GUIDEDSAMPLER: Coverage-guided Sampling of SMT Solutions

Rafael Dutra, Jonathan Bachrach and Koushik Sen
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{rtd, jrb, ksen}@cs.berkeley.edu

***Abstract*—The problem of sampling a large number of random solutions to SAT and SMT constraints is essential for constrained-random verification and testing. However, most current sampling techniques lack a problem-specific notion of coverage, considering only general goals such as uniform distribution. We have developed a new technique for *coverage-guided sampling* that allows the user to specify the desired coverage points in order to shape the distribution of solutions. Our tool GUIDEDSAMPLER can efficiently generate high-quality stimuli for constrained-random verification, by sampling solutions to a SMT constraint that also cover a large number of user-defined coverage classes.**

I. INTRODUCTION

Given a logical constraint, the problem of generating a large number of random solutions to the constraint has important applications in *constrained-random verification* (CRV) [1], testing [2] and synthesis [3]. In CRV, the constraints represent preconditions required by the design under test, which can be manually specified by the verification engineer using domain-specific knowledge [4], [5] or automatically generated from a formal specification of the design [6]. The solutions to such constraints can then be used as random stimuli to exercise the design under test.

In *constraint-based fuzzing* [2], path constraints for a given execution path of interest can be obtained through *dynamic symbolic execution* [7], [8], [9]. Solutions sampled from such constraints can be used to randomly and thoroughly explore execution paths that start from the same prefix of interest, mitigating the computational cost associated with the symbolic execution and constraint solving of those paths.

Random sampling can also be applied to synthesis problems. For example, in *counterexample-guided inductive synthesis* (CEGIS) [3], a verifier is responsible for checking the candidate expressions produced by the learner and producing counterexamples for invalid candidates. If, instead of producing a single counterexample, the verifier could sample a diverse set of counterexamples, this could help the learner in finding a valid candidate faster.

For all those applications, constraints might be expressed as *satisfiability modulo theories* (SMT) [10] formulas, involving theories such as bit-vectors, arrays and uninterpreted functions. The constraints can be quite complex, typically including hundreds of variables. SMT-LIB [11] provides a formal language and theories for specifying the constraints, as well as a large set of industrial benchmarks for evaluation.

Despite its importance, the problem of efficiently sampling diverse solutions to complex SMT constraints is still challenging today. If one does not carefully specify the desired distribution of solutions, the generated samples can be too biased and uninteresting. For example, consider a constraint of the form $x > 4 \ \vee \ \phi(x, y, z)$ where ϕ is some complex formula involving variables x, y, z. If x is a bit-vector of size 32, it is trivial to generate billions of solutions to the constraint by just choosing a value of x that satisfies $x > 4$. However, this generates solutions that are only trivially different and does not explore scenarios where the second disjunct holds and values of y and z are also mutated.

General notions of coverage, such as uniform distribution [12], [13], [2], or internal coverage of a SMT formula [14], may not be the most appropriate for a particular problem. For example, when testing a hardware design that implements a finite-state machine, a better metric for coverage would be making sure all relevant states and transitions are reached. Uniform sampling, on the other hand, might be too frequently generating solutions from the most common state. Even in scenarios where a general notion of coverage is suitable, our evaluation shows that existing sampling algorithms are not always optimal in maximizing this coverage.

To address these challenges, we formulate the problem of *coverage-guided sampling*, where the user can specify not only the constraint that must be satisfied, but also any number of coverage predicates that will be used to guide the distribution of solutions. We define different classes of solutions based on the coverage predicates and establish the goal of sampling from each coverage class with equal weight.

We developed a technique, called GUIDEDSAMPLER, which dynamically guides the search of new solutions based on the coverage predicates. It requires only a small number of calls to an off-the-shelf constraint solver in order to generate millions of solutions. GUIDEDSAMPLER starts by finding one base solution from a random coverage class. Then, it finds some simple mutations that can be applied to this solution in order to generate another solution from a neighboring coverage class. It then combines multiple of those mutations together to generate new solutions from previously unseen coverage classes. Our evaluation shows that GUIDEDSAMPLER outperforms existing techniques in the number of coverage classes reached, both when using general coverage predicates and also problem-specific coverage predicates.

Our main contributions are:

- Formally specify the problem of *coverage-guided sampling*.
- Develop a technique GUIDEDSAMPLER and implement it in an open-source tool for efficient coverage-guided sampling from SMT formulas.
- Evaluate GUIDEDSAMPLER against existing techniques on a large set of complex benchmarks from SMT-LIB, both using a general notion of internal coverage of SMT formulas and a problem-specific coverage notion based on random predicates.

The paper is organized as follows. Section II presents existing work in sampling and weighted sampling from SAT and SMT constraints. Section III formally defines the problem of coverage-guided sampling and Section IV presents our proposed algorithm GUIDEDSAMPLER. Finally, Section V evaluates GUIDEDSAMPLER against existing techniques and Section VI concludes the results.

II. RELATED WORK

There is a large number of techniques dedicated to sampling solutions to logical constraints, especially for Boolean (SAT) problems [15]. For example, techniques may be based on model counting [12], Markov Chain Monte Carlo (MCMC) [16], [17], [18], SAT solver search heuristics [19], combination of mutations [2], [14] and universal hashing [20], [21]. Most sampling techniques have a general goal about the desired distribution of solutions, such as uniform distribution [22], [13], not allowing more specific notions of coverage.

For SMT constraints in particular, SMTSAMPLER [14] is a state-of-the-art sampler that can sample from formulas with bit-vectors, arrays and uninterpreted functions. This technique has a goal of generating solutions that maximize the internal coverage of the constraint. Our tool GUIDEDSAMPLER is based on SMTSAMPLER, but extends it with 3 important modifications to allow coverage-guided sampling based on arbitrary coverage predicates. Our modifications use information about the coverage class of solutions in order to guide the search into exploring new coverage classes. They will be described in detail in Section IV and evaluated in Section V.

