SMTSAMPER: 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:

1 INTRODUCTION
Constrained-random verification (CRV) [12] is one of the 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 \( x > 5 \lor \phi \), where \( x \) is a 32-bit integer and \( \phi \) 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...
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 SAT formulas into SMT. 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, which 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 ≤ 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 SAT formula to SMT 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 $V = \text{Bool} \cup \text{BV} \cup \text{Array} \cup \text{UF}$, where $\text{Bool}$, $\text{BV}$, $\text{Array}$, and $\text{UF}$ 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 $\land$, $\lor$, $\neg$. 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.
Typically, only a small number of indices will be relevant when solving a constraint, even for array domains such as BV. In the example shown, an array function can have any arity. The example shows the function denoted by \( \sigma \), which maps a function to its arguments.

Nothing is known about the result of applying a function to its arguments. Nothing is known 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 \( b, v, a, f \) 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 \( v \in V \) we denote by \( \sigma[v] \) the concrete assignment to \( v \) under \( \sigma \).

Variables in \( BV \) are fixed-size bit-vectors, such as the variable \( v \in BV[8] \). Arrays must have bit-vector domains and ranges, such as the array \( a \), with domain \( BV[3] \) and range \( BV[4] \). Uninterpreted functions can have any arity. The example shows the function \( f : BV[1] \times BV[2] \rightarrow BV[2] \), of arity 2. A concrete instance \( a \) for an array \( a \) is constructed by defining its value for a finite set of indices \( l(a) \) and defining a default value \( d(a) \) for all other indices. In the example shown, \( l(a) = \{001, 011, 101\} \) 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.

### Table 1: Types of variables allowed

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b \in \text{Bool} )</td>
<td>( b = \text{False} )</td>
<td>( v = 01100111 )</td>
</tr>
</tbody>
</table>
| \( v \in BV \) | \( v = 01100111 \) | \( a[x] = \) \[
\begin{align*}
&\text{if } x = 001: \quad 0110 \\
&\text{if } x = 011: \quad 1001 \\
&\text{if } x = 101: \quad 0101 \\
&\text{otherwise: } \quad 0010
\end{align*}
\] |
| \( a \in \text{Array} \) | \( a[x] = \) \[
\begin{align*}
&\text{if } x = 0 \land y = 10: \quad 10 \\
&\text{if } x = 1 \land y = 00: \quad 01 \\
&\text{otherwise: } \quad 11
\end{align*}
\] |
| \( f \in UF \) | \( f(x, y) = \) \[
\begin{align*}
&\text{if } x = 0 \land y = 10: \quad 10 \\
&\text{if } x = 1 \land y = 00: \quad 01 \\
&\text{otherwise: } \quad 11
\end{align*}
\] |

For example, the formula \( \text{select}(a, 011) + v > 0110 \) contains an array variable \( a \) and a bit-vector variable \( v \), two bit-vector constants \( 011 \) and \( 0110 \), an array function \( \text{select} \), 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 \( \text{select} \) and \( \text{store} \) that satisfy the usual axiom

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

Here, \( x, x' \in BV[s_x] \) are bit-vectors of a certain size \( s_x \) and \( y \in BV[s_y] \) is a bit-vector with a possibly different size \( s_y \). \( a \) is an array of domain \( BV[s_x] \) and range \( BV[s_y] \). The function \( \text{select} \) returns the value at a given index from the array, while \( \text{store} \) 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.

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 \( \phi \). The SMTSAMPLER algorithm works over several epochs. In each epoch, we first sample one initial solution \( \sigma \) to the formula, which we call a base solution. This is done by generating a random assignment \( \sigma' \) to the variables of the formula in line 3 and then calling \( \text{findClosestSolution} \) to obtain the solution \( \sigma \) which is closest to \( \sigma' \) in line 4. The details of this procedure are given in §4.2. Then, in line 6, we use the function \( \text{computeNeighboringSolutions} \) to compute a set \( \Sigma_\sigma \) of neighboring solutions for \( \sigma \). This function will be described in §4.3.

