SMTSampler: Efficient Stimulus Generation from Complex SMT Constraints

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

Stimulus generation is an essential part of hardware verification, being at the core of widely applied constrained-random verification techniques. However, as verification problems get more and more complex, so do the constraints which must be satisfied. In this context, it is a challenge to efficiently generate random stimuli which can achieve a good coverage of the design space. We developed a new technique SMTSampler which can sample random solutions from Satisfiability Modulo Theories (SMT) formulas with bit-vectors, arrays, and uninterpreted functions. The technique uses a small number of calls to a constraint solver in order to generate up to millions of stimuli. Our evaluation on a large set of complex industrial SMT benchmarks shows that SMTSampler can handle a larger class of SMT problems, outperforming state-of-the-art constraint sampling techniques in the number of samples produced and the coverage of the constraint space.

CCS CONCEPTS

• **Hardware → Test-pattern generation and fault simulation**; *Theorem proving and SAT solving*; Semi-formal verification;

KEYWORDS

constrained-random verification, stimulus generation, sampling, SMT, arrays, bit-vectors, uninterpreted functions

ACM Reference Format:
Rafael Dutra, Jonathan Bachrach and Koushik Sen. 2018. SMTSampler: Efficient Stimulus Generation from Complex SMT Constraints. In *IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD '18), November 5–8, 2018, San Diego, CA, USA.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3240765.3240848

1 INTRODUCTION

Constrained-random verification (CRV) [12] is one of most widely used verification techniques in industry. At the core of CRV, a stimulus generator is responsible for generating multiple inputs that satisfy some user-specified constraints. Those inputs are then used to drive the design under test, in an attempt to cover the design space and trigger faults.

The constraints used in CRV can be manually specified by the verification engineer, taking into account preconditions required by the hardware and other domain-specific knowledge [11, 15]. However, the constraints are increasingly being synthesized by automated formal methods. Such methods can generate constraints from a high-level specification of the hardware interfaces [13]. Such constraints can be large and complex, involving higher-order theories, such as arrays and bit-vectors.

These constraints obtained from formal specification of hardware interfaces can be specified in the framework of Satisfiability Modulo Theories (SMT), using high-level theories such as bit-vectors, arrays and uninterpreted functions. The problem of finding one solution to SMT constraints is well studied, with off-the-shelf constraint solvers available [5]. There is also a standardized library SMT-LIB with multiple SMT benchmarks for different solvers to use [1]. However, the problem of generating multiple diverse solutions from one SMT constraint is much less studied in literature.

One big challenge to generating random stimuli from such constraints is that they can be quite complex, involving linear and non-linear arithmetic over a large number of bit-vectors, arrays and uninterpreted functions. When solutions are sparse and non-linearly distributed, traditional techniques such as MCMC samplers do not perform well, while techniques that use constraint solvers to obtain each solution become too expensive.

Another challenge is making sure the solutions are diverse and cover a large portion of the solution space. A good stimulus generator should avoid generating solutions which are only trivially different, because those are less likely to trigger new behaviors in the circuit. For example, if we have a constraint of the form $x > 5 \vee \phi$, where x is a 32-bit integer and ϕ is a complex SMT formula possibly involving x and other variables, there are billions of possible values for x which satisfy this constraint by simply satisfying the sub formula $x > 5$. However, producing billions of solutions which only differ in the value of x while ignoring other variables will likely not lead to new coverage and faults.

We developed a technique SMTSampler which can efficiently sample millions of solutions from a SMT formula. SMTSampler works by computing simple atomic mutations that can be applied to a satisfying assignment while preserving the satisfiability of the formula. Those mutations represent minimal sets of bits that can be flipped from the SMT variables of the formula to transform one solution into another solution to the formula. The key insight is that several such mutations can be merged together to produce valid solutions with high probability. We collect as many atomic mutations as possible and then adaptively combine subsets of those mutations together, while avoiding invalid samples and enabling the generation of a large number of valid solutions to the formula. SMTSampler works for SMT formulas including theories of bit-vectors, arrays and uninterpreted functions. We define atomic mutations for variables of each of those types, along with operations to combine them. Our evaluation shows that SMTSampler

---

ICCAD '18, November 5–8, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240848

can typically generate millions of solutions, using only hundreds of calls to the constraint solver.

In order to evaluate the coverage of the constraint space, we define a metric for the internal coverage of a SMT formula. The metric is defined by regarding the formula as a circuit, so that it can serve as a proxy for the coverage that would be obtained in the design under test.

Our main contributions are:

- Develop a technique SMTSampler and implement it in an open source tool for efficient sampling from SMT formulas.
- Evaluate SMTSampler against existing techniques on a large set of complex benchmarks from SMT-LIB.
- Define a metric for internal coverage of SMT formulas and use it in evaluating different sampling algorithms.

The paper is organized as follows. Section 2 presents the existing work in hardware stimulus generation and sampling from logical constraints. Section 3 defines the constraint format and Section 4 describes how our technique SMTSampler produces samples from those constraints. Finally, Section 5 evaluates it in terms of samples generated and coverage, and Section 6 concludes the results.

2 RELATED WORK

There is a large body of work in sampling solutions to Boolean satisfiability (SAT) formulas [9]. In principle, methods to sample solutions to SAT formulas can also be applied to SMT, as there are techniques for eager encoding of SMT formulas into SAT. However, one limitation of this conversion is the loss of the higher-level structure of the formula, which could be leveraged to generate samples more efficiently and also to ensure the samples are diverse. In SMTSampler, we have found that working at the SMT level without converting the formula into SAT leads to a larger number and diversity of samples.

