Modeling Design Constraints and Biasing in Simulation Using BDDs

Jun Yuan Kurt Shultz Carl Pixley
Motorola Inc.
5918 W. Courtyard Dr.
Austin, TX 78730
{yuan,shultz,pixley}@adttx.sps.mot.com

Hillel Miller
Motorola Inc.
1 Shenkar Str. E.
Herzelia 46120 Israel
hillelm@msil.sps.mot.com

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

Abstract

Constraining and *input biasing* are frequently used techniques in functional verification methodologies based on randomized simulation generation. Constraints confine the simulation to a legal input space, while input biasing, which can be considered as a probabilistic constraint, makes it easier to cover interesting "corner" cases. In this paper, we propose to use constraints and biasing to form a simulation environment instead of using an explicit testbench in hierarchical functional verification. Both constraints and input biasing can depend on the state of the design and thus are very expressive in modeling the environment. We present a novel method that unifies the handling of constraints and biasing via the use of Binary Decision Diagrams (BDDs). The distribution of input vectors under the effect of constraints and input biasing are determined by what we refer to as the *constrained probabilities*. A BDD representing the constraints is first built, then an algorithm is applied to bias the *branching probabilities* in the BDD. During simulation, this annotated BDD is used to generate input vectors whose distribution match their predetermined constrained probabilities. The simulation generation is a one-pass process, i.e., no backtracking or retry is needed. Also, we describe a partitioning method to minimize the size of BDDs used in simulation generation. Our techniques were used in the verification of a set of commercial designs; experimental results demonstrated their effectiveness.

1 Introduction

A typical integrated circuit interacts with its environment. During simulation-based functional verification, the environment is modeled by a testbench. In this paper we provide an alternative approach to environment modeling — we introduce a tool, SimGen, and an associated methodology which employs user-specified *constraints* to model the interaction between the design and its environment. Constraints are Boolean formulas involving the design signals. Note that constraints can depend on state variables. So input constraints can change depending upon the current state of the design.

As an example, we have employed the following constraint in the verification of a bus interface unit:

```
st != 2'b11 ? IN_a == st_PREV_a :
 (!IN_b & !IN_c) ? IN_u == st_PREV_u : TRUE;
```

It implies that when `st` is `11`, and inputs `IN_b, IN_c` are both low, then the input `IN_u` should be equal to the value of the state variable `st_PREV_u`.

Constraints are specified in a declarative manner, as opposed to the imperative approach of testbenches, and thus need less effort on the part of the user. This is especially helpful in a prototyping stage when all that is known about the environment are some abstract specifications in the architecture book. Furthermore, constraints form a modular and more formal interface documentation about design blocks; they automatically convert to properties to be monitored at a higher level of hierarchy. By contrast, testbench modules constitute an unmaintainable and unverifiable documentation of the environment.

SimGen also supports user-specified *dynamic biasing* on the inputs, i.e., assignment of probabilities to individual input bits depending on the current state. Dynamic biasing can facilitate discovering difficult "corner" cases. Given a current state in the design, and a set of constraints and biases, SimGen automatically generates random inputs according to a probability distribution which satisfies the constraints and respects the biases. If there exists an input satisfying the constraints, SimGen will generate it directly, without backtracking. When no input exists, SimGen flags an error and halts. The following construct illustrates dynamic biasing:

```
setbias1(in7, addr_state == IDLE ? 0.9, 0.5);
```

This sets the probability of the input `in7` having value 1 to 0.9 when `addr_state` is `IDLE`, and to 0.5 otherwise.

Our contribution is that we handle constraints and biasing in a unified way via the use of Binary Decision Diagrams [1] (BDDs). We pay special attention to handle the BDDs in an efficient manner, to avoid the BDD size explosion, which is a common problem in BDD-based formal verification [2]. We stress that we employ BDDs only for representation of constraints and biasing. In symbolic model checking (SMC), a complete traversal of the state space is performed using a BDD representation of the next state logic, and the reached state set [3, 4]. Thus our procedure is not as susceptible to the BDD explosion problem as SMC.

Related work in random test generation includes the following: Freeman *et al* [5] present a static-biased random test generation

0-7803-5832-5/99/ $10.00 © 1999 IEEE

technique in a tool called RIS. RTPG in [6] and AVPGEN in [7, 8] implements dynamic-biased instruction generation and utilizes constraint solving techniques tailored for specific instructions. A problem is that one may need to backtrack and perform heuristic search to resolve the "dead end" cases.

