DNNF compilers?

Our experiments showed that KUS outperformed both SPUR and UniGen2 by a factor of up to 3 orders of magnitude in terms of runtime in some cases while achieving a geometric speedup of 1.7× and 8.3× over SPUR and UniGen2 respectively. Also, KUS achieves a lower PAR-2[3] score equal to 396.49 compared to 484.01 of SPUR and 1059.42 of UniGen2. Furthermore, KUS achieves speedups of upto 3 orders of magnitude for *incremental sampling*. The distribution generated by KUS is statistically indistinguishable from that generated by an ideal uniform sampler. Moreover, KUS is almost oblivious to the number of samples requested. Finally, we observe that KUS can benefit from different d-DNNF compilers, therefore suggesting development of portfolio samplers in future.

5.1 Results

We present results for only a subset of representative benchmarks here. Detailed data along with expanded versions of all the Tables presented here is available at https://github.com/meelgroup/KUS.

Number of instances solved

We compared the runtime performance of KUS to two other state-of-the-art tools, SPUR [5] and UniGen2 [10] by generating 1000 samples from each tool with a timeout of 700 secs. Figure 3 shows the cactus plot for SPUR, KUS, and UniGen2. We present the number of benchmarks on $x-$axis and the time taken on $y-$axis. A point (x, y) implies that x benchmarks took less than or equal to y seconds to solve. All our runtime statistics for KUS ***include*** the time for the knowledge compilation phase (via D4).

We see that UniGen2 was able to complete only 370 formulas, while SPUR and KUS were able to complete 980 and 1049 formulas respectively. Table 1 shows the runtimes of some of the benchmarks on the three tools. The columns in the table give the benchmark name, number of variables, number of clauses, time taken in seconds by UniGen2, SPUR, and KUS divided into Compilation and A+S: Annotation and Sampling followed by speedup of KUS with respect to SPUR. Table 1 clearly shows that KUS outperforms UniGen2 and SPUR for most of the

[3]PAR-2 scheme, that is, penalized average runtime, used in SAT-2017 Competition [3], assigns a runtime of two times the time limit (instead of a not solved status) for each benchmark not solved by a solver

