Hardware accelerated constrained random test generation

B. Pal, A. Sinha, P. Dasgupta, P.P. Chakrabarti and K. De

Abstract: Recent design and verification languages, such as SystemVerilog, support a rich test bench language, which provides significant support towards developing layered, structured, constrained random test bench architectures. Typically, the test bench language offers many features that are not synthesisable and therefore cannot be carried into the hardware for hardware accelerated simulation. One of the main challenges in improving the performance of hardware accelerated simulation is to run the task of random value selection under specified constraints in hardware. This problem (possibly for the first time) is addressed and a two-step approach is presented. In the first step, the constraints are pre-processed in software to generate a set of entailed regions. In the second step, random value selection is performed in hardware using the entailed regions pre-computed in the first step. It is shown that this method has modest area overhead and produces constraint satisfying random valuations within very few cycles. Results on test bench architectures for the ARM AMBA Bus and IBM CoreConnect protocol suites have been reported.

1 Introduction

In recent times, there has been a significant change in the hardware verification paradigm from a directed approach to a coverage-driven randomised one, where the verification engineer attempts to develop a single constrained random test bench architecture that potentially covers all interesting behaviours, as opposed to the older practice of writing multiple directed tests targeting different behaviours. Recent experience shows that although the validation engineer has to invest more effort in modelling the test bench architecture in the new style, the coverage of behaviours is significantly faster eventually because automatic random test selection overtakes the time required to write directed tests individually.

It is therefore not surprising that most recent hardware verification languages, including Open-Vera [1], Specmen-Elite [2], and SystemVerilog [3], support constructs for defining constraints as well as constructs for random value selection under the given constraints. Simulation tools for these languages have built-in support for constraint solving and constrained random value selection. Since these solvers are implemented in software, they use state-of-the-art constraint solving techniques, often using intelligent backtracking search algorithms [4, 5].

Fig. 1 shows the emerging trends in hardware accelerated simulation. Fig. 1*a* shows a pure simulation based framework where both the design-under-test (DUT) and the test bench are simulated in software, Fig. 1*b* shows a conventional hardware accelerated simulation framework where the DUT is mapped into a hardware emulation platform (such as a field programmable gate array (FPGA) board which runs much faster than the simulator) whereas the test bench runs in software. It may be noted that methods for FPGA prototyping of synthesisable Verilog designs are quite well developed [6–9].

With the growing size and complexity of layered constrained random test benches [10, 11], the test bench is becoming the major bottleneck in hardware accelerated simulation because the test bench must interact with the DUT in every cycle. In a layered constrained random test bench, the lower layers directly interact with the DUT in every cycle, whereas the upper layers react less frequently (say once in every transaction). It is believed that migrating the lower layers of the test bench into hardware (as shown in Fig. 1*c*) will significantly speed up the simulation. This is one of the emerging challenges in hardware accelerated simulation today. One of the main components of this problem namely the problem of automatic constrained random value selection in hardware is the focus of this paper.

The generic problem is as follows—given a set of variables and a set of constraints over these variables, we need to synthesise a circuit in hardware that is able to deterministically generate pseudo-random valuations of these variables subject to the given constraints. Our goal is to develop a tool that performs this synthesis from a given system of constraints. There are two notable features of this problem:

(1) The constraints are known a priori, because these are hard-coded in the test bench.
(2) Our circuit must guarantee that all valuations, satisfying the constraints have a non-zero probability of being selected.

The first feature gives us the liberty to precompute (in software) a set of regions that satisfy the original constraints, and then carry these simpler constraints into hardware for random value selection at runtime.

doi:10.1049/iet-cdt:20070016

Paper first received 22nd January and in revised form 12th April 2007

B. Pal, A. Sinha, P. Dasgupta and P.P. Chakrabarti are with the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur 721302, India

K. De is with Synopsys (India) Pvt. Ltd., RMZ infinity, Benniganahalli, Bangalore-16, India

E-mail: bhaskar@cse.iitkgp.ernet.in

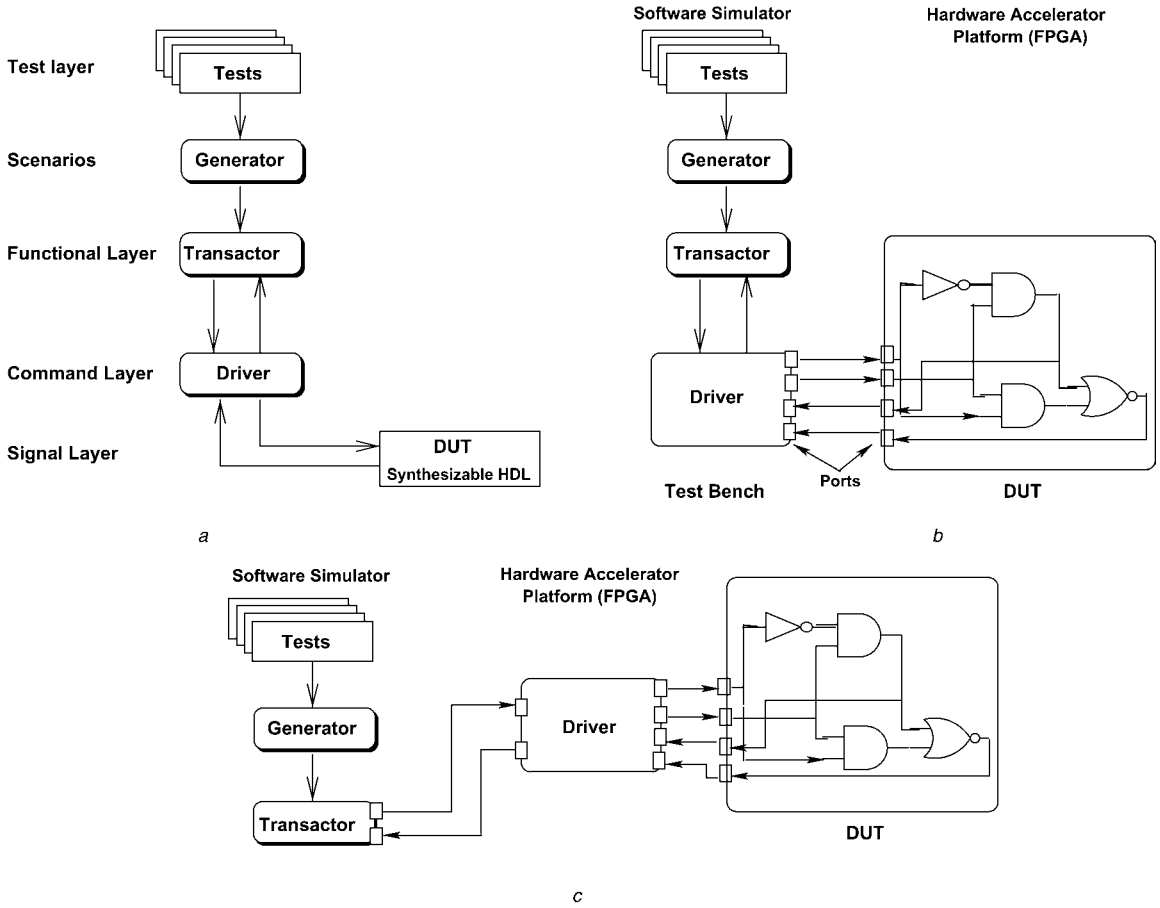

Fig. 1 *Emerging trends in hardware accelerated simulation*

a Pure software simulation
b Hardware accelerated simulation
c Outline of the proposal

From our experience in modelling the verification IPs for several standard bus protocols (including ARM AMBA [12], IBM CoreConnect (IBMCC) [13], and PCI Bus [14]), we have observed that constrained value selection in randomised test benches are broadly of two types:

(1) *Selection of control bits:* For example, the model of an arbiter in the test bench may randomly select a grant line from among requesting master devices. Also the test bench may randomly select the transfer type (single/burst) from a set of allowable transfer types.
(2) *Selection of data words:* For example, in a memory mapped architecture, the address must be within a specific range for a given device to be selected. As another example, the starting address may have to be aligned to 10 K byte boundaries in a burst transfer.

