International Journal of Control and Automation
Vol.7, No.2 (2014), pp.209-220
http://dx.doi.org/10.14257/ijca.2014.7.2.20

A Specified Coverage Analysis Method Used For Comparison Logic

Lirong Qiu

School of Information Engineering, Minzu University of China, Beijing, China
qiu_lirong@126.com

Abstract

The commonly used hardware verification languages, such as SystemVerilog have explicit coverage constructs which entitle verification engineers the power to model the function of design into coverage requirement. Verification Engineers usually subtract the functional coverage from their knowledge of the design requirements and map them into coverage points which can be expressed by hardware verification languages. However due to the limitation of syntax of these hardware verification languages, it is very hard in a reasonable way to specify the coverage points for the kind of module whose output is decided by scale-size relationship of its inputs. This paper presents a workaround to analyze the total effective scenarios for this kind of module. The paper also provides a method on how to model these effective scenarios with current syntax of hardware verification language.

Keywords: *SystemVerilog, coverage, scale-size relationship, comparison logic, coverage driven verification*

1. Introduction

The advent of new VLSI technology and SoC design methodologies has brought an explosive growth in the complexity of modern electronic circuits. As a result, functional verification has become the major bottleneck in any digital design flow [1]. In order to overcome this bottleneck, not only the methodology and techniques used for complex hardware system and SoC verification is greatly developed [3, 7, 12], but also abundant of well-defined modules and reusable IP are widely used to guarantee the quality of the whole chip [2, 11]. For highly reusable modules, they are preferred to be verified as completely as possible, as they may be used by different hardware systems [13]. The best wish is that quality of modules or reusable IP is not a concern before these modules or IP are integrated into high level system. We just assume they work correctly under correct input.

Just like a system, modules and reusable IP are also formed by lots of sub-modules. Some sub-modules or even modules themselves are just to process basic functions such as basic calculations or logic judgment. Sometimes these sub-modules may act so important role that if something goes wrong with them there will be a big impact on top-level modules. So such modules are usually given a thorough verification to make sure they can work properly under all kinds of scenarios. For a coverage-driven random based verification system, functional coverage, as the explicit functional requirements derived from the device and test plan specifications, can be an important feedback to verification engineers [8]. Verification engineers can use functional coverage to find the overlooked scenarios. But in some cases, the module's function is quite hard to be mapped into the coverage points with verification language syntax. Modules whose output are decided by scale-size relationship of its inputs are just one kind of these modules. Here is a simple example:

```
module func(clk, start, rst_n, ia, ib, ic, id, q );
input [7:0] ia, ib, ic;
input clk, start, rst_n;
output [31:0] q;
reg[1:0] q;
reg[1:0] cnt;
reg[31:0] d0, d1;
wire[31:0] cmp1,cmp2 ;
assign cmp1 = (cnt==0) ? ib : ic;
assign cmp2 = (cnt==0) ? ia : d0;
always @(posedge clk or negedge rst_n) begin
    if(!rst_n)
        cnt<=2;
    else if(start)
        cnt<=0;
    else if (cnt<2)
        cnt<=cnt++;
end
always @(posedge clk) begin
    if(cmp2>cmp1)    begin
        d0<=cmp2;
        if(cnt==1) q<=0;
    end else if(cmp2==cmp1) begin
        d0 <=id;
        if(cnt==1) q<=1;
    end else begin
        d0<=cmp1;
        if(cnt==1) q<=2;
    end
end
endmodule
```

In the above example, asserting of start signal will start a comparison process controlled by cnt. Output q of module func will be decided by scale-size relationship of its inputs ia, ib, ic and id. The syntax of defining coverage points in SystemVerilog is like "bins case1 =

{[7:100]}" which is for possible value range of one data or like "bins trans = (s0=>s1), (s2=>s3);" which is for state transition [9]. A simple way to follow SystemVerilog syntax is to make each possible value of ia, ib, ic and id into one bin and then cross them. But the coverage space will be huge. Putting so much energy to verify such a huge coverage space is not acceptable as most of them are redundant and can be replaced by the coverage point with the specified function mean. And code coverage is not dependable either, as some original inputs' comparison is hidden by the intermediate variable cmp1, com2 and the second round comparison when cnt is 1. So code coverage can't cover all the functional scenarios.

