

Stimulus Generation for Constrained Random Simulation

Nathan Kitchen¹

Andreas Kuehlmann^{1,2}

¹ University of California at Berkeley, CA, USA

² Cadence Research Labs, Berkeley, CA, USA

Abstract

Constrained random simulation is the main workhorse in today’s hardware verification flows. It requires the random generation of input stimuli that obey a set of declaratively specified input constraints, which are then applied to validate given design properties by simulation. The efficiency of the overall flow depends critically on (1) the performance of the constraint solver and (2) the distribution of the generated solutions. In this paper we discuss the overall problem of efficient constraint solving for stimulus generation for mixed Boolean/integer variable domains and propose a new hybrid solver based on Markov-chain Monte Carlo methods with good performance and distribution.

1 Introduction

Many contemporary verification flows in industry have adopted a combination of formal property checking and constrained random testing. Most research in the area of functional verification has focused on formal methods, and considerable theoretical and practical progress has been made in this field in the past 15 years. On the other hand, constrained random simulation has attracted significantly less attention, despite its importance in practical verification flows due to limited scalability of formal methods. At their core, both approaches require efficient constraint solving, but formal verification typically seeks out a single solution, whereas stimulus generation requires repeated generation of solutions with a good distribution (e.g., uniform) over the solution space.

In constrained random verification (CRV), the testbench for the design under test (DUT) includes (1) a generator of random stimuli, (2) monitors that check the correctness of the behavior, and (3) coverage analyzers for measuring which parts of the state space have been verified. Figure 1 depicts the structure of a CRV testbench, including these components.

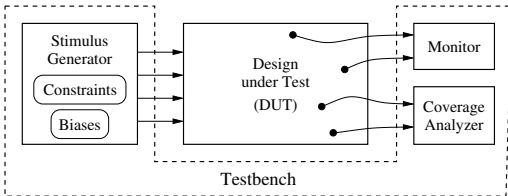
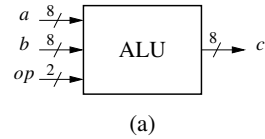


Figure 1: Structure of a setup for constrained random verification.

In order to avoid generating invalid stimuli that might lead to false negative verification results, the stimulus generator must obey input constraints. For example, the implementation of a protocol interface may only be correct for data packets with valid headers, so the input constraints would exclude invalid headers. The constraints may be implicit in the generator code, or they may be given explicitly as input to the generator.

Specific languages used in practice for specifying testbenches include SystemC [1], SystemVerilog [2], and e [3]. An example of constraints for an 8-bit ALU, written in SystemVerilog, is shown in Figure 2. The constraints specify input values such that the output c is computed without overflow or divide-by-zero exceptions.



```
rand enum { ADD, SUB, MUL, DIV } op;
rand bit signed [7:0] a, b;
constraint valid {
    op == ADD -> (-128 <= a + b && a + b <= 127);
    op == SUB -> (-128 <= a - b && a - b <= 127);
    op == MUL -> (-128 <= a * b && a * b <= 127);
    op == DIV -> (b != 0);
}
```

Figure 2: An example DUT—an 8-bit ALU: (a) block diagram, (b) constraints in SystemVerilog.

The distribution of the generated input vectors has a direct effect on the amount of time it takes to meet the coverage goals. In order to achieve coverage as quickly as possible, the input distribution should closely match the distribution of coverage points, e.g. the firing conditions for the transitions in the state-transition graph. In the absence of information about the distribution of coverage points, the best choice for the input distribution is the one with maximum entropy—i.e., the uniform distribution—so as to maximize the level of “surprise” and thus the chance of entering an unexplored region of the state space. When many of the coverage points lie in the corners of the state space, it is better to use a distribution with increased probability at the corners, such as a piecewise-uniform distribution, which may be specified by the use of biases on the input variables.

The efficiency of the constraint solver is as important as the quality of the stimulus distribution. An inefficient solver impacts the overall verification effort by long runtimes for stimulus generation, whereas a highly skewed distribution can dramatically increase the number of simulation steps required to execute a target transition of the design. This dual challenge is particularly demanding as the search for a single solution is already NP-hard for non-trivial Boolean constraints. Thus far little research has been focused on this problem despite its high relevance to practical verification methods. The difficulty of the problem is reflected in the limitations of state-of-the-art tools for CRV. Some tools generate random values efficiently, but from highly non-uniform value distributions, resulting in increased time to achieve coverage. Moreover, the distributions are unstable—they vary significantly with

small changes in the specification, such as changes in variable declaration order. Other approaches generate values uniformly but rely on methods (e.g., binary decision diagrams) that do not scale well to large designs.

2 Related Work

In previous work, there are various methods proposed for generating stimuli from a declarative description. In the following, we give a quick overview.

Weighted Binary Decision Diagram (BDD) sampling was proposed in [4, 5]. The idea is to build a BDD from the input constraints and to weight the branches of the vertices in such a way that a simple linear walk procedure from the root to the terminal vertex generates stimuli with a desired distribution [4, 5]. The weighted BDD approach is quite adequate to sample from control-dominated input constraints as long as they are not too complex. However, for integer constraints originating from arithmetic operations that have conflicting variable assignments, contain multiplications, or have complex select functions, the BDDs easily blow up and make this approach unusable. Section 5 provides a detailed analysis of a BDD-based sampler with the method proposed in this paper.

Another BDD-based approach [6] involves building a circuit whose structure matches the constraint BDD. This method does not require the variables to be ordered, and so it may avoid the common memory blowups, but its output distribution may be highly skewed.

Interval-propagation-based sampling is a technique that is used in multiple industrial applications such as [7]. It applies an iterative process where an interval of possible values is maintained for each variable. Each variable is successively assigned a randomly chosen value from its interval, and the intervals of the remaining variables are subsequently refined. The advantage of the interval-propagation sampling algorithm is its simplicity and relatively high performance. However, the sampling distribution of this algorithm is highly non-uniform. This results in a non-optimal stimulus entropy which reduces the overall coverage in the verification setup. To illustrate this problem, Figure 3 depicts the skewed sampling distribution for 505000 samples for an integer region given by: $y_1 + y_2 \leq 101; y_1, y_2 \geq 1$. Moreover, it can be shown that for an n -dimensional simplex of the form $\sum_{i=1}^n y_i \leq c$ there is a gap between the expected number of samples for $(0, 0, \dots, 0)$ and any corner $(0, \dots, c, \dots, 0)$ that is exponential in n .

