A Markov Chain Monte Carlo Sampler for Mixed Boolean/Integer Constraints

Nathan Kitchen[1] and Andreas Kuehlmann[1,2]

[1] University of California, Berkeley, CA, USA
[2] Cadence Research Labs, Berkeley, CA, USA

Abstract. We describe a Markov chain Monte Carlo (MCMC)-based algorithm for sampling solutions to mixed Boolean/integer constraint problems. The focus of this work differs in two points from traditional SAT Modulo Theory (SMT) solvers, which are aimed at deciding whether a given set of constraints is satisfiable: First, our approach targets constraint problems that have a large solution space and thus are relatively easy to satisfy, and second, it aims at efficiently producing a large number of samples with a given (e.g. uniform) distribution over the solution space. Our work is motivated by the need for such samplers in constrained random simulation for hardware verification, where the set of valid input stimuli is specified by a "testbench" using declarative constraints. MCMC sampling is commonly applied in statistics and numerical computation. We discuss how an MCMC sampler can be adapted for the given application, specifically, how to deal with non-connected solution spaces, efficiently process equality and disequality constraints, handle state-dependent constraints, and avoid correlation of consecutive samples. We present a set of experiments to analyze the performance of the proposed approach.

1 Introduction

Simulating design models for random input sequences and comparing their responses with the expected behavior has become common practice in functional hardware verification. A "testbench" that encapsulates the design model and is executed (or "simulated") with it specifies the valid input sequences and expected design behavior. In their early days, testbenches were written in imperative programming languages such as C or C++, and they specified explicitly the process of generating input stimuli. They supported nondeterminism through the use of random number generators (e.g. `rand()`) for generating data or choosing alternative generation paths. However, with increasingly complex constraints, such an unstructured approach quickly became unworkable. First, in the absence of a concise means to encode complex constraints, the inputs can easily be under- or over-constrained, resulting in spurious verification failures or uncovered input stimuli. Second, the ad-hoc use of random number generators for diversifying the stimuli makes it impossible to ensure a "good" distribution over the solution space. This can dramatically decrease the verification coverage and increase the required lengths of simulation runs.

© Springer-Verlag Berlin Heidelberg 2009

The need for a more concise constraint specification led to the use of declarative languages such as *e* [1] or SystemVerilog [2]. However, since declarative constraints are not directly executable, an online constraint solver is required to generate the stimuli. Since the testbench can have an internal state and the constraints may depend on it or the state of the design itself (e.g., to produce specific data during the different phases of a communication protocol), the solver must run in lockstep with the simulator to generate stimuli that correspond to the current state. There are two key requirements on constraint solvers that are geared for this application domain: First, the solver must be fast to avoid becoming a performance bottleneck; in practice not more than 20-30% of the overall verification runtime should be spent solving the input constraints. Second, the distribution of the generated stimuli must be well defined. In the absence of user-specified biases or knowledge of the design's state space, the generated stimuli should be distributed uniformly over the entire set of solutions.

The focus of this work is on sampling from constraints over Boolean and bounded-domain integers. We assume that the constraint formula is given in conjunctive normal form (CNF) where the clause literals are either Boolean or predicates over integer variables using linear or multilinear arithmetic; as predicates we support equality, inequality, and disequality. We note that, unlike decision procedures for word-level hardware verification, we do not require the support of bit-vector arithmetic. This is because testbench specifications typically do not rely on the side effects of two's-complement encoding or overflow as is often the case in hardware models. Operations on individual bits of the integer variables can be handled by full or partial bit-blasting the corresponding terms.

In this paper, we extend our work on MCMC sampling [3] into a new and more robust approach. After summarizing the overall approach in Section 2, we address specific challenges in Section 3, including a new approach to handle non-connected solution spaces, how to process constraint inputs used for expressing stimuli dependency on the design state, the handling of equality and disequality constraints, and how to avoid correlation between consecutive samples. In Section 4 we provide experiments that evaluate the performance of the proposed sampler on various benchmarks.

2 Preliminaries and Related Work

2.1 Mixed Boolean/Integer Constraints

Let $\hat{x} = (\hat{x}_1, \ldots, \hat{x}_{\hat{m}})$ and $x = (x_1, \ldots, x_m); \hat{x}_i, x_i \in \mathbb{B}$ be Boolean input and output variables, respectively, and $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_{\hat{n}})$ and $y = (y_1, \ldots, y_n); \hat{y}_i, y_i \in \mathbb{Z}$ be integer input and output variables, respectively. The predicate $\varphi(\hat{x}, \hat{y}, x, y)$ specifies the constraints on the outputs (x, y) for given inputs $(\hat{x}, \hat{y})$. In a practical verification setting, φ is given in a declarative form in the testbench; the $\hat{x}$ and $\hat{y}$ are constraint inputs projected from the current state of the design under test and the x and y are the stimuli to be produced for simulating the design. Figure 1 gives a simplified view of such a verification setup.

