A Framework for Constrained Functional Verification

Jun Yuan
Verplex Systems
Milpitas, CA 95035
jyuan@verplex.com

Carl Pixley
Synopsys
Hillsboro, OR 97124
cpixley@synopsys.com

Adnan Aziz
University of Texas at Austin
Austin, TX 78712
adnan@ece.utexas.edu

Ken Albin
Motorola Inc.
Austin, TX 78729
ken.albin@motorola.com

Abstract

We describe a framework for constrained simulation-vector generation in an industry setting. The framework consists of two key components: the constraint compiler and the vector generator. The constraint compiler employs various techniques, including prioritization, partitioning, extraction, and decomposition, to minimize the internal representation of the constraints, and thus the complexity of constraint solving. The vector generator then uses the compiled data together with input biasing to generate random simulation vectors. Constraints and input biases are treated in a unified manner in the vector generator. Although there are many alternative ways of generating vectors from constraints, the framework uniquely suits a practical constrained verification environment because of its ability to handle complicated constraints and its seamless treatment of constraints and biases. We illustrate the effectiveness of the framework with real examples from commercial designs.

1 Introduction

Constraint-based verification is the idea of defining an environment for the Design Under Verification (DUV) by using constraints. These constraints can take several forms such as Boolean formulas whose variables reference inputs and state bits in the design or in auxiliary finite state machines, or in the form of temporal logic expressions. An environment is often called a testbench or bus functional model in conventional simulation. It is used to inject inputs into a design possibly reacting to the design's outputs or to monitor the outputs of the design. An environment is also necessary for a formal analysis of the design. Several commercial tools such as Vera and Verisity use constraints to help define a testbench [11]. In addition, Yuan *et al.* [17], Kukula and Shiple [9] and Shimizu and Dill [13] have presented algorithms to implement constraints as stimulus generators for simulation.

Constraints are declarative in nature thus are easier to develop and maintain than traditional testbenches. Another key advantage of using constraints to model environments is its generation/monitor duality [10]. This duality means that the very same syntax can be used to monitor the interaction between designs and to inject or "drive" inputs to a design fragment. Therefore, the generators for a DUV can be "flipped" to become monitors when a DUV is integrated with its true environment. In contrast, in conventional simulation testbenches, stimulus generators specific to a DUV are discarded when the DUV is integrated with the chip environment.

In our experience, sometimes hundreds of constraints are used to model the environment of a commercial DUV. This requires that the stimulus generator be able to handle high complexity. In addition, so as not to inordinately slow down simulation, the generator must solve the constraints every clock cycle very quickly, depending upon the value of the state-holding variables sampled from the DUV.

In addition to satisfying certain constraints, a controllable input distribution is also desirable in functional verification. For example, the reset signal of a design should not be asserted too often, and the condition of a critical branching in a state machine should have about the same probability being true and false so that both branches are explored sufficiently in random simulation.

This paper describes a framework, called Simgen, for constrained random simulation verification. Simgen has been used in an industry setting for years. The underlying algorithm generates vectors according to a certain distribution defined by constraints and input biases together. Both constraints and biases are dynamic in the sense they can be dependent on the state of the DUV. The framework also allows multiple clocks and constraints specific to a particular clock domain. Prioritization is used to order the solving of constraints so that one constraint can be solved before another even if the two are interdependent. There are also various techniques available in the framework for constraint simplification.

Constraint solving is a core problem in functional verification. Besides the well-known satisfiability (SAT) approach in formal verification, there have been many various attempts in modeling design environment using constraint solving. For example, the *cofactor* based *functional vector* generation [5], using circuits to represent constraints [9], and constraint synthesis [16]. These approaches have one thing in common - they all represent the solution space of the constraints using the range of a set of functions, which are realizable in logic gates. Therefore, unlike Simgen, they are directly applicable in verification methods such as emulation and formal verification where synthesizable a design environment is required. However, they also lack the ability of controlling the distribution of generated vectors, which is important for random simulation. The vector generation method in [13] is similar to Simgen but with one major difference: the constraints are conjoined *during* instead of *before* simulation. The advantage of their approach is obvious: after the state information is resolved in the constraints, conjoining them becomes easier. However, the drawback is that the conjoining would have to be done in every clock cycle, which can potentially slow down the simulation. We feel that the ideal is a "smart" combination of the two that maximizes the simulation throughput while maintaining the ability to handle complicated constraints. This can be a future direction.