Some sampling techniques, such as MCMC, can be adapted to the context of weighted sampling. However, they frequently fail to converge to the desired distribution in practice. There are techniques that can sample from a literal-weighted distribution [23], such as WAPS [24], however, not all distributions can be specified by assigning weights to literals in the formula. GUIDEDSAMPLER, on the other hand, allows the desired distribution to be specified by coverage points, which are typically already present in testing and verification applications. GUIDEDSAMPLER also enables sampling directly from SMT formulas, without converting into SAT.

III. PROBLEM FORMULATION

Definition III.1. *SMT formula.* In this work, we only consider constraints in the QF_AUFBV logic of SMT, which are quantifier-free formulas over the theories of bit-vectors, bit-vector arrays, and uninterpreted functions. Given a SMT formula ϕ, let $Vars[\phi]$ represent the set of variables of ϕ. Let $Bool$, BV, and $Array$ be the sets of variables of type Boolean, bit-vector, and array, respectively. Also let $Sols[\phi]$ represent the set of solutions to ϕ. Given an assignment σ to the variables of ϕ, we define $\phi[\sigma] \in Bool$ as the Boolean value resulting from evaluating ϕ under the assignment σ. For example, $b \vee (x + 2 > y)$ represents a SMT formula over Boolean variable b and bit-vector variables x and y. One example solution would be σ_0 that satisfies $\sigma_0[\![b]\!] = False$, $\sigma_0[\![x]\!] = 5$ and $\sigma_0[\![y]\!] = 2$.

In the following definitions, let ϕ be a satisfiable SMT formula (i.e., $Sols[\phi] \neq \varnothing$) over variables $V = Vars[\phi]$ and let $\psi_1, \psi_2, \ldots, \psi_n$ be n SMT formulas which only include variables from V (i.e., $Vars[\psi_i] \subseteq V$). The formulas ψ_i represent the coverage predicates of interest.

Definition III.2. *Coverage class.* Each vector of possible Boolean values $b = (b_1, b_2, \ldots, b_n) \in Bool^n$ defines one coverage class G_b of formula ϕ over the coverage predicates $\psi_1, \psi_2, \ldots, \psi_n$, as follows:

$$G_b = \{\sigma \in Sols[\phi] \mid \psi_i[\sigma] = b_i,\ i \in \{1, 2, \ldots, n\}\}$$

That is, the set of solutions to ϕ that satisfy $\psi_i[\sigma] = b_i$ for each predicate defines a class of solutions of ϕ. Some of those classes might be empty, if they include no solutions to ϕ. Let N be the number of non-empty coverage classes, $N = |\{b \in Bool^n \mid G_b \neq \varnothing\}|$. For each assignment σ to the variables of ϕ, its coverage class is defined as

$$G[\sigma] = G_{(\psi_1[\sigma], \psi_2[\sigma], \ldots, \psi_n[\sigma])}$$

For our example formula $\phi = b \vee (x + 2 > y)$, if we have coverage predicates $\psi_1 = b$ and $\psi_2 = x + 2 > y$, then $G_{(False, False)} = \varnothing$ and there are only $N = 3$ non-empty coverage classes.

Definition III.3. *Ideal coverage-guided sampler.* An ideal coverage-guided sampler for formula ϕ with coverage predicates $\psi_1, \psi_2, \ldots, \psi_n$ is a random process that returns each possible solution $\sigma \in Sols[\phi]$ with probability

$$P(\sigma) = \frac{1}{N \cdot |G[\sigma]|}$$

What this means is that we want to give equal weight to each of the non-empty coverage classes. Every non-empty coverage class should have the same probability $1/N$ of being sampled. Within each individual coverage class, all solutions should be sampled uniformly.

If we could enumerate all solutions to the constraint ϕ, an ideal coverage-guided sampler could be constructed as follows.
(1) First, pick one non-empty coverage class uniformly.
(2) Then, uniformly pick one solution from that class.

However, for most practical problems, the number of solutions and coverage classes is astronomically large. Our goal is to devise an efficient algorithm that can approximate an ideal coverage-guided sampler.

IV. ALGORITHM

GUIDEDSAMPLER uses a small number of calls to an off-the-shelf constraint solver in order to generate a large number of solutions. The core idea is to learn interesting ways that the solutions of a formula can be modified minimally to generate new solutions that belong to different coverage classes. We call those modifications *atomic mutations*, which are minimal changes that can be applied to a solution in order to obtain another neighboring solution to the formula. We then define a combination function which can be used to merge the effects of several distinct atomic mutations. This function generates a compound mutation and applies it to the original solution, producing a possibly new solution. It is important to note that computing mutations requires expensive solver calls. However the combination function does not, making it very efficient. This efficient combination of solutions can be leveraged to generate millions of samples from just a few hundreds of atomic mutations. The samples generated by the combination function are assignments which may or may not satisfy the formula. However, our experiments show that they have a high probability of satisfying the formula, even on large and complex industrial benchmarks. Moreover, the new samples typically belong to previously unseen coverage classes.

Next we present the full details of the algorithm. In Section IV-A we describe the main sampling procedure of GUIDEDSAMPLER. Next, we explain in Section IV-B how one base solution is chosen for each epoch, and in Section IV-C how we discover a set of neighboring solutions to the base solution. Finally, in Section IV-D, we describe how those neighboring solutions are used to generate new samples.

A. Main GUIDEDSAMPLER Algorithm

Algorithm 1 presents the main GUIDEDSAMPLER procedure, which takes as input a SMT formula ϕ and n coverage predicates $\psi_1, \psi_2, \ldots, \psi_n$. The main algorithm is based on our prior tool SMTSAMPLER [14], which is a state-of-the-art technique for sampling solutions to SMT formulas. However, we added 3 important modifications to this base technique to allow coverage-guided sampling based on the coverage predicates (changes are underlined in Algorithm 1):

- M1 Generate neighboring solutions from a neighboring coverage class, by flipping the values of coverage predicates in function COMPUTENEIGHBORSOLS.
- M2 Discard solutions that belong to a previously seen coverage class in the current epoch (lines 24 and 36).
- M3 Randomize the coverage class of the initial base solution σ (lines 3-7), instead of generating σ based only on a random assignment σ'.