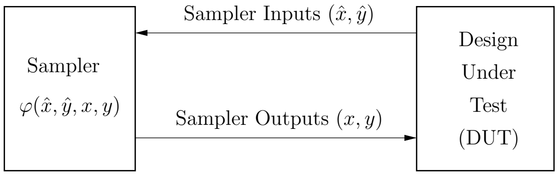
Sampler
$\varphi(\hat{x}, \hat{y}, x, y)$
Sampler Inputs $(\hat{x}, \hat{y})$
Sampler Outputs (x, y)
Design Under Test (DUT)

Fig. 1. Simplified view of a hardware design verification setup

For a given assignment to $(\hat{x}, \hat{y})$, we denote by $\Phi(\hat{x}, \hat{y})$ the set of solutions for φ, i.e., $\Phi(\hat{x}, \hat{y}) = \{(x, y) \mid \varphi(\hat{x}, \hat{y}, x, y) = 1\}$. Let $d(x, y|\hat{x}, \hat{y})$ be the desired distribution function for the samples (x, y) specified in the testbench. When sampling φ for a stream of inputs $((\hat{x}^1, \hat{y}^1), \ldots, (\hat{x}^s, \hat{y}^s))$, we want a sequence of solutions $S = ((x^1, y^1), \ldots, (x^s, y^s))$ with $(x^i, y^i) \in \Phi(\hat{x}^i, \hat{y}^i)$ that obeys this distribution; i.e., each sample in the sequence is chosen independently of the other samples and with probability proportional to its distribution value:

$$\forall (x^i, y^i) : \Pr((x^i, y^i) = (x, y)) = \frac{d(x, y|\hat{x}^i, \hat{y}^i)}{\sum_{(x,y) \in \Phi(\hat{x}^i, \hat{y}^i)} d(x, y|\hat{x}^i, \hat{y}^i)}$$

where $\Pr(A)$ denotes the probability of event A.

For this work, we assume that the constraints are specified as a CNF formula with the following grammar:

$$\begin{aligned}
\textit{formula} &: \textit{clause} \mid \textit{formula} \wedge \textit{clause} \\
\textit{clause} &: \textit{literal} \mid \textit{clause} \vee \textit{literal} \\
\textit{literal} &: \textit{BooleanIdentifier} \mid \neg\textit{BooleanIdentifier} \mid \\
&\quad (\ \textit{expression rel Constant}\ ) \\
\textit{rel} &: \leq \mid \geq \mid = \mid \neq \\
\textit{expression} &: \textit{term} \mid \textit{expression} + \textit{expression} \\
\textit{term} &: \textit{Constant} \mid \textit{term} * \textit{IntegerIdentifier}
\end{aligned}$$

We note that for predicates on integers, strict inequalities $<$ ($>$) can be handled by using non-strict inequalities $\leq$ ($\geq$) and decrementing (incrementing) the constant on the right-hand side by one.

A contemporary approach to solve the constraint satisfaction problem given in the above form would be a SAT Modulo Theory (SMT) solver (see [4,5] for an overview) combining propositional logic with the theory of integers. Applying the latest advances of DPLL SAT solvers, these algorithms enumerate variable assignments over the Boolean space and dynamically instantiate integer constraint problems which are then solved with a specialized theory solver. The problem with using a DPLL-style method for stimulus generation is that it does not produce a good distribution of solutions. Although randomness can be introduced into such an approach (e.g., by modifying the decision heuristic to choose random initial assignments for literals), the distribution is difficult to control.

We refer to [3] for a detailed discussion of a DPLL-style sampler for Boolean constraints, including experiments demonstrating the distribution problems.

2.2 Markov Chain Monte Carlo Methods

MCMC methods are widely used in the statistics, computational physics, and computational biology communities for sampling from complex distributions. Instead of generating each sample independently, these methods take as samples the successive states visited in a simulation of a Markov chain [6]. If the Markov state transitions are set up appropriately (such that all states are always reachable), the distribution of states converges over time to a unique stationary distribution. MCMC methods provide a basis for solving constraints randomly with a well-defined, useful distribution. Although adaptations for practical use can violate the requirements of the MCMC theory and thus distort the distribution, in practice we find that the distortion is usually small.

Metropolis Sampling. One of the most commonly used MCMC algorithms is Metropolis sampling [7]. The Metropolis algorithm implements the Markov state transitions in terms of a target stationary distribution d. It is not necessary for d to be a normalized probability distribution, which is useful in our setting where the number of solutions is generally not known a priori.

In the Metropolis algorithm, a state transition begins with the generation of a proposed new state $\tilde{s}^{(t+1)}$ at random from the current state $s^{(t)}$. The state $\tilde{s}^{(t+1)}$ is then accepted as the new state $s^{(t+1)}$ with probability α:

$$\alpha = \min\left\{1, \frac{d(\tilde{s}^{(t+1)})}{d(s^{(t)})}\right\}$$

If $\tilde{s}^{(t+1)}$ is not accepted, the new state is taken as $s^{(t+1)} = s^{(t)}$.

In order for the state distribution to converge to d, the Metropolis algorithm requires that the proposal distribution be symmetric, i.e., $q(s'|s) = q(s|s')$, where $q(s'|s)$ is the probability of proposing a move to s' given current state s. A generalization of the algorithm known as Metropolis-Hastings [8] relaxes this restriction and uses an adjusted acceptance probability:

$$\alpha = \min\left\{1, \frac{d(\tilde{s}^{(t+1)})}{d(s^{(t)})} \frac{q(s^{(t)}|s^{(t+1)})}{q(s^{(t+1)}|s^{(t)})}\right\}$$

For simplicity in the remainder of this paper we use the name "Metropolis algorithm" to refer to this generalization and omit the "Hastings" qualification.