For the first type, the constraints are typically Boolean, or can be modelled as Boolean constraints without much overhead. Value selection under Boolean constraints has been well studied [15–17]. There has also been several attempts towards synthesising Boolean constraints in hardware, including attempts to synthesise boolean decision diagrams (BDD) that model the solution space [18, 19].

Our interest in this paper is on constraints of the second type. Since these are constraints on words, any Boolean encoding of the problem will lead to combinatorial explosion because there will be too many variables. For example, BDDs grow alarmingly after about 100 variables, whereas each word in a 64 bit architecture will add 64 variables.

How complex are our constraints? This is an important question in terms of feasibility, since solving constraint satisfaction problems using real variables is a notoriously complex problem in general and it is hard to conceive any efficient implementation of existing algorithms in hardware.

On the other hand, the types of constraints typically used by verification engineers in constrained random test benches are often very simple. The most common constraints are range constraints, that specify the range of individual variables. Another common form of constraint specifies that the value of a variable must be within a given range of another variable (e.g. a randomly generated address variable may be constrained within some specified offset of a base address). Such constraints involve two variables. Individual constraints involving more than two variables are rare in most applications. For example in the entire test bench architectures for ARM advanced microcontroller bus architecture (AMBA), IBM CC and PCI Bus, we never required any single constraint involving more than two variables although many variables were constrained. Sample test benches of IBMCC device control register (DCR), on-chip peripheral bus (OPB) protocols and AMBA AHB Master component are given in our website for reference [20].

We target systems of linear constraints where each constraint involves at most two variables. Since all the constraints are hard-coded in the test bench, we have the liberty to simplify the set of constraints prior to hardware accelerated simulation. Our methodology works in two steps.

(1) *Constraint pre-processing in software:* In the first step, we use an integer linear programming (ILP) solver to

```
`define AnySlave 3'b001 //defining a state
program master;
  ...
  class MDriver;

            rand bit [9:0] base;
            rand bit [4:0] offset;
            ...
            task main_t ();
            ...
          endclass

          MDriver TB = new();
          ...

          task MDriver :: main_t () begin
            ...
            case (MasterState)
              ...
              `AnySlave: begin
                ...
                success=TB.randomize() with
                  {base >=0 && offset>=0 && (base+offset) <= 511 &&
                  (base+2*offse) <= 1023 && (base+2*offset) >= 512;};
              end
              ...
            endcase
          endtask
          ...
        endprogram
```

Fig. 2 *Example constraint random test-bench*

deduce a set of regions covering the constraint satisfying valuations. Since this is a pre-processing step executed in software, and since this step is executed only once before the start of the emulation, the complexity of this step is not a major issue and we can afford to use sophisticated algorithms.

(2) *Constrained random selection in hardware:* In the second step we synthesise a constrained random test generator in hardware. This circuit uses a pseudo-random generator based on a linear feedback shift register (LFSR) and uses constraint specific circuitry to constrain the generator within the regions entailed by the first step.

The main challenge is that the first step must produce entailed regions in some form such that the second step can work efficiently, both in terms of the number of clock cycles required to produce the random valuations, and in terms of the area overhead of the generating circuit. We present a simple example for illustration as follows.

Example 1.1: Consider a Bus connecting a Master device and two Slave devices. The Master, when required, can initiate transfers (read or write) to the Slave devices. The address map of the Slave is [0, 1023], where $Slave_1$ has the range [0, 511] and $Slave_2$ has the range [512, 1023]. For slave-specific tests, a test bench that models the Master needs to constrain the values of base and offset in order to provide addresses within the desired range. A SystemVerilog constraint random test bench fragment is given as follows. For readability, only relevant portions of the test bench are given (Fig. 2).

The proposed methodology is performed in two steps. In the first step, we take the input test bench constraints and pre-compute the solution region and entail this in some well-defined data structure. For example, the solution region (see Fig. 3) for the above constraints is entailed by the lines:

$$C1\text{:offset} \geq 0;\ C2\text{:base} + \text{offset} \leq 511;$$
$$C3\text{:base} + 2\text{offset} \geq 512$$

This processing is done in software. In the second step, we use the stored information to search for a random valuation of the inputs in hardware. The pre-computation and entailment of the solution region make the search algorithm deterministic, non-backtracking, synthesisable and workable within limited hardware resources.

The paper is organised as follows. We present the overall idea around suitable examples in Section 2. Section 3 presents the software preprocessing phase. Section 4 describes the synthesis methodology with suitable examples. Section 5 extends the basic pre-processing step in presence of inequality constraints. Section 6 presents the application aspects of the tool for the layered verification

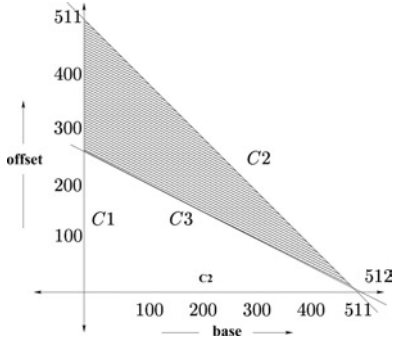

Fig. 3 *Solution region*

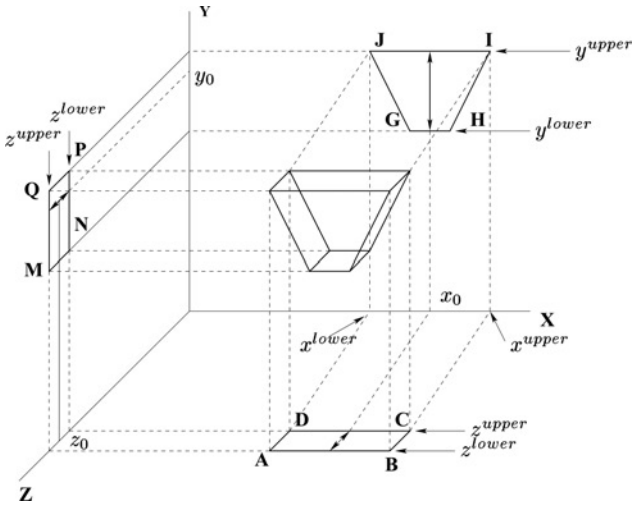

Fig. 4 *Overall idea*

architecture of reference verification methodology (RVM) [11]/verification methodology (VM) [10]. Section 7 presents results on standard Bus protocols.