In fact from first thought, we can manually model the scenarios by making scale-size relationship of inputs into a list table and use an intermediate variable to record it. Here is the list:

When ia = ib = ic= id, x1=1;

When ia < ib = ic =id, x1=2;

When ia < ib < ic = id, x1=3;

.....

When ia > ib > ic > id, x1=m;

Then set "bin x1 = {[1: m]} as coverage points in test bench. That is exact coverage definition needed for this module's function coverage. However if the inputs is increased to more ones such as 6, it will turn to an impossible job to make the list table.

Another commonly used syntax for coverage definition, cross coverage, is not the answer either. When two data are compared, we can get three relationships: bigger, smaller and equal. But if there are 3 data need to be compared with each other, we could not simply cross the two-two scale-size relationships as there are some unreasonable cases such as a<b, b<c a>c. With the number of compared data increasing, more and more such unreasonable cases will appear. It is impossible to filter these unreasonable cases manually. What make things worse is that each scale-size relationship has almost the same importance as the other one. Neither of them can be missing.

How to implement the program about the coverage points for DUT's verification whose output is decided by its inputs' scale-size relationship, as a mere engineering technique issue, there is nearly no paper which is made research on it. This paper provides its study on this issue. The major contribution in this paper is as followed.

- By transferring the scenario of such specified DUT's verification into point position issue on a line segment position, a method is proposed on how to calculate the total number of scenarios. The statistic on the scenarios provides a good verify on the coverage point implementing method.
- An easy implemented coverage definition method with verification language is proposed. In this method coverage points are defined by crossing all inputs' possible position and the unreasonable coverage bins are excluded in a way to be hit in advance by making a special test case.

The remainder of this paper is structured as three sections. In section I, the model of scenarios and deducing process of equation of calculating total number of scenarios is introduced. In section II, the method to mark the coverage points with current verification language is introduced. The related work and conclusion will be given in section III.

2. Method to Calculate Total Number of Scenarios

Before talking about the way to list scenarios, let's see how many scenarios the coverage should have for the module whose output is decided by scale-size relationship of its inputs. The statistic on the scenarios can give us an evaluation on how many test sets should be used in our test suits. Moreover the statistic can be used verify our automatic list method.

2.1. Model the scenarios

To solve this problem, we model such module as a functions $y = F(x_1, x_2, \ldots, x_n)$ (Here capital F is used because it stands for the classified function whose output is decided by scale-size relationship of inputs). As the output y is determined by the comparison of inputs: $x_1, x_2, \ldots, x_n$ we can regard each input as one point on line segment. Then the processed scenarios of this function can be regarded as the position relationship on the line segment. Just like the Figure 1 shows and Figure 2 shows. When no input is equal to any of the other one, there will be totally n points on the line segment, like the Fig 1 shows. As some inputs may be equal to each other, the corresponding points of these inputs will be overlapped on the line segment and t points can be got on the line segment. Here $1 \le t < n$, just like Figure 2 shows.

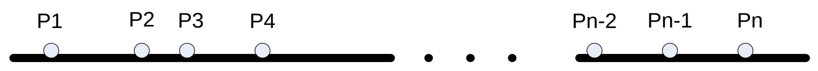


Figure 1. no point is overlapped with each other

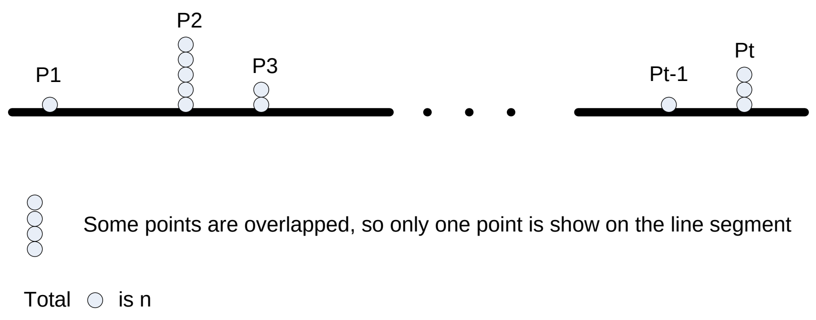

P2
P1
P3
Pt
Pt-1
Some points are overlapped, so only one point is show on the line segment
Total ○ is n

Figure 2. Some points are overlapped, only t points are showed on line segment

2.2. Recurrence derivation

After the scenario of the function is model as points' position in a line segment, a recurrence relationship can be found to help us on the calculation of the statistic of all scenarios.