DPLL-based sampling utilizes CNF-based DPLL-style SAT solvers [8, 9, 10] to generate stimuli from constraints. The advantage is their fairly good scalability for a large set of practical constraints. A random pre-assignment of variables is used to attempt a good distribution for the generated samples. Starting from this random variable assignment, a DPLL-style search is then applied. At the first conflict, the solver goes into backtracking mode, effectively finding a smallest cube that contains both the pre-assignment and a solution. The problem with SAT-based methods is that, depending on the constraints, the distribution can be highly non-uniform. Furthermore, their runtime can be slow, especially for constraints involving arithmetic. In Section 5 we report a detailed comparison of our proposed solver with a DPLL-style approach.

Random-walk sampling for Boolean constraints was achieved in SAMPLESAT [11] by combining Metropolis sampling steps [12] with greedy steps from WALKSAT [13]. This method works well for small constraint sets, but due to the incomplete and greedy nature of the WALKSAT search its performance quickly deteriorates for larger constraint sets, especially for constraints that have been generated from data paths.

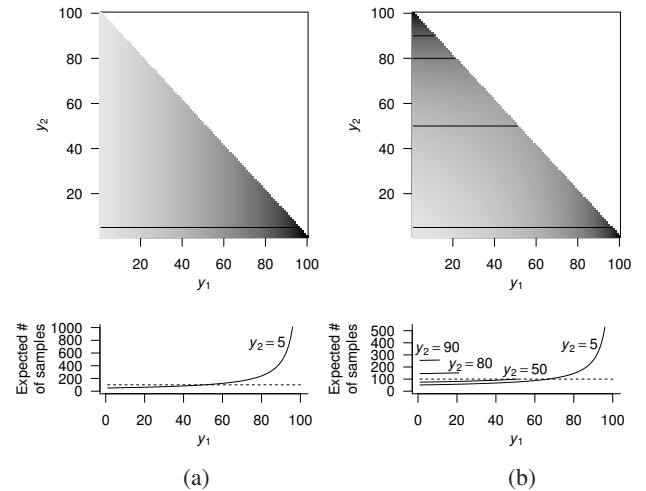


Figure 3: Distribution of 505000 samples when using interval-propagation-based sampling for sampling $y_1 + y_2 \leq 101; y_1, y_2 \geq 1$: (a) using fixed sampling order y_1, y_2 , (b) using random sampling order. The lower parts show the expected number of samples for each y_1 for fixed values of y_2 . The dashed lines indicate the expected values for uniform distribution.

Another line of research [14, 15] converts the constraints into **belief networks** and performs approximately uniform sampling based on estimating the size of the solution space. As reported in [15], the performance of this method does not match that achieved by SAMPLESAT, despite some improvements adopted from modern constraint solvers.

Various **specialized stimulus generators** [16, 17] have been developed for specific verification domains utilizing particular domain knowledge for constraint specification and solving.

In this paper we first introduce a general normal form for constraint specification, the *Mixed Boolean/integer constraint normal form*. We then present a constraint solver that utilizes multiple concepts from previous work for solving the Boolean domain and combines it with a new solver based on Markov-chain Monte Carlo (MCMC) methods [12, 18] for the integer domain. Specifically, we use the combination of Metropolis and WALKSAT steps suggested in SAMPLESAT for gaining a good distribution for the Boolean part but improve its performance by switching to a DPLL-style search for samples that “time out”. For the integer part we use a sampler that was inspired by the Gibbs method [19, 20] which provides excellent performance and a flexible control of the distribution.

The presented stimulus generator offers a key component for contemporary random simulation methodologies but is also part of a larger line of research which attempts to combine formal methods with simulation [21, 22] and coverage-driven verification flows.

3 Constraint Specification

In this section we discuss the specification of random stimuli. The presented approach is sufficiently expressive for many practical applications, but also lends itself to efficient algorithms for approximately uniform sampling. Reflecting the typical application for hardware verification, we use a combination of Boolean variables and integer variables of bounded domain (i.e., of fixed bit-width) for the inputs of the DUT. As constraints we support a combination of Boolean constraints and restricted classes of inequalities on the integers.

Formally, let $x = (x_1, \dots, x_m)$ denote a vector of m Boolean variables and $y = (y_1, \dots, y_n)$; $-2^{B-1} \leq y_i \leq 2^{B-1} - 1$ denote a vector of n integer variables, where B is a positive integer, i.e., the maximum bit-width of y_i . The constraints on assignments to x are specified in terms of a Boolean function $f(x)$, where $f(x) = 1$ for all valid assignments x . We allow the constraints on y to be conditional on x , i.e., different assignments for x may “trigger” different constraints on the y values. The constraint which is active for a particular set of values of x is denoted by $g^{(x)}(y)$. Then the set of valid stimuli is the set of assignments

$$\{(x, y) : f(x) = 1 \wedge g^{(x)}(y) = 1\}.$$

In the most general form, each $g^{(x)}(y)$ is expressed as a disjunction of conjunctions of predicates $g_{ij}^{(x)}$ of the y variables:

$$g^{(x)}(y) = \bigvee_i \bigwedge_j g_{ij}^{(x)}(y).$$

Each of the $g_{ij}^{(x)}(y)$ denotes the indicator function for a constraint on the values of y . The $\bigwedge_j g_{ij}^{(x)}$ can be seen as bounding a single region in the y space, whereas their disjunction \bigvee_i combines multiple regions that are valid for particular x values.

3.1 Constraints on y

In general, the $g_{ij}^{(x)}$ can be arbitrary arithmetic constraints on the y variables. However, for performance reasons, it is beneficial to restrict the type of constraints to a set that is of practical value and, at the same time, can be solved efficiently. In our case, we support inequalities on the y variables. Note that an equality can be expressed as a pair of inequalities, i.e., $(a = b) \Leftrightarrow (a \leq b) \wedge (a \geq b)$.