2 Central approach

Given a set of linear constraints over N variables, such that each constraint has no more than two variables, our solution space represents a region in the N-dimensional hyperspace. Our goal is to randomly choose points in that space.

In Fig. 4 we illustrate our approach with an example having constraints in three variables, x, y and z. The 3D model (in the centre of Fig. 4) represents the solution space. We take the projection of the multi-dimensional space into 2D planes. In Fig. 4 **GHIJ**, **MNPQ** and **ABCD** are the projections in XY, YZ and ZX planes, respectively. Within the bounds of x (i.e. $[x^{\text{lower}}, x^{\text{upper}}]$), we first select a random value (call it x_0). This restricts the bounds of the other variables, namely a refined bound for y (i.e. $[y^{\text{lower}}, y^{\text{upper}}]$ from XY plane) and z (i.e. $[z^{\text{lower}}, z^{\text{upper}}]$ from XZ plane). Next, we choose a random value for y within the refined bound. This again might refine the bounds on successive variables (i.e. z in our case). At this stage only z needs to be assigned a random value. We choose a random value within $[z^{\text{lower}}, z^{\text{upper}}]$. In general, in the k-th step, a random value is assigned to the k-th variable and the given problem reduces to a sub-problem having $(n - k)$ variables. More importantly, the choice of the value of the k-th variable is done in such a way that guarantees the existence of a solution for the reduced problem involving $(n - k)$ variables.

By virtue of projections, our target in hardware is to get a random value for a constrained variable from the projected polygon in any given plane. Essentially, the hardware should be simple enough to churn out random values in a stipulated time frame, otherwise the whole purpose of hardware acceleration would be defeated. For this, suitable logic is required to store and propagate the information obtained from each plane. We partition our methodology into two halves–software preprocessing of the 2D data we have from each plane, followed by hardware synthesis of the oracle targeted to generate valid assignments to the set of constrained variables.

3 Software pre-processing

In this section, we will show the various stages of entailment of the solution space for a given problem $\boldsymbol{C}$, along with the supporting theorems and their proofs. Our goal is to decompose the solution space of each variable into a finite number of entailed regions. The stages of the computation are described in the following subsections.

We use the following example to demonstrate the software preprocessing of the projections on the xy plane.

$$\tilde{C} = \langle \{x, y\}, \{C_1, C_2, \ldots, C_5\} \rangle \text{ where}$$
$$C_1{:}x + y \geq 1;\ C_2{:}x + y \leq 5;$$
$$C_3{:}y \leq 3;\ C_4{:}x - y \leq 1;$$
$$C_5{:} -x + y \leq 1$$

3.1 Computation of bounds

The linear constraints when represented graphically on a 2D plane will give a polygon representing the solution space as shown in Fig. 5. In order to check the satisfiability of $\boldsymbol{C}$, we compute the convex polygon (encapsulating the solution region). In this example, we compute **ACDE**. There exists

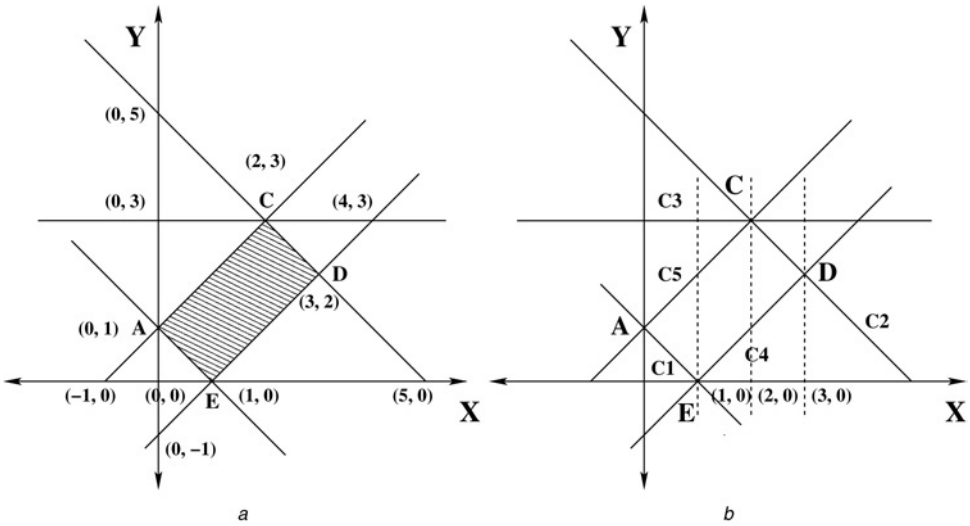

Fig. 5 *Constraints plotting in X–Y plane*

a Constraints plotted as the ploygon
b Set of strips of the solution polygon

ComputeStrip(C, V(polygon))

begin

1: $S \leftarrow$ NULL, where S is set of strips.

2: $StripCount \leftarrow$ **Sort** (V(polygon)).

3: for (i=1 to StripCount) do

3.1: $S_{x_i x_{i+1}} \leftarrow \langle \{x_i, x_{i+1}\}, \{c_j \mid c_j \text{ crosses } S_{x_i x_{i+1}}\} \rangle$

3.2: $S \leftarrow S \cup S_{x_i x_{i+1}}$

Endfor

4: Return S.

end

Fig. 6 *Procedure compute strip*

standard methods [21, 22] for computing the boundary points of a polygon. In this work, we use a standard ILP solver ILOG CPLEX [23] to compute the set of vertex (boundary points) of the polygon. We refer this set as V (polygon).

3.2 Strip computation

Our next step is to cut the 2D polygon into a collection of trapeziums or triangles. We sort the vertices of the polygon by one of the dimensions. The dimension is decided by the user. For example, for a polygon in the XY projection, we sort the vertices by the non-decreasing x co-ordinates and drop vertices having same x-coordinate. We use the following definitions.

Definition 3.1: A strip is defined as the region bounded by $[x_j, x_{j+1}]$ where $x_j < x_{j+1}$ and no other vertex (v) exists with x-coordinate (x_v) such that $x_j < x_v < x_{j+1}$.

Definition 3.2: The set of constraints of the convex (solution) polygon which crosses through a strip is called the overlapping constraint set (OCS) for the strip.

The algorithm for strip computation is given in Fig. 6

In our example, the strips along with the associated OCS are (see the solution polygon **ACDE** in Fig. 5):

$$\langle \{0, 1\}, \{C_1, C_5\} \rangle,$$
$$\langle \{1, 2\}, \{C_4, C_5\} \rangle, \langle \{2, 3\}, \{C_2, C_4\} \rangle$$

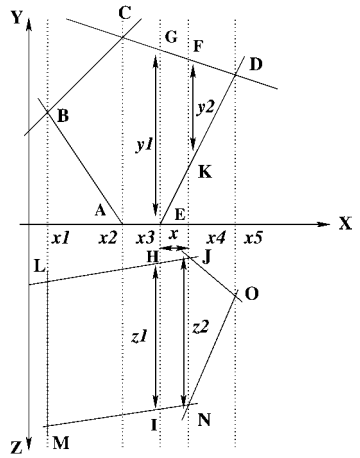

Fig. 7 *Estimate of volume from different plane-profiles for a particular strip*

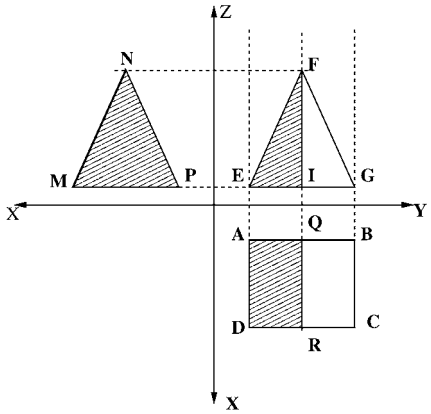

Fig. 8 *Different profiles of a tetrahedron*

We have seen that such OCS for a strip always has the cardinality 2. This is important as we use this during synthesis to generate the solution points.