The particular choice of proposal distribution $q(s'|s)$ does not affect the correctness of the Metropolis algorithm (as long as the reachability requirement is satisfied), but it does determine how fast samples are generated and the rate at which the distribution converges. If the proposed moves are small, then the number of moves needed to cross the sample space is large, and the distribution converges slowly. On the other hand, if large moves are proposed without taking

Algorithm 1. MIXED BOOLEAN/INTEGER SAMPLER

```
1: {Given: formula $\varphi(x, y)$; parameter $p_{ls}$}
2: $(x, y)$ := random assignment
3: loop
4:   $(x, y)$ := METROPOLISMOVE$(x, y)$
5:   while $\neg\varphi(x, y)$ do
6:     with probability $p_{ls}$ do
7:       $(x, y)$ := LOCALSEARCHMOVE$(x, y)$
8:     else
9:       $(x, y)$ := METROPOLISMOVE$(x, y)$
10:  output $(x, y)$
```

into account the target distribution, the probability of rejection can be high, so that moves to new states are infrequent and convergence is still slow.

One proposal method that allows efficient large moves with a rejection rate of zero is Gibbs sampling [9,10]. In this approach, a move consists of changing only a single variable's value, which is sampled from the target distribution, conditioned on the current values of the remaining variables:

$$q(s'|s) = \begin{cases} d(s'_i|s \setminus s_i) & \text{if } s' \setminus s'_i = s \setminus s_i \\ 0 & \text{otherwise} \end{cases}$$

MCMC Sampling with Constraints. One challenge that arises when applying the Metropolis algorithm to sampling with constraints is the requirement that each solution be reachable from every other. In general, moves that can be proposed efficiently, such as Gibbs moves, do not fully connect the solution space. This problem can be addressed by allowing moves to invalid states and using a target distribution $\tilde{d}$ that favors solutions, such as $\tilde{d}(s) = d(s)e^{-U(s)}$ where $U(s)$ is the number of clauses unsatisfied under s.

In theory, this is sufficient for the state distribution to converge. However, in practice it can take too long to reach a solution from an invalid state. In our previous work [3], we strengthened the solving power of an MCMC sampler by interleaving the Metropolis moves with iterations of a local-search solver [11]. This approach was inspired by SAMPLESAT [12], which applies it to purely Boolean constraints. Since the local-search moves do not use the Metropolis acceptance rule, they distort the stationary distribution away from the target distribution; however, the distortion is usually small in practice.

Our approach from [3] for the specific case of uniform sampling with constraints is outlined in Algorithms 1., 2., and 3.. Algorithm 1. is the main sampling loop. Algorithms 2 and 3 are the Metropolis and local-search moves, respectively. Both types of moves use a subroutine PROPOSE(y_k, x, y), which samples a new value for one variable y_k similarly to Gibbs sampling.

PROPOSE constructs proposal distributions as a combination of indicator functions for individual integer relations; the structure of the combination is isomorphic to the structure of the clauses: Let $\{C'_i\}$ be a projection of the clauses $\{C_i\}$

Algorithm 2. MetropolisMove

```
 1: {Given: assignment $(x,y) = (x_1,\dots,x_m; y_1,\dots,y_n)$; temperature $T$}
 2: select variable $v$ from $\{x_1,\dots,x_m,y_1,\dots,y_n\}$ uniformly at random
 3: if $v$ is Boolean $x_k$ then
 4:     $(x',y') := (x_1,\dots,x_{k-1},\neg x_k,x_{k+1}\dots,x_m;y)$
 5:     $Q := 1$
 6: else {$v$ is integer $y_k$}
 7:     $(x',y') :=$Propose$(y_k,x,y)$
 8:     $Q := q(y|y')q(y'|y)^{-1}$
 9: $U :=$ # clauses unsatisfied under $(x,y)$
10: $U' :=$ # clauses unsatisfied under $(x',y')$
11: $p := \min\{1, Q\,e^{-(U'-U)/T}\}$
12: with probability $p$ do
13:     return $(x',y')$
14: else
15:     return $(x,y)$
```

Algorithm 3. LocalSearchMove

```
 1: {Given: formula $\varphi$; assignment $(x,y) = (x_1,\dots,x_m; y_1,\dots,y_n)$}
 2: select unsatisfied clause $C_i \in \varphi$ uniformly at random
 3: for each literal $l_j \in C_i$ do
 4:     if $l_j$ is Boolean $x_k$ or $\neg x_k$ then
 5:         $(x^j,y^j) := (x_1,\dots,x_{k-1},\neg x_k,x_{k+1}\dots,x_m;y)$
 6:     else {$l_j$ is integer relation $R$}
 7:         select $y_k \in$ Support$(R)$ uniformly at random
 8:         $(x^j,y^j) :=$Propose$(y_k,x,y)$
 9:     $U^j :=$ # clauses unsatisfied under $(x^j,y^j)$
10: $j^* := \arg\min_j U^j$
11: return $(x^{j^*},y^{j^*})$
```

onto the current assignments of x and $y \setminus y_k$, so they are constraints on y'_k only. Then the proposal distribution q is

$$q(y'_k|x, y \setminus y_k) = \min_{C'_i} \max_{R_{ij} \in C'_i} d(x, y'_k, y \setminus y_k)\, \sigma_{R_{ij}}(y'_k) \tag{1}$$

where $\sigma_R(y'_k)$ is a "soft" indicator that decays exponentially in the invalid range. The indicator for $R \Leftrightarrow y_k \leq c$ ($y'_k \geq c$), with softness parameter r, is given by:

$$\sigma_R(y'_k) = \begin{cases} 1 & \text{if } y'_k \leq c \quad (y'_k \geq c) \\ e^{-|y'_k - c|/r} & \text{otherwise} \end{cases} \tag{2}$$

The min and max in (1) are isomorphic to $\wedge$ and $\vee$, respectively, in φ. We refer to this type of proposal distribution as *Gibbs-style*, since it resembles a relaxed form of Gibbs sampling; in the limit as $r \to 0$, it *is* Gibbs sampling.