One important class of sampling techniques is based on Markov Chain Monte Carlo (MCMC) methods [7, 8]. They generate samples from a probability space by applying some form of random walk through the solution space, using techniques such as simulated annealing and Metropolis-Hastings. With MCMC, it is possible to guarantee that the distribution of samples will eventually converge to the desired distribution (such as the uniform one). However, in practice, this convergence is too slow for real-world problems, and heuristics are also applied which make the sampling more biased [8, 14]. One common approach is combining Metropolis moves with a random walk through the assignments of the formula [14]. MCMC is the basis of most constrained-random verification techniques [7, 8, 12, 15]. MCMC techniques are typically effective for linear constraints, where the space of solutions is composed of polytopes which can be efficiently covered with random walks [8]. However, they are not so effective on arbitrary non-linear constraints, that lead to a more sparse distribution of solutions. SMTSampler, on the other hand, is designed to be applied even to arbitrary, complex non-linear constraints.

A different strategy for sampling is using a constraint solver to produce each sample. The internal search heuristics of the solver can be modified to generate more diverse samples [10]. An important limitation of this approach is that it requires one constraint solver call per each sample produced, which is expensive. SMTSampler, on the other hand, generates several samples per solver call.

On the more theoretical side, there are techniques based on universal hashing which can sample solutions from SAT formulas with a provably uniform distribution, such as UniGen [4] and UniGen2 [2]. However, these techniques are expensive, as they require solving constraints which include complex hash functions that are hard to solve. In addition, the goal of sampling uniformly from the solution space does not necessarily lead to the best coverage of the constraint space. We designed SMTSampler with the goals of quickly generating samples and achieving the best possible coverage of the design space.

Following the universal hashing approach, SMTApproxMC [3] is an approximate model counter for SMT formulas. It is applicable only to formulas in the bit-vector theory and works similarly to UniGen [4] and UniGen2 [2], but using different hash functions that work at the word level. Although SMTApproxMC is a model counter, it can be adapted to work as a random sampler of bit-vector solutions, by outputting the solutions in a given cell, after the solution space is uniformly partitioned into cells. In contrast to SMTApproxMC, SMTSampler is designed to be more efficient and to also work with formulas containing the theories of arrays, uninterpreted functions and bit-vectors.

A recent technique developed to sample solutions to SAT formulas is QuickSampler [6]. It works by computing some simple patterns of bit-flips, called *atomic mutations*, which can be applied to a valid solution to generate another valid solution to the formula. QuickSampler produces samples by combining k such atomic mutations together, for each $k \leq 6$. Those samples are not guaranteed to be solutions for the formula, but they were solutions with high probability on hundreds of SAT benchmarks. Our technique SMTSampler also uses the same idea of computing atomic mutations and combining them to generate samples. However, SMTSampler is adapted to work over the higher-level theories of bit-vectors, arrays, and uninterpreted functions, producing and combining mutations over those data types directly. This leads to more efficient solving, while also eliminating the cost to convert the SMT formula into SAT. We show in our experimental evaluation section that on most benchmark programs SMTSampler outperforms a naïve approach that converts a SMT formula to SAT and then applies QuickSampler. Moreover, unlike QuickSampler, SMTSampler only outputs valid samples and adaptively increases the number k of atomic mutations combined based on the accuracy in the samples that are tried.

3 CONSTRAINT SPECIFICATION

SMTSampler works over any 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. We define the set of variables in the formula as $V = Bool \cup BV \cup Array \cup UF$, where $Bool$, BV, $Array$, and UF are the sets of variables of type Boolean, bit-vector, array, and uninterpreted function, respectively.

A SAT formula is a logical formula constructed from only Boolean variables and operators from combinatorial logic, such as $\wedge, \vee, \neg$. A SMT formula, on the other hand, is a logical formula with terms (variables, constant symbols and function symbols) originated not only from the SAT logic, but also from different theories.

Table 1: Types of variables allowed

Type	Example Value
$\boldsymbol{b} \in Bool$	$b = False$
$\boldsymbol{v} \in BV$	$v = 01100111$
$\boldsymbol{a} \in Array$	$a[x] = \begin{cases} \text{if } x = 001: & 0110 \\ \text{if } x = 011: & 1001 \\ \text{if } x = 101: & 0101 \\ \text{otherwise}: & 0010 \end{cases}$
$\boldsymbol{f} \in UF$	$f(x, y) = \begin{cases} \text{if } x = 0 \land y = 10: & 10 \\ \text{if } x = 1 \land y = 00: & 01 \\ \text{otherwise}: & 11 \end{cases}$

For example, the formula $select(\boldsymbol{a}, 011) + \boldsymbol{v} > 0110$ contains an array variable $\boldsymbol{a}$ and a bit-vector variable $\boldsymbol{v}$, two bit-vector constants 011 and 0110, an array function *select*, a bit-vector function +, and a bit-vector predicate >.