3.3 Computation of the strip probability

Since we compute the strips of the projection of the solution space on the different 2D planes, the area of the strips does not reflect the density of the solution points on a given strip. Ideally we wish to assign to each strip a weight that is proportional to the total volume of the solution space that is projected on that strip. We quantify this estimated volume as follows.

Consider Fig. 7, where the vector space is 3D. Consider the strip $S_{x_3x_4}$. The XY plane contains the trapezium $GFKE$ whereas the ZX plane contains the trapezium $HJNI$ corresponding to the given strip. Let $W(S_{x_3x_4})$ define the weight of choosing $S_{x_3x_4}$

$$W(S_{x_3x_4}) = \frac{\text{Area}(GFKE) \times \text{Area}(HJNI)}{\text{Area}(ABCDE) \times \text{Area}(LMNOJ)}$$

Formally, we define

$$W(S_{x_ix_{i+1}}) = \frac{\prod_{\text{all2D-planes}} \text{Area\_induced\_by\_}S_{x_ix_{i+1}}}{\prod_{\text{all2D-planes}} \text{Area\_of\_projected\_polygon\_in\_that\_plane}}$$

The intuitive justification of defining such a metric is as follows. Consider Fig. 8. It shows the different plane-profiles of a tetrahedron. We are interested in computing the probability of the strip S_{EI}. From our definition

$$W(S_{EI}) = \frac{\text{Area}(\Delta EIF) \times \text{Area}(ADRQ) \times \text{Area}(\Delta MNP)}{\text{Area}(\Delta EFG) \times \text{Area}(ABCD) \times \text{Area}(\Delta MNP)}$$

$$= \frac{1/2\text{Area}(\Delta EFG) \times 1/2 \text{Area}(ABCD) \times \text{Area}(\Delta MNP)}{\text{Area}(\Delta EFG) \times \text{Area}(ABCD) \times \text{Area}(\Delta MNP)}$$

$$= \frac{1}{4}$$

We want to estimate the volume. Thus, $W(S_{EI})$ should be equal to the ratio of the volume cut by the strip S_{EI} to the whole volume. Therefore ideally

$$W(S_{EI}) = \frac{1}{2}$$

Our estimate gives the square of this ratio. From the intuitive dimensional analysis, we find the following (the operator dim() gives the dimension of a quantity)

$$\dim(W(S_{x_i x_{i+1}})) = \dim\left(\frac{\prod_{\text{all2D-planes}} \text{Area\_induced\_by\_}S_{x_i x_{i+1}}}{\prod_{\text{all2D-planes}} \text{Area\_of\_projection\_in\_given\_plane}}\right) = \dim\left(\frac{(\text{volume induced by } S_{x_i x_{i+1}})^{n-1}}{(\text{total volume induced})^{n-1}}\right) = \dim\left(\left(\frac{\text{volume induced by } S_{x_i x_{i+1}}}{\text{total volume induced}}\right)^{n-1}\right)$$

This is just an intuitive rationale behind the estimate. For many regular shaped objects the estimate truly gives ratio $(v/\mathrm{V})^{n-1}$, where v is the volume induced by a particular strip, and V is the total volume of the object. To illustrate further, in our example

$$W(S_{EI}) = \left(\frac{1}{2}\right)^{(3-1)} = 1/4$$

In order to obtain the selection probabilities of the strips, the strip weights are normalised as follows

$$P(S_{x_i x_{i+1}}) = \frac{W(S_{x_i x_{i+1}})}{\sum_i W(S_{x_i x_{i+1}})}$$

To make the strip probabilities usable by the synthesis algorithm, we map the strip probabilities to integer values. Let for the ith dimension, the strip probabilities for the variable x_i be 0.6, 0.36 and 0.04. After the map, these would change to weights 60, 36 and 4. Further optimization would produce weights 15, 9 and 1.

During synthesis, for each variable (say x_i), we use a variable TWt_{x_i} that is assigned a value equal to the sum of weights of all the strips for variable x_i. For example, in the above case, TWt_{x_i} would have the value $(15 + 9 + 1)$ that is 25. Now we define range for each of these strips to be: [1, 15], [16, 24], [25, 25]. During synthesis, we use a LFSR that generates a random value between 1 and 25, and we select a strip depending on this value. For example, when the LFSR generates 12, strip 1 will be selected.

4 Synthesis methodology

The synthesis algorithm takes the following inputs (for every variable x) from the software preprocessing phase.

(1) The set of strips and the OCS (defined in Section 3.2) for each of the strip. In our example (see Fig. 5), OCS for each of the strips are given as follows

$$\langle\{0, 1\}, \{C_1, C_5\}\rangle,$$
$$\langle\{1, 2\}, \{C_4, C_5\}\rangle, \langle\{2, 3\}, \{C_2, C_4\}\rangle$$

(2) Sum of weights of all the strips that is TWt_x.

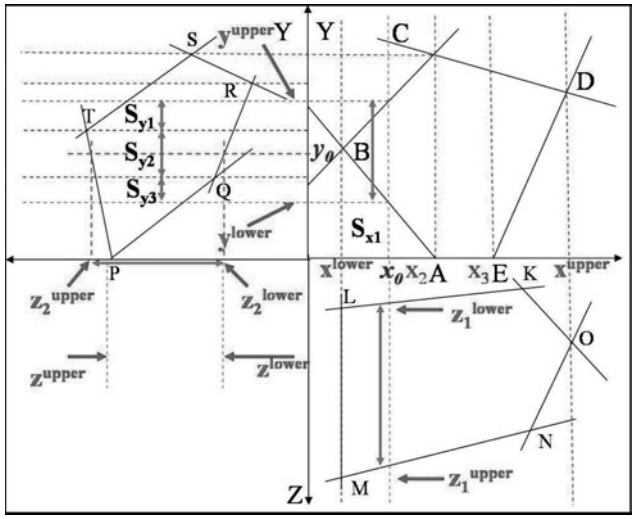

Fig. 9 *Synthesis methodology*

We define lower and upper bounds of a variable x as

$$x^{\text{lower}} = \text{MIN(first co-ordinate of all the strips)}$$
$$x^{\text{upper}} = \text{MAX(second co-ordinateof all the strips)}$$

In our example (see Fig. 5), $x^{\text{lower}} = \text{MIN}\ (0, 1, 2) = 0$ and $x^{\text{upper}} = \text{MAX}\ (1, 2, 3) = 3$.

Given the above-mentioned inputs for every variable, we do the following.