Gibbs-style sampling on such constrained regions can be done fast if every inequality can be resolved explicitly for each y_k in its support. This makes it possible to execute the inner-loop Gibbs step efficiently (see Algorithm 4), which requires the computation of the valid value ranges for the sampled y_k given values of the remaining variables $y \setminus y_k$. By precompiling a set of explicitly resolved inequalities of the form $y_k \leq g(y \setminus y_k)$ or $y_k \geq g(y \setminus y_k)$ the value range computation for y_k is executed in linear time in the number of constraints involving y_k .

In the following sections we discuss two types of inequalities, linear constraints and multilinear constraints, which can be resolved for the y_k and cover the vast majority of constraints that are needed in practical verification flows. Other types of constraints can easily be handled in the presented solver as long as the above stated condition is fulfilled for them.

3.1.1 Linear Constraints

One class of constraints $g_{ij}^{(x)}(y)$ that can be handled efficiently is the class of linear inequalities. In this case, the valid assignments to y are the solutions to a system of inequalities whose coefficients depend on x :

$$\bigvee_{v=1}^n a_{uv}^{(x)} y_v \leq b_u^{(x)}; \quad a_{uv}^{(x)}, b_u^{(x)} \in \mathbb{Z}$$

yielding

$$g_{ij}^{(x)}(y) = \left[\bigvee_{v=1}^n a_{uv}^{(x)} y_v - b_u \leq 0 \right]$$

where $[P]$ denotes the indicator function of event P .

As an example, suppose that for the 8-bit ALU in Figure 2 the ADD and SUB operations are encoded as 00 and 01, respectively.

By using the variables x , y_1 , and y_2 to represent the inputs op , a , and b , respectively, we express the input constraints:

$$\begin{aligned} x_1 = 0 \wedge x_2 = 0 &\Rightarrow -128 \leq y_1 + y_2 \leq 127 \\ x_1 = 0 \wedge x_2 = 1 &\Rightarrow -128 \leq y_1 - y_2 \leq 127 \end{aligned}$$

leading to the following constraints for these two operations:

$$\begin{aligned} x_1 \vee x_2 \vee ([-y_1 - y_2 - 128 \leq 0] \wedge [y_1 + y_2 - 127 \leq 0]) \\ x_1 \vee \neg x_2 \vee ([-y_1 + y_2 - 128 \leq 0] \wedge [y_1 - y_2 - 127 \leq 0]) \end{aligned} \quad (1)$$

3.1.2 Multilinear Constraints

Besides linear inequalities, our approach also works well for a class of non-linear constraints that are of broad practical relevance. In particular, we allow the use of *multilinear inequalities*, a class of constraints in which the terms may multiply several variables but are linear in the individual variables. Formally, a system of multilinear constraints is expressed as

$$\sum_v a_{uv}^{(x)} \pi_v \leq b_u^{(x)}; \quad a_{uv}^{(x)}, b_u^{(x)} \in \mathbb{Z}$$

where each π_v is the product of some subset of $\{y_1, \dots, y_n\}$. The corresponding constraint functions are then defined as follows:

$$g_{ij}^{(x)}(y) = \left[\sum_u a_{uv}^{(x)} \pi_v - b_u^{(x)} \leq 0 \right]$$

Multilinear inequalities can express multiplicative constraints that cannot be encoded as linear inequalities. For example, they allow us to extend the ALU input constraints in (1) to handle the MUL operation. If MUL is encoded as 10, the added constraint is:

$$\neg x_1 \vee x_2 \vee ([-y_1 y_2 - 128 \leq 0] \wedge [y_1 y_2 - 127 \leq 0])$$

Efficient handling of multilinear inequalities is one significant advantage of our approach over BDD-based methods. Since BDDs for multiplication are exponentially large [23], they cannot express multiplicative constraints in a scalable way.

3.2 Normal Form for Boolean and Integer Constraints

Using the examples from the previous section, we now introduce a mixed Boolean/integer constraints normal form (MBINF). The MBINF is a simple extension of the conjunctive normal form (CNF) in which the literals are composed from the x variables and inequalities on the y variables. More formally:

$$\begin{aligned} MBINF &:= \bigwedge C_i \\ C_i &:= \bigvee l_{ij} \\ l_{ij} &\in \{x_k, \neg x_k, g_{ij}^{(x)}(y)\} \end{aligned}$$

Following the conventions from CNF we denote the C_i as *clauses* and the l_{ij} as *Boolean literals* ($x_k, \neg x_k$) and *inequality literals* ($g_{ij}^{(x)}(y)$). A constraint of the form $f(x) \wedge g^{(x)}(y)$ can simply be generated by composing the CNF for $f(x)$ with conditional clauses for $g^{(x)}(y)$. These clauses, which may be expressed initially in the natural “triggered” form $g'(x) \Rightarrow g''(y)$, can be translated using the

Boolean identity ($g'(x) \Rightarrow g''(y) \Leftrightarrow (\neg g'(x) \vee g''(y))$). For the ALU example of Figure 2 the MBINF is as follows:

$$\begin{aligned}
&(x_1 \vee \neg x_2 \vee [-y_1 - y_2 - 128 \leq 0]) && \wedge \\
&(x_1 \vee \neg x_2 \vee [y_1 + y_2 - 127 \leq 0]) && \wedge \\
&(x_1 \vee \neg x_2 \vee [-y_1 + y_2 - 128 \leq 0]) && \wedge \\
&(x_1 \vee \neg x_2 \vee [y_1 - y_2 - 127 \leq 0]) && \wedge \\
&(\neg x_1 \vee \neg x_2 \vee [-y_1 y_2 - 128 \leq 0]) && \wedge \\
&(\neg x_1 \vee \neg x_2 \vee [y_1 y_2 - 127 \leq 0]) && \wedge \\
&(\neg x_1 \vee \neg x_2 \vee [y_2 \leq -1] \vee [-y_2 \leq -1]) &&
\end{aligned}$$

Note that this form is general and allows the expression of arbitrary combination of constraints on the x and y variables within the expressiveness of the $g^{(x)}(y)$. For example, a clause may contain multiple inequality literals which supports the encoding of multiple x regions for particular x values as well as non-convex regions.

4 Sampling with Boolean and Integer Constraints

In the following we present an algorithm which generates for a given MBINF a set of valid solutions with approximately uniform distribution. We do not cover the handling of biases, i.e., how the sampling can be adjusted to obey user-specified densities for the x regions. However, the algorithms presented can be applied in a straightforward manner to obtain piecewise-uniform distributions.