One of the theories that is commonly used in an SMT formula is the theory of fixed-size bit-vectors. Here, we denote by $BV[n]$ the sort of bit-vectors of size n. The theory of bit-vectors includes the customary arithmetical and logical operations on bit-vectors, such as additions, comparisons and bit-wise operations. Another theory common in SMT formulas is the theory of arrays, which consists of two functions *select* and *store* that satisfy the usual axiom

$$select(store(\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{y}), \boldsymbol{x'}) = \begin{cases} \boldsymbol{y}, & \text{if } \boldsymbol{x'} = \boldsymbol{x} \\ select(\boldsymbol{a}, \boldsymbol{x'}), & \text{otherwise.} \end{cases}$$

Here, $\boldsymbol{x}, \boldsymbol{x'} \in BV[s_x]$ are bit-vectors of a certain size s_x and $\boldsymbol{y} \in BV[s_y]$ is a bit-vector with a possibly different size s_y. $\boldsymbol{a}$ is an array of domain $BV[s_x]$ and range $BV[s_y]$. The function *select* returns the value at a given index from the array, while *store* produces an array with a new value assigned to the given index. The theory of uninterpreted functions is a free theory, so it does not add any new axioms. Nothing is known *a priori* about the result of applying such a function to its arguments.

Table 1 shows example values for variables of each type. Bold names $\boldsymbol{b}, \boldsymbol{v}, \boldsymbol{a}, \boldsymbol{f}$ are used for variable names, while b, v, a, f represent concrete instances that can be assigned to those variables in a given solution. Let S be the set of all possible assignments to the variables in V. Given an assignment $\sigma \in S$ and a variable $\boldsymbol{v} \in V$ we denote by $\sigma[\![\boldsymbol{v}]\!]$ the concrete assignment to $\boldsymbol{v}$ under σ.

Variables in BV are fixed-size bit-vectors, such as the variable $\boldsymbol{v} \in BV[8]$. Arrays must have bit-vector domains and ranges, such as the array $\boldsymbol{a}$, with domain $BV[3]$ and range $BV[4]$. Uninterpreted functions can have any arity. The example shows the function $\boldsymbol{f} : BV[1] \times BV[2] \to BV[2]$, of arity 2. A concrete instance a for an array $\boldsymbol{a}$ is constructed by defining its value for a finite set of indices $I(a)$ and defining a default value $d(a)$ for all other indices. In the example shown, $I(a) = \{001, 011, 101\}$ and $d(a) = 0010$. Typically, only a small number of indices will be relevant when solving a constraint, even for array domains such as $BV[64]$, which allows 2^{64} possible indices. An analogous construction is used for uninterpreted functions, where its value is defined for a finite set of argument tuples.

4 SMTSAMPLER ALGORITHM

SMTSampler 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. 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. It generates a compound mutation and applies it to the original solution, producing a possibly new solution. The combination function 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, SMTSampler checks each generated sample for validity and only outputs valid solutions.

Next we present the full details of the algorithm. In §4.1 we describe the main sampling procedure of SMTSampler. Next, we explain in §4.2 how one base solution is chosen for each epoch, and in §4.3 how we discover a set of neighboring solutions to the base solution. Finally, in §4.4, we describe how those solutions are used to generate new samples.

4.1 Main SMTSampler Algorithm

Algorithm 1 presents the main SMTSampler procedure, which takes as input a SMT formula ϕ. The SMTSampler algorithm works over several epochs. In each epoch, we first sample one initial solution σ to the formula, which we call a *base solution*. This is done by generating a random assignment σ' to the variables of the formula in line 3 and then calling findClosestSolution to obtain the solution σ which is closest to σ' in line 4. The details of this procedure are given in §4.2. Then, in line 6, we use the function computeNeighboringSolutions to compute a set Σ^1_σ of *neighboring solutions* for σ. This function will be described in §4.3.

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* $\Psi : S \times S \times S \to S$, where S is the space of all possible assignments to the variables V in the formula. 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 valid 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 §4.4.

We next describe how function combine uses Ψ to generate new samples. We denote by Σ^1_σ the set of neighboring solutions to σ obtained from computeNeighboringSolutions. 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

Algorithm 1 SMTSampler algorithm

```
function SMTSampler($\phi$)
    while not done do
        $\sigma' \leftarrow$ generateRandomAssignment($\phi$)
        $\sigma \leftarrow$ findClosestSolution($\phi$, $\sigma'$)
        output($\{\sigma\}$)
        $\Sigma^1_\sigma \leftarrow$ computeNeighboringSolutions($\phi$, $\sigma$)
        output($\Sigma^1_\sigma$)
        $\alpha \leftarrow 1, k \leftarrow 1, \Sigma_\sigma \leftarrow \Sigma^1_\sigma$
        while $\alpha \geq \alpha_{min} \land k < 6$ do
            $(\Sigma^{k+1}_\sigma, \alpha, \Sigma_\sigma) \leftarrow$ combine($\Sigma^k_\sigma, \Sigma^1_\sigma, \Sigma_\sigma, \phi$)
            output($\Sigma^{k+1}_\sigma$)
            $k \leftarrow k + 1$

function computeNeighboringSolutions($\phi$, $\sigma$)
    $C_\sigma \leftarrow$ getConditions($\phi$, $\sigma$)
    $\Sigma^1_\sigma \leftarrow \{\}$
    for $c$ in $C_\sigma$ do
        $\Sigma^1_\sigma \leftarrow \Sigma^1_\sigma \cup$ findNeighboringSolution($\phi$, $c$, $C_\sigma$)
    return $\Sigma^1_\sigma$

function combine($\Sigma^k_\sigma, \Sigma^1_\sigma, \Sigma_\sigma, \phi$)
    $valid \leftarrow 0, checks \leftarrow 0$
    for $(\sigma_a, \sigma_b)$ in $\Sigma^k_\sigma \times \Sigma^1_\sigma$ do
        $\tilde{\sigma} \leftarrow \Psi_\sigma(\sigma_a, \sigma_b)$
        if $\tilde{\sigma} \notin \Sigma_\sigma$ then
            $\Sigma_\sigma \leftarrow \Sigma_\sigma \cup \{\tilde{\sigma}\}$
            $checks \leftarrow checks + 1$
            if isValidSolution($\tilde{\sigma}$, $\phi$) then
                $\Sigma^{k+1}_\sigma \leftarrow \Sigma^{k+1}_\sigma \cup \{\tilde{\sigma}\}$
                $valid \leftarrow valid + 1$
    return $(\Sigma^{k+1}_\sigma, valid/checks, \Sigma_\sigma)$
```

the current epoch, we maintain a set Σ_σ of samples which were computed so far, both valid and invalid. Initially, $\Sigma_\sigma = \Sigma^1_\sigma$.