- We first choose the variable order. In our case, we take the variables in a pre-determined order.
- For each variable x, we do the following.

(1) Within the bounds of the variable, we randomly choose a strip according to the strip weights. We use the range [start, TWt_x] for the choice. Variable 'start' indicates the lower bound of the first strip within the bounds of x. In this example (see Fig. 9) start = 1.
(2) Within the bounds of the chosen strip, we choose a random value for the variable, x.
(3) After each assignment of a variable, the bounds on the remaining variables may change. For example, choice of value x_0 for variable x (see Fig. 9) refines the initial bounds of variable y (new boundary contains only strips S_{y_1}, S_{y_2}, and S_{y_3}).
We use the OCS of the chosen strip to compute the new bounds of the remaining variables.
(4) For each variable, whose bounds have been refined, we re-compute the value of TWt_x.

Note that steps 1 and 2 are similar. Both of these work based on randomly generating a number within a given bound. There are mainly three basic hardware blocks which are used for computing constraint satisfying random valuations of the set of variables. In the following sub-sections, we illustrate the synthesis of each of these blocks in details.

4.1 Generating a random number within a given bound

To generate a random valuation of a variable x, within a given bound $[x^{\min}, x^{\max}]$, we use the Procedure GenRandNum which is illustrated in Fig. 10.

We compute the difference between the initial upper and lower bounds of the variable. For x it is, $D = x^{\text{upper}} - x^{\text{lower}}$. Let us assume that D be of k bits, that is, $2^k \geq D > 2^{k-1}$.

Procedure GenRandNum(***UpperBound*** (x^{max}), ***LowerBound*** (x^{min}), ***k-bit-LFSR***, ***RandNum***)

begin

1. We compute the difference between the given bounds, d = x^{max} - x^{min}. Let us assume that d be of k_1 bits (note that $k_1 \leq k$), that is $2^{k_1} \geq d > 2^{k_1-1}$.
2. We use the k-bit-LFSR to generate a k-bit random number R. Note that we could have used a k_1 ($\leq k$) bit LFSR to generate R. However, the value of k_1 is not known a priori and may vary. Thus we use the k-bit LFSR.
3. Repeat k times the following.
   - 3.1. Compare R with d.
   - 3.2. If R $\leq d$, ***RandNum*** $\leftarrow R$. Break.
   - 3.3. Else if R > d, flip the first set most significant bit (MSB) of R.

   EndRepeat
4. ***RandNum*** $\leftarrow$ ***RandNum*** + x^{min}.

end

Fig. 10 *Outline of the procedure GenRandNum*

We shift the LFSR only once (in step 2) such that it requires one cycle to generate R. A sample schematic of the associated logic is shown Fig. 11. The logic is produced by Synopsys Design Compiler.

Example 4.1: Let $x_{min} = 1$ and $x_{max} = 21$. Therefore $d = 20$. Also assume that $D = 25$. Thus the width of the LFSR becomes 5-bits. Suppose the 5 bit LFSR generates R(=5′b11100). So, $R > d$. Thus we check R from the MSB. It has a '1' at the MSB. So we flip it. The modified R becomes 5′b01100 (decimal 12), which becomes less than d. Thus the generated value for x becomes 13 (1 + 12). □

A detailed analysis of the circuit complexity is given in Table 1. The first column shows width (in bits) of R (i.e. D). The second column shows width (in bits) of d. The third column shows the gate count of the generated logic. The fourth column enumerates the area overhead and the fifth column shows the critical path delay (CPD). Note that the circuit complexity grows linearly with the width of R. The synthesis has been done using Design Compiler [24] (of Synopsys).

4.2 Refinement of the bounds of a variable

Each strip for a variable x has its own OCS (defined in Section 3.2). After choosing a random value for a variable x within a chosen strip, we need to refine the bounds of the other variables associated with the overlapping constraints for that strip. The refinement is done by a (simple) combinational logic. The logic uses the linear constraints involving x and every other associated variable y, and substitutes the value of x (in the constraints) to get a refined domain for y. As the OCS for a strip is constant, the amount of logic is also constant.

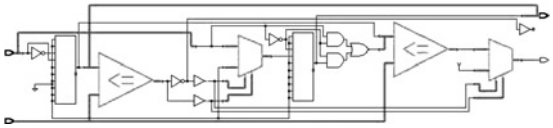

Fig. 11 *Sample schematic for the value adjustment logic*

Table 1: Logic complexity of the adjustment hardware

Width of R, bit	Width of d, bit	Gate count	Area, sq. micron	CPD, (ns)
8	3	34	8901.27	1.55
10	3	42	10 874.5	1.64
12	3	49	12 811.73	1.55
15	3	58	15 680.46	1.64

Example 4.2: Consider the following two constraints on two variables x and y.

$$Con_1: x + y \geq 1; \quad Con_2: x + 2 * y \leq 6$$

Initially, the lower and upper bounds of x and y are given as [1, 6] and [1, 3]. Assume that during randomization, we take x first and then y. For this, a sample bound updation logic (for variable y) is given as follows.

$$y^{ltemp} = \lceil 1 - x \rceil y^{lower} = (y^{lower} \geq y^{ltemp}) \,?\, y^{lower} : y^{ltemp}$$

$$y^{htemp} = \lfloor 3 - (x/2) \rfloor$$

$$y^{upper} = (y^{upper} \leq y^{htemp}) \,?\, y^{upper} : y^{htemp}$$

Now assume we get value 3 for variable x from the LFSR. For this i the refined bounds of y (from the above logic) become [1, 2].

Note that with reference to Fig. 9, the choice of x_0 (for variable x) makes only the strips S_{y2} (completely covered), S_{y1} and S_{y3} (partially covered) belong to the updated bound [y^{lower}, y^{upper}] of y. We will refer to this logic as a procedure called *RefineBound*.

A sample synthesised circuit (by Synopsys Design Compiler) for the updation logic (of Example 4.2) is

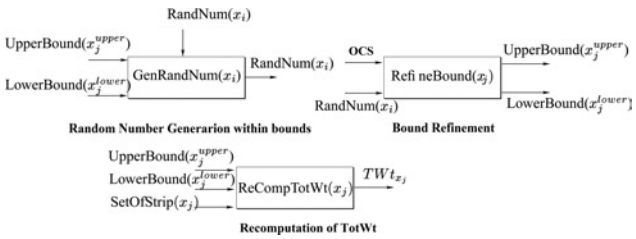

Fig. 12 *Outline of three primary procedures*

Table 2: Logic complexity of the adjustment hardware

Width of x, bit	Width of y, bit	Gate count	Area, sq. micron	CPD, ns
3	2	19	5535.0	1.57

given in Fig. 12. An estimate of the complexity of the circuit is given in Table 2. A schematic of this hardware block is shown in Fig. 13. For every other associated variable y, this block takes the appropriate OCS and the generated random number as inputs and updates the bounds of y.

4.3 Re-computation of the value of TWt

Once the bound refinement step for a variable y is over, we need to compute which strips of y now lie within the refined bounds. In general any refinement cancels out a number of strips. This step is done by the following logic. For each strip S_i of y that is partially/totally covered by the refined bound, we add the associated weight W_{S_i} to TWt_y. Variable 'start' is assigned the lower bound of the first strip that lies within the refined bound. A sample schematic of the associated logic is shown Fig. 14. We will refer to this logic as a procedure called *ReCompTotWt*. Fig. 14 shows the logic for a given strip S_i. Note that *ReCompTotWt* may involve more than one strip. For each strip, the schematic remains the same. S_i^{lower} and S_i^{upper} indicates the lower and upper bounds of strip S_i.