3 Challenges in Practical Verification

3.1 Non-connected Solution Spaces

The solution spaces for constraints used in verification often consist of multiple components that are not connected by single-variable moves; that is, the sampler must change multiple variables in order to move between components. Sometimes the non-connectedness is due to the presence of equality constraints, for which we use special handling as described in Section 3.2. For other cases we use the approach described in Section 2.2: We allow moves to invalid states and use local-search moves to bring the sampler more quickly back to a solution. We refer to the extra moves from invalid states as *recovery moves* (lines 5–9 in Algorithm 1.).

Both the time for recovery and the ability to move effectively between solution-space components are affected by the choice of proposal distribution. In our previous work, we used the Gibbs-style distribution described in Section 2.2. This type of distribution usually works well for quick recovery, since it strongly favors valid states, but it puts low probability on moves through wide gaps to other components.

To move more easily between widely separated components, we propose a new proposal distribution based on the number of unsatisfied clauses rather than the distance from satisfying values. For this we define the *cost* function $U(y'_k)$, which gives for each value of y'_k the number of clauses not satisfied by the assignment $(x, y \setminus y_k, y'_k)$. $U(y'_k)$ is easily computed from the intervals for y'_k that satisfy the integer relations. To construct the proposal distribution $q(y'_k|x, y \setminus y_k)$ we first sample a cost u from the range of $U(y'_k)$ with probability proportional to $e^{-u/T}$, where T is a temperature parameter. Then we sample a value for y'_k from the set $\{\tilde{y}_k : U(\tilde{y}_k) = u\}$.

In our experiments, we found cases where recovery moves using Gibbs-style proposal distributions succeeded while cost-based proposals failed, and vice versa. Therefore, in our new algorithm we use both types, selecting one of them at random in each move, for greater robustness than our previous work.

3.2 Equality Constraints

An equality constraint can be expressed by and handled as a pair of inequality constraints. However, generic MCMC sampling with equality is inefficient because any change of state requires moving through invalid states. On the other hand, equality constraints can be viewed as reducing the dimensionality of the problem, which we exploit in the approach described below.

General Case. For simplicity, we focus the following discussion on the integers y and omit the Boolean variables x and inputs $\hat{x}$ and $\hat{y}$. This does not result in any loss of generality; in fact, the same arguments hold equivalently for the Boolean variables x. For a constraint formula φ, if there exists a variable $y_i \in y$ and a function $f_i(y \setminus y_i)$ such that:

$$\varphi(y) \Rightarrow y_i = f_i(y \setminus y_i)$$

then the dimensionality of the sampling problem can be reduced by 1. That is, we can simply sample the quantified formula $\exists y_i.\varphi$ over the variables $y \setminus y_i$ and determine at each step the value of y_i by evaluating f_i.

Theorem 1. *If $\varphi(y) \Rightarrow y_i = f_i(y \setminus y_i)$ then sampling φ is equivalent to sampling $\exists y_i.\varphi$ for $y^t \setminus y_i^t$ and completing each sample with $y_i^t = f_i(y^t \setminus y_i^t)$.*

Proof. First, we need to show that the solution space has not changed, that is, $(\varphi(y) \Rightarrow y_i = f_i(y \setminus y_i)) \Rightarrow (\varphi(y) \Leftrightarrow (\exists y_i.\varphi(y) \wedge y_i = f_i(y \setminus y_i)))$. Let D_i be the domain of y_i, then $\varphi(y) \Leftrightarrow \bigvee_{\tilde{y}_i \in D_i}(\varphi(y_1, \ldots, \tilde{y}_i, \ldots, y_n) \wedge (\tilde{y}_i = y_i))$ and $\exists y_i.\varphi(y) \Leftrightarrow \bigvee_{\tilde{y}_i \in D_i} \varphi(y_1, \ldots, \tilde{y}_i, \ldots, y_n)$. The condition $\varphi(y) \Rightarrow y_i = f_i(y \setminus y_i)$ implies that only one disjunct for the quantified expression will evaluate to true, specifically the one for $\tilde{y}_i = f_i(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n)$. The required equivalence is established when y_i is restricted to this value by conjoining the quantified expression with $y_i = f_i(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n)$.

Second, we need to show that the distribution has not changed. For this, we notice that each satisfying assignment to the variables $y \setminus y_i$ has a unique value assignment to the y_i. Therefore, $\Pr(y) = \Pr(y \setminus y_i)$. $\square$

This theorem shows that if we can extract dependent variables from φ by either syntactical or functional analysis, we can reduce the dimensionality of the problem and thus increase the efficiency for the solver. This applies for Boolean variables as well as the integer variables. In this context, we handle a specific case of linear equality constraints, which are common in practical applications.

Linear Equality Constraints. In the common case where equality constraints are linear, computing the quantified formula $\exists y_{i_1}, \ldots, y_{i_k}.\varphi(y)$ for dependent variables $y_{i_1}, \ldots, y_{i_k}$ is straightforward because it is equivalent to solving each inequality simultaneously with the equations. However, the set of dependent variables and relevant equations is, in general, not unique. In the following we describe how we select the variables and equations.