The vector generation algorithm and hold-constraint extraction based simplification in Simgen were previously presented in [17] and [15], respectively. The contribution of this paper is the introduction of several new techniques and a thorough discussion of how the new and existing techniques form a constrained random simulation framework in an industry setting. The new techniques include graph based decomposition of constraints, vector generation from a functional decomposition tree, and multiple clock domains.

The remainder of this paper is organized as follows: Section 2 gives an overview of the Simgen flow. Section 3 defines the constraints and biases discussed in this paper. The next two sections describes the two key components of Simgen, namely vector generation and constraint compilation. Section 6 briefly discusses the impact of multiple clock domains. In addition to the experiment results (all using real designs) shown in their pertaining subsections, two separate simulation experiments are presented in Section 7. The paper is summarized in Section 8.

2 Overview

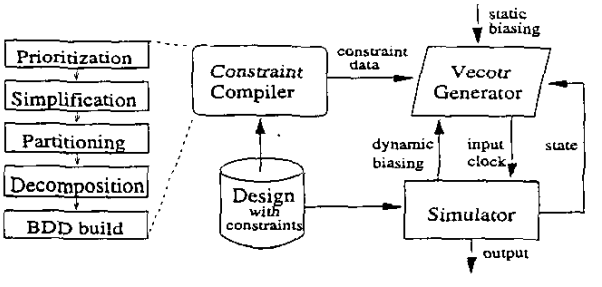

Figure 1: Simgen flow

The flow of constrained random simulation in Simgen is shown in Figure 1. The input to Simgen consists of the design files, constraints and input biases. Constraints and biases are given in a verilog-like language, as we will describe later. The constraints are first compiled into a database. The generator engine operates on this database to

ICCAD'03, November 11-13, 2003, San Jose, California, USA.

provide inputs to the simulator. At each simulation cycle, the generator engine first samples the current state and determines the dynamic biases, then it uses the state and bias information and the constraint database to generate a vector that is allowed by the constraints under the sampled state, and according to a distribution determined by the constraints and the biases. Simulators that Simgen interfaces to include Verilog-XL, VCS and a proprietary simulator.

The compilation stage are divided into several steps. Constraints are first levelized according to their priority levels assigned by the user. A simplification is then applied to extract special constraints that can be solved independently and prior to the original constraints. These derived constraints are also used to simplify the original constraints. This simplification may help refine the constraint partitioning that happens next – a process that breaks up the constraints into groups. The constraints inside each group are then subjected to structural and functional decomposition for further simplification.

In the sequel, we describe how the generator engine works and provide details on the constraint compilation part.

3 Constraints and Biases

In Simgen, constraints are (Boolean) functions defined over design signals. Since all these signals are determined by the design state and input, constraints are essentially functions of state and input variables. As an example, a typical assumption about bus interfaces " the 'transaction start' input (*ts*) is asserted only if the design is in the 'address idle' state" can be captured in the following constraint:

$$\$constraint(\text{ts} \rightarrow (\text{addr\_state} == \text{ADDR\_IDLE})).$$

Constraints depending on state information are general enough to describe a rich class of scenarios. For example, the temporal behavior of inputs can be modeled by constraints that use auxiliary variables to remember past states.

An input bias is a function of state variables that evaluates to $(0,1)$. For example, the following statement assigns the bias (towards 1) of input *freeze_in* to 0.9 when *addr_state* is idle and 0.5 other wise.

$$\$setbias(\text{freeze\_in}, \text{addr\_state} == \text{ADDR\_IDLE} \ ? \ 0.9, 0.5)$$