Binary graphs were used by Blum *et al* [9] to probabilistically check equivalence of Boolean functions. In their work, a probability distribution of the output of a function is computed recursively from the input probabilities. A similar approach was adopted in [10] to compute exact fault detection probabilities. We use an algorithm similar in spirit to the above, though for a different purpose, namely, the computation of biases under input constraints.

The rest of this paper is organized as follows. In Section 2 we describe a mechanism for simultaneously considering environment constraints and input biasing. Section 3 contains details of the vector generation procedures in SimGen. Experiments and a case study are discussed in Section 4. We summarize and point out future research topics in Section 5.

2 Constraints and Input Biasing

A constraint is a Boolean formula involving any signals occurring in a design, including inputs to the design. Involvement of design signals other than the inputs makes constraints state-dependent. Furthermore, auxiliary memory-elements (registers) can be instantiated in the design to remember the "past state" so that constraints referring to these registers can define inputs based on the history of the design.

The Boolean formulas of constraints are represented by BDDs defined over input and state variables. In the sequel, the *constraint BDD* refers to the conjunction of the BDDs of all constraints, unless otherwise stated. Observe that for any given state of the design, the *legal input space* is simply the cofactor of the constraint BDD with respect to that state. If the set is empty, we define the design to be in an *illegal* state. We call these *deadend* states because the simulator cannot proceed if it is in this state.

A *biasing on an input variable* is specified by a function mapping the state space to real number in $[0,1]$. If the function is a constant, the biasing is said to be *static*; otherwise, it is *dynamic*. We denote an input u_i's biasing towards the value 1 and 0 by $p(u_i = 1)$ and $p(u_i = 0)$, respectively.[1]

Given an input vector $\vec{\alpha}$, the term $\prod_{\alpha_i \in \vec{\alpha}} p(u_i = \alpha_i)$ will be referred to as the "*weight of the input vector* $\vec{\alpha}$", and denoted by $\pi(\vec{\alpha})$. Given a set of constraints and input biases, and a state of the design, the distribution of an input vector can be defined as the *constrained probability* as follows:

Definition 1 Let $\{u_0, ..., u_{n-1}\}$ be the design inputs, and U_s be the legal input space under state s. The constrained probability of an input vector $\vec{\alpha} = \langle \alpha_0, \ldots, \alpha_{n-1} \rangle$, at the state s, is 0 if $\vec{\alpha} \notin U_s$. Otherwise, the constrained probability is

$$\frac{\pi(\vec{\alpha})}{\sum_{\vec{\beta} \in U_s} \pi(\vec{\beta})}$$

Conceptually, the constrained probability of an input vector is the weight of that vector divided by the sum of the weights of all vectors that satisfy the constraint. With our definition, an input vector is *legal* if and only if its constrained probability is greater than zero. Also, the constrained probabilities are closely correlated to input biasing as we will show in Section 3.

3 Vector Generation

In this section, we develop the p-tree algorithm. This takes a state, a constraint BDD, and a set of biases, and then employs two procedures, namely *Weight* and *Walk*, to generate a random input vector according to the distribution given by the constrained probabilities at that state. *Weight* labels BDD nodes with *branching probabilities*; *Walk* traverses the BDD according to the branching probabilities. The resulting path represents a vector whose distribution matches its *constrained probability* as given by Definition 1. Since the p-tree algorithm is called at each simulation cycle to generate an input, it is imperative that *Weight* and *Walk* be fast. Both theory and experimental results show this is the case.

The Branching Probabilities

Generating vectors from a constraint BDD is analogous to evaluating the BDD for an input vector and a state, except that in the former we need to probabilistically select an input value, rather than looking up its value when deciding what branch to take. In the following, we define the "*branching probabilities*" upon which we base our selections.

First, we define what we mean by the *weight* of a BDD node for a particular state. The weight of the constant *ONE* node is 1, and the weight of the *ZERO* node is 0. The weight of a nonleaf node, σ_j, corresponding to variable v_i, given the weight of its *then* node, t_j, and the weight of its *else* node, e_j, is given by the following:

$$w_j = \begin{cases} p(v_i = 1) \cdot t_j + p(v_i = 0) \cdot e_j & \text{if } v_i \text{ is an input variable} \\ t_j, & \text{if } v_i \text{ is a state variable, and } v_i = 1 \\ e_j, & \text{if } v_i \text{ is a state variable, and } v_i = 0 \end{cases}$$

Conceptually, the weight of a node is the sum of weights of vector suffixes represented by the set of paths from that node to *ONE*. By induction on the length of the paths, the weight of the root node is $\sum_{\vec{\beta} \in U_s} \pi(\vec{\beta})$, where U_s is the set of vectors satisfying the constraints.