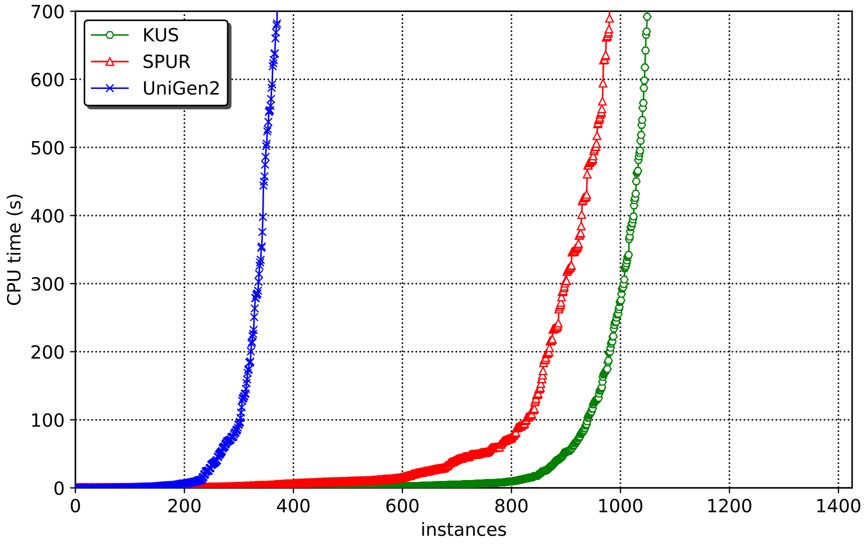

Figure 3: Cactus Plot comparing UniGen2, SPUR, and KUS

Table 1: Run time (in seconds) for 1000 samples

Benchmark	Vars	Clauses	UniGen2	SPUR	KUS			Speedup on SPUR
					Compile	A+S	Total	
LoginService	11511	41411	80.78	TO	TO	-	TO	-
IssueServiceImpl	1393	4319	6.28	4.71	1.48	9.47	10.95	0.43
UserServiceImpl	1509	5009	4.44	2.55	0.13	3.54	3.67	0.69
GuidanceService	988	3088	11.65	1.88	0.21	1.81	2.02	0.93
mastermind_06_08_03	3979	8833	TO	78.71	18.05	47.72	65.77	1.20
cnt07.shuffled	1786	5856	0.08	6.54	0.06	4.96	5.02	1.30
ais10	181	3151	457.53	5.10	2.35	0.61	2.96	1.72
uf250-031	250	1065	TO	29.22	5.74	0.41	6.15	4.75
bugsynthesisnc6	723	1273	TO	9.89	0.08	1.37	1.45	6.82
qg1-08	512	148957	3.34	31.58	3.12	0.74	3.86	8.18
or-50-20-1-UC-40	100	250	85.66	11.87	0.01	0.32	0.33	35.97
case10	328	878	TO	171.85	0.16	1.17	1.33	129.21
or-60-10-1-UC-40	120	300	TO	147.36	0.03	0.53	0.56	263.14
or-70-10-1-UC-30	140	350	TO	215.77	0.02	0.44	0.46	469.07

benchmarks. Over the entire set of benchmarks, there were 190 benchmarks for which SPUR failed to complete while KUS could successfully discharge the samples. On the other hand, for 121 benchmarks, KUS failed to complete but SPUR could sample successfully. Furthermore, for the 859 benchmarks that were successfully sampled by both KUS and SPUR, KUS outperformed SPUR for 446 benchmarks; KUS had speedup of (i) more than 10× for 229 instances, (ii) more than 100× for 102 cases, and (iii) more than 1000× for 14 cases.

Incremental Sampling

One of the biggest advantage of sampling from knowledge compilations is in incremental sampling—fetching multiple, relatively small-sized samples, repeatedly. The typical use case of

Table 2: Comparing runtimes (in seconds) of SPUR and KUS for incremental sampling

Benchmark	Vars	Clauses	SPUR		KUS		Speedup
			1000	10,000	1000	10,000	
PhaseService	1686	5655	2.6	26.61	4.11	31.10	0.89
IterationService	1896	6732	3.39	35.06	7.13	40.62	0.95
log-4	2303	20963	7.95	80.97	68.02	137.38	1.05
mastermind_03_08_04	4720	10920	51.46	539.66	62.29	277.50	2.27
C209_FC	1922	4805	33.19	325.92	306.61	355.33	6.01
uf250-054	250	1065	7.68	79.03	4.06	9.86	12.30
bugsynthesisnc20	500	1113	16.83	183.19	2.24	11.86	17.29
qg1-08	512	148957	33.46	320.84	3.88	10.50	43.41
or-50-5-2-UC-30	100	250	23.21	229.50	0.30	3.01	76.12
case145	219	558	62.57	637.18	0.63	4.74	139.81
or-60-10-9-UC-30	120	300	107.17	1073.19	0.31	2.99	360.46
or-70-20-6-UC-40	140	350	661.51	6676.41	0.32	3.47	1909.49

iterative sampling can be in repeated invocation of a sampling tool until the objective (such as desired coverage or violation of property) is achieved.

We evaluate KUS (with the D4 compiler) and SPUR for incremental sampling by invoking the respective tool for 1000 samples in 10 successive calls. The separation of compiling and sampling phase for KUS allows us to invoke Annotate only once and save its dag for the subsequent invocations. Table 2 presents the results for a subset of the benchmarks. The first column presents the benchmark id while the second and third column represent the number of variables and clauses in the benchmark. The fourth and fifth column represent the time taken by SPUR for the first 1000 samples and the total time taken for 10,000 samples. The sixth and seventh column show the time taken by KUS for the first 1000 samples and the total time taken for 10,000 samples. The final column shows the speedup of KUS over SPUR for the subsequent 9000 samples.