The mutations that can be applied to \( \sigma \) 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 \( V \) in the formula. We denote by \( \Psi_\sigma(\sigma_a, \sigma_b) \) the application of the combination function to the base solution \( \sigma \) and two other solutions \( \sigma_a \) and \( \sigma_b \). Intuitively, the combination function \( \Psi \) computes the mutations which can be applied to \( \sigma \) to generate \( \sigma_a \) and \( \sigma_b \), then merges those two mutations together to produce a new assignment. The assignment returned by \( \Psi \) 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 \( \Psi \) to combine those changes in an additive way. The full definition of \( \Psi \) is given in §4.4.

We next describe how function \( \text{combine} \) uses \( \Psi \) to generate new samples. We denote by \( \Sigma_k^\sigma \) the set of neighboring solutions to \( \sigma \) obtained from \( \text{computeNeighboringSolutions} \). Starting from \( \Sigma_k^\sigma \), our goal is to compute sets \( \Sigma_k^\sigma \) which will contain solutions generated by combining \( k \) atomic mutations, for \( 1 \leq k \leq 6 \). Throughout
Algorithm 1 SMTSampler algorithm

1: function SMTSampler(ϕ)
2:   while not done do
3:     σ′ ← generateRandomAssignment(ϕ)
4:     σ ← findClosestSolution(ϕ, σ′)
5:     output(σ)
6:     Σσ ← computeNeighboringSolutions(ϕ, σ)
7:     output(Σσ)
8:     α ← 1, k ← 1, Σσ ← Σσ
9:     while α ≥ αmin ∧ k < 6 do
10:        (Σσ+1, α, Σσ) ← combine(Σσ, Σσ, ϕ)
11:        output(Σσ+1)
12:        k ← k + 1
13:     end while
14: end while
15: function computeNeighboringSolutions(ϕ, σ)
16:   Cσ ← getConditions(ϕ, σ)
17:   Σσ ← {}  
18: for c in Cσ do
19:     Σσ ← Σσ ∪ findNeighboringSolution(ϕ, c, Cσ)
20:   return Σσ
21: function combine(Σσ, Σσ, ϕ)
22:   valid ← 0, checks ← 0
23: for (σa, σb) in Σσ × Σσ do
24:   σ ← Ψσ(σa, σb)
25:   if σ ∉ Σσ then
26:     Σσ ← Σσ ∪ {σ}
27:     checks ← checks + 1
28: if isValidSolution(σ, ϕ) then
29:     Σσ ← Σσ ∧ {σ}
30:     valid ← valid + 1
31: return (Σσ, valid/checks, Σσ)

the current epoch, we maintain a set Σσ of samples which were computed so far, both valid and invalid. Initially, Σσ = Σσ.

Now assume that we already constructed a set Σσ. We can inductively build the set Σσ+1 as follows. For each pair of samples σa ∈ Σσ and σb ∈ Σσ, we apply the combination function Ψ to generate a new sample σ = Ψσ(σa, σb). If σ is an element of Σσ, it has already been checked and is discarded. Otherwise, we add it to Σσ 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 σ. If σ is a valid solution, it is then added to Σσ+1. During the construction of Σσ+1 from Σσ, we keep statistics on which fraction α of the checked samples were valid. If this fraction is below a certain threshold αmin, such as 0.1, we do not generate Σσ+2 and 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 ∪0≤k≤KΣσ, where we define Σσ = {σ}. Those are all valid solutions to the formula, as the ones which were produced by the combination function for 2 ≤ k ≤ 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.

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 v, we add one soft constraint v = σ′[b] stating that the v should have the same value that it had in σ′. Analogously, we add one soft constraint b = σ′[b] for each Boolean variable 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 Cσ 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 Cσ, 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 Cσ. Function getConditions produces Cσ 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 v and an integer index i and returns the value of the bit at index i in v.

The conditions are generated as follows. For each Boolean variable, we add one condition b = σ′[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(v, i) = extract(σ′[v][i], 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.
We specify two hard constraints $\{v \in BV\}$.