We define Z_n as the total number of scenarios of function which has n inputs and define the total number of scenarios that t inputs are not equal to each other is $z_{n,t}$, here $1 \le t \le n$. Obviously, we can get

$$Z_n = z_{n,1} + z_{n,2} + z_{n,3} + \ldots + z_{n,n-1} + z_{n,n}$$

$$Z_{n-1} = z_{n-1,1} + z_{n-1,2} + z_{n-1,3} + \ldots + z_{n-1,n-2} + z_{n-1,n-1}$$

We assume that $Z_{n-1}, z_{n-1,1}, z_{n-1,2}, z_{n-1,3}, \ldots, z_{n-1,n-2}, z_{n-1,n-1}$ are all known values.

As Figure 3 shows, when one point is inserted to the line segment which has total n-1 points and t points left as un-overlapped, we will get one scenario which belongs to F function with n inputs. As the added point has t choices to be inserted onto the t existing un-overlapped points, which means this input is equal to some other inputs, we can get $t * z_{n-1,t}$ new scenarios and the new scenarios are all belongs to the cases that there are n points on line segment and t points are left un-overlapped. We mark them as $z^{2}_{n,t}$.

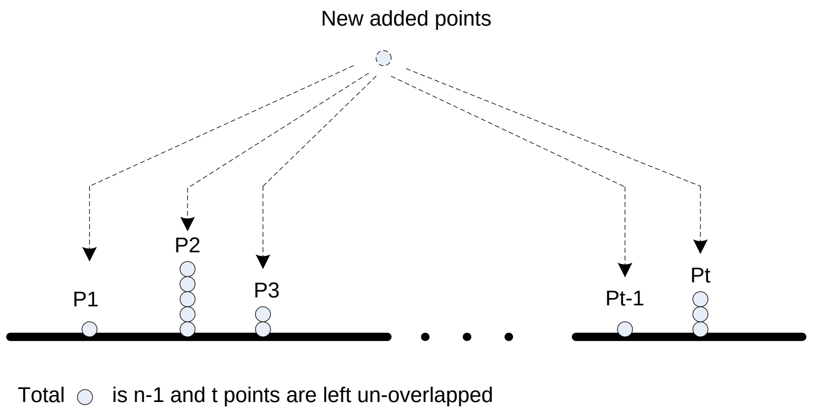


Figure 3. Added points has t choices to be inserted onto t existing un-overlapped points

Meanwhile as Figure 4 shows, the added point has t+1 choices to be inserted to the t+1 divided sub line segments by existing un-overlapped points, which means this input is not equal to any other inputs, we can get $(t + 1) * z_{n-1,t}$ new scenarios and the new scenarios are all belongs to the cases that there are n points on line segment and t+1 points are left un-overlapped. We marked them as $z^{1}_{n,t+1}$

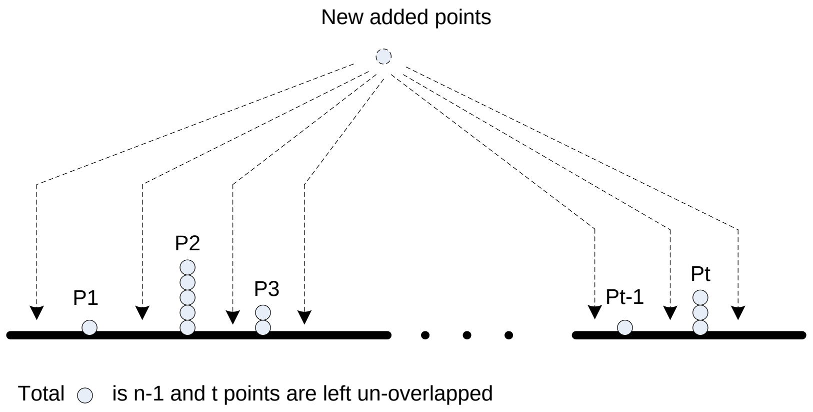


Figure 4. Added points has t+1 choices to be inserted the divided sub line segment by existing un-overlapped points