Next, we explain the algorithm in detail. The GUIDEDSAMPLER algorithm works over several epochs. In each epoch, we start by generating an initial random solution σ to the formula, which we call a *base solution*. This is done by generating a random coverage class G_b, as well as a random assignment σ' to the variables of the formula in lines 3-6 and using a MAX-SMT [25] solver call to obtain a solution σ which is closest to G_b and to σ' in line 7. The details of this procedure are given in Section IV-B. Then, in line 9, we use the function COMPUTENEIGHBORSOLS to compute a set Σ^1_σ of *neighboring solutions* to σ that belong to different coverage classes than σ. This function will be described in Section IV-C.

The mutations that can be applied to σ in order to produce neighboring solutions are called atomic mutations. Our key idea to producing new samples is by defining a *combination function* $\Psi : S \times S \times S \rightarrow S$, where S is the space of all possible assignments to the variables in $V = Vars[\phi]$.

Algorithm 1 GUIDEDSAMPLER($\phi, \{\psi_1, \psi_2, \ldots, \psi_n\}$)

```
1:  function GuidedSampler
2:      while not done do
3:          (b_1, b_2, ..., b_n) ← genRandomClass(n)
4:          S_b ← {ψ_1 = b_1, ψ_2 = b_2, ..., ψ_n = b_n}
5:          σ' ← genRandomAssignment(φ)
6:          C_σ' ← getConditions(φ, σ')
7:          σ ← MAX-SMT({φ}, S_b ∪ C_σ')
8:          output({σ})
9:          (Σ^1_σ, Γ_σ) ← computeNeighborSols(σ)
10:         output(Σ^1_σ)
11:         α ← 1, k ← 1, Σ_σ ← Σ^1_σ
12:         while α ≥ α_min ∧ k < 6 do
13:             (Σ^{k+1}_σ, α, Σ_σ, Γ_σ) ← join(Σ^k_σ, Σ^1_σ, Σ_σ, Γ_σ)
14:             output(Σ^{k+1}_σ)
15:             k ← k + 1
16:
17: function computeNeighborSols(σ)
18:     (b_1, b_2, ..., b_n) ← (ψ_1[σ], ψ_2[σ], ..., ψ_n[σ])
19:     S_b ← {ψ_1 = b_1, ψ_2 = b_2, ..., ψ_n = b_n}
20:     C_σ ← getConditions(φ, σ)
21:     Σ^1_σ ← {}, Γ_σ ← {}
22:     for c in S_b do
23:         σ̃ ← MAX-SMT({φ ∧ ¬c}, C_σ ∪ S_b \ {c})
24:         if σ̃ ∧ G[σ̃] ∉ Γ_σ then
25:             Γ_σ ← Γ_σ ∪ G[σ̃]
26:             Σ^1_σ ← Σ^1_σ ∪ {σ̃}
27:     return (Σ^1_σ, Γ_σ)
28:
29: function join(Σ^k_σ, Σ^1_σ, Σ_σ, Γ_σ)
30:     valid ← 0, checks ← 0
31:     for (σ_a, σ_b) in Σ^k_σ × Σ^1_σ do
32:         σ̃ ← Ψ_σ(σ_a, σ_b)
33:         if σ̃ ∉ Σ_σ then
34:             Σ_σ ← Σ_σ ∪ {σ̃}
35:             checks ← checks + 1
36:             if φ[σ̃] ∧ G[σ̃] ∉ Γ_σ then
37:                 Γ_σ ← Γ_σ ∪ G[σ̃]
38:                 Σ^{k+1}_σ ← Σ^{k+1}_σ ∪ {σ̃}
39:                 valid ← valid + 1
40:     return (Σ^{k+1}_σ, valid/checks, Σ_σ, Γ_σ)
```

We denote by $\Psi_\sigma(\sigma_a, \sigma_b)$ the application of the combination function to the base solution σ and two other solutions σ_a and σ_b. Intuitively, the combination function Ψ computes the mutations which can be applied to σ to generate σ_a and σ_b, then merges those two mutations together to produce a new assignment. The assignment returned by Ψ is not guaranteed to satisfy the formula, but in practice it is a solution with high probability. This is because the atomic mutations capture the minimal changes that preserve the satisfiability of the formula, and we designed Ψ to combine those changes in an additive way. The full definition of Ψ is given in Section IV-D.

We next describe how function JOIN called by GUIDEDSAMPLER in line 13 uses Ψ to generate new samples. We denote by Σ^1_σ the set of neighboring solutions to σ obtained from COMPUTENEIGHBORSOLS. Starting from Σ^1_σ, our goal is to compute sets Σ^k_σ which will contain solutions generated by combining k atomic mutations, for $1 \leq k \leq 6$. Throughout the current epoch, we maintain a set Σ_σ of samples which were computed so far, both valid and invalid. Initially, $\Sigma_\sigma = \Sigma^1_\sigma$. We also maintain a set Γ_σ of all the coverage classes for which we have generated a solution in the current epoch.

Now assume that we already constructed a set Σ^k_σ. We can inductively build the set Σ^{k+1}_σ as follows. For each pair of samples $\sigma_a \in \Sigma^k_\sigma$ and $\sigma_b \in \Sigma^1_\sigma$, we apply the combination function Ψ to generate a new sample $\tilde{\sigma} = \Psi_\sigma(\sigma_a, \sigma_b)$. If $\tilde{\sigma}$ is an element of Σ_σ, it has already been checked and is discarded. Otherwise, we add it to Σ_σ and check if it is a solution to the formula. We also check if its coverage class $G[\tilde{\sigma}]$ is one for which we have already found a solution in the current epoch. Those checks are fast, as they only need to evaluate the formulas using the assignments in $\tilde{\sigma}$. If $\tilde{\sigma}$ is a solution to ϕ and comes from a previously unseen class, it is added to Σ^{k+1}_σ. During the construction of Σ^{k+1}_σ from Σ^k_σ, we also keep statistics on which fraction α of the checked samples were valid, so we can stop the process earlier if we detect that the algorithm is generating too many invalid samples.

All the samples which are ultimately output by GUIDEDSAMPLER are the ones in $\cup_{0 \leq k \leq 6} \Sigma^k_\sigma$, where we define $\Sigma^0_\sigma = \{\sigma\}$. Those are all solutions to the formula, as the ones which were produced by the combination function for $2 \leq k \leq 6$ have been checked for validity.