Assume that we selected variable y_k for the next Propose step in line 7 of Algorithm 2. or line 8 of Algorithm 3. First we describe the case where each equality constraint E_i with y_k in its support is the only literal in its clause. Each such equation has only one solution for y_k as long as the values of $y \setminus y_k$ are held fixed. In order to generate moves to solutions with other values of y_k, we must allow a simultaneous change to at least one other variable in each E_i. These variables whose values change in the same move are the *dependent* variables.

In the general case, each dependent variable may be constrained by other equations that do not have y_k in their support. An additional dependent variable must be chosen for each of these equations, and so on. Different choices of the initial dependent variables may lead to different sets of equations to solve. In our approach we select the variables randomly; if the resulting system of equations does not have a unique solution we fall back to a single-variable move.

Algorithm 4. outlines our procedure for selecting dependent variables and equations. Note that equations are taken from a priority queue which is ordered

Algorithm 4. Selection of Dependent Variables

```
1: {Given: primary sampling variable $y_k$; equations $\{E_i\}$}
2: $Y := \{\}$ {dependent variables}
3: $F := \{\}$ {dependency equations}
4: $Q := ()$ {priority queue sorted by increasing $|\textsc{Support}(E_i)|$}
5: $y_j := y_k$
6: loop
7:    for $E_i$ such that $y_j \in \textsc{Support}(E_i)$ and $E_i \notin Q$ do
8:       insert $E_i$ in $Q$
9:    if $|Q| = 0$ then
10:      return $(Y, F)$
11:   pop $E_i$ from $Q$
12:   $V := \textsc{Support}(E_i) \setminus \{y_j\} \setminus Y$
13:   if $|V| > 0$ then
14:      select new $y_j$ from $V$ uniformly at random
15:      $Y := Y \cup \{y_j\}$
16:      $F := F \cup \{E_i\}$
```

by increasing size of support (see line 4). This strategy reduces the likelihood that all the variables in an equation's support have already been selected by the time it is processed.

After collecting the dependent variables and dependency equations, we solve these equations simultaneously with each inequality on y_k in order to get transformed bounds for y_k. We combine these bounds to construct a proposal distribution for y_k as described in Section 3.1. After sampling a new value for y_k, we substitute it into the collected equations and solve them to obtain new values for the dependent variables.

Example 1. A two-dimensional problem is established by the following inequalities and equality constraint:

$$\begin{array}{l} (\ y_1 - \ y_2 \geq -2\ ) \wedge \\ (\ y_1 + \ y_2 \leq \ 32\ ) \wedge \\ (\ y_1 + \ y_2 \geq \ 17\ ) \wedge \\ (\ y_1 - 4y_2 \leq \ 1\ ) \wedge \\ (\ 2y_1 - \ y_2 = \ 16\ ) \end{array}$$

Let $s^{(t)} = (13, 10)$ be the current Markov chain state and assume that we select y_1 as the primary sampling variable; y_2 is the only possible dependent variable. Solving the equation with each of the four inequalities gives the following transformed bounds for y_1:

$$y_1 \leq 18 \qquad y_1 \leq 16 \qquad y_1 \geq 11 \qquad y_1 \geq 9$$

leading to the sampling range $11 \leq y_1^{(t+1)} \leq 16$. After sampling $y_1^{(t+1)}$, the value for $y_2^{(t+1)}$ is computed from the equality constraint, i.e., $y_2 = 2y_1 - 16$. Note that if y_2 is selected as the primary sampling variable and an odd value is chosen for it, no integer solution exists for y_1. In this case the nearest integer state is taken. Even though this state is invalid it does not break the algorithm; it will either be rejected in the Metropolis acceptance rule or be handled by recovery moves.

Disjunctions of Equations. When a clause contains an equality constraint and another constraint that shares support with it, the dependencies between variables may not be well defined. For example, suppose that the primary sampling variable is y_1 and one of the clauses is $(y_1 + y_2 = a) \vee (y_1 - y_3 \geq b)$. If $y_1 + y_2 = a$ is enforced, then y_2 is dependent on y_1, but if $y_1 - y_3 \geq b$ is enforced, then this clause requires no dependent variable. The dependency between y_1 and y_2 holds only for possible values of y_1 that do not satisfy $y_1 - y_3 \geq b$.

We handle this case by distributing such clauses over the rest of the formula, thus creating a disjunction of formulas where the dependencies in each sub-formula are well defined, e.g.:

$$(E_1 \vee E_2) \wedge (I_1 \vee I_2) \quad \Rightarrow \quad (E_1 \wedge (I_1 \vee I_2)) \vee (E_2 \wedge (I_1 \vee I_2))$$

for equations E_1, E_2 and inequalities I_1, I_2. We construct the proposal distributions for the disjuncts by solving each inequality simultaneously with the equations in its disjunct (e.g., I_1 with E_1, etc.) and combining the transformed inequalities as described in Section 2.2 or Section 3.1. For Gibbs-style proposals we take the pointwise maximum of the distributions for the sub-formulas as the overall distribution for the primary sampling variable. For cost-based proposals we take the pointwise minimum of the cost functions for the sub-formulas and construct the distribution from this composite cost function as in Section 3.1.

The number of sub-formulas may be exponential in the number of clauses, so we impose a bound on the number of them that we use. If the total number exceeds the bound, we select a random subset to use.