The *then* and *else branching probabilities* of σ_j are defined to be $[p(v_i = 1) \cdot t_j]/w_j$, and $[p(v_i = 0) \cdot e_j]/w_j$, respectively. Note that they add up to 1.

[1] Note that the biasing function is restricted to biasing on individual input variables, rather than cubes or more general subsets of the Boolean space of inputs. We discuss this issue at the end of the paper.

```
Weight(node,cur_st) {
  if (node == ONE) return 1;
  if (node == ZERO) return 0;
  if (node is visited) return node.wgt;
  set_visited(node);

  if (node.var is a state variable) {
    if(cur_st[node.var] == 1)
      node.wgt = Weight(node.then,cur_st);
    else
      node.wgt = Weight(node.else,cur_st);
    return node.wgt;
  } else {
    t = Weight(node.then,cur_st);
    e = Weight(node.else,cur_st);
  }

  node.wgt =
  p(node.var=1,cur_st) * t + p(node.var=0,cur_st) * e;
  node.then_branch =
  p(node.var=1,cur_st) * t / node.wgt;
  node.else_branch = 1 - node.then_branch;
  return node.wgt;
}
```

Figure 1: Computing branching probabilities.

The *Weight* Procedure

Weight computes node weights and branching probabilities bottom-up in the constraint BDD, as shown in Figure 1. We use the following notations: *node.var* is the variable associated with a BDD *node*; *node.then* and *node.else* are the two child nodes (branches) of *node*, taken when *node.var* is assigned to 1 and 0, respectively; *p(var=0/1,cur_st)* returns the input biasing of *var* under the current state *cur_st*. In the implementation, an input biasing, being a Verilog expression, can be evaluated by the simulator dynamically.

A straight-forward upper bound on the time complexity of *Weight* is $O(n)$, where n is the number of nodes in the constraint BDD. Note, however, that the procedure traverses only a subset of the BDD nodes, dependent on the current state. As a result, in practice, even when the constraint BDD is quite large, *Weight* is fairly efficient.

The *Walk* Procedure

A vector is generated by the procedure *Walk* in a top-down traversal of the constraint BDD as follows: at a state node, take the *then* (resp. *else*) branch if the corresponding state variable is assigned to 1 (resp. 0) in the current state; at an input node, take a branch according to its *branching probabilities*, and set the value of the corresponding input variable accordingly.

Since on any path in a BDD, a variable can be visited at most once, the procedure *Walk* is guaranteed to terminate within m steps where m is the number of input and state variables in the constraint BDD. At end of the traversal, we must be in one of the following situations:

$cmd[3:0]$	weight of vector	const. prob.
1000	$1/2 \cdot 2/3 \cdot 3/4 \cdot 4/5 = 24/120$	24/50
0100	$1/2 \cdot 1/3 \cdot 3/4 \cdot 4/5 = 12/120$	12/50
0010	$1/2 \cdot 2/3 \cdot 1/4 \cdot 4/5 = 8/120$	8/50
0001	$1/2 \cdot 2/3 \cdot 3/4 \cdot 1/5 = 6/120$	6/50

Table 1: Computing constrained probabilities.

1. We are at the *ZERO* node, which indicates an illegal state — exit the simulation.
2. We are at the *ONE* node. We just generated a legal input vector.

An input that is not visited at the end of the above traversal is randomly assigned according to its biasing. The default bias, 0.5, is used for an input if its biasing is not given by the user.

Correctness and Properties

Recall that our goal was to generate input vectors according to their constrained probabilities. The following theorem (the proof of which is omitted for brevity) demonstrates that the p-tree algorithm achieves this goal:

Theorem 1 At a given legal state, p-tree will generate a random input vector with a probability equal to its constrained probability.

The p-tree algorithm enjoys the following properties.

Lemma 1 For a given state, the probability of generating an input vector $\vec{\alpha}$ in which u_i equals 1 monotonically increases as $p(u_i = 1)$ increases.

Lemma 2 The probability of generating an input vector is independent of the input and state variable ordering of the constraint BDD.