Following this rule, we can get:

When t=1, that is 1 points left un-overlapped in total n-1 points on line segment, an point insertion on such line segment can make $z^2_{n,1} = 1 * z_{n-1,1}$ and $z^1_{n,1+1} = (1+1) * z_{n-1,1}$

When t=2, that is 2 points left un-overlapped in total n-1 points on line segment, an point insertion on such line segment can make $z^2_{n,2} = 2 * z_{n-1,2}$ and $z^1_{n,2+1} = (2+1) * z_{n-1,2}$

…...

When t=m-1, that is m-1 points left un-overlapped in total n-1 points on line segment, an point insertion on such line segment can make $z^2_{n,m-1} = (m-1) * z_{n-1,m-1}$ and $z^1_{n,m} = m * z_{n-1,m-1}$

When t=m, that is m points left un-overlapped in total n-1 points on line segment, an point insertion on such line segment can make $z^2_{n,m} = m * z_{n-1,m}$ and $z^1_{n,m+1} = (m+1) * z_{n-1,m}$

When t=m+1, that is m +1points left un-overlapped in total n-1 points on line segment, an point insertion on such line segment can make $z^2_{n,m+1} = (m+1) * z_{n-1,m+1}$ and $z^1_{n,m+2} = (m+2) * z_{n-1,m+1}$

……

When t=n-2, that is n-2 points left un-overlapped in total n-1 points on line segment, an point insertion on such line segment can make $z^2_{n,n-2} = (n-2) * z_{n-1,n-2}$ and $z^1_{n,n-2+1} = (n-2+1) * z_{n-1,n-2}$

When t=n-1, that is n-1 points left un-overlapped in total n-1 points on line segment, an point insertion on such line segment can make $z^2_{n,n-1} = (n-1) * z_{n-1,n-1}$ and $z^1_{n,n-1+1} = (n-1+1) * z_{n-1,n-1}$

To add all the new got scenarios number up Zn can be got. So we will have

$$\begin{aligned}
&Z_n \\
&= z_{n,1} + z_{n,2} + z_{n,3} + . + z_{n,m} + z_{n,m+1} + ... + z_{n,n-1} + z_{n,n} \\
&= z^2_{n,1} + (z^1_{n,1+1} + z^2_{n,2}) + z^1_{n,2+1} + ... + \\
&z^2_{n,m-1} + (z^1_{n,m} + z^2_{n,m}) + (z^1_{n,m+1} + z^2_{n,m+1}) + z^1_{n,m+2} \\
&+ ... + z^2_{n,n-2} + (z^1_{n,n-2+1} + z^2_{n,n-1}) + z^1_{n,n-1+1} \\
&= 1 * z_{n-1,1} + ((1+1) * z_{n-1,1} + 2 * z_{n-1,2}) + (2+1) * z_{n-1,2} + ... + \\
&(m-1) * z_{n-1,m-1} + (m * z_{n-1,m-1} \\
&+ m * z_{n-1,m}) + ((m+1) * z_{n-1,m} \\
&+ (m+1) * z_{n-1,m+1}) + (m+2) * z_{n-1,m+1} \\
&+ ... + (n-2) * z_{n-1,n-2} + ((n-2+1) * z_{n-1,n-2} + (n-1) * z_{n-1,n-1}) + (n-1+1) * z_{n-1,n-1}
\end{aligned}$$

From the formula above, we also get

$z_{n,1} = z^2_{n,1} = 1 * z_{n-1,1}$

$z_{n,2} = (z^1_{n,1+1} + z^2_{n,2}) = (2 * z_{n-1,1} + 2 * z_{n-1,2})$

……

$z_{n,m} = (z^1_{n,m} + z^2_{n,m}) = (m * z_{n-1,m-1} + m * z_{n-1,m})$

$z_{n,m+1} = (z^1_{n,m+1} + z^2_{n,m+1}) = (m+1) * z_{n-1,m} + (m+1) * z_{n-1,m+1}$

……

$z_{n,n-1} = (z^1_{n,n-2+1} + z^2_{n,n-1}) = ((n-1) * z_{n-1,n-2} + (n-1) * z_{n-1,n-1})$

$z_{n,n} = z^1_{n,n-1+1} = n * z_{n-1,n-1}$

2.3. Original condition for recurrence derivation

It is also quite easy to calculate the original condition:

When total number points on line segment is 2, 2 points left un-overlapped, $z_{2,2} = 2$。

When total number points on line segment is 2, 1 points left un-overlapped, $z_{2,1} = 1$。

With this recursive rule and the original condition, we can calculate the total number of scenarios with computer.