B. Computing the Base Solution

Now, we describe how the initial base solution σ is obtained. We first generate a random coverage class G_b by picking n random Boolean values $b = (b_1, b_2, \ldots, b_n)$. We also generate a random assignment σ' by choosing values to the Boolean and bit-vector variables in the formula uniformly at random. After choosing G_b and σ', we want to find a solution σ which is as close as possible to the coverage class G_b and to the random assignment σ'. This is done to explore as much of the coverage classes and the solution space as possible.

The problem of finding a solution σ which is as close as possible to G_b and σ' can be encoded as a MAX-SMT [25] optimization problem to be solved by the constraint solver. The MAX-SMT$(Hard, Soft)$ optimization problem is the problem of finding a solution to a SMT formula that must satisfy a set $Hard$ of hard constraints and should also satisfy the maximum possible number of constraints in the set $Soft$ of soft constraints. We encode the MAX-SMT query as follows. We add one hard constraint stating the the formula ϕ must be satisfied. We also add two sets of soft constraints, S_b and $C_{\sigma'}$. For each coverage predicate, we add one soft constraint to S_b saying that the predicate ψ_i should be equal to the random Boolean b_i. For each bit-vector or Boolean variable $v \in Vars[\phi]$, we add one soft constraint to $C_{\sigma'}$ stating that variable v should have the same value that it has in σ'.

C. Computing Atomic Mutations

After generating a base solution σ, we compute neighboring solutions of the base solution σ, so that their atomic mutations can be combined to generate new samples. When doing so, we want to generate neighboring solutions that are from different classes than the base solution σ. We want our neighboring solutions to flip the value of some of the coverage predicates. The first step is collecting a set of conditions S_b which are true about the coverage predicates for the base solution σ. Then, MAX-SMT queries are used to produce new neighboring solutions. Each MAX-SMT query attempts to flip one of those conditions, while maintaining the remaining conditions valid, if possible. We also add soft constraints for the variables in ϕ, saying that those variables should keep the same value they had in the base solution, if possible. This is done because we want the neighboring solution to be as close as possible to the original base solution, so that the atomic mutations are simple. We found that this is essential for having a good probability of generating valid samples when combining mutations.

a) Constructing S_b: The definition of S_b is the same as in Section IV-B. For each coverage predicate, we add one condition to S_b saying that this predicate has the same value it had in the base solution σ. Then, in our MAX-SMT queries, we will try to flip one of those conditions, while maintaining as many as possible of the other conditions in S_b true. The goal is that the neighboring solution produced should come from a neighboring coverage class, that only flips a minimal number of coverage predicates.

b) Constructing C_σ: Function GETCONDITIONS produces C_σ by collecting conditions for each variable in the formula. The conditions can be, for example, $extract(v, 5) = 1$, saying that the bit of index 5 in the bit-vector v has value 1. Those are the same conditions generated by SMTSAMPLER using its default strategy (SMTbit). For each bit inside each variable in the formula, we add a condition asserting that this bit has the same value as in the base solution.

c) Computing Σ^1_σ: After collecting the conditions in S_b and C_σ, we want to compute neighboring solutions by picking one condition $c \in S_b$ and using the constraint solver to find a solution to $\phi \wedge \neg c$. However, the neighboring solution should be as similar as possible to σ. We express such constraint by requiring that the new solution should satisfy the maximum possible number of the remaining conditions

$$
\begin{array}{rlll}
v: & 1\,1\,0\,0\,0\,1\,0\,1 & & 0\,1\,0 \\
v_a: & 0\,1\,0\,0\,0\,0\,1\,1 & & 1\,1\,0 \\
v_b: & 1\,1\,0\,1\,0\,0\,0\,1 & & 0\,0\,1 \\
\delta_a = v \oplus v_a: & 1\,0\,0\,0\,0\,1\,1\,0 & & 1\,0\,0 \\
\delta_b = v \oplus v_b: & 0\,0\,0\,1\,0\,1\,0\,0 & & 0\,1\,1 \\
(\delta_a \vee \delta_b): & 1\,0\,0\,1\,0\,1\,1\,0 & & 1\,1\,1 \\
\psi_v(v_a, v_b) = v \oplus (\delta_a \vee \delta_b): & 0\,1\,0\,1\,0\,0\,1\,1 & & 1\,0\,1
\end{array}
$$

Fig. 1. Combining two mutations over bit-vectors of size 8. We also list values for 3 coverage predicates.

in $S_b \backslash \{c\}$ and C_σ. Those requirements can be specified as a MAX-SMT optimization problem. We specify two hard constraints $\{\phi, \neg c\}$, stating that we want a solution to ϕ that does not satisfy c. And we also specify as soft constraints the conditions in $S_b \backslash \{c\}$ and in C_σ.

D. Combining Mutations

Now we define the combination function Ψ which we use to generate new samples. Assume that we already know the base solution σ and two additional solutions to the formula σ_a and σ_b, which are close to σ. Those additional solutions can be obtained by calling COMPUTENEIGHBORSOLS or they could be already generated by an application of the Ψ function.

The combination function Ψ, which combines entire solutions, is constructed by defining a method ψ to combine the values of each of the variables in the formula. We define

$$\Psi_\sigma(\sigma_a, \sigma_b)[\![v]\!] = \psi_{\sigma[\![v]\!]}\left(\sigma_a[\![v]\!], \sigma_b[\![v]\!]\right)$$

This means that, in order to produce the assignment $\Psi_\sigma(\sigma_a, \sigma_b)$, we simply use ψ to combine the assignments for each variable $v \in V$.

Consider the bit-vectors presented in Figure 1. Given the values of v, v_a and v_b, we define the differences $\delta_a = v \oplus v_a$ and $\delta_b = v \oplus v_b$ computed by a bit-wise XOR. Those differences δ_a and δ_b indicate exactly which bits differ between the base value and each of the additional values. One can think of those differences as mutations that can be applied to the base value in order to produce a different value. For example, we can compute v_a as $v \oplus \delta_a$, where the XOR operator is used to apply mutation δ_a to v.