Table 2 clearly demonstrates the advantage of uniform sampling on knowledge compilations; while KUS can reuse the d-DNNF representation for the subsequent calls, SPUR is required to perform a completely new exhaustive DPLL exploration to collect samples for the subsequent phase. To further demonstrate the advantage of separation of compiling and sampling steps, we show the trend of one of the benchmarks (*or-60-20-8-UC-10*) in Figure 4. The $x-$axis represents the the number of samples while $y-$axis represents the time taken by KUS and SPUR. We see that while KUS takes longer to compute the first 1000 samples than SPUR but the subsequent sampling time for KUS is significantly smaller than SPUR.

Uniformity Comparison

Theorem 1 shows that KUS is a uniform sampler. In line with previous, we seek to compare the distribution of samples generated by KUS vis-a-vis *an ideal uniform sampler* (similar to [10]) henceforth denoted as IS: given a formula F, IS enumerates all the models and, then, picks a model uniformly at random. We measured the KL-divergence between the distribution generated by samplers (KUS, SPUR, UniGen2 and IS) and the expected Poisson distribution. The divergence between the distribution generated by the IS and the Poisson distribution provide a measure for all the other samplers.

Figure 6 reports the KL-divergence for case110 that has 16,834 models: the ratio of divergence is close to 1. For this formula we generated 4×10^6 samples. Since the number of samples is much larger than the number of models, each model occurred multiple times in the list of

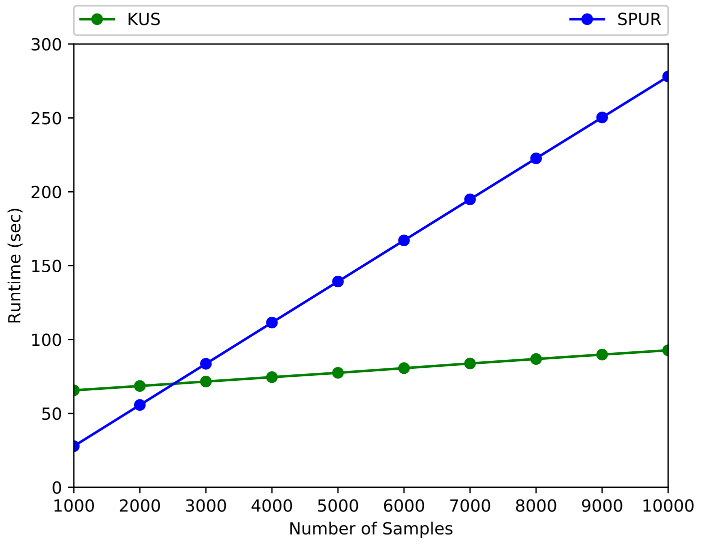

Figure 4: Cumulative runtimes for incremental sampling (*or-60-20-8-UC-10*)

samples. We then computed the frequency of generation of individual model and grouped the models that had the same frequency. Figure 5 plots the output of KUS, UniGen2, SPUR and IS with each (x, y) indicating that the x distinct models are generated y times. It is clear from Figure 5 that the distribution generated by KUS is practically indistinguishable from IS, which is also expected from the theoretical analysis of KUS.

Effect of number of samples

To evaluate the scalability of KUS (with D4) with the number of samples drawn, we invoked our tool for fetching different number of samples: 100, 200, 400, 800 and 1000. Table 3 presents the runtime of KUS for different samples. The first column gives the benchmark name and the next five columns show time taken by KUS for 100, 200, 400, 800 and 1000 samples. Table 3 clearly demonstrates that KUS is almost oblivious to the number of samples requested.