2.4. Calculation of an example with 6 inputs

Here we take an F function with 6 inputs as an example to show how to calculate the total scenarios

For 2 inputs

$z_{2,1} = 1$, $z_{2,2} = 2$

$Z_2 = z_{2,1} + z_{2,2} = 3$

For 3 inputs

$z_{3,1} = z_{2,1} = 1$, $z_{3,2} = 2 * z_{2,1} + 2 * z_{2,2} = 6$, $z_{3,3} = 3 * z_{2,2} = 6$

$Z_3 = z_{3,1} + z_{3,2} + z_{3,3} = 13$

For 4 inputs

$z_{4,1} = z_{4,1} = 1$, $z_{4,2} = 2 * z_{3,1} + 2 * z_{3,2} = 14$, $z_{4,3} = 3 * z_{3,2} + 3 * z_{3,3} = 36$, $z_{4,4} = 4 * z_{3,3} = 24$

$Z_4 = z_{4,1} + z_{4,2} + z_{4,3} + z_{4,4} = 75$

For 5 inputs

$z_{5,1} = z_{4,1} = 1$, $z_{5,2} = 2 * z_{4,1} + 2 * z_{4,2} = 30$, $z_{5,3} = 3 * z_{4,2} + 3 * z_{4,3} = 150$, $z_{5,4} = 4 * z_{4,3} + 4 * z_{4,4} = 240$, $z_{5,5} = 5 * z_{4,4} = 120$

$Z_5 = z_{5,1} + z_{5,2} + z_{5,3} + z_{5,4} + z_{5,5} = 541$

For 6 inputs

$z_{6,1} = z_{6,1} = 1$, $z_{6,2} = 2 * z_{5,1} + 2 * z_{5,2} = 62$, $z_{6,3} = 3 * z_{5,2} + 3 * z_{5,3} = 540$, $z_{6,4} = 4 * z_{5,3} + 4 * z_{5,4} = 1560$, $z_{6,5} = 5 * z_{5,4} + 5 * z_{5,5} = 1800$, $z_{6,6} = 6 * z_{5,5} = 720$

$Z_6 = z_{6,1} + z_{6,2} + z_{6,3} + z_{6,4} + z_{6,5} + z_{6,6} = 4683$

So with 6 inputs, we will get 4683 scenarios. What a tough job if we make the coverage table manually! Statistic on scenarios gives us a good evaluation on the overload of verification effort.

3. Method to Mark the Coverage Points

In Chapter 2 we have discussed the difficulty on how to model the scenarios of the modules whose output is determined by scale-size relationship of inputs into System Verilog coverage point syntax. Now we still face this problem.

3.1. Make a cross coverage based on each point's possible position

Look back chapter 3, one important clue to find the statistic of scenarios is we regard each input of the module as one point on line segment. So one processed scenarios of such module can be regarded as all points position relationship on the line segment. Following this clue, we can map the scale-size relationship scenario into System Verilog coverage syntax easily: If the module has n inputs which should be compared with each other, each input will have n possible positions on the line segment. Mark the position with position number on line segment, we get coverage point for one input and its value range is from 0 to n-1 (here we mark first position as 0, second position as 1 and other position can be listed in turn). As every input has n possible positions too, we need to cross them. Then we can get a big set which will include all the scenarios for the module. The testbench program is quite easy to be put down. Let take module in chapter 1 as an example, the coverage should be like this

Coverpoint ia {Bins ia [] = = {[0:3]} ;}

Coverpoint ib {Bins ib [] == {[0:3]} ;}

Coverpoint ic {Bins ic [] == {[0:3]} ;}

Coverpoint id {Bins id [] == {[0:3]} ;}

Cross ia, ib, ic, id;

When we do coverage collection, we should sort ia, ib, ic and id and sample the each one's position number. For example, for the case ia=10, ib=28, ic=20, id=92, ia's position is 0, ib's position is 2, ic's position is 1, ic's position is 3. One point of (0, 2, 1, 3) will be sampled to the cross coverage.