An Example of the p-tree Algorithm

The constraint below specifies that when the state "reset" is 0, exactly one bit of the 4-bit input cmd must be 1.

```
!reset->((cmd[3:0] == 4'b1000)||
         (cmd[3:0] == 4'b0100)||
         (cmd[3:0] == 4'b0010)||
         (cmd[3:0] == 4'b0001)));
```

Assume that under any state in which "reset" is 0, the user-specified input biases evaluate to: $p(cmd[3] = 1) = \frac{1}{2}$, $p(cmd[2] = 1) = \frac{1}{3}$, $p(cmd[1] = 1) = \frac{1}{4}$, $p(cmd[0] = 1) = \frac{1}{5}$. By Definition 1, we compute the values for the constrained probabilities, as illustrated in Table 1.

Now we show how the *p-tree* algorithm generates vectors matching the distribution in Table 1. The constraint BDD is shown in Figure 2. Each node is labeled with its weight under

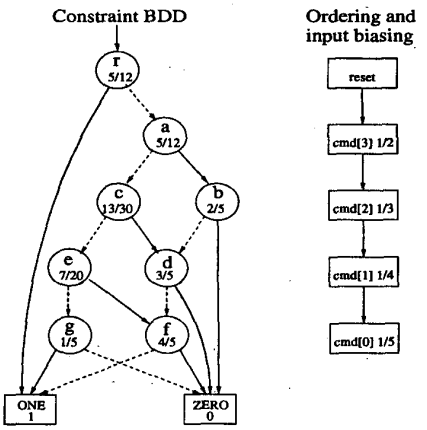

Figure 2: A constraint BDD labeled with node weight.

the state "reset"=0. Solid and dashed arcs represent the *then* and *else* branches, respectively. The variable ordering and input biasing are shown to the right of the BDD, with each variable lined up with its BDD nodes.

To illustrate the procedure *Weight*, consider node *f*:

$$\begin{aligned} weight(f) &= p(f.var=1) * 1 + p(f.var=0) * 0 \\ &= 4/5 * 1 + 1/5 * 0 = 4/5 \end{aligned}$$

Similarly, $weight(g) = 1/5$. Now we compute $weight(e)$:

$$\begin{aligned} weight(e) &= p(e.var=1) * weight(f) + p(e.var=0) * weight(g) \\ &= 1/4 * 4/5 + 3/4 * 1/5 = 7/20 \end{aligned}$$

Consider the input vector $\langle 0,1,0,0 \rangle$. The probability that *Walk* chooses this vector is the product of branching probabilities along the path $\{a, c, d, f, ONE\}$:

$$\frac{(1-1/2)\cdot 13/30}{5/12} \cdot \frac{1/3\cdot 3/5}{13/30} \cdot \frac{(1-1/4)\cdot 4/5}{3/5} \cdot \frac{(1-1/5)\cdot 1}{4/5} = 6/25$$

As expected (cf. Theorem 1) this is exactly equal to the constrained probability we calculated for vector $\langle 0,1,0,0 \rangle$ in Table 1.

Finally, recall that only a subset of the nodes in the constraint BDD is visited in *Weight* — in this particular example, if the state was "reset"=1, then *Weight* would only visit the nodes *r* and *ONE*.

Constraint Partitioning

The time complexity of *Weight* is closely linked to the size of the constraint BDD. Recall that the constraint BDD is the conjunction of all the BDDs corresponding to the user-specified constraints. Intuitively, when we are given constraints which have no input variables in common, we should be able compute an input vector by computing the component inputs with the p-tree algorithm on each constraint separately, and then concatenating the results to form a global input. For brevity, we omit a formal proof of the fact that the vectors generated in this manner have the same probability distribution as those generated when applying p-tree to the monolithic constraint BDD. We partition the set of BDDs corresponding to constraints into sets with disjoint input variable support as follows:

1. for each input variable u_i, create a partition;
2. for each constraint c_j, if c_j depends on u_i, put c_j in u_i's partition;
3. merge all partitions that share a constraint until each constraint appears in at most one partition.

It is possible that the conjunction of all the constraints is empty while none of the constraint BDDs of the partitions are empty. However, if for a given state, *s*, each element of the partition allows a partial input vector, then there is an input vector for the conjunction of the constraints. So if the conjunction of constraints is vacuous this will become apparent immediately after the vector generation starts — for some element of the partition the *Walk* procedure will generate no vector and exit, since there is no *legal* state, as described in Section 3.