Runtime impact of d-DNNF generation tool

We evaluate how KUS responds to two of the state-of-the-art d-DNNF compilers: D4 and Dsharp. For this experiment, we generate 1000 samples from the respective d-DNNF DAG; we run both KUS with Dsharp and KUS with D4 with a timeout of 700 secs. Figure 7 shows a scatter plot for KUS with D4 vis-a-vis KUS with Dsharp. A point (x, y) indicates that KUS with D4 took x seconds to solve the particular formula while KUS with Dsharp took y seconds. First of all, KUS with D4 was able to sample from 1049 benchmarks while KUS with Dsharp was able to sample from 1005 benchmarks. Secondly, KUS with D4 outperformed KUS with Dsharp for 485 instances against 458 instances for the opposite case. As the knowledge compilation technique continue to develop, we expect development of tools that would further speedup the compilation process. Our decision to separate the compiling and sampling phase would allow

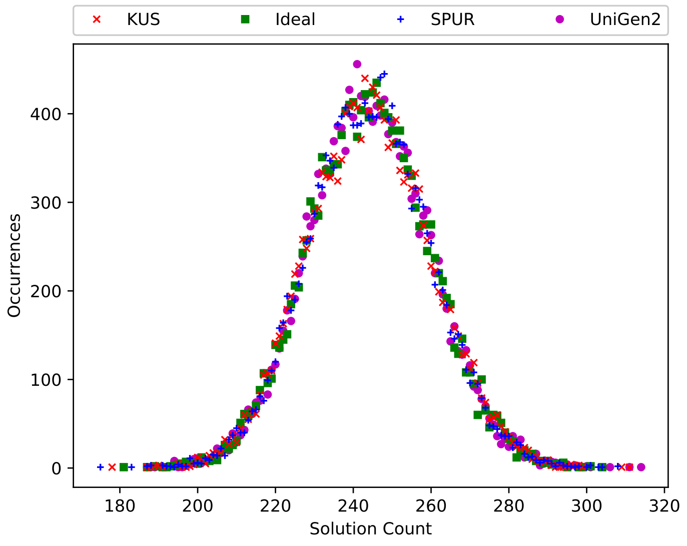

Figure 5: Uniformity Comparison between IS, KUS, UniGen2 and SPUR on case110 with 4×10^6 samples.

Sampler	KL-Divergence
Ideal	0.092818
KUS	0.093250
SPUR	0.092849
UniGen2	0.093867

Figure 6: KL-Divergence between the distribution generated by the samplers and Poisson distribution for case110

Table 3: Runtime (in seconds) of KUS to generate different size samples

Benchmarks	Vars	Clauses	Sample Size				
			100	200	400	800	1000
fs-01	32	38	0.18	0.17	0.21	0.18	0.18
2bitcomp_5	125	310	2.45	2.47	2.59	2.82	2.81
s1238	539	1549	3.74	3.84	4.09	4.43	4.62
IssueServiceImpl.sk	1393	4319	8.53	8.74	9.14	10.06	10.58
bugsynthesisnc24	1544	3048	13.51	13.63	14.17	13.77	14.13
uf250-016	250	1065	17.70	17.46	19.40	18.21	16.98
s1423a_7_4	795	1964	55.57	56.77	58.63	61.82	55.95
logistics.b	843	7301	51.26	52.42	53.72	53.72	53.98
or-70-10-4-UC-30	140	350	117.23	110.87	116.18	113.03	111.79
or-100-10-2-UC-20	200	500	131.28	129.41	128.34	130.65	129.97
or-60-20-2-UC-10	120	300	355.89	353.98	355.19	357.53	359.12
case139	846	2163	378.40	382.39	363.97	366.49	371.82
sat-grid-pbl-0010	110	191	529.87	555.27	511.70	513.42	521.67

KUS to benefit from advances in knowledge compilation.