3.2. Exclude the unreasonable bins

However you will find a hole for this coverage collection or coverage point's definition. Looking at this case: ia = 10, ib = 20, ic = 20, id = 92, the position set will be (0, 1, 1, 2). We will not mark the position set to be (1, 2, 2, 3). Similarly we also have cases like ia = ib = ic = id, we can only mark the position set to be (0,0,0,0) not (1,1,1,1), (2,2,2,2), (3,3,3,3). The sorting operation in coverage collection made the reasonable position set follows a sorting rule which is quite like we rank a test score of one class students: a large rank can only be used when all little ranks are occupied. Unreasonable position set doesn't follow this sorting rule. However these position sets are one part of the crossed bins of ia, ib, ic, id and they could never be hit if we use above method to do coverage collection.

Obviously excluding them manually just like making the coverage list table will not be a good choice. Putting some unreasonable position sets into line segment again and examining these line segments, you will find a rule which make this position set so special: check each point on line segment, you will find there is at least one point which has empty position before it. See Figure 5 show two examples. One is the position set for (1,2,2,3) which has position 0 not filled, the other is the position set for (3,3,3,3) which has position 1 and position 2 not filled. According this rule, we can easily exclude the unreasonable cases from the big set got by crossing every point's possible position number. For example make a little perl program to print these set into ignore bins. But the drawback of this method is we will get a huge table for ignore bins in our testbench.

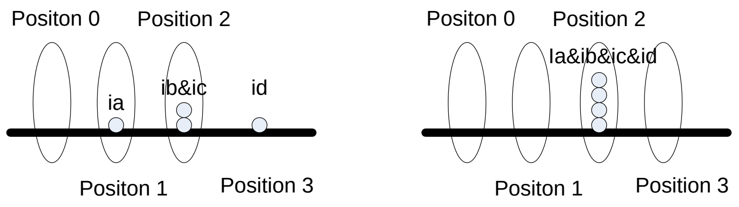
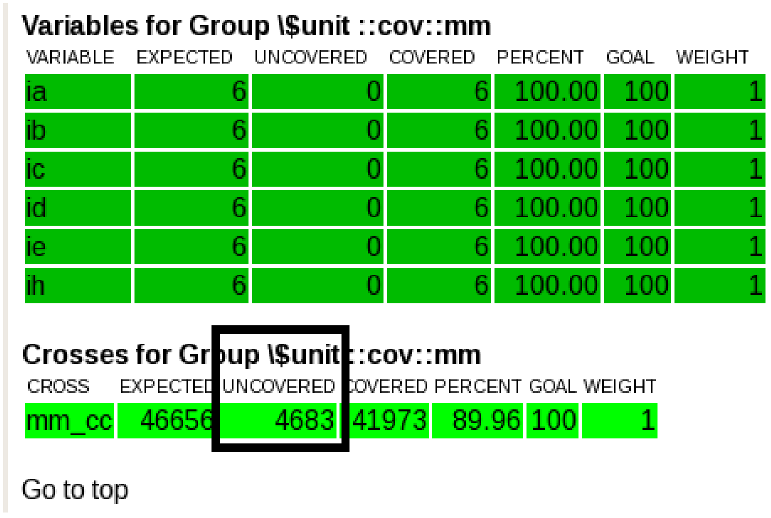


Figure 5. line segment with position set (1, 2, 2, 3) and (3, 3, 3, 3)

Another method is to make a program to process the collected coverage database by either excluding the unreasonable coverage point or making the unreasonable coverage point hit. But additional knowledge about the coverage database format is needed for this job. In fact we have easier way to make the unreasonable coverage point hit: make a special case. Such special case is not to test our DUT. It just set sample variable in coverage point to the values belong to the unreasonable position set and them sample them. Such case can generate a special coverage database which will cover these bins we don't case.