Example	|*Vars*|	|*Constraints*|	|*Cons Vars*|
block-1	76	13	26
block-2	178	10	59
block-3	1437	11	153
block-4	446	33	175
block-5	407	34	297
block-6	185	107	156
block-7	1396	171	677

Table 2: Design & constraint statistics.

4 Experimental Results

SimGen has been developed at Motorola. This has given us access to a suite of meaningful examples. We present experimental results on seven real designs. All experiments were conducted on a 233 MHz UltraSPARC-60 machine with 512 MB main memory. Simulation was performed using Verilog-XL.

Constraint BDDs

Dynamic variable reordering was enabled in all experiments. Table 2 reports the statistics of the designs. The column labeled *vars* denotes the total number of inputs and latches in the design; Column 3 gives the number of constraints, and Column 4, the total number of input and state variables which appear in the constraints.

The result of building constraint BDDs without partitioning is shown in Table 3. Note that *block-4*, *block-5*, and *block-7* each have close to 100,000 nodes in the final constraint BDDs.

Table 4 shows the effectiveness of using partitioning. Column 5 lists the total number of constraints, and the number of partitions formed. Partitioning gives modest improvement to BDD size for *block-1, block-2, block-3* and *block-6*; it dramatically reduces both time and space complexity for the larger designs, namely *block-4, block-5* and *block-7*. The complexity of the designs and constraints, together with the size of the constraint BDDs demonstrate that our technique is feasible for medium or even large designs.

Example	*time*	*peak*	*final*
block-1	0.0	6312	54
block-2	5.0	5110	119
block-3	26.0	6132	774
block-4	885.5	303534	110858
block-5	727.7	181243	82405
block-6	8.3	19418	4094
block-7	365.0	218708	98658

Table 3: Building the constraint BDD without partitioning.

Example	*t(sec)*	*peak*	*final*	*part*
block-1	0.0	1022	43	13/5
block-2	4.0	5110	103	10/7
block-3	20.3	6132	609	11/9
block-4	38.0	13286	1595	33/10
block-5	33.4	22484	1962	34/9
block-6	2.0	10220	2722	107/16
block-7	53.0	33726	12535	171/18

Table 4: Building the constraint BDDs with partitioning.

A Case Study

We discuss in detail the experiment with *block-5*. This is a bus interface unit. Conceptually, it consists of three FSMs, namely the master, data, and address units, together with considerable glue logic. Our goal was to fully "exercise" the states in these FSMs. Specifically, the individual units issue Read/Write requests under various conditions, and we wanted to generate as many such requests under as many different conditions as possible.

It took about two person-days to write the constraints according to the (English language) specification for the block and its interface. The end result was a concise specification of the environment in a 200-line Verilog file consisting of 34 constraints.

The benefit of constraining the input space cannot be overemphasized. Unconstrained random simulations invariably produced false negatives when checking properties in simulation. We observed that in unconstrained simulations X values were constantly generated on tri-state buses, indicating bus contentions, which made the simulations meaningless.

Developing input biasing was straightforward. For instance, we wanted to limit the frequency of external errors when testing the essential functionality of a design. This was expressed as `setbias1(error, 0.2);`. There were also cases where we needed to employ dynamic biasing. For example, even after we statically biased over a dozen critical input signals, the three FSMs stayed largely idle (that is, they did not generate Read-Write transactions) through simulation runs. After studying the design for about an hour, we were able to find a set of dynamic input biases that stimulated many more Read-Write transactions.

Table 5 shows the effect of biasing on the number of Read-Write transactions made by the master, data, and address units over the course of simulating an input sequence of length 1000. The dynamically biased simulation resulted in 130 times as many transactions as the unbiased simulation.

biasing	*R/W requests*	*t (sec)*
none	14	6.8
static	811	6.2
dynamic	1918	6.3

Table 5: Effectiveness of biasing.

setting	*Total time (sec)*	*SimGen overhead*
random	44.9	12.0%
SimGen	48.2	21.3%
SimGen with dump	63.6	16.0%
SimGen with monitors	635.6	1.7%

Table 6: Overhead of SimGen.

Of course, the true test of a verification tool is not simply that it excites more behavior — it is the number of bugs found. In this specific example, SimGen, together with a simulation monitoring tool, discovered 30 design bugs. These were found not only by assertion failures in the HDL, but also by the design entering states where the set of legal inputs was empty.