The insight that allows the generation of a large number of samples is that such mutations can be combined together. We define a combined mutation through the OR operator, producing $(\delta_a \vee \delta_b)$. This resulting mutation can be applied to the base value v, producing a new value $v \oplus (\delta_a \vee \delta_b)$. Thus, the combination method ψ is defined as

$$\psi_v(v_a, v_b) = v \oplus ((v \oplus v_a) \vee (v \oplus v_b))$$

This definition of ψ is generalized to all other types of variables in the formula ϕ. The motivation for this definition of $\Psi_\sigma(\sigma_a, \sigma_b)$ is that it attempts to obtain the mutation that generates σ_a from σ and the mutation that generates σ_b from σ and then combine those two mutations in an additive way. If σ_a and σ_b are neighboring solutions obtained from a MAX-SMT query, those mutations are atomic mutations, which represent a minimal set of bits that can be flipped and still preserve the satisfiability of the formula. Therefore, it is likely that there exist some clauses in the formula which establish a strong dependence between those bits. Since in the resulting sample we flip the bits which were flipped by either of the two atomic mutations, it is likely that such clauses would still be satisfied.

Our experiments demonstrate that this combination of mutations is effective at generating valid samples, even for large and complex SMT formulas. We also found that the coverage predicates tend to follow the same pattern. That is, when we compute the value of a coverage predicate ψ_i on the resulting sample, most of the time it will be the same value that would result from combining mutations of the predicate values, as shown in the last 3 columns of the Figure 1 example. This helps our algorithm generate solutions from new diverse coverage classes.

V. EVALUATION

We have implemented GUIDEDSAMPLER as an open-source tool[1] in C++, using Z3 [26] as the constraint solver. Z3 has native support for the MAX-SMT queries [27] that are required by GUIDEDSAMPLER. For evaluation, we have used 213 benchmarks from the QF_AUFBV logic of SMT-LIB [11], and its sublogics, such as QF_ABV and QF_BV. The benchmarks come from 22 different families, which will be presented in Table I. They are the same benchmarks which were used for the evaluation of SMTSAMPLER [14], and include a diverse set of complex, non-linear constraints. We used a time budget of 30 minutes for each sampling experiment.

A. Coverage Predicates

We performed two sets of experiments, varying the way coverage predicates are generated. The goal is to evaluate the technique both using a general notion of coverage that stems from the formula itself and also a problem-specific notion of coverage, which can be quite different from the original constraint.

The first set of experiments uses *internal predicates*, which are subparts of the formula itself. Considering the representation of the formula ϕ as an abstract syntax tree (AST), we look at the internal nodes of type bit-vector or Boolean. For example, in the formula $\phi = \phi_1 \vee \phi_2$, two of the internal nodes are the disjuncts ϕ_1 and ϕ_2. We sample solutions to ϕ using a generic sampler, SMTSAMPLER, and collect coverage statistics on those Boolean nodes and individual bits inside the bit-vector nodes. We identify a subset of those internal nodes that have exhibited diverse and independent coverage behaviors and choose the subformulas represented by those nodes as the coverage predicates. For example, in the formula $\phi = \phi_1 \vee \phi_2$, we could pick ϕ_1 and ϕ_2 to be two of the coverage predicates, and choose additional predicates as subparts of ϕ_1 and ϕ_2. This is intended to represent a generic coverage notion

[1] Source code at https://github.com/RafaelTupynamba/GuidedSampler.

based on the formula itself. The number of predicates collected varies from only a couple on some benchmarks up to hundreds of predicates on others.

The second set of experiments uses *random predicates*, which are randomly generated constraints involving the variables in $Vars[\phi]$. The coverage predicates are randomly sampled from a context-free grammar that includes all arithmetical, logical and bitwise operations from SMT-LIB, as well as the variables from $Vars[\phi]$. We generate from 16 to 48 predicates for each benchmark, with each predicate having on average a depth of 4 operations (i.e., any path from the root to a leaf in the formula AST has 4 operations on average). The random predicates are intended to represent a problem-specific measure of coverage, as they are independent from the original formula.

B. Approaches

In our evaluation, we compare the following six approaches for coverage-guided sampling.

a) BH*: Baseline with hard constraints:* This simple baseline approach consists of choosing n random Boolean values $b = (b_1, b_2, \ldots, b_n)$ that define a coverage class and trying to find one solution from that class. We then use the SMT solver to generate one solution to $\phi \wedge (\psi_1 = b_1) \wedge (\psi_2 = b_2) \wedge \cdots \wedge (\psi_n = b_n)$, where the predicates are added as hard constraints. Then repeat the process for different random vectors b. No solution is generated for a given b if class G_b is empty.

b) BS*: Baseline with soft constraints:* This baseline approach consists of choosing n random Boolean values $b = (b_1, b_2, \ldots, b_n)$ that define a coverage class and trying to find the solution closest to that class. We use the query MAX-SMT($\{\phi\}, S_b$), where the predicates are added as soft constraints $S_b = \{\psi_1 = b_1, \psi_2 = b_2, \ldots, \psi_n = b_n\}$. Then repeat the process for different random vectors b. Each MAX-SMT call is guaranteed to eventually find a solution if ϕ is satisfiable, but the optimization problems can be expensive.

c) S0*:* SMTSAMPLER*:* This baseline approach consists of applying SMTSAMPLER to sample solutions to formula ϕ. The coverage predicates are not used.

d) S1 = S0 + M1*:* This approach includes our modification M1, which consists of flipping the values of coverage predicates to generate neighboring solutions in function COMPUTENEIGHBORSOLS. This ensures neighboring solutions come from neighboring coverage classes.

e) S2 = S0 + M1 + M2*:* This approach includes modifications M1 and also M2, which consists of discarding solutions from repeated coverage classes in lines 24 and 36 in Algorithm 1. A coverage class is considered repeated if we have already generated a solution from that class in the current epoch of the algorithm.

f) S3 = S0 + M1 + M2 + M3*:* This is the GUIDEDSAMPLER approach, including modifications M1, M2 and also M3, which randomizes the coverage class of the initial base solution in lines 3-7 of Algorithm 1.

C. Evaluating Diversity of Classes

We first evaluate the number of coverage classes reached by our sampling approaches. For most benchmarks, the total number of non-empty coverage classes is large, so a good coverage-guided sampler must find solutions from a large number of classes during the fixed time budget of 30 minutes and not keep visiting the same few classes multiple times.

