Use of Iterative Weight-Age Constraint to Implement Dynamic Verification Components

by Mahak Singh, Design Engineer, Siddhartha Mukherjee, Sr. Design Engineer, Truechip

Conferences, discussions, or evaluations, engineers everywhere are asking the same questions:

• How to achieve the maximum coverage with randomization?
• What level of cross coverage to be checked for?

Achieving maximum randomization and coverage, how much of verification cycle time hit will it cost? Can I control it?

Today, each verification component is loaded with modes and features. How can we use these modes or features to create the maximum possible scenarios, and still be able to control the potential run time.

With a variety of modes and options in your verification component you need a way to fire them randomly to stress test the DUT. Here comes the importance of dynamism and the effectiveness of the EDA tools. How easily the verification component can change the mode depends on the types of SV constructs that the simulators support these days. SystemVerilog provides a way called weight-age constraints using which, one can implement dynamism in today’s verification components. Support for normal implementation of weight-age constraints is available across almost all simulators; however, Mentor’s QuestaSim is one of the most effective EDA tools, which implements iterative use of the weight-age constraint function where in you can randomize your configurations for a number of times in a loop.

For example:

```verilog
constraint c_dist {
    foreach (a[i])
        weight dist {{a[i]:b[i]} :/ c[i];} };
```

This type of constraint helps you create a nonlinear random distribution.

If you want a bathtub shaped distribution, high on both ends and low in the middle, you could build an elaborate distribution constraint.

The graph in Figure 1 shows how you can combine two exponential curves to make a bathtub curve.

For a perfect bathtub shape you may require lot of tweaking. This can be easily done by creating multiple sample points in the bathtub curve and Mentor’s support of iterative loop in weight-age constraints in their EDA tool.

In this article we will explain how weight-age constraints can be used for randomized verification with maximum coverage near the corners and some coverage in the middle, thus saving verification time and getting good coverage faster.

INTRODUCTION

With the growing complexity of system-on-chip and number of gates in design, verification becomes a huge task. With the increase in design complexity verification is also evolving and growing day by day. From an underdog to a champion it has conquered more than 70% of the man power and time of the whole tape out cycle. The case was not the same earlier when the designs were simple and less focus was given on verification. The industry was set for a change. More minds and companies jumped into the verification domain and within a decade it was a different ball game altogether; open groups, new ideas, better strategies backed up by global verification standards (such as IEEE standard). The verification enigma was about to reveal all its secrets. Let us have a look.

VERIFICATION IN LATE 19TH CENTURY

In the 1970’s automation had started taking place. We had our first design compiler by then. The concept of HDL coding was born with the advent of first design compiler. In 1980 Dracula was released, first physical design tool. Then came the first place and route tool. By 1980 major EDA giants like Synopsys, Mentor and Cadence were setting firm steps to rule the verification industry and hence front
end verification came into existence. Formal Verification was also introduced by the late 1990's by Bell Design Automation and later Verplay and Jasper came in the same field. By this time verification was not given much importance because the designs were not that complex. There were no concepts of functional verification and coverage driven verification. Directed Verification model was followed everywhere. It had many limitations and was prone to human errors.

**VERIFICATION IN 2000's**

In early 2000 and later part of the decade, design complexity was increased by almost 100% and it became very difficult for the designers to verify their designs. All around the globe people were fighting with verification hazards and the industry was looking yet another change. This was the time when HVLs were introduced. Specman was one of the early Hardware Verification Languages used by Cadence which remained a proprietary for the latter till 2005. SystemVerilog was also released by 2001 as a standard by IEEE. The EDA giants were also active and everybody had SV based simulation tools by this time. There were also many wave form viewers and protocol analyzers in the market for the ease of verification engineers. Later SystemC was also introduced further with the concept of synthesizing behavioral code, another effort to reduce the cycle time.

All these developments had affected the verification industry immensely and reduced the verification process much. New concepts like constraint driven random stimulus and coverage driven verification plans were adopted with these new standards. Though we had achieved much in the last decade, still something was missing. The design complexity still kept on growing and we still could not meet the verification challenges to the maximum.

In the present time we focus on Functional Verification and Coverage has a major role in that. It gives you the bottle neck for the non-ending verification process. Though the cycle time takes a hit in this process, much iteration is required to achieve this goal. Being a verification engineer and designing verification components (VIPs), it is my constant effort to make sure optimum quality is maintained and there should be minimum possible tradeoff in this path. Usually people have to compromise on either run time or memory usage if they go for quality and variety in their verification components. Providing multiple configurations in the components makes your VIP bulky and obviously takes more time to process. Increasing the number of modes will also increase the time to verify all the scenarios and also cracking 100% of the coverage matrix. Multiple iterations might be required to cover corner case scenarios. Even test plans and test vectors may change at this point and these last minute changes take much more time. So in short corner case testing requires more time and good quality of verification components.
OUR AIM
To propose a way for acute corner case testing in less time using iterative weight-age based constraints.

CONCEPT BEHIND IT
SystemVerilog provides exponential distribution which can be realized to make a bathtub shaped distribution (refer Figure 1) in constrained random stimulus.

BATHTUB CURVE GENERATION
In verification, it is always desired to generate maximum number of corner case scenarios. More number of corner case hits means we need to use a good simulation tool and some randomization logic to control the number of hits for a particular scenario. The number of corner case hits can be controlled by using a non-linear random distribution method. One such method is generating a bathtub shaped distribution.