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 valid solution to the formula. This checking is relatively fast, as it only needs to evaluate the formula using the assignments in $\tilde{\sigma}$. If $\tilde{\sigma}$ is a valid solution, it is then added to Σ^{k+1}_σ.

During the construction of Σ^{k+1}_σ from Σ^k_σ, we keep statistics on which fraction α of the checked samples were valid. If this fraction is below a certain threshold α_{min}, such as 0.1, we do not generate Σ^{k+2}_σ and instead just proceed to the next epoch. This adaptive generation of samples allows us to avoid trying out too many invalid samples.

All the samples which are ultimately output by SMTSampler are the ones in $\cup_{0 \leq k \leq 6} \Sigma^k_\sigma$, where we define $\Sigma^0_\sigma = \{\sigma\}$. Those are all valid 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. We have found that this adaptive generation of samples is essential in some SMT formulas to avoid the generation of large number of invalid samples. We always use valid solutions as arguments to the combination function, which enables it to generate valid solutions with high probability.

Table 2: Example conditions to be flipped

Type	Example Condition
$\boldsymbol{b} \in Bool$	$\boldsymbol{b} = False$
$\boldsymbol{v} \in BV$	$extract(\boldsymbol{v}, 5) = 1$
$\boldsymbol{a} \in Array$	$extract(\boldsymbol{a}[011], 3) = 1$
$\boldsymbol{f} \in UF$	$extract(\boldsymbol{f}(0, 10), 1) = 0$

4.2 Computing the Base Solution

Now, we describe how the initial base solution σ for the epoch is obtained. We first generate a random assignment σ' by choosing values to the Boolean and bit-vector variables in the formula uniformly at random. We do not assign values to the arrays and uninterpreted functions in σ', because we do not know initially which indices will be relevant for those variables. After generating σ', we choose σ as a solution which is as close as possible to σ'. This is done to explore as much of the solution space as possible, generating base solutions σ from different parts of the space.

The problem of finding a solution σ which is as close as possible to σ' can be encoded as a MAX-SMT optimization problem to be solved by the constraint solver. The MAX-SMT optimization problem is the problem of finding a solution to an SMT formula that must satisfy a set of hard constraints and should also satisfy the maximum possible number of soft constraints. We encode the MAX-SMT query as follows. We add one hard constraint stating the the formula ϕ must be satisfied. For each bit-vector variable $\boldsymbol{v}$, we add one soft constraint $\boldsymbol{v} = \sigma'[\![\boldsymbol{v}]\!]$ stating that the $\boldsymbol{v}$ should have the same value that it had in σ'. Analogously, we add one soft constraint $\boldsymbol{b} = \sigma'[\![\boldsymbol{b}]\!]$ for each Boolean variable $\boldsymbol{b}$.

4.3 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. The first step is collecting the set of conditions C_σ which are true for σ. Then, MAX-SMT queries are used to produce new neighboring solutions. Each MAX-SMT query attempts to flip one condition, while maintaining the remaining conditions valid, if possible. We specify a maximum time budget allowed for this phase, such as 20 minutes. If the time budget is enough to solve queries flipping each of the conditions in C_σ, then all those queries will be made. Otherwise, we select randomly and uniformly a maximum subset of the conditions to be flipped and solved in MAX-SMT queries within the time limit.

Constructing C_σ. Function getConditions produces C_σ by collecting conditions for each variable in the formula. Table 2 shows one example condition for each of the variables types. Those are conditions that are valid for the example values from Table 1. Here, *extract* is a function that takes a bit-vector $\boldsymbol{v}$ and an integer index i and returns the value of the bit at index i in $\boldsymbol{v}$.

The conditions are generated as follows. For each Boolean variable, we add one condition $\boldsymbol{b} = \sigma[\![\boldsymbol{b}]\!]$ asserting that the variable has the same value as in the base solution. For each bit-vector variable, we add one condition for each of its bits. The condition is of the form $extract(\boldsymbol{v}, i) = extract(\sigma[\![\boldsymbol{v}]\!], i)$, asserting that, when extracting the given bit from the bit-vector, we obtain the same value that would be obtained from the base solution.

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

Figure 1: Combining two mutations over $\boldsymbol{v} \in BV[8]$.

For each array $\boldsymbol{a}$, we look at each of the indices $I\left(\sigma[\![\boldsymbol{a}]\!]\right)$ assigned in the concrete instance of the array $\sigma[\![\boldsymbol{a}]\!]$. For each such index x, we consider the concrete bit-vector $\sigma[\![\boldsymbol{a}]\!][x]$ returned by the array on such index and we add one condition for each bit in this bit-vector, such as $\mathit{extract}\left(\boldsymbol{a}[x], i\right) = \mathit{extract}\left(\sigma[\![\boldsymbol{a}]\!][x], i\right)$. The procedure for uninterpreted functions is analogous. For each argument tuple that is assigned a value in the base solution, we recursively add conditions according to the value type.