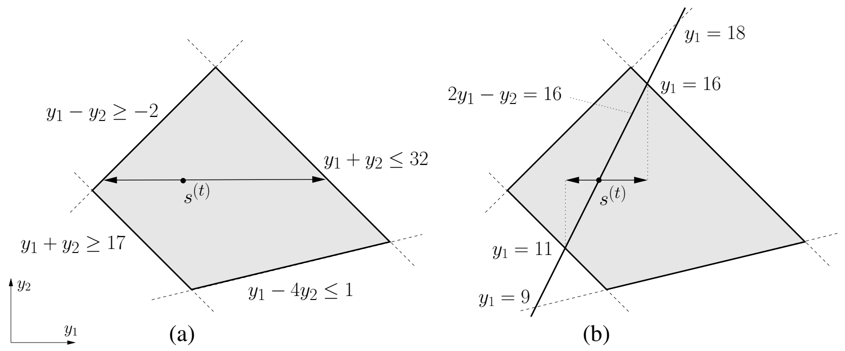

Fig. 2. Illustration of handling equality constraints for Example 1: (a) sampling range for y_1 to produce state $s^{(t+1)}$ without equality constraint, (b) sampling range with equality constraint

3.3 Disequality Constraints

A disequality constraint $g(y) \neq 0$ can be translated into a pair of inequalities $g(y) < 0 \vee g(y) > 0$. However, when a disequality appears in a clause with an equation, this transformation forces distribution of the disjunction as described in the previous section, creating more instances for the equation solver. To avoid this, we ignore disequality constraints while constructing proposal distributions. If the proposed next state violates the disequality, the violation is counted in the evaluation for the Metropolis acceptance rule and the move may be rejected.

3.4 Input-Dependent Constraints

As described in Section 2.1, constraints may be dependent on input variables $\hat{x}$, $\hat{y}$ coming from the state of the design under test. In practice, state-dependency is applied to select between different sets of constraints. When input values change, the current assignment may be far from a solution. In our MCMC framework, we utilize the the recovery moves to guide the search back into valid space.

Our practical observation is that state-dependency does not trigger a large amount of jumping between disjoint solution spaces. However, in theory it can cause distorted distributions. When the inputs move frequently between values with very different solution sets, the distribution can be distorted. The solutions that are closest (in number of moves) to the previous solution set will be sampled more often. This is because we do not use the Hastings rule in recovery moves.

3.5 Serial Correlation

A fundamental property of Markov-chain-based sampling is the high correlation of consecutive states. Since for verification purposes consecutive stimuli must not be correlated, the generated state sequence cannot be applied directly as stimuli. Instead, we apply a *mixing pool* to cache the generated states and randomly shuffle them before the next move. The mixing pool is implemented as a simple array of memory and shuffling is obtained inexpensively by a memory read and write. In addition to mixing, subsampling can be applied to de-correlate samples. For this, a fixed number of Markov chain state transitions are executed at each sampling step. Note that when accessing a random element of the mixing pool its state may have been for different inputs. Similar to directly processing inputs as described in Section 3.4, this case is handled by recovery moves.

3.6 Overall Algorithm

Algorithm 5. shows the complete flow for the proposed sampling approach. It includes the initialization of the mixing pool (lines 4–6), input processing (line 8), actual sampling (lines 10–12), and mixing (lines 9 and 13).

Algorithm 5. MCMC Stimulus Generator

```
 1: {Given: formula $\varphi(x, y)$; parameters $p_{ls}$, $K$; mixing pool $(s_P^1, \ldots, s_P^M)$}
 2: $(x, y)$ := random assignment
 3: $(\hat{x}, \hat{y})$ := random assignment
 4: for $i := 1$ to $M$ do {fill mixing pool}
 5:    $(x, y) :=$ MetropolisMove$(x, y)$
 6:    $s_P^i := (x, y)$
 7: loop {generate samples}
 8:    $(\hat{x}, \hat{y}) :=$ ReadInputs()
 9:    select $i \in \{1, \ldots, M\}$ uniformly at random
10:    for $k := 1$ to $K$ do {subsample}
11:       $(x, y) :=$ MetropolisMove$(s_P^i)$
12:       recovery: see lines 5–9 in Algorithm 1.
13:    $s_P^i := (x, y)$
14:    output $(x, y)$
```

4 Experimental Results

We have implemented our MCMC-based stimulus generation algorithm. In this section we analyze its performance on benchmarks designed to illustrate practical challenges arising in verification.

As we stated in Section 3.1, the Gibbs-style proposal distributions that we proposed in [3] are not effective for moving between disconnected components of solution spaces that are separated by wide multidimensional gaps. For this reason we introduced a new type of proposal distribution based on the number of unsatisfied clauses (the cost). To test the effectiveness of the cost-based distributions in moving between disconnected solution regions, we generated two groups of constraint sets. The solution sets for the benchmarks in the first group consist of two squares of width 8 separated by distance D: $0 \leq y_1, y_2 < 16 + D$, $(y_1 < 8 \wedge y_2 < 8) \vee (y_1 \geq 8 + D \wedge\ y_2 \geq 8 + D)$. We generated 100 000 solutions for each of $D = 0, 1, 2, 4, 8, \ldots, 2^{14}$ and for each type of proposal distribution. We disabled local-search moves to avoid distortion of the basic Metropolis distribution. Figure 3(a) shows the relative frequencies of transitions between the regions that we observed, i.e., the frequencies with which consecutive solutions were from different regions. The transition frequencies are constant with increasing distance for cost-based proposal distributions and decrease quickly to zero for Gibbs-style proposals. A combination of both proposal types gives intermediate transition frequencies.