Table I shows average statistics about the benchmarks and experimental results using random predicates. For each benchmark family, the column n lists the number of benchmarks used from that family. All the following columns present average numbers among all benchmarks in that family. First, we list the number of internal nodes in the SMT formula and the number of variables of type array, bit-vector and Boolean. The 'bits' column presents the total number of bits in the bit-vector and Boolean variables in the formula, and the 'preds' column represents the number of coverage predicates in the experiments that used internal predicates.

Next, for each of the six approaches, we list the total number of unique coverage classes found in the time budget. We only show the results for experiments with random predicates, but the results with internal predicates were similar. From Table I, we can see that the GUIDEDSAMPLER approach S3 was able to find the largest number of classes for most benchmarks. S2 tends to be the second best approach, with S1 being the third best one. This shows the effectiveness of our modifications M1, M2 and M3 in helping find more coverage classes. On only two benchmark families S0 performed better. We found that this happened because the MAX-SMT queries were too complex for those benchmarks and were frequently timing out.

The following are average statistics for our approach S3 with internal predicates (random predicates in parenthesis). The percentage of time spent in Z3 API calls was 82% (88%). The percentage of candidate solutions that turned out to be real solutions to the formula was 76% (75%). The percentage of solutions which were unique (distinct solutions) was 65% (94%). The percentage of unique solutions which were from unique coverage classes (distinct classes) was 46% (54%).

Figure 2 shows a more thorough comparison of the number of coverage classes found across all benchmarks. We define N_X as the number of unique coverage classes for which a solution was found using approach X during our time budget. The graphs in Figure 2 show the ratio of classes found $N_{\mathrm{S3}}/N_{\mathrm{X}}$ between approach S3 and each of the baseline approaches X, in a logarithmic scale. Whenever no solutions were generated by one approach, the logarithm would evaluate the $+\infty$ or $-\infty$. This is represented on the graphs by bars that reach the top or the bottom of the graph.

Figure 2 includes results using both internal predicates and random predicates. From the graphs, we can see that the GUIDEDSAMPLER approach S3 is vastly superior to the baseline approaches on most benchmarks. Baseline BH frequently tries to find solutions from empty classes, leading to unsatisfiable queries to the SMT solver. Baseline BS makes MAX-SMT queries which are satisfiable, but still expensive

TABLE I
AVERAGE RESULTS OVER THE BENCHMARKS USING RANDOM PREDICATES

Benchmarks	Benchmark statistics							Unique coverage classes					
	n	nodes	$\|Array\|$	$\|BV\|$	$\|Bool\|$	bits	preds	BH	BS	S0	S1	S2	S3
QF_AUFBV/ecc	4	179	0	27	7	1931	12	188	854	6579	826	969	11377
QF_ABV/bmc-arrays	3	855	1	1	0	53	82	44	660	1337	19003	19374	19122
QF_ABV/stp_samples	15	1139	1	24	0	192	58	9831	15082	11360	97511	679380	744289
QF_ABV/dwp_formulas	5	613	3	32	0	428	38	7028	5688	404	2641	4227	31735
QF_ABV/egt	15	90	1	0	0	0	5	8656	30984	16432	15561	49864	480017
QF_ABV/bench_ab	15	314	1	0	0	2	3	151	3406	3118	2765	2530	6000
QF_ABV/platania/...member	12	595	10	89	0	2856	2	15	0	399	83	84	235161
QF_BV/bmc-bv	10	256	0	4	0	134	6	19	122	437	201	200	208
QF_BV/bmc-bv-svcomp14	8	7518	0	205	1055	7607	223	178	2866	2004	16363	19503	23861
QF_BV/spear/zebra_v0.95a	9	571	0	185	0	2012	21	50	2797	618	8135	9622	7384
QF_BV/RWS	9	1086	0	21	0	3628	95	0	0	187	84	80	53
QF_BV/gulwani-pldi08	5	1146	0	130	0	950	185	130	2524	9605	499158	566758	622999
QF_BV/stp_samples	14	793	0	22	0	200	10	18015	81052	5387	431305	908703	1068203
QF_BV/brummayerbiere2	3	632	0	2	0	149	27	0	5	7	9	9	10
QF_BV/tacas07	3	8812	0	345	588	16620	91	0	0	53	51	58	78
QF_BV/bench_ab	13	21	0	1	0	24	1	2	134	99	153	149	152
QF_BV/sage/app2	13	231	0	24	0	206	7	19501	20581	1583	769464	1004453	1035877
QF_BV/sage/app9	8	271	0	35	0	391	17	1940	8829	13025	180458	257994	318534
QF_BV/sage/app8	15	978	0	93	0	1047	31	1312	20627	14635	209286	269363	328543
QF_BV/sage/app5	12	268	0	29	0	354	3	1577	23736	6428	300547	437180	628252
QF_BV/sage/app1	10	115	0	20	0	268	5	346	5731	3458	19765	20768	21473
QF_BV/sage/app12	12	247	0	31	0	358	5	4371	9065	11766	330629	608030	683599