Computing Σ^1_σ. After collecting the fine-grained conditions in C_σ, we want to compute neighboring solutions by picking one condition $c \in C_\sigma$ and using the constraint solver to find a solution to $\phi \wedge \neg c$, where ϕ is the original formula. 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 in $C_\sigma \setminus \{c\}$. Those requirements can be specified as a MAX-SMT optimization problem, by defining a set of hard constraints and soft constraints. We specify two hard constraints $\{\phi, \neg c\}$, stating that we want a valid solution that does not satisfy c. And we also specify as soft constraints the $|C_\sigma| - 1$ conditions in $C_\sigma \setminus \{c\}$, so that the maximum number of those conditions is preserved.

One challenge in solving such optimization problems is that they are expensive when the number of soft constraints is too large. For this reason, as an alternative, SMTSAMPLER also allows the strategy of specifying only one soft constraint per bit-vector variable, instead of one for each bit in a bit-vector. For example, one would specify one condition as $\boldsymbol{v} = 00100111$, instead of 8 different conditions such as $\mathit{extract}(\boldsymbol{v}, 0) = 0$ and $\mathit{extract}(\boldsymbol{v}, 1) = 0$. We evaluate this strategy in addition to our original strategy in Section 5. This alternative approach only changes the soft constraints that are added to the MAX-SMT query. For the hard constraint $\neg c$, we chose to always use conditions on the individual bits of each bit-vector, because we found that this is important to generate a larger number of atomic mutations and consequently a larger number of samples.

4.4 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 COMPUTENEIGHBORINGSOLUTIONS 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

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

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 $\boldsymbol{v} \in V$.

Next, we define how the combination method ψ is applied to each of the variable types. We first present the procedure for bit-vector variables and then generalize it to the other types. Let $\boldsymbol{v} \in BV$ be a bit-vector variable in the formula. We use the notations v, v_a, v_b to represent the values assigned to variable $\boldsymbol{v}$ in each of the solutions $\sigma, \sigma_a, \sigma_b$, i.e. we define $v = \sigma[\![\boldsymbol{v}]\!], v_a = \sigma_a[\![\boldsymbol{v}]\!], v_b = \sigma_b[\![\boldsymbol{v}]\!]$.

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. For bit-vectors, 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, for bit-vectors, ψ is defined as

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

Now we generalize the definition of ψ to other types of variables. For Boolean values, we use the same technique: $\psi_b(b_a, b_b) = b \oplus ((b \oplus b_a) \vee (b \oplus b_b))$. This way, Boolean values behave the same as bit-vectors of size 1.

Now we define how to apply the combination method ψ to a base array $a = \sigma[\![\boldsymbol{a}]\!]$ and two neighboring arrays $a_a = \sigma_a[\![\boldsymbol{a}]\!]$ and $a_b = \sigma_b[\![\boldsymbol{a}]\!]$. Remember that our array models only define explicit values for a finite set of indices. Assume that array a has explicitly defined values for indices in the set $I(a)$, and a default value $d(a)$ for all other indices. Arrays a_a and a_b are constructed analogously, with possibly different sets of assigned indices $I(a_a)$ and $I(a_b)$. We define the combination function for arrays as

$$\psi_a(a_a, a_b)[x] = \begin{cases} \psi_{a[x]}\left(a_a[x], a_b[x]\right), & \text{if } x \in I(a) \cup I(a_a) \cup I(a_b) \\ \psi_{d(a)}\left(d(a_a), d(a_b)\right), & \text{otherwise.} \end{cases}$$

This means that the assigned indices of the generated array will be $I = I(a) \cup I(a_a) \cup I(a_b)$, the union of the assigned indices of each of the three arrays. If $x \in I$, then x may or may not have a non-default value assigned for each of the three arrays, while if $x \notin I$, we know that x has a default value assigned for all the arrays. This definition keeps the generated array model $\psi_a(a_a, a_b)$ simple, with explicitly defined values only for the set of indices I. For uninterpreted functions, the combination function is defined analogously, with the set of assigned argument tuples being the union of the assigned tuples for the base solution and the two neighboring solutions. This completes the definition of ψ and, consequently, Ψ.

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

Table 3: Z3 tactics for conversion into SAT

Tactic	Description
`simplify`	Simplify the formula
`bvarray2uf`	Encode arrays as UFs
`ackermannize_bv`	Apply Ackermann's encoding to UFs
`bit-blast`	Bit-blast the bit-vector variables

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 solutions, not only for SAT formulas, but also for such complex SMT formulas.

We note that the combination functions are commutative and associative with respect to the atomic mutations applied. More explicitly, we have $\Psi_\sigma(\sigma_a, \sigma_b) = \Psi_\sigma(\sigma_b, \sigma_a)$ and $\Psi_\sigma\ (\Psi_\sigma(\sigma_a, \sigma_b), \sigma_c) = \Psi_\sigma\ (\sigma_a, \Psi_\sigma(\sigma_b, \sigma_c))$. In SMTSampler, the samples generated in Σ_σ^k can be seen as the combination of k atomic mutations.

5 EVALUATION

We have implemented SMTSampler as an open source tool[1] in C++. We used Z3 [5] as the constraint solver, which natively supports MAX-SMT queries. Each benchmark is run in one core, with a maximum virtual memory of 4GB and a time budget of 1 hour. We stop each execution after generating 1 million solutions or reaching the timeout of 1 hour, whichever occurs first. The experiments were run on a machine with a 64-core Intel Xeon CPU E5-4640 and 264GB of memory. For each epoch, we establish a maximum time budget of 20 minutes for the generation of all neighboring solutions. We also set a timeout of 5 seconds for each individual MAX-SMT solver call. If the call cannot be completed in 5 seconds, we remove the MAX-SMT soft constraints and retry the call with only the hard constraints (such as the condition to flip one bit in a bit-vector) for 5 more seconds. This allows us to still make progress and obtain some solutions in case the MAX-SMT problems are too expensive.