A detailed analysis of the circuit complexity is given in Table 3. Column 1 shows the number of strips. Note that complexity of the above logic depends on the number of strips. Column 2 shows the gate count of the generated logic. Column 3 enumerates the area overhead and Column 4 shows the CPD. Note that the circuit complexity grows almost linearly with the number of strips. However, we have observed that in general the number of strips does not become too large. The synthesis has been done using Design Compiler [24] (of Synopsys). A schematic of this hardware block is shown in Fig. 12. For every variable y, this block takes the (refined) bounds of y and the set of strips of y and it it updates the value of the variable $TWty$.

A block diagram for the overall flow of the above mentioned hardware blocks (right from the strip selection logic) is shown in Fig. 15. The block diagram demonstrates

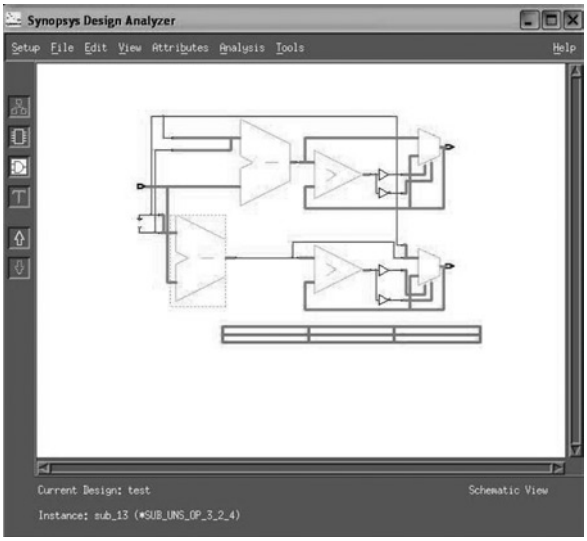

Fig. 13 *Sample circuit for the re-computation logic*

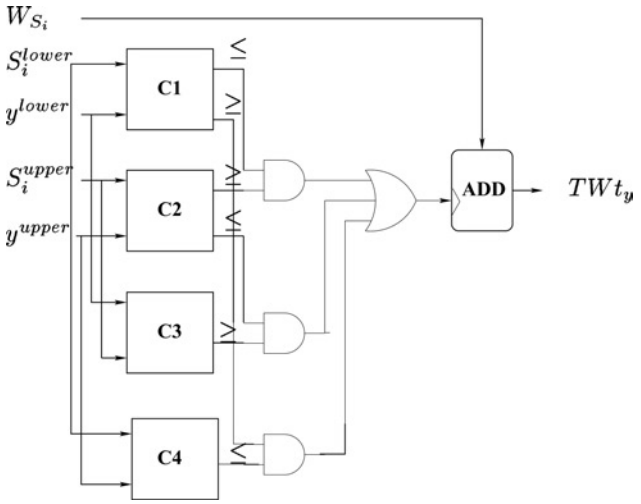

Fig. 14 *Sample schematic for the updation logic*

the logic for only variable x_i (from N variables). Logic for the other variables would follow the same pattern.

4.4 Synthesis algorithm

Let CT be the set of all the constraints and VT be the set of all variables. At any given instant, the set of variables with already generated random values are defined as GV (generated variables), and the set containing the rest is defined as NGV (non-generated variables). Initially, NGV is equal to VT. Moreover, the bound for each variable is initialised to its extreme feasible values. In the ith iteration, we choose a variable (say x_i) from NGV. After getting a random fair (using the weights for the strips) solution for x_i (steps 3.2 to 3.4 of Fig. 16), we push x_i in GV and update the bounds for x_{i+1} by using the Bound Updation logic (referred as Procedure *RefineBound*). We also update the variable $TWt_{x_{i+1}}$ using the recomputation logic (referred as Procedure *ReCompTotWt*). We iterate till we get a random value for all the variables, that is until NGV becomes empty. The outline of the algorithm is as follows.

4.4.1 Complexity analysis: Note that for each variable, both steps 3.2 and 3.4 in Fig. 16 (i.e. the random value generation) require one cycle. Steps 3.3 and 3.7 are executed by combinational logics. So the number of cycles required to generate the final solution depends on the number of variables in VT. Thus Fig. 16 requires $2 * |n|$ cycles to generate a random valuation for the set of n variables.

Theorem: Algorithm GenerateSolution guarantees that the valuation of the first k variables is done in a way that the remaining $(n - k)$ variables have a satisfiable assignment.

Proof: We start with maximum feasible bound B_{x_i} for ith variable ($i = 1$ to $n - 1$). Let, after obtaining a satisfying valuation $\tilde{x}_k$ for kth variable, there exists no satisfying value for $(k + 1)$th variable. This implies the bound for x_{k+1} (call it $B_{x_{k+1}}$) after refinement (say $B'_{x_{k+1}}$) is infeasible.

Table 3: Logic complexity of the adjustment hardware

No of strips	Gate count	Area, sq. micron	CPD, ns
2	31	5935.74	1.69
3	35	7215.7	2.04
5	42	9149.08	2.55

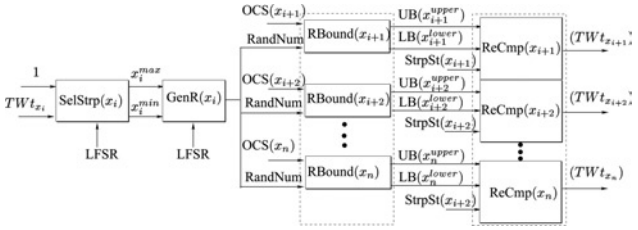

Fig. 15 *Overall logic flow*

$B_{x_{k+1}}$ was a feasible bound. Also, $B'_{x_{k+1}} \subseteq B_{x_{k+1}}$. Moreover, $B'_{x_{k+1}} \neq \phi$ (since otherwise it implies that no part of the polygon intersects with $x_k = \tilde{x}_k$, i.e. $\tilde{x}_k \notin B_{x_k}$ which is not possible). This implies $B_{x_{k+1}}$ contained an infeasible region, which is a contradiction. Hence, there exists a satisfying assignment for $(k+1)$th variable whenever we choose a valid valuation of k-th variable. □

Theorem 4.2: All valid valuations occur with a non-zero probability using GenerateSolution.

Proof: Consider a valid tuple $\tilde{x} = (\tilde{x}_0, \tilde{x}_1, \ldots, \tilde{x}_{n-1})$. If we show that on choosing '$\tilde{x}_k$' for k-th variable '$\tilde{x}_{k+1}$' lies in the refined bound for $(k+1)$-th variable, then applying this reason recursively we prove that all the n-elements of a tuple are considered with non-zero probability and hence the tuple has non-zero probability of occurrence.