For each such index $x$, we consider the concrete bit-vector $[a][x]$ returned by the array on such index and we add one condition for each bit in this bit-vector, such as $\text{extract}(a[x], i) = \text{extract}(\sigma[a][x], i)$. 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 $\Sigma^2_\sigma$. After collecting the fine-grained conditions in $C_\sigma$, we want to compute neighboring solutions by picking one condition $c \in C_\sigma$ and using the constraint solver to find a solution to $\phi \land \neg c$, where $\phi$ is the original formula. However, the neighboring solution should be as similar as possible to $\sigma$. 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 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, 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 $\psi = 00100111$, instead of 8 different conditions such as $\text{extract}(a[0], 0) = 0$ and $\text{extract}(a[1], 1) = 0$. 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 $\Psi$ which we use to generate new samples. Assume that we already know the base solution $\sigma$ and two additional solutions to the formula $\sigma_a$ and $\sigma_b$, which are close to $\sigma$. Those additional solutions can be obtained by calling computeNeighboringSolutions or they could be already generated by an application of the $\Psi$ function.

The combination function $\Psi$, which combines entire solutions, is constructed by defining a method $\psi$ to combine the values of each of the variables in the formula. We define $\Psi(\sigma_a, \sigma_b)[v] = \psi(\sigma_a[v], \sigma_b[v])$.

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

Next, we define how the combination method $\psi$ 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. For each $v \in BV$ be a bit-vector variable in the formula. We use the notations $\psi(v, v_a, v_b)$ to represent the values assigned to variable $v$ in each of the solutions $\sigma, \sigma_a, \sigma_b$, i.e. we define $\psi = \sigma[v], v_a = \sigma_a[v], v_b = \sigma_b[v]$.

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 $\delta_a$ and $\delta_b$ 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 $\delta_a$ 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 $(\delta_a \lor \delta_b)$. This resulting mutation can be applied to the base value $v$, producing a new value $v \lor (\delta_a \lor \delta_b)$. Thus, for bit-vectors, $\psi$ is defined as $\psi(\sigma_a, \sigma_b) = v \lor (v_a \lor v_b)$.

Now we generalize the definition of $\psi$ to other types of variables. For Boolean values, we use the same technique: $\psi(\sigma_a[b], \sigma_b[b]) = b \lor (b \lor b)$, this way, Boolean values behave the same as bit-vectors of size 1.

Now we define how to apply the combination method $\psi$ to a base array $\sigma = \sigma[a]$ and two neighboring arrays $\sigma_a = \sigma[a]$ and $\sigma_b = \sigma[b]$. 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 $\sigma_a$ and $\sigma_b$ are constructed analogously, with possibly different sets of assigned indices $I(\sigma_a)$ and $I(\sigma_b)$. We define the combination function for arrays as $\psi(\sigma_a[a], \sigma_b[a])[x] = \begin{cases} \psi_a(a[x], a_b[x]), & \text{if } x \in I(a) \cup I(\sigma_a) \cup I(\sigma_b) \\ \psi_{d(a)}(d(a), d(a)), & \text{otherwise} \end{cases}$.

This means that the assigned indices of the generated array will be $I = I(a) \cup (I(\sigma_a) \cup I(\sigma_b))$, the union of the assigned indices of each of the three arrays. If $x \notin I$, 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 $\psi(\sigma, \sigma_a, \sigma_b)$ 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 $\psi$ and, consequently, $\Psi$.

The motivation for this definition of $\Psi(\sigma_a, \sigma_b)$ is that it attempts to obtain the mutation that generates $\sigma_b$ from $\sigma$ and the mutation that generates $\sigma_b$ from $\sigma$ and then combine those two mutations in an additive way. If $\sigma_a$ and $\sigma_b$ 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
We did not compare against techniques such as the MCMC-based approach from Ambigen because those are only applicable over constraints 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.

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.

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.

5.2 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.
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 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 +∞. 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
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.

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.