Our sampler includes ingredients from Markov-chain Monte Carlo (MCMC) methods (including Metropolis-Hastings [12, 18] and Gibbs sampling [19, 20]) and WALKSAT [13]. The Metropolis algorithm is widely used in the statistics and computational physics communities to generate samples from distributions that are difficult to sample directly. In each iteration, it proposes a random change to the current assignment and accepts it with a probability that depends on the relative weights of the current and proposed assignments. In the limit, i.e., as the number of samples goes to infinity, the distribution converges to the desired distribution.

Gibbs sampling [19, 20] is a special case of Metropolis sampling that can be applied when the conditional distribution of each variable can be computed. The sampler may move across the entire range of a variable in a single step, so it can travel through the sample space faster than other versions of the Metropolis algorithm that use only local moves.

There are three main issues that need to be addressed for adapting Gibbs sampling to stimulus generation:

- Gibbs sampling works only for fully connected sampling regions, which is generally not the case for Boolean constraints and in the limit also not for integer constraints. For example, the region given by $y_1 + y_2 = 100; 0 \leq y_1, y_2 \leq 100$ cannot be handled by Gibbs since a change of only y_1 or y_2 is not sufficient to move from one valid assignment to another. We address this by allowing invalid values with nonzero probability. As a result, some moves leave the valid sampling region. When such an excursion occurs, we use a fast search to get back to a legal assignment.
- Related to the problem above, while sampling an MBINF, a change of a Boolean variable x_k may cause a switch from one y region to another causing that the current y assignment becomes illegal. We utilize the same search mechanism mentioned above to get back to a legal y assignment.
- Successive samples are highly correlated as they often differ in the assignment of a single variable only. To use them in

a verification flow, this correlation must be removed. We apply a *mixing queue* which stores the generated samples in an array and draws randomly from it in each iteration. In our experiments we found that most of the correlation for practical verification constraints can be removed with a short queue.

4.1 Overview of MBINF Sampler

Our sampling algorithm uses the framework of the Metropolis-Hasting algorithm with the relaxed Gibbs step described in the previous section. Each new sample (x', y') is generated from the previous sample (x, y) by the following main steps:

1. Variable selection:

Select uniformly at random a variable $x_k \in x$ or $y_k \in y$.

2. Metropolis move:

Make a random change for the selected variable. For a Boolean variable the assignment for x_k is flipped, i.e., $x'_k = \neg x_k$; for an integer variable a new assignment y'_k is selected for y_k with uniform probability from all valid ranges and exponentially decaying probability from the ranges adjacent to them. Depending on the change in the number of satisfied clauses, the change is then accepted or not.

3. Recovery move:

If the new assignment (x', y') is illegal, i.e., violates at least one constraint of the MBINF, then apply a recovery move to get back to a legal region. The recovery move is composed of a sequence of greedy and Metropolis steps. The greedy steps is a generalization of the WALKSAT step [13] by applying a generalized literal flip. Boolean literals are flipped in the usual manner, whereas flipping of an inequality literal involves the attempt to satisfy the corresponding constraint. We limit the number of generalized WALKSAT steps and utilize a DPLL-style search on the Boolean variable after that limit is reached. This allows to bound the time spend in the recovery phase without significantly impacting the sampling distribution.

We notice that for the recovery from an illegal assignment (x, y) , we could attempt to directly solve the corresponding set of inequalities that are active for the given x assignment. However, the discrete nature of the problem can make such approach expensive and would further restrict the type of supported inequalities beyond the fairly weak limitation of “explicit resolvability” for the y_k .

4.2 Metropolis Move

The Metropolis move selects an x or y variable at random, changes its assignment and then accepts the move with a probability that is dependent on the change in the number of satisfied clauses C_i . Changing a Boolean variable x_k is straightforward. For an integer variables y_k the change is more complex, consisting of a Gibbs-style step to propose a new value. For given values (x, y) the new assignment for the selected y_k is obtained by first projecting the constraints $\{C_i\}$ onto the current assignments x and $y \setminus y_k$, leading to a general set of constraints $\{C'_i\}$ on y_k of the form:

$$\bigwedge_i \bigvee_j g'_{ij}(y_k)$$

The $\{C'_i\}$ can be computed quickly by scanning through the clauses C_i , dropping the ones with at least one true Boolean literal and collecting the inequality literals for the remaining ones. Following up on the introductory discussion of Section 3.1, for linear and multilinear constraints, the $g'_{ij}(y_k)$ can be analyzed quickly

by simply evaluating the precompiled inequalities resolved for y_k . This converts the $g'_{ij}(y_k)$ into a set of constraints $y_k \leq c$ or $y_k \geq c$. We then compose a density function $P(y_k)$ that defines the proposal distribution of y_k :

$$P(y_k) \propto \min_i \max_j P_{ij}(y_k) \quad (2)$$

The $P_{ij}(y_k)$ are computed for the inequalities of the form $y_k \leq c$ ($y_k \geq c$) as follows:

$$P_{ij}(y_k) = \begin{cases} 1 & \text{if } y_k \leq c \text{ (} y_k \geq c \text{)} \\ e^{-|c-y_k|/T} & \text{otherwise} \end{cases} \quad (3)$$

where T is a parameter called the *temperature*.

Figure 4 provides three illustrations of the computation of the proposal density. The purpose of the exponentially decaying distribution for the invalid range is to address the shortcomings of the Gibbs sampler discussed earlier by allowing moves out of the legal regions. Specifically, the definition of the distribution for an empty valid range for y_k (see Figure 4(b)) is chosen to heuristically minimize the effort to get back to a valid assignment after a flip of an x_i changes the valid region for y . Figure 5 illustrates this case.

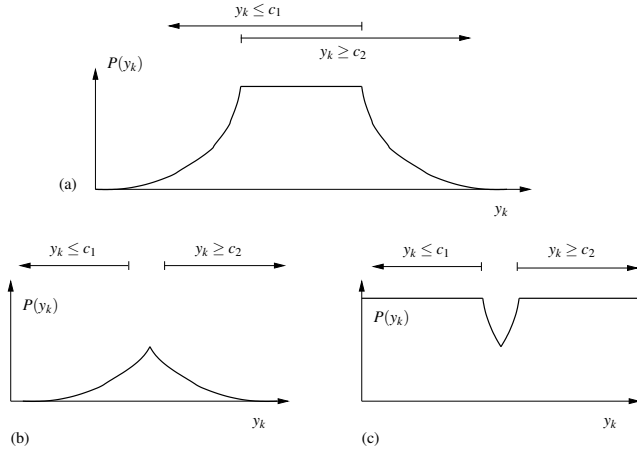


Figure 4: Example for distribution function computed by (2) and (3): (a) $[y_k \leq c_1] \wedge [y_k \geq c_2]$ with $c_1 > c_2$, (b) $[y_k \leq c_1] \wedge [y_k \geq c_2]$ with $c_1 < c_2$, and (c) $[y_k \leq c_1] \vee [y_k \geq c_2]$.