4 Vector Generation

Constraints are represented in BDDs and conjoined for vector generation. The generator engine is an $O(n)$ algorithm where n is the size of the conjunct BDD.[1] The algorithm consists of two phases: first, a bottom-up computation in the BDD is performed to determine the *weight* of each node, wherein the weights of the ONE and ZERO nodes are defined as 1 and 0, respectively; second, if the weight of the root node is 0, then we have encountered a deadend state for which there is no valid input (debugging of the constraints or design ensues); otherwise, an input vector is generated in a top-down random walk in the BDD according to *branching probabilities* derived from the weights. The weight computation is done using the recursion:

$$w(i) = p(x) \cdot w(t) + (1 - p(x)) \cdot w(e) \qquad (1)$$

where i is a BDD node, x the variable associated with i, and t and e the *then* and *else* child of i, respectively. $p(x)$ returns the input bias if x is an input variable, otherwise it returns the value of x under the current state. The random walk procedure that generates an input vector is shown in Figure 2. Note the branching probability for node i is computed as:

$$i.t\_branch = p \cdot t / w(i) \qquad (2)$$

For input variables not visited, their input biases are used in assigning the value. The probability of each vector so generated is the product of the branching probabilities (for nodes where *then* was taken), or their 1's complement (for nodes where *else* was taken), or the input biases (if the variable was not visited). We call this product the *constrained probability* of the vector. Constrained probabilities are insensitive to BDD variable ordering. Further, the vector generation algorithm has the following properties:

1. Only legal input vectors are generated, and

[1]Generating a satisfying vector from a BDD in general is linear in the number of variables, but this is not true in the case when state variables are involved: they are determined by the design and may cause backtracking

```
Walk(i) {
    if (i == ONE)  return;
    if (v is a state variable) {
        if (v==1) Walk(t);
        else Walk(e);
    } else {
        i.t_prob = p · t.weight / i.weight;
        let r = random(0,1);
        if (r < i.t_prob) {
            assign v to 1;
            Walk(t);
        } else {
            assign v to 0;
            Walk(e);
        }
    }
}
```

Figure 2: Random walk for vector generation.

2. Every legal input vector has a constrained possibility of greater than 0.

5 Constraint Compilation

5.1 Constraint Prioritization

Constraint prioritization stems from a frequently encountered situation where some constraints assume higher priority over the others because the inputs in the former have to be decided first. For example, in a bus protocol, one usually needs to choose a transaction type first, then the attributes associated with it. This is impossible using conjunctive constraints but can be achieved by constraint prioritization.

Given a set of constraints C, a constraint prioritization is a partition of C with ordered components $\{C_1, \ldots, C_n\}$, where C_1 has the highest priority. This partition in turn defines a partition $\{X_1, \ldots, X_n\}$ of input variables X as follows: let $sup(C_i)$ be the input support of C_i, then $X_1 = sup(C_1)$, and $X_i = sup(C_i) - \cap_{j=1}^{i-1} X_j$ for $1 < i \leq n$. X_i defines the set of input variables to be solved at priority level i: the set of input variables in C_i that have not been solved in a higher level.

In vector generation, the algorithm given in Section 4 is first applied to C_1, then to C_2 conditioned upon the assignment to X_1, then to C_3 upon the assignments to X_1 and X_2, and so on. Obviously, the distribution of vectors so generated is determined by the product of the constrained probabilities of the assignments to $X_1, \ldots, X_n$.

Constraint prioritization not only is necessary for enhanced expressiveness of constraints, but also helps to divide complicated constraint system into manageable blocks which is beneficial to both constraint solving and debugging. We observe wide-spread use of constraint prioritization among Simgen users; many sampled designs involves more than 3 levels of constraints, some involve up to 10 levels.

5.2 Constraint Partitioning

Constraints in each priority level can be partitioned into groups of disjoint input variable support and each group solved separately. The partitioning is done as follows:

1. for each input variable, create a group
2. for each constraint depending on a variable, add the constraint to the variable's group
3. merge all groups that share a common constraint until each constraint appears in at most one group

Solving constraints of a priority level can now be parallelized by applying the vector generation algorithm to the groups in the partition simultaneously. The soundness of this approach is guaranteed by

the fact that the probability of generating a vector with and without the partitioning remains the same.