We compare two versions of SMTSampler. The first, abbreviated SMTbv, uses one soft constraint per bit-vector. The second, abbreviated SMTbit, uses soft constraints for each bit inside a bit-vector. As a baseline, we use a technique abbreviated SAT, which works by bit-blasting the SMT formula to convert it into a SAT formula and then sampling solutions from the SAT formula. This would be similar to applying the QuickSampler [6] to the problem, although still taking advantage of our adaptive combination of mutations. The goal is to evaluate the advantage of operating directly over high-level SMT formulas, as opposed to bit-blasting. We did not compare against techniques such as the MCMC-based approach from Ambigen [7] because those are only applicable over constraints which are linear on each variable and would not be able to handle the general SMT-LIB benchmarks. Moreover, experimental evaluation of QuickSampler showed that QuickSampler is 1000× faster than MCMC-based approaches on SAT problems. SMTSampler focuses on enabling the sampling of solutions from complex SMT formulas, which are generally non-linear.

For the baseline bit-blasting approach, the `expand_select_store` rewriter option is used to replace *select*(*store*(. . .), . . .) patterns by if-then-else terms. In addition, the Z3 tactics from Table 3 are applied to encode arrays as uninterpreted functions, apply Ackermann's encoding to those functions, and bit-blast bit-vectors. In our experiments, we chose not to encode the SAT problem into conjunctive normal form (CNF) because we found that this conversion lead to slower solving due to the introduced auxiliary variables. Our conversion approach enables the conversion of most benchmarks into SAT, as long as they do not use the theory of arrays with extensionality, including equality comparisons between arrays.

5.1 Coverage Metric

When sampling from SMT formulas, we noticed that the number of unique solutions generated is an incomplete metric for coverage. Sometimes, it is easy to sample a large number of solutions which are only trivially different and thus not interesting inputs for verification. For example, if a bit-vector variable x of size 32 in a formula is only constrained by a condition such as $x > 5$, there are billions of values for x that would satisfy this constraint. However, enumerating all those possibilities would probably not generate interesting inputs and a better strategy would be mutating other variables in the formula.

To better evaluate the coverage of the constraint space, we propose the use of a different coverage metric. We notice that the SMT formula has an abstract syntax tree (AST) structure where internal nodes better consolidate higher-level information than the leaf variable nodes. Thus, as a coverage metric we use coverage statistics about the internal nodes of the formula. For each internal node of type Boolean, we remember whether this node ever received the values of *True* or *False* in the generated solutions. Additionally, for internal nodes of type bit-vector, we remember if each of its bits ever received the values 1 or 0 in the generated solutions. The coverage metric is the number of such internal Booleans and bits which received both possible values among the set of generated solutions.

This metric can be thought of as a measure of the coverage of a circuit that evaluates the constraint. One could synthesize a circuit that takes as inputs assignments to the variables of the formula and produces a Boolean output of *True* or *False* indicating whether the formula is satisfied. The values computed by the internal nodes of the formula correspond to the intermediate values computed by the internal wires of this circuit. In this sense, the coverage metric we defined is equivalent to the coverage of internal wires in this circuit, when it is exercised by the generated solutions. Therefore, we use this metric as a proxy for the coverage that could be obtained when executing the design under test with the generated stimuli.

5.2 Experimental Results

Our benchmarks are obtained from SMT-LIB [1], specifically the problems in the logic QF_AUFBV and its sublogics, such as QF_ABV, and QF_BV. The benchmarks include problems from the verification of hardware and software, bounded model checking, symbolic execution, static analysis and others.

---

[1] The source code is available at https://github.com/RafaelTupynamba/SMTSampler.

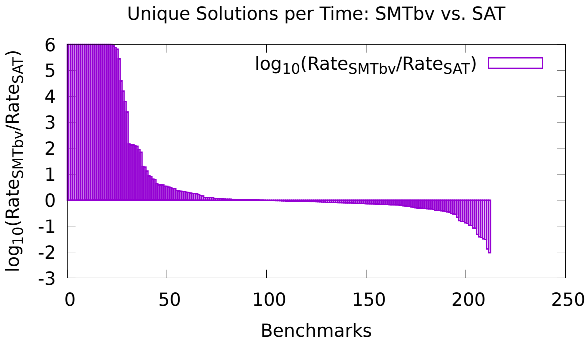

Figure 2: Speed comparison between SMTbv and SAT

We have tried all techniques on benchmarks from each directory available from those logics of SMT-LIB. Some directories had benchmarks which were inadequate for the problem, so we discarded them from the results. Those are cases where the formula is unsatisfiable, or the number of unique solutions that can be produced is less than 100, or where no coverage can be obtained. We ran the experiments over the remaining 22 directories, by randomly choosing 15 benchmarks from each, when there were at least 15 benchmarks available. A total of 274 benchmarks were chosen following this procedure. From those, we excluded the benchmarks for which none of the techniques were able to produce more than one solution, leaving a final set of 213 benchmarks.