The second group of benchmarks has solution regions at the corners of n-dimensional simplices for $n = 2, 3, 4, 5, 10, 15, \ldots, 100$, with 100 instances for each value of n. The constraints have the form $0 \leq a_i y_i \leq b$, $\sum_i a_i y_i \leq b$, $\bigvee_i (y_i \geq c_i)$. We generated the benchmarks randomly, constraining the parameters $\{a_i\}, \{c_i\}$ so that the solution regions would not be connected. For each benchmark we sampled 100 000 solutions for each type of proposal distribution. Figure 3(b) shows the relative frequencies of transitions between regions. The

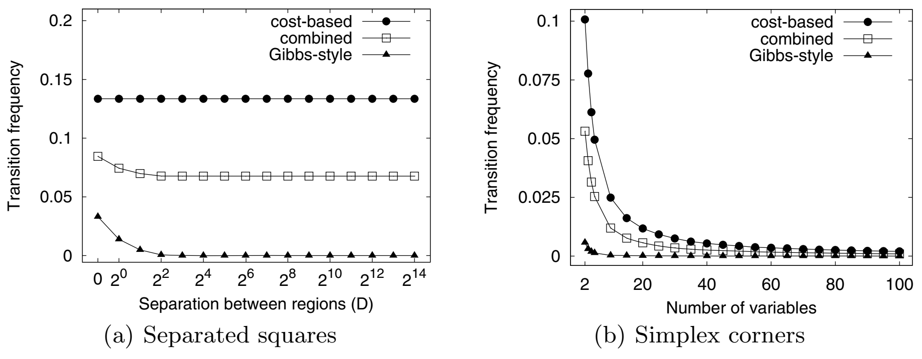

Fig. 3. Frequencies of transitions between regions for different types of proposal distributions. Parameters: $T = 1$, $r = 1$, $K = 1$, $M = 1$; for (a) $p_{ls} = 0$; for (b) $p_{ls} = 0.5$.

sampler moved between regions much more often when using cost-based proposal distributions than when using Gibbs-style proposals. Even for the cost-based proposals, the transition frequency decreases rapidly with the number of variables; this fits with the fact that the average probability of an invalid value decreases exponentially as the average cost increases linearly.

The results from these first benchmarks demonstrate that cost-based proposal distributions are more effective for moving between disconnected regions than Gibbs-style proposals, especially when the regions are widely separated.

To test the effectiveness of the mixing pool at reducing serial correlation, we generated benchmarks with solution spaces consisting of a narrow space between oblique hyperplanes: $-c \leq y_i \leq c$, $-b \leq \sum_i a_i y_i \leq b$, $b = \frac{c}{10}$. We generated 100 instances with each of 4, 16, and 64 variables and sampled 10 000 solutions for each instance with mixing pool sizes $1, 2, 4, \ldots, 256$. Figure 4(a) shows the autocorrelation of the solutions, computed as an average of the autocorrelation

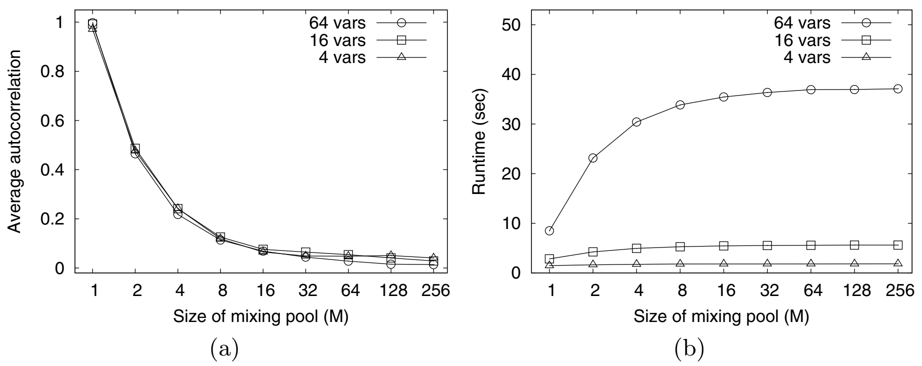

Fig. 4. Effect of mixing pool on (a) average autocorrelation and (b) average runtime in seconds. Parameters: $p_{ls} = 0.5$, $T = 1$, $r = 1$, $K = 1$.

for each variable. The correlation decreases quickly as the size of the mixing pool increases, showing the effectiveness of our mixing approach.

Figure 4(b) shows the average runtimes of the instances for each mixing pool size M. The scaling of runtime with the number of variables for $M = 1$ is expected, since each benchmark has constraints with all the variables in their support and these constraints are evaluated for every move. However, the trend of the runtimes for the 64-variable benchmarks requires further explanation. The rapid rise for small values of M and the disproportionately long runtimes in general are likely due to cache misses. This suggests a strategy for improving runtime: Subsample with a longer period and use a smaller mixing pool. Subsampling on the same position in the pool gives higher cache hit rates than mixing but reduces serial correlation similarly.