We sample from the distribution defined by $P(y_k)$ by separating the density function into uniform and exponential segments, selecting a segment at random, and sampling a value within the segment. To select a segment, we compute the total weight $w(s_i)$ of each segment s_i , then select segment s_j with probability $w(s_j) / \sum_i w(s_i)$. The sampling procedure is linear in the number of segments because the weight of each segment and the sampling probabilities within segments can be expressed with closed-form expressions.

One consequence of including exponential segments in the density function is that the proposal distribution is not symmetric; i.e., if $Q(y_k, y'_k)$ is the probability of proposing a move from y_k to y'_k , $Q(y_k, y'_k) \neq Q(y'_k, y_k)$ in general. In order to guarantee convergence to the right distribution with an asymmetric proposal distribution, it is necessary to use the Hastings extension of the Metropolis algorithm [18], in which the probability of accepting a move is weighted by the ratio of the proposal probabilities.

4.3 Core Sampling Algorithm

Algorithm 1 provides the pseudo-code for the top level of the presented sampling algorithm. It starts from a random assignment

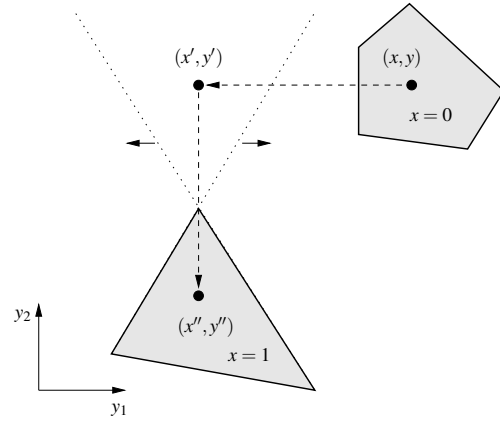


Figure 5: Motivation for the sampling distribution illustrated in Figure 4(b): A switch of the y region through a change from $x = 0$ to $x = 1$ resulted in an empty valid range for y_1 . The chosen distribution enables a quick return to a valid region with few recovery moves (e.g. the two moves shown).

to x and y which may or may not satisfy the constraints $\{C_i\}$. It then fills the mixing queue by generating a sufficient number of samples using the SAMPLE procedure. Once the queue is warmed up, the sampler generates one sample per iteration and applies the mixing queue to shuffle them.

Algorithm 1 Mixed Boolean/integer sampler

- 1: {Given: clauses $\{C_i\}$; number of samples N ; length of mixing queue M ; parameter t ; decimation period K }
 - 2: $(x, y) :=$ random assignment
 - 3: **for** $j := 1$ to M **do** {warm up queue}
 - 4: **for** $k := 1$ to t **do**
 - 5: $(x, y) :=$ SAMPLE(x, y)
 - 6: $QUEUE[j] := (x, y)$
 - 7: **for** $j := 1$ to N **do** {generate samples}
 - 8: **for** $k := 1$ to K **do** {decimate sample stream}
 - 9: $(x, y) :=$ SAMPLE(x, y)
 - 10: select $r \in \{1, \dots, M\}$ uniformly at random
 - 11: $(x', y') := QUEUE[r]$
 - 12: $QUEUE[r] := (x, y)$
 - 13: **output** (x', y')
-

The pseudo-code for the individual sampling step is given in Algorithm 2. It starts with a METROPOLISMOVE, which generates, with a certain probability, a new assignment for (x, y) . If the new assignment is not valid, the algorithm executes a recovery move by first performing a fixed number (R) of METROPOLISMOVES or WALKSATMOVES. If no valid solution is found during these moves, the algorithm finishes with a DPLLMOVE that searches for a valid x assignment. In Section 5 we demonstrate that for practical constraints the number of steps necessary for recovery is typically small, which validates the overall effectiveness of the proposed Metropolis-style sampling approach.

Algorithm 3 provides the details of the METROPOLISMOVE for mixed Boolean/integer constraints. It either flips a Boolean literal x (line 5), or it changes the assignment of a single variable y_k using a GIBBS step (line 10). Depending on the impact on the number of satisfied clauses and the proposal probabilities, the step is then accepted (line 17) or rejected (line 19).

Algorithm 2 SAMPLE

```
1: {Given: clauses  $\{C_i\}$ ; current assignment  $(x,y)$ ; parameters  $R,p$ }
2:  $(x,y) := \text{METROPOLISMOVE}(x,y)$ 
3: for  $i := 1$  to  $R$  do {recovery moves}
4:   if  $(x,y)$  satisfies all clauses  $C_i$  then
5:     return  $(x,y)$ 
6:   with probability  $p$  do
7:      $(x,y) := \text{METROPOLISMOVE}(x,y)$ 
8:   else
9:      $(x,y) := \text{WALKSATMOVE}(x,y)$ 
10:  $(x,y) := \text{DPLLMOVE}(x,y)$ 
11: return  $(x,y)$ 
```

Algorithm 3 METROPOLISMOVE

```
1: {Given: clauses  $\{C_i\}$ ; assignment  $(x,y) = (x_1, \dots, x_m; y_1, \dots, y_n)$ ; temperature  $T$ }
2:  $U := \#$  clauses unsatisfied under  $(x,y)$ 
3: select  $j \in \{1, \dots, m+n\}$  uniformly at random
4: if  $1 \leq j \leq m$  then {Update Boolean variable}
5:    $x'_j := x; x'_j := \bar{x}_j$  {Flip  $x_j$ }
6:    $y' := y$ 
7: else {Update integer variable}
8:    $x' := x$ 
9:    $k := j - m$ 
10:   $y'_k := y; y'_k := \text{GIBBS}(x,y,y_k)$ 
11:   $U' := \#$  clauses unsatisfied under  $(x',y')$ 
12:   $\Delta U := U' - U$ 
13:   $Q :=$  probability of proposing move  $(x,y) \rightarrow (x',y')$ 
14:   $Q' :=$  probability of proposing move  $(x',y') \rightarrow (x,y)$ 
15:   $p := e^{-\Delta U/T} Q' Q^{-1}$ 
16: with probability  $\min\{p, 1\}$  do
17:   return  $(x',y')$ 
18: else
19:   return  $(x,y)$ 
```

Algorithm 4 outlines the GIBBS step. In line 2, the constraints $\{C_i\}$ are projected onto the current x and y assignments, except for the assignment of y_k . In line 3 a new value for y_k is selected by sampling from the distribution that is computed by formula (2) from the $\{C'_i\}$.