Table 4 shows the directories of benchmarks used, along with average statistics from the benchmarks in each directory. We first list the number n of benchmarks which were used from each directory. All other values in the table are averages computed over those n benchmarks in a directory. We list the number of internal nodes in the SMT formula, as a measure of the benchmark size. We also list the number of variables from each type *Array*, *BV*, *Bool*, UF. The 'bits' column represents the total number of bits in all the bit-vector and Boolean variables in the formula.

The next columns present average results from the experiments with the three techniques. First, we list the number of unique solutions produced, then the ratio of unique solutions over time and, finally, the total coverage obtained. When computing the rate of unique solutions over time, we only include the time spent executing Z3 API calls. This is to ensure that the result is fair and not influenced by our implementation of the methods to store, process and combine solutions. In those Z3 API calls we include the time for solving constraints, checking the validity of solutions and converting solutions from SAT into SMT format. We do not include the time spent computing the coverage achieved by the solutions, as the coverage computation is done only for evaluation and is not required to apply the techniques.

Overall, we see that the SMT-based techniques tend to perform better than the bit-blasting approach. For a more thorough evaluation, we present graphs representing the rate of solution generation and the coverage on all 213 benchmarks.

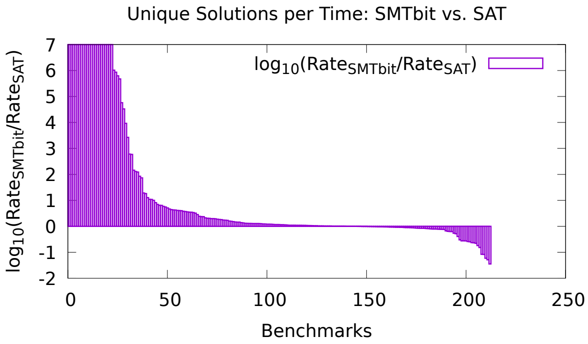

Figure 3: Speed comparison between SMTbit and SAT

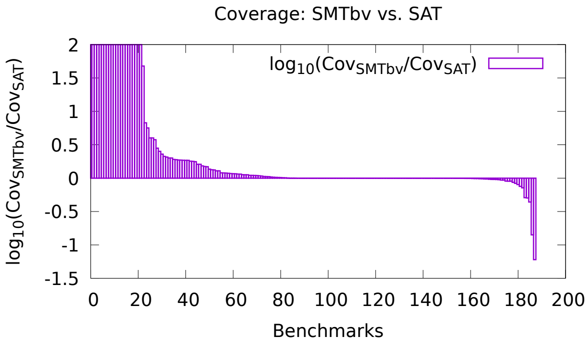

Figure 4: Coverage comparison between SMTbv and SAT

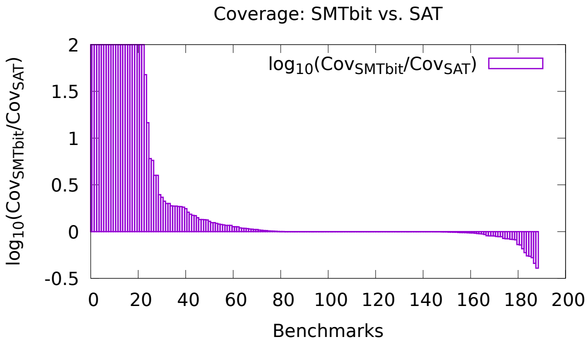

Figure 5: Coverage comparison between SMTbit and SAT

Figure 2 compares the rate of generation of unique solutions for the techniques SMTbv and SAT. Figure 3 is the analogous graph comparing SMTbit and SAT. The rates are defined as the number of unique solutions produced divided by the time spent in calls to the Z3 APIs. The y axis represents the logarithm in base 10 of these rates for both techniques. Higher bars indicate that the SMT-based approach performed better than the SAT-based approach on that benchmark. For 23 benchmarks, the SAT approach was unable to produce any solutions because of a solver timeout. In these cases, the logarithm would be $+\infty$. Those are represented by bars that reach the top of the graph.

Figures 2 and 3 show that, in general, the approaches that work over SMT formulas can generate more unique solutions in a given time budget, compared to bit-blasting. There were some benchmarks for which the SAT approach was able to generate more samples, such as some of the benchmarks from QF_ABV/egt and QF_ABV/bench_ab. Analyzing those benchmarks, we found that they were mostly composed of Boolean operations from combinatorial logic, with very few bit-vector operations. It is natural that, in