Let on choosing $\tilde{x}_k$ for kth variable we exclude $\tilde{x}_{k+1}$ from $B_{x_{k+1}}$. This is possible only if the X_kX_{k+1}-plane possesses some constraints (since in order to refine $B_{x_{k+1}}$ the kth and $(k+1)$th variables should constitute some constraint, otherwise no valuation of kth variable can affect the bound of $(k+1)$th variable). As $\tilde{x}$ is valid, we have the following for all $C_i - s$ in X_kX_{k+1}-plane

$$(a_{i1}x_k + a_{i2}x_{k+1})\theta a_{i3} \quad \text{where } \theta \in \{ \geq, \leq \} \qquad (1)$$

Let C_p and C_q be the member of the overlapping constraint set for the strip constituting $\tilde{x}_k$. As $\tilde{x}_{k+1}$ is out of the refined bound

$$(a_{p1}x_k + a_{p2}x_{k+1}) \neg \theta a_{p3} \text{ or, } (a_{q1}x_k + a_{q2}x_{k+1}) \neg \theta a_{q3}$$

But that is a contradiction to (1). □

GenerateSolution(CT, VT)

begin

1: NGV ← VT , $B_{x_i} \leftarrow [x_i^{\text{lower}}, x_i^{\text{upper}}]$, $\forall i \in [1,n]$ obtained from CPLEX // Initialization step

2: $i \leftarrow 1$.

3: while (| NGV |) do

3.1: Take x_i from NGV.

3.2: $R_{x_i} \leftarrow$ *GenRandNum* ([start, TWt_{x_i}]).

3.3: Choose strip $S_{x_i}^k$ with bounds $[x_i^{\text{lower}}, x_i^{\text{upper}}]$ based on R_{x_i}.

3.4: $X_i \leftarrow$ *GenRandNum* ($[x_i^{\text{lower}}, x_i^{\text{upper}}]$).

3.5: $\boldsymbol{GV} \leftarrow \boldsymbol{GV} \cup \{x_i\}$.

3.6: NGV ← VT - GT .

3.7: For each variables $x_j \in$ NGV $(j > i)$ do

3.7.1: *RefineBound* (X_i, OCS($S_{x_i}^k$), $x_i^{\text{lower}}, x_i^{\text{upper}}$).

3.7.2: Update TWt_{x_j} by *ReCompTotWt* ($x_i^{\text{lower}}, x_i^{\text{upper}}$, *SetOfStrip*($x_j$), TWt_{x_j}).

3.8: $i \leftarrow i + 1$;

EndWhile

4: Solution ← $(X_i, X_{i+1}, X_{i+2}, \ldots, X_n)$.

Fig. 16 *Generation solution*

Theorems 4.1 and 4.2, respectively, show that our methodology is sound and complete.

5 Constraints with inequality

In order to extend the methodology for handling constraints having inequality we take the following approach. Let us consider the following set of constraints

$$C_1{:}x + y \leq 2;\ C_2{:} -x + y \leq 2;\ C_3{:}x \leq 5;\ C_4{:}y \leq 5$$

Now, let us include another constraint which has an inequality, say

$$C_5{:}x \neq 2$$

Clearly, this constraint would prevent any previous solution (e.g. point (2, 2)) which has x value of 2. For any linear inequality constraints, enumerating such bad points is very difficult (specially in hardware). Thus to address this issue, we split these inequality constraints into two parts as follows

$$C_5^1{:}x \geq 3$$

$$C_5^2{:}x \leq 1$$

Therefore any satisfying assignment should satisfy the following conjunction of constraint

$$C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$$

which is equivalent to

$$C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge (C_5^1 \vee C_5^2)$$

which can be distributed over the disjunction

$$(C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5^1) \vee (C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5^2)$$

Observe that the set of constraints $\{C_1, C_2, C_3, C_4, C_5^1\}$ and $\{C_1, C_2, C_3, C_4, C_5^2\}$ produce two disjoint convex regions in the four-dimensional hyperspace. Therefore the task is to compute the strips for both the regions during the software preprocessing stage and during the actual hardware test-generation, first randomly choose the solution region and next, apply the aforesaid methodology.

6 RVM/VM integration

Recently, several test-bench modelling techniques have been standardised such as RVM [11]/VM [10]. These propose a layered test-bench architecture. The upper layers (Generator, Transactor etc.) are involved in generating the high-level transaction scenarios. Each of these transactions involve several cycles of signal transitions with the DUT. In general, these signal transitions (in accordance with the associated protocol) are controlled by the lower level Bus Functional Models (BFMs). These BFMs are generally called the Drivers that take part in actual handshaking with the DUT. The task of moving the high-level layers into the hardware is a mammoth task which includes developing methodology for synthesis of several high-level language constructs. In recent years, researchers have proposed to move only the lower-level BFMs/Drivers into the hardware [25]. This movement necessarily speeds up the overall simulation as the most frequent interaction with the DUT are done in the hardware. The high-level layers are connected with this hardware via a transaction-based interface. This permits the high-level layers to quickly send large fragments of transactions to enable rest of the processing done by the BFMs (in the hardware). Note that to model the

Table 4: Results on industry-standard Bus protocol

		Var stat					Run-times		Area overhead		
		No of Vars	Width of Vars	No of planes ($^nC_2$)	No of cons	No of strips	Preproc time, ms	Synth time, ms	Bit count	Gate count	No of cycles
AHB	arbiter	6	(32, 3, 3, 2, 2, 2)	15	11	21	2	3	70 (LFSR:11)	156	12
	slave	5	(32, 3, 3, 2, 2)	10	8	14	2	2	67 (LFSR:10)	144	10
	master	2	(2, 4)	1	4	2	1	2	25 (LFSR:5)	76	4
IBMCC	DCR slave	2	(32, 2)	1	4	2	1	1	56 (LFSR:7)	90	4
	DCR master	2	(10, 2)	1	4	2	1	1	33 (LFSR:6)	75	4
	OPB arbiter	3	(32, 2, 2)	3	8	4	2	2	59 (LFSR:8)	96	6
	OPB master	3	(2, 2, 2)	3	6	4	1	1	24 (LFSR:3)	70	6

```
success = randomize() with {
   trans >= 2 && trans <= 3 &&
   ((fracad + size >= 1) && (fracad + size <= 3)) &&
   ((addr + burst * 16) <= 255 && addr >= 128) &&
   burst >= 4 && burst <= 7 && size <= 2 && resp >= 1 && resp <= 2;};
```

Fig. 17 *In-line constraint for the enviornment*

protocol (for interaction with the DUT), these BFMs are designed using several in-line constraints. The methodology presented in this paper generates hardware for these in-line constraints.

7 Results

We have applied our prototype tool on the test-benches of two industry standard on-chip bus protocols, namely ARM AMBA AHB and IBM CoreConnect. The test-benches have been modelled following standard test-bench modelling guidelines (such as RVM/VM). These test-benches have also been used as environment models for the verification of different components of the two protocols. There are two goals of presenting the experimental results. Firstly, it establishes a proof of concept on industry standard designs and justifies our claim that systems of constraints involving only bi-variate constraints are both relevant and often adequate in practice. Secondly, it demonstrates that our synthesis methodology generates circuits that have modest area overheads and can generate random valuations within a few cycles.

In order to provide an idea of the complexity of constraint synthesis for a complex test-bench, we present only a few instances of the constraints (see Table 4). Sample constrained random test-benches for IBM CC OPB Master and DCR Master device are given in [20].