The WALKSATMOVE procedure is outlined in Algorithm 5. It first randomly selects an unsatisfied clause C_j . It then chooses, controlled by probability q , between random and greedy moves. In a random move, it flips a randomly chosen literal from C_j (line 5). In a greedy move, it flips the literal in C_j that maximizes the number of satisfied clauses among $\{C_i\}$. The FLIPLIT procedure is outlined in Algorithm 6. A Boolean literal is flipped simply by complementing the corresponding x variable. In case of an inequality literal, an integer variable y_k is chosen randomly from the support set of the inequality and resampled using the GIBBS procedure.

Algorithm 4 GIBBS

```
1: {Given: clauses  $\{C_i\}$ ; assignment  $(x,y) = (x_1, \dots, x_m; y_1, \dots, y_n)$ ; variable  $y_k$  to be resampled}
2:  $\{C'_i\} := \text{PROJECT}(\{C_i\}, x, y \setminus y_k)$   $\{C'_i$  constraints on  $y'_k$  only}
3:  $y'_k := \text{SAMPLEFROM}(\{C'_i\})$ 
4: return  $y'_k$ 
```

Algorithm 5 WALKSATMOVE

```
1: {Given: clauses  $\{C_i\}$ ; assignment  $(x,y) = (x_1, \dots, x_m; y_1, \dots, y_n)$ ; noise parameter  $q$ }
2: select unsatisfied clause  $C_j$  uniformly at random
3: with probability  $q$  do {Random move}
4:   select literal  $l \in C_j$  uniformly at random
5:   return FLIPLIT( $l, x, y$ )
6: else {Greedy move}
7:   for each literal  $l_k \in C_j$  do
8:      $(x^k, y^k) := \text{FLIPLIT}(l_k, x, y)$ 
9:      $U_k := \#$  clauses unsatisfied under  $(x^k, y^k)$ 
10:   $k^* := \arg \min_k \{U_k\}$ 
11:  return  $(x^{k^*}, y^{k^*})$ 
```

Algorithm 6 FLIPLIT

```
1: {Given: clauses  $\{C_i\}$ ; generalized literal  $l$ ;  $(x,y) = (x_1, \dots, x_m; y_1, \dots, y_n)$ }
2: if  $l$  is Boolean then
3:    $j := \text{VARINDEX}(l)$ 
4:    $x'_j := x; x'_j := \bar{x}_j$  {Flip  $x_j$ }
5:   return  $(x',y)$ 
6: else {Integer constraint}
7:    $S := \text{SUPPORT}(l)$  {Indices of variables in constraint}
8:   select  $k \in S$  uniformly at random
9:    $y' := y; y'_k := \text{GIBBS}(x,y,y_k)$ 
10:  return  $(x,y')$ 
```

4.4 Extensions

The presented algorithm can naturally be extended to handle most scenarios that are of practical relevance.

First, the verification of complex protocols often requires sequential constraints, which use an internal testbench state to specify conditions on sequences of stimuli, e.g. using PSL [24] or SystemVerilog [2]. Sequential constraints can be handled in two ways: If the number of testbench states is small (which is usually the case), a separate sampler and mixing queue is instantiated for each state. The sampling is then performed dependent on the testbench state; i.e., the testbench state progresses in lock-step with the state of the DUT. During each iteration, a new input stimulus and testbench state are generated using the sampler of the current state. Alternatively, if the number of testbench states is too large, then the states must be encoded using auxiliary variables which are treated as additional sampling variables. Additional constraints encode the dependencies between the testbench states and thus indirectly the sequential input constraints of the DUT. The entries for the mixing queue are now state-dependent; samples from different states cannot be mixed. If the number of samples stored in the queue for a particular state is too small to guarantee sufficient mixing, temporal mixing might be required; i.e., a sequence of samples must be generated and ignored before one can be applied as to the inputs.

Second, testbenches sometimes use constraints that are conditioned on projected internal states of the DUT. Similar to the case of testbench states, this dependency can be handled explicitly or implicitly, dependent on the number of states.

Another extension of our constraint solver that has not been discussed in this paper is constraint partitioning. As described in previous work [5], the variables and constraints can be partitioned into sets which are sampled separately. These techniques are orthogonal to the constraint solving method, so they can be applied within our

Test case	MBINF					BDD		Boolean CNF	
	Bool vars	int vars	cls	lits	mult	vars	nodes	vars	cls
T1	2	2	11	26	2	34	*	2634	7855
T2	10	4	33	60	0	46	6073	330	961
T3	5	67	146	162	0	539	84530	6640	18347
I1	0	6	30	30	0	116	143848	879	2444
I2	1	9	28	34	6	127	*	29290	87684
I3	0	30	214	455	0	232	*	1080	2954
I4	0	213	497	514	0	244	565977	7428	21827

Table 1: Characteristics of benchmarks: Columns 2–6 describe the mixed Boolean/integer constraints in MBINF. Columns 7–8 describe the BDDs for Boolean versions of the constraints (*=memory blowup). Columns 9–10 describe the Boolean versions in CNF.

approach as well. This provides a major boost in terms of sampler efficiency.

5 Experimental Evaluation of Proposed Sampler

We implemented our sampling algorithm for Boolean and integer variables with multilinear constraints in a random-value generator called Ambigen (A mixed Boolean/integer generator). For comparison, we also implemented the BDD- and DPLL-based sampling algorithms described in Section 2. We selected these two approaches because of their complementary characteristics: The BDD-based method is fast and its distribution is uniform, while the DPLL-based method is more robust. In this section we compare the performance, robustness, and distribution of both methods with our algorithm.