We also observe considerable partitioning of constraints and reduction in BDD size and build time in sampled designs, as shown in Table 1. We ignored the prioritization to show just the effect of partitioning. To save space, the data without partitioning is not shown since the BDD build was much more expensive, and in many cases timed out.

design	#cons	#part	size	time
mmq	117	26	24.54	17.0
qbc	93	34	2.69	2.5
qpag	215	29	142.9	272.4
qpcu	109	11	4.56	1.0
rio	198	66	375.72	3.3
sbs	108	40	2.94	1.5
smi	74	25	9.67	4.3
sdca	135	32	58.37	180.5
max	506	69	26.34	90.5

Table 1: Effect of constraint partitioning

5.3 Constraint Extraction and Simplification

The disjoint-input-support partitioning can be refined by inferring from the original constraints a special type of constraints called the *hold-constraints*. This dichotomy comes from a frequently encountered scenario in synchronous designs – under certain condition, an input variable maintains its value from the previous clock cycle. In our case, this definition is generalized to include all constraints in which the input variables, under certain design states, depend only on state variables. For example, the constraint example given in Section 3 is a hold-constraint. A hold-constraint on an input variable x can be written in the normal form

$$k \rightarrow (x = g) \tag{3}$$

where k and g are both functions of state variables. Hold-constraints have the following advantages: (1) they can be solved separately from and before the original constraints; (2) they can be used to simplify other constraints, and the simplified constraints usually contain fewer input variables which can lead to a finer partition.

To show how hold-constraints can be inferred, we denote *Boolean differential* of constraint f with respect to variable x by $\partial f/\partial x$, computed as $f_x \cdot \overline{f_{\bar{x}}} + \overline{f_x} \cdot f_{\bar{x}}$. Then for a given constraint f and an input variable x in f, a hold-constraint regarding x exists iff

$$k = \frac{\partial(\exists_x f)}{\partial x} \neq 0. \tag{4}$$

And if this is the case, the extracted hold-constraint is

$$k \rightarrow x = [g^{on}, \overline{g^{off}}] \tag{5}$$

where

$$g^{on} = k \cdot (\exists_x f)_x,\ g^{off} = k \cdot (\exists_x f)_{\bar{x}}. \tag{6}$$

This extraction is applied to every constraint on every input variables in that constraint. Then for each extracted hold-constraint on $e := k \rightarrow (x = g)$ (if there is any), each constraint f that contains x is replaced by the *conditional substitution*

$$\tau(f, e) = k \cdot f_{x:=g} + \bar{k} \cdot f$$

where $f_{x:=g}$ is the substitution of variable x with function g.

The above extraction and simplification is repeated until there is no more hold-constraint to be extracted. Table 2 shows the hold-constraint extraction based simplification.

5.4 Tree-decomposition

The partitioning resulted from the hold-constraint simplification can still be refined further using *Tree-decomposition* (TD) [12]. TD is a simplification technique extensively studied in Constraint Satisfaction problems (CSPs) (e.g., [7]) and the similar conjunctive query problem in relational database (e.g., [8]). Definition of TD varies with specific applications. We use the following definition aiming to minimize the number of variables occurring in a BDD.

design	#parts	size	time
mmq	58	9.53	6.2
qbc	92	1.29	0.0
qpag	216	20.61	12.3
qpcu	136	0.80	0.0
rio	140	143.0	1.5
sbs	503	1.61	0.1
smi	66	3.02	1.9
sdca	107	11.22	41.8
max	111	23.67	172.3

Table 2: Hold-constraint extraction based simplification.

Let $C = \{f_1, \ldots, f_n\}$ be the set of constraints, X the set of input variables in C, and $var(f_i) \subseteq X$ the set of input variables in f_i. A TD of C is a tree $T = (I, E)$, with a set $var(i) \in X$ labeling each vertex $i \in I$, such that

1. For each constraint $f_i \in C$, there is a vertex $i \in I$ such that $var(f_i) \subseteq var(i)$
2. For each variable $v \in X$, the set of vertices $\{i \in I \mid v \in var(i)\}$ induces a (connected) tree

The width of a TD is the maximum size of $var(i)$ over all $i \in I$. Finding the TD with the smallest width among all TDs of a constraint system is expense thus we use the linear time *triangulation* algorithm that heuristically search for a TD with small width [14]. Once a tree is built, a root selected, and the nodes made *arc-consistent* [1], each node can then be solved separately.