Generating bathtub curve using SystemVerilog’s randomization techniques along with Mentor’s simulation tool is a powerful and efficient method to achieve all corner case scenarios. This approach uses less number of verification cycles for complex designs, most likely to show undesired behavior, on extreme limit stimuli. Using conventional or linear random distribution methods for the verification of such designs, will take a lot of time with less guarantee of robust testing. With such methods, the number of times of hitting a desired scenario will either be very less or zero. But, by using bathtub curve we can get the maximum hits of the scenarios where input values fall more in the range towards minimum and maximum of all possible values. This can be done by setting the probability of values in these ranges as highest and setting the probability of the values falling in the middle range as lowest so as to generate the desired values more often than others. Here comes the use of SystemVerilog’s iterative weight-age constraints.

ITERATIVE WEIGHT-AGE CONSTRAINT
An example of an iterative weight-age constraint is shown below:

```verilog
rand int value;
int a[5], b[5];
int c[5];
a = {1, 11, 21, 31, 41};
b = {10, 20, 30, 40, 50};
c = {30, 15, 10, 15, 30};
constraint c_value { foreach (a[i])
    value dist { [a[i]:b[i]] :/ c[i] };
}
```

Here, “c_value” is an iterative weight-age constraint which is assigning a weight “c[i]” for each range of the random variable named “value”. Five different ranges of values have been selected which are defined by “a[i]” and “b[i]”.

Using this basic iterative weight-age constraint, we have developed a logic for generating random values in selective ranges with desired probabilities at multiple sample points which is supported by Mentor’s QuestaSim Simulator.

BATHTUB CURVE GENERATION USING ITERATIVE WEIGHT-AGE CONSTRAINT: DEVELOPMENT OF LOGIC, TESTING AND PROOF OF CONCEPT
When the logic was tested with a simple testbench, we achieved the desired weighted random distribution of the corner case scenarios. We set the minimum and maximum values of the random variable “value” as ‘1’ and ‘50’ respectively. We set different weights for different ranges of the possible values. To achieve a proper bathtub curve much iteration and tweaking will be required. So selection of values for weights and their respective ranges must be done carefully. Using this method developed for bathtub curve generation with the help of iterative weight-age constraints along with the optimum distribution weights, we can definitely reduce the time to achieve the target of maximum coverage for the corner case scenarios with lesser repetitive efforts.

We took two arrays “count” and “hits” in our testbench with the size equal to the number of possible values of the random variable “value”:

```verilog
int count [1:50];
int hits [1:50];
```

We initialized the arrays to some default values:

```verilog
foreach (count[i])
    begin
        count[i] =i;
        hits[i] =0;
    end
```
Then in a “repeat” loop the task “bathtub_dist” was called which implements the randomization logic with iterative weight-age constraint. Every time this task is called the value of random variable “value” is updated in such a way so as to generate multiple sample point values for bathtub shaped distribution. Every time after getting a new randomized value of the random variable from function “bathtub_dist”, the array “hits” incremented to count the total number of hits for that particular value:

```
repeat (1500)
    begin
        bathtub_dist();
        hits[value]++;
    end
```

At the end, asterisks (*) were printed for all randomly generated values. For each value, asterisks were printed in a row for the number of times the value has been generated or hit:

```
foreach(count[i])
    begin
        $write("count[%0d]=%0d	", i, hits[i]);
        repeat(hits[i]) $write("*");
        $display;
    end
```

Finally, the output that we received was a bathtub curve of the weighted distribution of random values. Here, we can see the result in the format “count[i] = hits[i]” followed by asterisks which depicts which value has been hit how many times:

**APPLICATION IN REAL WORLD**

We can apply the above learnings in the real world. Let us take an example of an AXI Master DUT and an AXI Slave BFM. Now to stress test the DUT we need to randomize the signals and modes in the BFMs. Few of the possible options are:

- **Write Response Modes** : AXI supports four response types
  - OKAY
  - EX-OKAY
  - SLVERR
  - DECERR

Slave BFM should generate all these responses to test the master DUT. For this, there can be four modes in the Slave to generate all four types of responses. In a testcase, we can randomize the mode of Slave to randomly generate any of the response. But, if it is required that the responses “OKAY” and “SLVERR” should hit maximum number of times as compared to “DECERR”, here we can use bathtub distribution for random generation of modes.

- **xValid signals of the Slave BFM should be toggled after random delays.**

These random delays should be kept in ranges of very quick response time to a long response time. This is because if a slave responds after a fixed delay every time then we cannot say that the master DUT has been fully tested,
but in the case where random delays are generated using bathtub distribution we may very well expect that the DUT will sample all the responses from the slave in any condition.

From the above examples, we can conclude that dynamism comes into existence with the use of iterative weight-age constraint in our verification components.

**CONCLUSION**

From here we conclude that by usage of iterative weight-age constraints corner case testing can be done easily, in comparatively less time. As this is not the simplest style of coding, it needs simulator support for execution and requires skilled engineers to implement this methodology to stress test the DUT. It also improves the quality of product, both the DUT and any third party verification component. It allows the VIP developer to implement various testing modes and dynamism also comes into existence which provides better control over the modes from test cases.

**BENEFITS FOR USERS**

- SoC verification becomes fast
- More regress testing
- Good controllability
- Maximum corner case hits
- Dynamism
- Maximum coverage in less time
VERIFICATION ACADEMY

The Most Comprehensive Resource for Verification Training

20 Video Courses Available Covering
- Intelligent Testbench Automation
- Metrics in SoC Verification
- Verification Planning
- Basic and Advanced UVM
- Assertion-Based Verification
- FPGA Verification
- Testbench Acceleration
- Power Aware Verification
- Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 4400 topics
UVM Connect and UVM Express Kits

www.verificationacademy.com