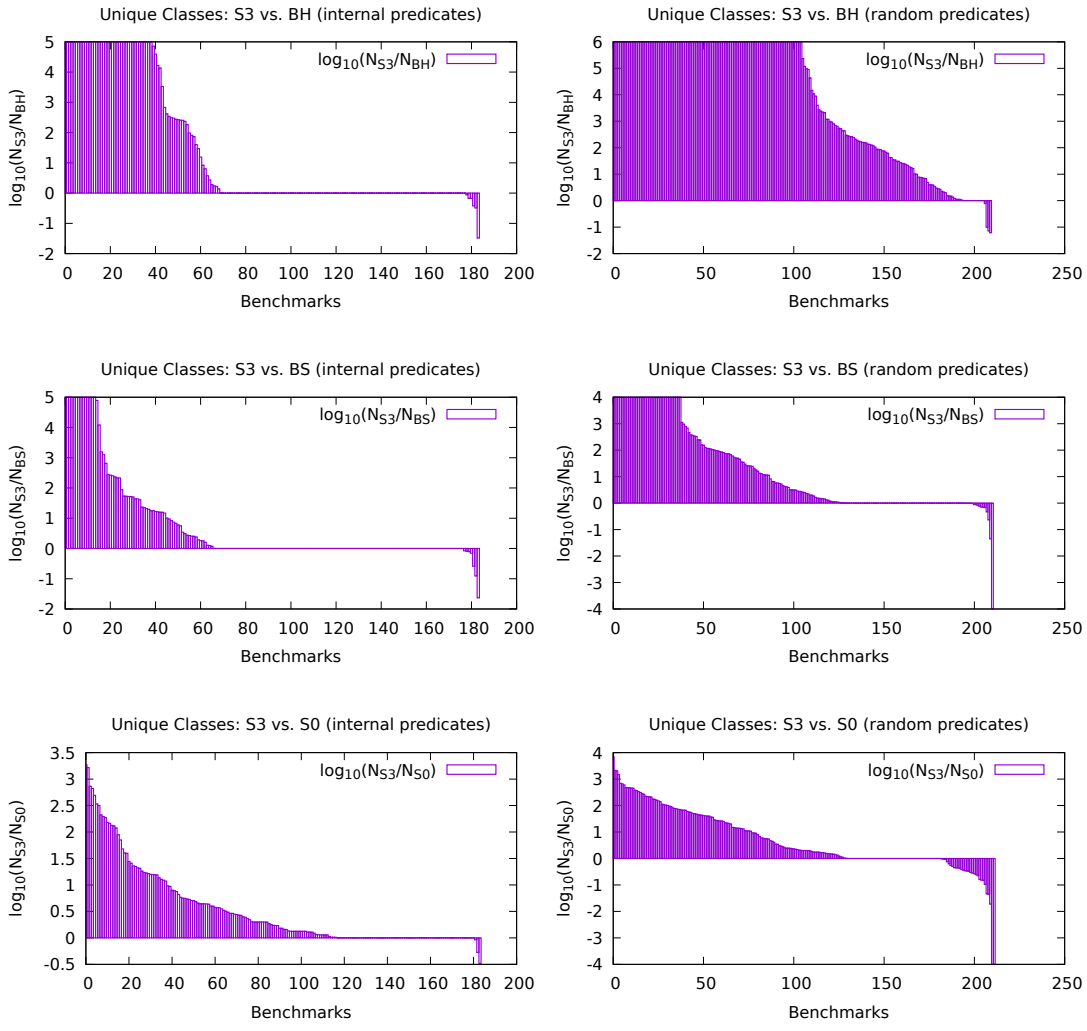

Fig. 2. Unique classes covered: GUIDEDSAMPLER vs. baselines.

TABLE II
MEAN VALUES ACROSS ALL BENCHMARKS

Expression	Internal Predicates	Random Predicates
N_{S3}/N_{BH}	2.6 $(\times 10^{\pm 1.0})$	38 $(\times 10^{\pm 1.4})$
N_{S3}/N_{BS}	2.5 $(\times 10^{\pm 0.9})$	4.1 $(\times 10^{\pm 0.9})$
N_{S3}/N_{S0}	3.4 $(\times 10^{\pm 0.8})$	5.4 $(\times 10^{\pm 1.0})$
N_{S3}/N_{S1}	1.1 $(\times 10^{\pm 0.3})$	1.7 $(\times 10^{\pm 0.5})$
N_{S3}/N_{S2}	1.08 $(\times 10^{\pm 0.2})$	1.3 $(\times 10^{\pm 0.4})$
H_{BH}	1.3 $(\times 10^{\pm 0.1})$	1.2 $(\times 10^{\pm 0.2})$
H_{BS}	1.3 $(\times 10^{\pm 0.2})$	2.2 $(\times 10^{\pm 0.3})$
H_{S0}	14 $(\times 10^{\pm 1.0})$	19 $(\times 10^{\pm 0.8})$
H_{S1}	2.7 $(\times 10^{\pm 0.6})$	8.5 $(\times 10^{\pm 0.7})$
H_{S2}	1.4 $(\times 10^{\pm 0.2})$	3.1 $(\times 10^{\pm 0.3})$
H_{S3}	1.3 $(\times 10^{\pm 0.2})$	2.9 $(\times 10^{\pm 0.3})$

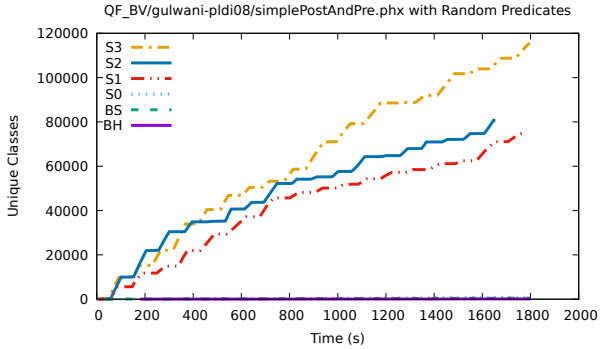

Fig. 3. Unique classes over time for one benchmark

to solve. BS does not scale well because it still requires one MAX-SMT query for each new solution generated. The SMTSAMPLER approach S0 mitigates this cost by using a small number of MAX-SMT queries to learn atomic mutations and then quickly combines those mutations to generate a large number of solutions. However, S0 ignores the coverage predicates, so it frequently keeps generating solutions from the same few classes.

Table II shows mean values for the ratios N_{S3}/N_X, in the format 10^{μ} $(\times 10^{\pm\sigma})$, where μ and σ are the average and standard deviation of $\log_{10}(N_{S3}/N_X)$ across all benchmarks. The numbers show that S1, S2 and S3 can find a much larger number of classes when compared to S0. This demonstrates that M1 is the most important of our modifications. M1 ensures that the atomic mutations flip only a small number of the coverage predicates, so those mutations can be combined and generate solutions from more diverse classes. The modifications M2 and M3 also contribute to a small increase in the number of visited classes.

In Figure 3, we can see an example of the number of unique classes visited over time, for one representative benchmark. The GUIDEDSAMPLER approach S3 is the quickest to discover new classes, followed closely by approaches S2 and S1. We can also see that in this case the baseline approaches S0, BS and BH can find a much smaller number of classes.