The effect of TD can be seen in Table 3, where the "time" and "size" columns are for the BDD build time (in seconds) and final BDD size (in 1000 nodes), respectively. The last column gives the ratio of the tree width to the total number of variables, indicating the reduction of variable count in BDDs after TD.

design	without TD		with TD		
design	*time*	*size*	*time*	*size*	*ratio*
qpag	103.1	200.3	0.09	2.65	0.25
sqbc	0.01	6.13	0.00	2.08	0.40
qpcu	0.02	3.07	0.00	0.79	0.65
mmq	14.16	6.60	0.16	1.33	0.30
rio	0.97	142.7	1.53	133.9	0.66
ccu	t/o	-	74.1	4486.2	0.50

Table 3: Effect of tree-decomposition.

5.5 Functional Decomposition

The Simgen framework has provided several methods described previously for simplifying constraint solving. In extreme cases, we need even heavier preprocessing to make vector generation sufficiently fast for simulation. One possibility is to try to break the large constraints using *disjunctive functional decomposition*, wherein the subfunctions have disjoint support. In Section 4, we have shown a vector generation algorithm operating on BDDs. In fact, the algorithm can be extended to any *read-once* decision graph, wherein a variable is visited at most once on all decision making paths. BDDs and disjunctive functional decomposition trees are such examples.

It is shown in [6] that *fully sensitive* functions, including all Boolean functions, have a finest disjunctive decomposition. Although finding such a decomposition is expensive, there are efficient methods (e.g., [4, 3, 2]) for obtaining coarser decompositions.[2] For a decomposition tree of a constraint $f(X)$, we build a set of BDDs for the tree starting from the leaves, inserting a cut-point if the size of the BDD under construction reaches a predefined threshold. The resulting BDDs, in the set $G = \{g_0, g_1, \ldots, g_n\}$, are functions of variables X and the set of cut-points $Y = \{y_1, \ldots, y_n\}$, where g_0 is the

[2] A more conservative approach is to transform the circuitry behind the constraint into a tree by collapsing the *fanout-reconvergent regions*.

biasing	idle states	active states	total (sec)
none	2977	14	6.8
static	2179	811	6.2
dynamic	1073	1918	6.3

Table 4: Results of Biasing.

setting	overall (sec)	SimGen overhead
random	44.9	-
SimGen	48.2	21.3%
w. dump	63.6	16.0%
w. monitor	635.6	1.7%

Table 5: Overhead of SimGen.

BDD at the root of the tree, and g_i ($1 \leq i \leq n$) the BDD for cut-point y_i.

To apply the previous vector generation algorithm to this decomposition tree, the weight computation in Equation (1) needs to be augmented by adding to the definition of $p(x)$ the case "if x is a cut-point corresponding to BDD g_x, then $p(x)$ returns the weight of the root of g_x." Also, the random walk used to generate vectors needs to be augmented as follows: if the node being visited is associated with a cut-point c_i, then after assigning c_i according to the branching probability, the traversal recurs into the corresponding BDD g_i before it explores the selected branch in the current BDD. The random walk in g_i is done similarly but with an important difference: if c_i was assigned to 0, then each node in g_i will have its branching probability replaced by its complement (of 1).

6 Implication of Multiple Clocks

System components often operate under their local clock. Simgen handles multiple clock domains by introducing clock constraints for clock generation and clock qualifiers that govern normal (input) constraints. Clock constraints often involves clock generation circuitry, e.g., state machines, but they differ from the traditional explicit generation method by being able to introduce randomness in the interrelationship of clocks. This is an important requirement for verification in multiple clock domains. Clock-qualified constraints are natural consequence of a multiple-clock design since each constraint should only be sampled by the clock governing the involved signals. The new constraints are handled using the same core procedure but the sequence of sampling state and solving constraints will be different, as shown in Figure 3.

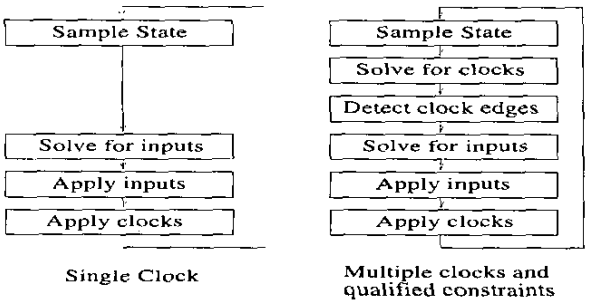

Figure 3: Events in each simulation cycle.