6 Conclusion

In this paper, we proposed a new approach for uniform sampling that builds on breakthrough progress in knowledge compilation. The primary contribution of this paper is marrying knowledge compilation with uniform sampling. Our algorithm, KUS, employs the state-of-the-art knowledge compilers, first to compile constraints into d-DNNF form and then generates sam-

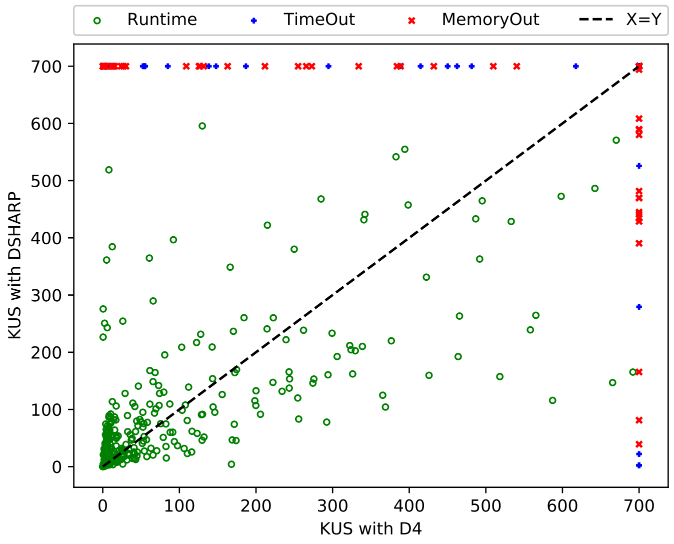

Figure 7: Runtime comparison of sampling with KUS by using D4 and Dsharp

ples by making a single pass over the compiled representations. We show that KUS is able to significantly outperform existing state-of-the-art algorithms, solving more instances than any of the competing tools.

In our experiments, KUS outperforms both SPUR and UniGen2 by a factor of up to 3 orders of magnitude in terms of runtime while achieving a geometric speedups of 1.7× and 8.3× over SPUR and UniGen2 respectively. Also, KUS achieves a lower PAR-2 score, around 0.82× that of SPUR and 0.38× that of UniGen2. Furthermore, KUS achieves speedups of upto 3 orders of magnitude for *incremental sampling*. The distribution generated by KUS is statistically indistinguishable from that generated by an ideal uniform sampler. Moreover, KUS is almost oblivious to the number of samples requested. Finally, we observe that KUS can benefit from different d-DNNF compilers, therefore suggesting development of portfolio samplers in future. We believe that the success of KUS will motivate researchers in verification and knowledge compilation communities to investigate a broader set of logical forms amenable to efficient uniform generation.

Acknowledgments

This work was supported in part by NUS ODPRT Grant, R-252-000-685-133. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg. We are thankful to Mate Soos for useful discussions and for guiding us with various useful scripts used in the experimental comparisons. We also thank the reviewers for their insightful comments.

References

[1] Buddy - a binary decision diagram package. http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/.

[2] Muddy - an sml interface for buddy. https://www.itu.dk/research/muddy/.

[3] Proc. of SAT COMPETITION 2017 Solver and Benchmark Descriptions. https://baldur.iti.kit.edu/sat-competition-2017/index.php?cat=results.

[4] The SDD Package: Version 1.1.1. http://reasoning.cs.ucla.edu/sdd/.

[5] Dimitris Achlioptas, Zayd S. Hammoudeh, and Panos Theodoropoulos. Fast sampling of perfectly uniform satisfying assignments. In *Proc. of SAT*, pages 135–147, 2018.

[6] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proc. of FMCAD*, pages 1–8, 2013.

[7] Mihir Bellare, Oded Goldreich, and Erez Petrank. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation*, 163(2):510–526, 2000.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proc. of ACM SIGSAC*, pages 1032–1043, 2016.