Table 4: Average Results Over the Benchmarks

								Unique solutions			Unique solutions per second			Coverage		
Benchmarks	n	nodes	$\|Array\|$	$\|BV\|$	$\|Bool\|$	bits	$\|UF\|$	SMTbv	SMTbit	SAT	SMTbv	SMTbit	SAT	SMTbv	SMTbit	SAT
QF_AUFBV/ecc	4	291	1	42	12	2785	1	209535	26860	49087	579	748	680	407	856	596
QF_ABV/bmc-arrays	3	855	1	1	0	53	0	807570	1229174	1096836	1347	1757	1085	3090	3480	3134
QF_ABV/stp_samples	15	1139	1	24	0	192	0	258440	437702	287537	179	309	220	4484	4768	4035
QF_ABV/dwp_formulas	5	613	3	32	0	428	0	898530	1300388	319558	746	989	259	1276	1016	377
QF_ABV/egt	15	90	1	0	0	0	0	769975	916283	1333783	879	1093	2114	138	136	140
QF_ABV/bench_ab	15	317	1	0	0	6	0	181716	341169	689368	761	1295	2621	129	129	114
QF_ABV/platania/...member	12	4152	36	463	0	14816	0	2085	113046	0	558	515	0	83	44	0
QF_BV/bmc-bv	10	782	0	13	0	422	0	439867	1250125	297913	6161	7011	5604	98	94	95
QF_BV/bmc-bv-svcomp14	8	7518	0	205	1055	7607	0	40809	5915	131089	194	212	153	3081	3108	3655
QF_BV/spear/zebra_v0.95a	9	571	0	185	0	2012	0	196822	18633	35	858	1319	216	530	534	28
QF_BV/RWS	9	1086	0	21	0	3628	0	4776	476	201	31	71	23	4766	4766	2517
QF_BV/gulwani-pldi08	5	1146	0	130	0	950	0	167745	153184	127868	268	305	279	2113	2150	2107
QF_BV/stp_samples	14	793	0	22	0	200	0	505214	501507	438752	375	373	399	1426	1331	969
QF_BV/brummayerbiere2	3	632	0	2	0	149	0	263405	531815	712535	362	625	857	224	187	224
QF_BV/tacas07	3	8812	0	345	588	16620	0	33928	2444	28465	165	183	112	1520	12535	4781
QF_BV/bench_ab	13	23	0	2	0	41	0	1109129	3385658	2783503	7797	10677	10143	11	11	10
QF_BV/sage/app2	13	240	0	25	0	211	0	218827	213889	217598	162	159	180	861	289	433
QF_BV/sage/app9	8	271	0	35	0	391	0	1137172	1984923	1495072	1301	1616	1595	466	464	465
QF_BV/sage/app8	15	978	0	93	0	1047	0	389719	543033	260763	202	375	317	1515	1523	1506
QF_BV/sage/app5	12	269	0	29	0	355	0	1146022	1397666	1008205	1351	1638	1657	391	205	261
QF_BV/sage/app1	10	117	0	21	0	271	0	243452	509655	527177	1171	2421	2362	281	294	267
QF_BV/sage/app12	12	247	0	31	0	358	0	470245	847513	334077	1168	1322	844	313	298	201

those cases, a SAT representation for the formula is more efficient and can be solved faster.

However, we also noticed that for many of those formulas, the larger number of solutions produced by SAT did not give any increase in the coverage metric. This reinforces our hypothesis that the speed to generate unique solutions is an incomplete metric and we should also be analyzing the coverage obtained by the different approaches. Figures 4 and 5 present the graphs comparing the coverage obtained by the SMT-based approaches and the SAT-based approach. Here, we see even more noticeable differences in favor of the SMT-based approaches, especially SMTbit. On the benchmarks from QF_ABV/bmc-arrays, QF_ABV/dwp_formulas, QF_BV/stp_samples and QF_BV/tacas07, for example, those approaches obtained significantly more coverage than SAT.

Overall, we see that the SMT-based approaches proposed by SMTSampler perform better than the baseline bit-blasting approach. They are more robust, being able to produce solutions and obtain coverage on a larger range of benchmarks, while also obtaining a higher constraint coverage and generating solutions at a higher speed in most cases. Between the two SMT-based approaches, we could not identify a clear winner. We noticed that the fine-grain soft constraints from SMTbit approach can help obtaining more precise atomic mutations and produce higher coverage. However, SMTbv tends to be more robust on formulas where the MAX-SMT queries are harder to solve, since it uses a smaller number of soft constraints.

6 CONCLUSION

We proposed a technique SMTSampler for efficient stimulus generation from SMT constraints. SMTSampler works by learning atomic mutations over the SMT solutions and combining those mutations to generate new samples. Our evaluation over a large set of industrial SMT benchmarks shows that working over SMT solutions allows SMTSampler to be effective on a larger set of formulas, generate more unique samples and obtain a better coverage of the constraint space.

ACKNOWLEDGMENTS

Research partially funded[2] by Brazilian SwB CAPES 13245/13-9; NSF grants CCF-1409872 and CCF-1423645; DARPA CRAFT HR0011-16-C-0052; Intel Science and Technology Center for Agile Design; and ADEPT Lab industrial sponsors and affiliates Intel, Google, Siemens and SK Hynix.

REFERENCES

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`. (2016).

[2] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation.. In *TACAS*. 304–319.

[3] Supratik Chakraborty, Kuldeep S Meel, Rakesh Mistry, and Moshe Y Vardi. 2016. Approximate Probabilistic Inference via Word-Level Counting.. In *AAAI*, Vol. 16. 3218–3224.

[4] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2014. Balancing scalability and uniformity in SAT witness generator. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 1–6.

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

[6] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *ICSE'18*.

[7] Nathan Kitchen and Andreas Kuehlmann. 2007. Stimulus generation for constrained random simulation. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*. IEEE, 258–265.

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

[9] Kuldeep S Meel. 2014. Sampling techniques for boolean satisfiability. *Master's thesis* (2014).

[10] Alexander Nadel. 2011. Generating Diverse Solutions in SAT.. In *SAT*. Springer, 287–301.

[11] Reuven Naveh and Amit Metodi. 2013. Beyond feasibility: CP usage in constrained-random functional hardware verification. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 823–831.

[12] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. 2007. Constraint-based random stimuli generation for hardware verification. *AI magazine* 28, 3 (2007), 13.

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

[14] Wei Wei, Jordan Erenrich, and Bart Selman. 2004. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, Vol. 4. 670–676.

[15] Yanni Zhao, Jinian Bian, Shujun Deng, and Zhiqiu Kong. 2009. Random stimulus generation with self-tuning. In *Computer Supported Cooperative Work in Design, 2009. CSCWD 2009. 13th International Conference on*. IEEE, 62–65.

---

[2] 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.