To compare the algorithms, we used each of them to generate solutions to several benchmark constraint sets. We derived a few of our benchmarks (T1–T3) from simple verification scenarios, such as the ALU in Figure 2, and extracted the others (I1–I4) from industrial verification instances. For the BDD- and DPLL-based samplers, we converted the mixed Boolean/integer constraints to purely Boolean form by synthesizing circuits that evaluate the constraints. We used OpenAccess Gear [25] for synthesis.

For the BDD sampler, we built BDDs for the output functions of the constraint evaluation circuits. In the variable order of each BDD, we put the control variables first and interleaved the bits of the data variables, with the least significant bits first. We used CUDD [26] as our BDD package.

For the DPLL sampler, we converted the circuits, with their outputs asserted, to Boolean CNF. To generate each sample, we selected an ordered pre-assignment at random and used its values at decision points. When the solver found a conflict, we allowed it to backtrack and find a solution using its usual decision heuristic. We used MiniSat [27] as our SAT solver.

The characteristics of each version of the benchmarks are given in Table 1. For the benchmarks in MBINF, the table lists the numbers of Boolean variables, integer variables, clauses, literals (both Boolean and integer inequalities), and multiplicative (i.e., nonlinear) inequalities. For the constraint BDDs, the table lists the number of variables and the number of nodes. Each entry with a * indicates that the BDD could not be built with a memory limit of 1 GB. For the constraints in Boolean CNF, the table lists the numbers of variables and clauses.

We used each sampling algorithm to generate 1 million solutions for each benchmark, with a time limit of 10000 seconds. The results of these experiments are shown in Table 2. For Ambigen, the table lists the runtime, the number of unique solutions, and the number of moves per solution (both Metropolis and WALKSAT-style). For the BDD-based sampler, the table lists the total runtime

Test Case	Ambigen			BDD-based		DPLL-based	
	Time (sec)	#uniq	Avg #moves/solution	Time (sec)	Build time	Time (sec)	#samples
T1	35	975627	1.11	*	*	2091	1000000
T2	49	858590	1.56	41	< 1	423	1000000
T3	193	999986	1.05	382	70	10000	16845
I1	112	992391	1.40	111	1	4409	1000000
I2	59	999995	1.00	*	*	10000	252495
I3	179	999979	1.19	*	*	10000	854484
I4	586	920922	1.58	453	20	10000	726645

Table 2: Results of generating $N = 1000000$ solutions to constraints with a time limit of 10,000 sec and a memory limit of 1 GB. The parameters for Ambigen are $M = 1$, $t = 1$, $K = 5$, $R = \infty$, $p, q = 0.5$, $T = 1$.

and the time needed to build the BDD. Entries with *’s in columns 5 and 6 indicate that no solutions were generated because of memory blowup. For the DPLL-based sampler, the table lists the runtime and the total number of samples generated. The number of unique samples for the BDD- and DPLL-based samplers is not reported because almost all the solutions generated were unique; there were duplicate solutions in only three cases, and at most 100 duplicates in each of these.

A comparison of the runtimes shows that Ambigen has similar performance to the BDD-based sampler, and that both of these are faster than the DPLL-based algorithm by 1–3 orders of magnitude. However, Ambigen is more robust than the BDD-based algorithm, because it does not suffer from memory blowup. Note that the memory blowup is not limited to benchmarks with multiplicative constraints. In preliminary experiments, we found this was true even for other variable orders.

Column 4 of Table 2 lists the average number of moves per solution. We computed this value as the ratio of the total number of moves to the total number of solutions, including those that were decimated (thus the denominator is $K \times 10^6$). This value is an important measure of Ambigen’s ability to move through the solution space. For each benchmark, the number of moves per solution is close to 1 (at most 1.58), indicating that the sampler is most often able to find a legal solution in the initial Metropolis move; recovery moves are usually not needed.

The experiments described above are not well suited for comparing the sampling distributions of the algorithms. Each of the benchmarks has a solution space many orders of magnitude larger than the number of samples, so each solution could not be expected to occur more than once in a uniformly generated set of samples. Ambigen’s distribution seems to be skewed because it generates some duplicates. However, some duplication should be expected, given the local-walk nature of the algorithm. This does not contradict the theory on the convergence to the uniform distribution, since the time for convergence can be very long for a large sample space.

Decimation is one of the easiest ways to reduce the number of duplicate samples, i.e. reporting only every K th sample. To investigate the effect of decimation, we increased K from 1 to 10 for the benchmark T2, which had the least unique samples in Table 2. The results are shown in Figure 6. Increasing K from 1 to 5 raises the percentage of unique samples from 32% to 81%, while the runtime increases by 1.5x. Continued increases of K bring diminishing returns—increasing K to 9 yields an additional 9% unique samples at the cost of 2x the original runtime.

Given appropriate choices for the parameters and enough time to converge, the output distribution of our algorithm can be brought arbitrarily close to uniformity. In general, this is not the case for the DPLL-based algorithm. If the number of solutions is not a power

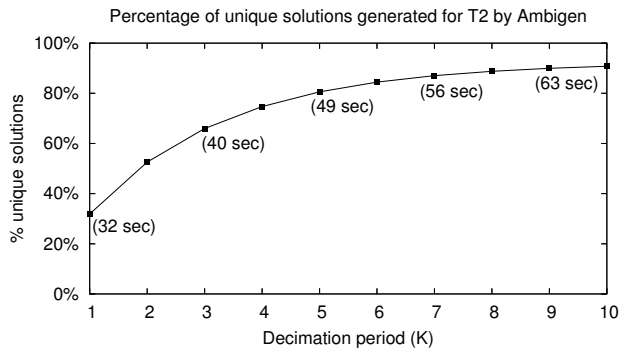


Figure 6: Percentage of unique solutions generated for test case T2 by Ambigen and selected runtimes.