[9] R. E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[10] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. On parallel scalable uniform sat witness generation. In *Proc. of TACAS*, pages 304–319, 2015.

[11] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A Scalable and Nearly Uniform Generator of SAT Witnesses. In *Proc. of CAV*, pages 608–623, 2013.

[12] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proc. of DAC*, pages 1–6, 2014.

[13] Mark Chavira and Adnan Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6):772–799, 2008.

[14] Lori A. Clarke. A program testing system. In *Proc. of ACM*, pages 488–491, 1976.

[15] Gregory F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial intelligence*, 42(2):393–405, 1990.

[16] A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proc. of IJCAI*, pages 819–826, 2011.

[17] Adnan Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proc. of ECAI*, pages 318–322, 2004.

[18] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[19] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proc. of TACAS*, pages 337–340, 2008.

[20] Rina Dechter, K. Kask, E. Bin, and R. Emek. Generating random solutions for constraint satisfaction problems. In *Proc. of AAAI*, pages 15–21, 2002.

[21] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Efficient sampling of SAT solutions for testing. In *Proc. of ICSE*, pages 549–559, 2018.

[22] Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman. Embed and project: Discrete sampling with universal hashing. In *Proc. of NIPS*, pages 2085–2093, 2013.

[23] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proc. of ESEC/FSE*, pages 416–419, 2011.

[24] Vibhav Gogate and Rina Dechter. A new algorithm for sampling CSP solutions uniformly at random. In *Proc. of CP*, pages 711–715. Springer, 2006.

[25] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proc. of USENIX*, Security'12, pages 38–38, 2012.

[26] Allen Householder and Jonathan Foote. Probability-based parameter selection for black-box fuzz testing. Technical Report CMU/SEI-2012-TN-019, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2012.

[27] Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *Proc. of IJCAI*, volume 5, pages 156–162, 2005.

[28] Mahesh A. Iyer. RACE: A word-level ATPG-based constraints solver system for smart random simulation. In *Proc. of ITC*, pages 299–308. Citeseer, 2003.

[29] Pierre Marquis Jean-Marie Lagniez. An improved decision-dnnf compiler. In *Proc. of IJCAI*, pages 667–673, 2017.

[30] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43(2-3):169–188, 1986.

[31] J.C. King. Symbolic execution and program testing. *Comm. ACM*, 19(7):385–394, 1976.

[32] Nathan Kitchen. *Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation*. PhD thesis, University of California, Berkeley, 2010.

[33] Nathan Kitchen and Andreas Kuehlmann. Stimulus generation for constrained random simulation. In *Proc. of ICCAD*, pages 258–265, 2007.

[34] James H Kukula and Thomas R Shiple. Building circuits from relations. In *Proc. of CAV*, pages 113–123, 2000.

[35] Kuldeep S Meel. *Constrained Counting and Sampling: Bridging the Gap between Theory and Practice*. PhD thesis, Rice University, 2017.

[36] Kuldeep S Meel. Sampling techniques for boolean satisfiability, MS Thesis, Rice University, 2014.

[37] Kuldeep S. Meel, Moshe Y. Vardi, Supratik Chakraborty, Daniel J Fremont, Sanjit A Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. Constrained sampling and counting: Universal hashing meets sat solving. In *Proc. of Beyond NP Workshop*, 2016.

[38] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proc. of DAC*, pages 530–535. ACM, 2001.

[39] Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric Hsu. DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Proc. of CAIAC*, 2012.

[40] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. In *Proc of IAAI*, pages 1720–1727, 2006.

[41] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1):273–302, 1996.

[42] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proc. of ESEC/FSE*, 2018.

[43] Wei Wei and Bart Selman. A new approach to model counting. In *Proc. of SAT*, pages 2293–2299. Springer, 2005.

[44] Jun Yuan, Adnan Aziz, Carl Pixley, and Ken Albin. Simplifying boolean constraint solving for random simulation-vector generation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(3):412–420, 2004.