7 More Results

We use two sets of experiment results to illustrate the effectiveness of input biasing and the low overhead incurred by vector generation in Simgen. In the first experiment, we simulated a design for multiple 1000-cycle runs and measured how many times the 3 state machines in the design are outside of their respective idle states. The simulation was performed under three modes: (1) with no input bias (i.e., all biases in Simgen are set to 0.5), (2) with static bias, and (3) with dynamic bias. The biases in (2) and (3) were obtained after a quick study of the state machines. Table 4 shows the improvement from (1) to (2) and to (3).

In the second experiment, we measured the overhead incurred by vector generation in Simgen. Simulations usually run with monitoring processes, or dumping the design state for post-processing. Table 5 summarizes the run time overhead of SimGen on the design used in the first experiment, using dynamic biasing. All simulations ran for 10000 cycles each. Row 2 to 5 respectively represent the simulations with pure random generation, stand alone SimGen, SimGen with Verilog dump, and with property monitoring. In realistic simulation settings (dumping or monitoring), the overhead of SimGen is fairly low.

8 Summary

We have described a framework for constrained simulation-vector generation in an industry setting. The contribution of Simgen is manifold: it provides a nice vector generation engine that unifies input constraint and biasing, and it employs novel techniques to preprocessing constraints so that the complexity of constraint solving is minimized. The strength and robustness of Simgen has been witnessed by its years' of practical usage in an industry verification flow and its consistent capability to discover design bugs that were categorized by engineers as "very hard or impossible to find using other verification methods."

References

[1] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.

[2] T. Bengtsson, A. Martinelli, and E. Dubrova. A Fast Heuristic Algorithm for Disjoint Decomposition of Boolean Functions. *Proc. Intl. Workshop on Logic Synthesis*, pages 51–55, 2002.

[3] V. Bertacco and M. Damiani. The Disjunctive Decomposition of Logic Functions. *Proc. Intl. Conf. on Computer-Aided Design*, pages 78–82, 1997.

[4] D. Bochmann, F. Dresig, and B. Steinbach. A New Decomposition Method for Multilevel Circuit Design. *Proc. European Design Automation Conf.*, pages 374–377, 1991.

[5] O. Coudert and J. C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 126–129, November 1990.

[6] E. V. Dubrova, J. C. Muzio, and B. von Stengel. Finding Composition Trees for Multiple-valued Functions. *Proceedings of the 27th International Symposium on Multiple-Valued Logic (ISMVL '97)*, pages 19–26, 1997.

[7] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural csp decomposition methods. *IJCAI*, 1999.

[8] M. Gyssens, P. Jeavons, and D. Cohen. Decomposing Constraint Satisfaction Problems Using Database Techniques. *Artificial Intelligence*, pages 57–89, 1994.

[9] J.H. Kukula and T.R. Shiple. Building circuits from relations. *Proc. of the Computer Aided Verification Conf.*, 2000.

[10] C. Pixley. Integrating Model Checking Into the Semiconductor Design Flow. *Computer Design's Electronic Systems journal*, pages 67–74, March 1999.

[11] S. Regimbal, J-F. Lemire, Y. Savaria, G. Bois, E-M. Aboulhamid, and A. Baron. Applying aspect-oriented programming to hardware verification with e. *Proceedings of HDLCON*, 2002.

[12] N. Robertson and P. D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, pages 7:309–322, 1986.

[13] Kanna Shimizu and David Dill. Deriving a simulation input generator and a coverage metric from a formal specification. *Proc. of the Design Automation Conf.*, pages 801–806, June 2002.

[14] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

[15] J. Yuan, A. Aziz, K. Albin, and C. Pixley. Simplifying boolean constraint solving for random simulation-vector generation. *Proc. Intl. Conf. on Computer-Aided Design*, pages 123–127, 2002.

[16] J. Yuan, A. Aziz, K. Albin, and C. Pixley. Constraint synthesis for environment modeling in functional verification. *Proc. of the Design Automation Conf.*, 2003.

[17] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling Design Constraints and Biasing in Simulation Using BDDs. *Proc. Intl. Conf. on Computer-Aided Design*, pages 584–589, 1999.