5 Other Work

There are several works in the area of stimulus generation for constrained random simulation. The authors of [13] use BDDs to representing the constraints and then apply a random walk from the root to the 1-leaf for producing samples. Although this approach can guarantee a uniform or other desired distribution, BDDs often blow up for practical problems. For example, many testbenches use multiplication operations on data variables, which BDDs cannot handle for larger bitwidths. In general, all bit-level sampling algorithm, including the work presented in [14] and SampleSat [12], require to bit-blast the constraints and thus lose the word-level structure and its information for efficient sampling.

Word-level samplers take advantage of the higher-level structure of the given formulas to quickly produce stimuli. Interval-propagation-based sampling [15] combines random variable assignment, iterative interval refinement, and backtracking on mixed bit- and word-level constraints. Although it is efficient for most practical constraints, the interval propagation scheme produces stimuli with skewed distributions, which in the worst case, can have exponential error. A number of other, specialized stimulus generators [16,17,18] utilize specific domain knowledge for constraint specification and solving.

The work closest to ours is presented in [19]. Similar to our approach, the authors propose the use of an MCMC sampler to generate stimuli for software verification. The sampler is placed in a feedback loop to learn a good biasing from the observed design coverage. In contrast to our work, this paper applies a bare-bones Metropolis-Hasting sampler and does not address the specific challenges present in large-scale practical verification as discussed here. The work in [20] discusses sampling from disconnected spaces; however, it focuses on the confidence level of the resulting distribution and is mainly of theoretical interest.

6 Conclusions

We presented an MCMC-based sampler for generating stimuli for hardware verification, although the same approach can also be applied for verifying software.

The focus of this work was on making an MCMC approach work for large-scale, practical applications. Although MCMC methods provide a solid theoretical foundation for sampling with a desired distribution, a naive implementation would not perform well and would be impractical. We showed how an MCMC-based approach can be adapted to deal with non-connected solution spaces, efficiently process equality and disequality constraints, handle state-dependent constraints, and avoid correlation of consecutive samples. Our future work is focused on a concurrent design of an MCMC sampler that exploits the probabilistic nature of the approach to maximally benefit from parallel execution.

References

1. Iman, S., Joshi, S.: The e Hardware Verification Language. Kluwer Academic, Norwell (2004)
2. Sutherland, S., Davidmann, S., Flake, P.: SystemVerilog for Design: A guide to using SystemVerilog for hardware design and modeling. Kluwer Academic Publisher, Norwell (2003)
3. Kitchen, N., Kuehlmann, A.: Stimulus generation for constrained random simulation. In: IEEE/ACM Int'l Conf. on CAD, pp. 258–265 (November 2007)
4. Sebastiani, R.: Lazy satisfiability modulo theories. Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 3, 141–224 (2007)
5. Kroening, D., Strichman, O.: Decision Procedures. Springer, Heidelberg
6. Brémaud, P.: Markov Chains: Gibbs fields, Monte Carlo Simulation, and Queues. Texts in Applied Mathematics, vol. 31. Springer, New York (1999)
7. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equations of state calculations by fast computing machines. J. Chem. Phys. 21, 1087–1092 (1953)
8. Hastings, W.K.: Monte Carlo sampling methods using Markov chains and their applications. Biometrika 57, 97–109 (1970)
9. Geman, S., Geman, D.: Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. IEEE Trans. Pattern Analysis and Machine Intelligence 6(6), 721–741 (1984)
10. Gelfand, A.E., Smith, A.F.M.: Sampling-based approaches to calculating marginal densities. J. Amer. Statist. Assoc. 85(410), 398–409 (1990)
11. Selman, B., Kautz, H.A., Cohen, B.: Local search strategies for satisfiability testing. In: Trick, M., Johnson, D.S. (eds.) Proc. 2nd DIMACS Challenge on Cliques, Providence RI (1993)
12. Wei, W., Erenrich, J., Selman, B.: Towards efficient sampling: Exploiting random walk strategies. In: Proc. Nat'l Conf. Artificial Intelligence, pp. 670–676 (July 2004)
13. Yuan, J., Shultz, K., Pixley, C., Miller, H., Aziz, A.: Modeling design constraints and biasing in simulation using BDDs. In: Digest Tech. Papers IEEE/ACM Int'l Conf. Computer-Aided Design, pp. 584–589 (November 1999)
14. Kim, H., Jin, H., Ravi, K., Spacek, P., Pierce, J., Kurshan, B., Somenzi, F.: Application of formal word-level analysis to constrained random simulation. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 487–490. Springer, Heidelberg (2008)
15. Iyer, M.A.: 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 (September 2003)

16. Chandra, A., Iyengar, V., Jameson, D., Jawalekar, R., Nair, I., Rosen, B., Mullen, M., Yoon, J., Armoni, R., Geist, D., Wolfsthal, Y.: AVPGEN— a test generator for architectural verification. IEEE Trans. Very Large Scale Int. 3(2), 188–200 (1995)
17. Shimizu, K., Dill, D.L.: 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 (June 2002)
18. Mihail, M., Papadimitriou, C.H.: On the random walk method for protocol testing. In: Computer Aided Verification. LNCS, pp. 132–141. Springer, Heidelberg (1994)
19. Sankaranarayanan, S., Chang, R.M., Jiang, G., Ivancic, F.: State space exploration using feedback constraint generation and Monte-Carlo sampling. In: Proc. ESEC-FSE 2007: – 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 321–330. ACM, New York (2007)
20. Bandyopadhyay, A., Aldous, D.J.: How to combine fast heuristic Markov chain Monte Carlo with slow exact sampling. Electronic Communications in Probability 6, 79–89 (2001)