To conclude the case study, we conducted an experiment on the run time overhead of SimGen on *block-5* with partitioned constraint BDDs and dynamic biasing. The result is reported in Table 6. All simulations consist of the application of 10000 inputs. Row 1 shows that even with a pure random generation, there is a certain amount of overhead associated with the interaction between the generator and the simulator. Row 2 shows the result of a SimGen-driven simulation with no other activities. Row 3 presents the case where the SimGen-driven simulation dumps signal values to a file for post-processing. In the final case, a set of multi-cycle properties are monitored during the simulation. It can be seen that the overhead of SimGen is fairly low.

5 Conclusion and Future Work

We have described a dynamically-constrained and dynamically-biased random simulation generation method, its algorithm, implementation and application to commercial designs.

An area of future work is to provide a feedback mechanism between simulation coverage and SimGen to provide some quantitative guidance so that SimGen can readjust the input biasing dynamically to focus on as yet unexplored design behaviors on-the-fly.

In its current form, the mechanism for specifying probabilistic biases on the input space is quite simple — biases are restricted to individual bits. A natural extension of this would be biases on subsets of the entire Boolean space of inputs. For example, one might want to write a bias to the effect that the probability of an input op-code being an ADD is 0.9.

Probabilistically generating input vectors which simultaneously satisfy a set of biases on subsets of the input space will be difficult, since the biases may be mutually inconsistent. Observe that, in general, checking for consistency is at least as hard as checking whether the intersection of a collection of subsets

$\{B_1, B_2, \ldots, B_n\}$ of the input space is empty: simply assign biases of 1 to each set — the intersection is nonempty iff a probability distribution exists on the input space simultaneously satisfying the constraints. When the subsets $\{B_1, B_2, \ldots, B_n\}$ are represented as BDDs, the problem of checking if their intersection is empty is NP-hard; this follows directly from the NP-hardness of 3-CNF-SAT.

In practice, we would expect that the subsets $\{B_1, B_2, \ldots, B_n\}$ would be very "simple". For example, the biases may be only on input cubes, e.g., the probability of $u_0u_1 = 00$ is 0.9 and the probability of $u_1u_2 = 01$ is 0.2. In this case one might hope that an efficient algorithm exists for probabilistically generating inputs according to the biases. However, Koller and Megiddo [11] show that even when the cubes contain no more than two literals, finding a distribution satisfying the biases is NP-hard.

Nevertheless, we have continued working on heuristics for the problem of probabilistic generation of inputs satisfying biases on subsets of the input space given as BDDs, because we view this as an important problem, with implications towards using SimGen as the underlying engine for an instruction level test generation tool.

Acknowledgements

The authors would like to thank Tom Shiple (Synopsys) for finding several mistakes in an earlier version of this paper, and Sanjeev Arora (Princeton) for suggesting we look at [11].

References

[1] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[2] R. Ranjan, J. Sanghavi, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. High Performance BDD Package Based on Exploiting Memory Hierarchy. In *Proc. of the Design Automation Conf.*, Las Vegas, NV, June 1996.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[4] C. Pixley. A Computational Theory and Implementation of Sequential Hardware Equivalence. In E. M. Clarke and R. P. Kurshan, editors, *Proc. of the Workshop on Computer-Aided Verification*, pages 293–320, June 1990.

[5] J. Freeman, R. Duerden, C. Taylor, and M. Miller. The 68060 microprocessor functional design and verification methodology. In *On-Chip Systems Design Conference*, pages 10.1–10.14, 1995.

[6] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd. Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-random Test Program Generator. *IBM Systems Journal*, 30(4):527–538, July 1991.

[7] A. K. Chandra and V. S. Iyengar. Constraint Solving for Test Case Generation - A Technique for High Level Design Verification. In *Proc. Intl. Conf. on Computer Design*, pages 245–248, 1992.

[8] 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 Case Generator for Architecture Verification. In *IEEE Transactions on VLSI Systems*, volume vol. 3, no. 2., pages 188–200, June 1995.

[9] M. Blum, A. Chandra, and M. Wegman. Equivalence of Free Boolean Graphs Can Be Decided Probabilistically in Polynomial Time. In *Information Processing Letters*, pages 10:80–82, 1980.

[10] R. Krieger, B. Becker, and R. Sinkovic. A BDD-based Algorithm for Computation of Exact Fault Detection Probabilities. In *International Symposium on Fault-Tolerant Computing*, pages 186–195, 1993.

[11] D. Koller and N. Megiddo. Constructing small sample spaces satisfying given constraints. *SIAM Journal on Discrete Mathematics*, pages 260–274, 1994.