Table 4 shows the results of our tool on IBM CoreConnect and AMBA AHB on-chip Bus protocols. Analysis is given for some specific constraints modeled in the environment of the DUT. The name of the DUT is given first (e.g. AHB Arbiter, Master, Slave etc.). Next details of the associated environment constraints are given. The third column shows the number of variables and the fourth column enumerates the width of these variables in the constraint. For example, a specific constraint in the environment for verifying the AHB Master includes six variables (of width 32, 3, 3, 2, 2 and 2 bits, respectively). Note that the width of the variables greatly affects the number of bits of memory in the generated hardware. The fifth column shows the number of planes and the seventh column shows the number of strips in all the planes. The sixth column indicates the number of input constraints (cons). The eighth and ninth columns show the run-times (preprocessing and synthesis times) in an Intel(R) Pentium(R) 2.8 GHz, 512 MB RAM machine. The run-times are given in mili-seconds (ms). The 10th and 11th columns enumerate the area-overhead (bit count and Gate count) in the generated circuit. Also we provide the number of bits of memory required by the LFSR's. We have used Design Compiler [24] of Synopsys for synthesising the hardware. Note that the area complexity of IBM CC OPB and AHB Master is quite less compared to the others. This is because the width of the variables associated with the environment of these components are small. Other components include constraints involving address buses. The last column shows the number of clock cycles required to generate a random valuation satisfying the constraints.

Example 7.1: To illustrate the above table further, consider the following scenario from the AMBA AHB protocol. A master component (master number: 4) gets a Split or Retry response for a transaction targeting towards a slave component (slave number: 1). The slave has an address range: [128:255]. The transfer is a half-word (16 bit) read transfer for specific byte lanes. The byte lanes are selected by two variables fracad (2 bits) and size (3 bits). The span of the transfer is characterised by variables addr (32 bits) and burst (3 bits). Transfer mode and response types are selected by variables trans (2 bits) and resp (2 bits), respectively. Now we analyse the complexity of one of the constraints required to model this scenario in verifying the AHB arbiter component. To verify the arbiter component, we require to model an environment consisting of the master and the slave device. The in-line constraint for this environment is given in Fig. 17.

Note that this constraint requires six variables (see Table 4). The variable widths are (32, 3, 3, 2, 2, 2). The number planes is $^6C_2 = 15$. The number of constraints are 11 (see above). Analysis for the other bus components are given in a similar fashion. □

8 Conclusions

By investigating several complex test benches for verifying on-chip Bus protocols, we have identified that these test benches in general use constraints which involve either ranges of individual variables or linear constraints involving two variables. In this work, we concentrate on synthesis of these types of test benches. However the problem of synthesis of general constraints (involving any number of variables) remains an open problem.

9 Acknowledgment

We take this opportunity to thank the referees for their time in reviewing this paper, and for their comments. The authors acknowledge Synopsys (India) Pvt. Ltd. for partial Support of this work. Pallab Dasgupta and P.P. Chakrabarti further acknowledge the Department of Science and Technology (DST), Govt. of India for partial support of this work.

10 References

1 OpenVera LRM 2.0, http://www.open-vera.com
2 e Reuse Methodology (eRM), www.verisity.com\products\erm.html
3 SystemVerilog 3.1a Language Reference Manual. http://www.eda.org/sv/SystemVerilog_3.1a.pdf
4 Dechter, R.: 'Constraint processing' (Morgan Kaufmann Publishers, 2003), ISBN 1-55860-890-7
5 Tsang, E.: 'Foundations of constraint satisfaction' (Academic Press, 1993)
6 Bauer, J., Bershteyn, M., Kaplan, I., and Vyedin, P.: 'A reconfigurable logic machine for fast event-driven simulation'. Proc. Design Automation Conf. (DAC), June 1998, pp. 668–671
7 Cadambi, S., Mulpuri, C., and Ashar, P.: 'A fast, inexpensive and scalable hardware acceleration technique for functional simulation'. Proc. Design Automation Conf. (DAC), June 2002, pp. 570–575
8 Suyama, T., Yokoo, M., Sawada, H., and Nagoya, A.: 'Solving satisfiability problems using reconfigurable computing'. Proc. IEEE Trans. on VLSI Systems, February 2001, vol. 9, pp. 109–116
9 Varghese, J., Butts, M., and Batcheller, J.: 'An efficient logic emulation system', *Proc. IEEE Trans. VLSI Syst.*, 1993, **1**, (2), pp. 171–174
10 Bergeron, J., Cerny, E., Hunter, A., and Nightingale, A.: 'Verification methodology manual for systemverilog' (Springer, 2005), vol. XVIII, p. 510
11 Reference Verification Methodology for Vera, http://www.synopsys.com/products/simulation/pdf/va_vol4_ids1_vera.pdf
12 AMBA Specification Rev2.0, http://www.arm.com/products/solutions/AMBA_Spec.html
13 IBM CoreConnect Bus Specification. www-306.ibm.com\chis\techlib\techlib.nsf\techdocs\
14 PCI Bus Specification 3.0,http://www.pcisig.com/membersdownloads/specifications/conventional/PCI_LB3.0-2-6-04.pdf
15 Chandra, A.K., and Iyengar, V.S.: 'Constraint solving for test generation–a technique for high level design verification'. Proc. Intl. Conf. on Computer Design (ICCD), October 1992, pp. 245–248, ISBN: 0-8186-3110-4
16 Shimizu, K., and Dill, D.: 'Deriving a simulation input generator and a coverage metric from a formal specification'. Proc. Design Automation Conf. (DAC), June 2002, pp. 801–806
17 Yuan, J., Shultz, K., Pixley, C., Miller, H., and Aziz, A.: 'Modeling design constraints and biasing in simulation using BDDs'. Proc. Int. Conf. on Computer-Aided Design (ICCAD), 1999, pp. 584–589
18 Albin, K., Yuan, J., Aziz, A., and Pixley, C.: 'Constraint synthesis for environment modeling in functional verification'. Proc. of the Design Automation Conf. (DAC), USA, June 2003, pp. 296–299
19 Kukula, J.H., and Shiple, T.R.: 'Building circuits from relations'. Proc. 12th Int. Conf. on Computer Aided Verification (CAV), LNCS, 2000, vol. 1855, pp. 113–123
20 Example Constrained Random Test Benches. http://www.facweb.iitkgp.ernet.in/~pallab/Constraint1.htm
21 Berg de, M., Kreveld, V.M., Overmars, M., and Schwarzkopf, O.: 'Computational geometry: algorithms and applications' (Springer Verlag, 2000, 2nd rev.,), p. 367, ISBN: 3-540-65620-0
22 Boissonnat, J., and Yvinec, M. (translated by Bronniman H.): 'Algorithmic geometry' (Cambridge University Press), p. 541, ISBN: 13:9780521565295
23 ILOG CPLEX: High-performance software for mathematical programming. http://www.ilog.com/products/cplex/
24 Design Compiler. www.synopsys.com\products\logic\logic.html
25 Young-II, K.: 'TPartition: testbench partitioning for hardware accelerated functional verification', *Proc. IEEE Design and Test Comput.*, 2004, **21**, (6), pp. 484–493