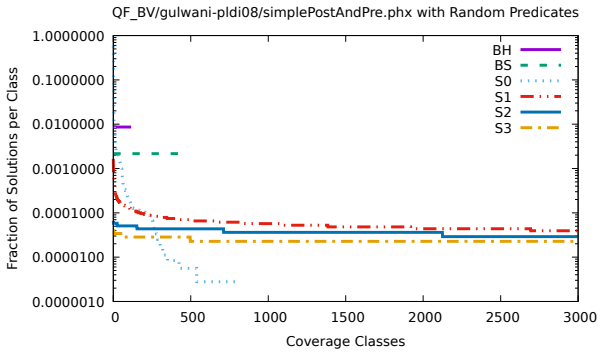

Fig. 4. Uniformity of solutions over coverage classes for one benchmark

D. Evaluating Uniformity Over Coverage Classes

In addition to looking at the number of classes covered, we also analyze the distribution of solutions among those classes. This is important, since a good coverage-guided sampler needs to sample from all the coverage classes with equal weight. Figure 4 shows the fraction of solutions generated from each coverage class for the same benchmark of Figure 3. S0 is the most biased sampler, with the most frequent class being covered by 58% of the solutions. BH and BS are very uniform, but could only reach less than 500 classes. Our approach S3 is very close to uniform, with its line at the bottom of the graph, extending to the right until 115196 classes (not shown).

If S solutions are generated, covering N distinct classes, we expect each of those classes to be covered by approximately S/N solutions. So as a first order estimate on how biased the distribution is, we look at the number M of solutions found in the most frequent class. We define $H = M/(S/N)$ as a measure of how far the frequency M is from the expected frequency S/N. The closer H is to 1, the more uniform is the distribution of solutions found among those classes.

Table II displays mean values of H across all benchmarks. As expected, the baseline approaches BH and BS are very uniform, since each new solution is generated inside or near a random coverage class. S0 is the most biased approach, often sampling a large number of solutions from a single class. From the table, we see that our modifications M1 and M2 are essential for producing a more uniform distribution over the coverage classes, almost as uniform as the one from BH and BS, with $H_{S3} \approx 1.3$ for internal predicates and $H_{S3} \approx 2.9$ for random predicates.

VI. CONCLUSION

In this paper, we presented the problem of *coverage-guided sampling*, to allow shaping the distribution of SMT solutions via user-specified coverage predicates. We described our technique GUIDEDSAMPLER for this problem, and showed that it is able to reach 2.5× to 38× more coverage classes than the baseline approaches, while having a distribution over coverage classes that is close to uniform. The approach works well for both internal coverage predicates based on the formula itself and also for random coverage predicates.

ACKNOWLEDGMENT

Research partially funded by Brazilian Science Without Borders CAPES 13245/13-9; NSF grants CCF-1409872 and CNS-1817122; DARPA CRAFT HR0011-16-C-0052; Intel Science and Technology Center for Agile Design; the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849; and ADEPT Lab industrial sponsors and affiliates Intel, Apple, Futurewei, Google, and Seagate. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors.

REFERENCES

[1] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI magazine*, vol. 28, no. 3, p. 13, 2007.

[2] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient Sampling of SAT Solutions for Testing," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 549–559.

[3] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," *ACM Sigplan Notices*, vol. 41, no. 11, pp. 404–415, 2006.

[4] Y. Zhao, J. Bian, S. Deng, and Z. Kong, "Random stimulus generation with self-tuning," in *Computer Supported Cooperative Work in Design, 2009. CSCWD 2009. 13th International Conference on*. IEEE, 2009, pp. 62–65.

[5] R. Naveh and A. Metodi, "Beyond feasibility: Cp usage in constrained-random functional hardware verification," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, pp. 823–831.

[6] O. Padon, K. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: interactive verification of parameterized systems via effectively propositional reasoning," *PLDI. ACM*, 2016.

[7] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *PLDI'05*, June 2005.

[8] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/FSE'05*, Sep 2005.

[9] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08*, Dec 2008.

[10] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[11] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[12] S. Ermon, C. P. Gomes, and B. Selman, "Uniform solution sampling using a constraint solver as an oracle," *Conference on Uncertainty in Artificial Intelligence*, 2012.

[13] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform sat witness generation." in *TACAS*, 2015, pp. 304–319.

[14] R. Dutra, J. Bachrach, and K. Sen, "SMTSampler: Efficient Stimulus Generation from Complex SMT Constraints," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018.

[15] K. S. Meel, "Sampling techniques for boolean satisfiability," *Master's thesis*, 2014.

[16] W. Wei, J. Erenrich, and B. Selman, "Towards efficient sampling: Exploiting random walk strategies," in *AAAI*, vol. 4, 2004, pp. 670–676.

[17] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*. IEEE, 2007, pp. 258–265.

[18] N. B. Kitchen, *Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation*. University of California, Berkeley, 2010.

[19] A. Nadel, "Generating diverse solutions in sat." in *SAT*. Springer, 2011, pp. 287–301.

[20] S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman, "Embed and project: Discrete sampling with universal hashing," in *Advances in Neural Information Processing Systems*, 2013, pp. 2085–2093.

[21] K. S. Meel, M. Y. Vardi, S. Chakraborty, D. J. Fremont, S. A. Seshia, D. Fried, A. Ivrii, and S. Malik, "Constrained sampling and counting: Universal hashing meets sat solving." in *AAAI Workshop: Beyond NP*, 2016.

[22] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Balancing scalability and uniformity in sat witness generator," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 2014, pp. 1–6.

[23] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "Distribution-aware sampling and weighted model counting for SAT," in *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 7 2014.

[24] R. Gupta, S. Sharma, S. Roy, and K. S. Meel, "Waps: Weighted and projected sampling," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 4 2019.

[25] R. Nieuwenhuis and A. Oliveras, "On SAT modulo theories and optimization problems," in *International conference on theory and applications of satisfiability testing*. Springer, 2006, pp. 156–169.

[26] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[27] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νz-an optimizing smt solver." in *TACAS*, vol. 15, 2015, pp. 194–199.