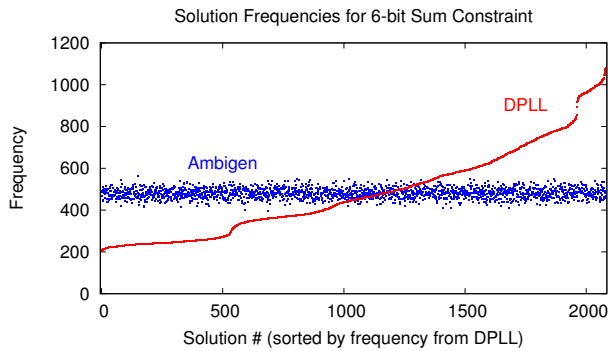


Figure 7: Frequencies of samples from 2-dimensional simplex for Ambigen and DPLL-based sampler.

of 2, the backtracking style of DPLL cannot map the full Boolean space—from which the pre-assignments are uniformly selected—into the solution space in a uniform manner. Depending on the particular constraints, the mapping may not even be close to uniform. For example, we generated solutions to a small set of constraints similar to the example in Figure 3. The resulting distributions of sample frequencies for Ambigen and the DPLL-based sampler is shown in Figure 7.

6 Conclusions

We proposed a normal form, MBINF, for specifying constraints on Boolean and integer variables. It is a generalization of Boolean CNF with integer inequalities as generalized literals. The constraint formulation is expressive enough to cover many practical verification needs and also lends itself to efficient algorithms for sampling. We proposed a new sampling algorithm that combines concepts from the Metropolis-Hastings algorithm, Gibbs sampling, and WALKSAT to efficiently generate solutions to the constraints with an approximately uniform distribution. Our algorithm is significantly faster than a sampler based on Boolean DPLL and substantially more robust than BDD-based sampling. Future work will include adding feedback from coverage analysis to dynamically bias input distributions.

References

- [1] T. Grötter, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Boston: Kluwer Academic, 2002.
- [2] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design: A guide to using SystemVerilog for hardware design and modeling*. Norwell, MA, USA: Kluwer Academic, 2003.

- [3] S. Iman and S. Joshi, *The e Hardware Verification Language*. Norwell, MA, USA: Kluwer Academic, 2004.
- [4] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, “Modeling design constraints and biasing in simulation using BDDs,” in *Digest Tech. Papers IEEE/ACM Int’l Conf. Computer-Aided Design*, pp. 584–589, Nov. 1999.
- [5] J. Yuan, A. Aziz, C. Pixley, and K. Albin, “Simplifying Boolean constraint solving for random simulation-vector generation,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 412–420, Mar. 2004.
- [6] J. H. Kukula and T. R. Shiple, “Building circuits from relations,” in *Proc. 12th Int’l Conf. Computer-Aided Verif. (CAV)*, pp. 113–123, Springer-Verlag, 2000.
- [7] M. A. Iyer, “RACE: A word-level ATPG-based constraints solver system for smart random simulation,” in *IEEE International Test Conference (ITC)*, (Charlotte, NC, United States), pp. 299–308, Sept. 2003.
- [8] M. Davis and H. Putnam, “A computing procedure for quantification theory,” *J. ACM*, vol. 7, pp. 201–215, 1960.
- [9] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem proving,” *Comms. ACM*, vol. 5, pp. 394–397, July 1962.
- [10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient SAT solver,” in *Proc. 38th ACM/IEEE Design Automation Conf.*, pp. 530–535, June 2001.
- [11] W. Wei, J. Erenrich, and B. Selman, “Towards efficient sampling: Exploiting random walk strategies,” in *Proc. Nat’l Conf. Artificial Intelligence*, pp. 670–676, Jul. 2004.
- [12] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, “Equations of state calculations by fast computing machines,” *J. Chem. Phys.*, vol. 21, pp. 1087–1092, June 1953.
- [13] B. Selman, H. A. Kautz, and B. Cohen, “Local search strategies for satisfiability testing,” in *Proc. 2nd DIMACS Challenge on Cliques, Coloring, and Satisfiability* (M. Trick and D. S. Johnson, eds.), (Providence RI), 1993.
- [14] R. Dechter, K. Kask, E. Bin, and R. Emek, “Generating random solutions for constraint satisfaction problems,” in *18th Nat’l Conf. Artificial Intelligence (AAAI02)*, pp. 15–21, July 2002.
- [15] V. Gogate and R. Dechter, “A new algorithm for sampling CSP solutions uniformly at random,” tech. rep., School of Information and Computer Science, University of California, Irvine, May 2006.
- [16] A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal, “AVPGEN—a test generator for architectural verification,” *IEEE Trans. Very Large Scale Integration*, vol. 3, no. 2, pp. 188–200, 1995.
- [17] K. Shimizu and D. L. Dill, “Deriving a simulation input generator and a coverage metric from a formal specification,” in *Proc. 39th Design Automation Conf.*, (New Orleans, LA, United States), pp. 801–806, Jun 2002.
- [18] W. K. Hastings, “Monte Carlo sampling methods using Markov chains and their applications,” *Biometrika*, vol. 57, pp. 97–109, 1970.
- [19] S. Geman and D. Geman, “Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images,” *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 6, pp. 721–741, Nov. 1984.
- [20] A. E. Gelfand and A. F. M. Smith, “Sampling-based approaches to calculating marginal densities,” *J. Amer. Statist. Assoc.*, vol. 85, no. 410, pp. 398–409, 1990.
- [21] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, “Smart simulation using collaborative formal and simulation engines,” in *Digest Tech. Papers IEEE/ACM Int’l Conf. Computer-Aided Design*, pp. 120–126, Nov. 2000.
- [22] S. Shyam and V. Bertacco, “Distance-guided hybrid verification with GUIDO,” in *Design Automation and Test in Europe*, Mar. 2006.
- [23] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. Computers*, vol. 35, pp. 677–691, Aug. 1986.
- [24] “Property specification language, reference manual, version 1.1,” Accellera Organization, June 2004.
- [25] Z. Xiu, D. A. Papa, P. Chong, C. Albrecht, A. Kuehlmann, R. A. Rutenbar, and I. L. Markov, “Early research experience with OpenAccess Gear: An open source development environment for physical design,” in *Proc. ACM Int’l Symp. Phys. Design (ISPD)*, pp. 94–100, Apr. 2005.
- [26] F. Somenzi, *CUDD: CU Decision Diagram Package, Release 2.4.0*. University of Colorado at Boulder, 2005.
- [27] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Proc. 6th Int’l Conf. Theory & Appl. Satisfiability Testing (SAT)*, pp. 502–518, May 2003.