3.3 Coverage reports for an example with 6 inputs

Following above method and using a 6 inputs function as an example, a testbench program based on SystemVerilog is developed. The program will define coverage points by crossing all 6 inputs 6 possible positions and call a function to fill the unreasonable bins. We use VCS to run the program and URG command to get the coverage report. Figure 6 is a snapshot of the coverage report. From the coverage report, we can see total number of uncovered bins is the coverage points left for verification job, which is 4683. This result matches with the conclusion in Chapter 2 and proves the correctness of this method. The total number of covered bins is 41973 which backups our conclusion that it is a tough job to exclude the unreasonable bins manually.



Variables for Group \$unit ::cov::mm

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
ia	6	0	6	100.00	100	1
ib	6	0	6	100.00	100	1
ic	6	0	6	100.00	100	1
id	6	0	6	100.00	100	1
ie	6	0	6	100.00	100	1
ih	6	0	6	100.00	100	1

Crosses for Group \$unit ::cov::mm

CROSS	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
mm_cc	46656	4683	41973	89.96	100	1

Go to top

Figure 6. A snapshot of coverage report

4. Related work and Conclusion

The logic used in digital circuit design can basically classify into three kinds of logics: mathematic logic such as “+, -, *”, logical logic such as “&&, ||, &” and comparison logic such as “>, <, =” [5,10]. For mathematic logic and logical logic, it is easier to map the function coverage into coverage points with verification language syntax. We can use the value range of inputs or intermediate results to mark the coverage points. We can slice the value range of inputs and intermediate results to several bins and choose largest value or smallest value as the corner case. There is no need to cover each possible value. So coverage space can be greatly reduced without worry about scenarios missing. But for comparison logic we lost the direct map from the logic’s function to verification language’s coverage syntax. This paper provides a method on how to model these effective scenarios with current syntax of hardware verification language. Moreover it also provides a mathematic method to calculate the statistic of scenarios. As comparison logic is widely used in digital circuit design, the coverage definition method presented in this paper can be good reference to the coverage point definition for digital circuit design.

Acknowledgements

Our work is supported by the Program for New Century Excellent Talents in University (NCET-12-0579) and the “Science Fund for Youths” project of Minzu University of China (No. 1112KYQN39).

References

[1] K. R. G. da Silva, E. U. K. Melcher, I. Maia and H. N. Cunha, “A methodology aimed at better integration of functional verification and RTL design”, Journal of Design Automation for Embedded Systems, (**2005**).
[2] C. Kuznik and W. Muller, “Functional Coverage-driven Verification with SystemC on Multiple Level of Abstraction”, Proceedings of DVCON, (**2011**).
[3] S. M. Mohan and J. C. Narayana Swamy, “VMM Based Constrained Random Verification of an SOC Block”, International Journal of Advances in Engineering & Technology, (**2012**).
[4] S. Saponara, L. Fanucci and M.Coppola, “Design and coverage-driven verification of a novel network-interface IP macrocell for network-on-chip interconnects”, Microprocessors and Microsystems, vol. 35, **(2011)**, pp. 579-592.

[5] V. Subedha and S. Sridhar, "An Efficient Coverage Driven Functional Verification System based on Genetic Algorithm", European Journal of Scientific Research, vol. 4, no. 81, (**2012**).
[6] C. Spear, Editor, "Systemverilog for Verification", In Plastics, Springer, New York, (**2006**).
[7] P. Rashinkar, P. Paterson and L. Singh, "System on Chip Verification – Methodology and Techniques" , Kulwer Publishers, (**2001**).
[8] A. Piziali, "Functional Verification Coverage Measurement and Analysis", In Plastics, Kluwer Academic, New York, (**2004**).
[9] Design Automation Standards Committee: IEEE Std 1800-2009, "IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language", In 3 Park Avenue, New York: The Institute of Electrical and Electronics Engineers, Inc.
[10] J. Bergeron, E. Cerny, A. Hunter and A. Nightingale, (Eds.), "Verification Methodology Manual for SystemVerilog", In Plastics, Springer, New York, (**2005**).
[11] S. J. E. Wilton and R. Sale, "Programmable logic IP cores in SoC design: opportunities and challenges" In Proceedings of the IEEE 2001 Custom Integrated Circuits Conference, (2001), San Diego, CA, (**2001)** May 06-09.
[12] Y. -N. Yun, J. -B. Kim, N. -D. Kim and B. Min, "Beyond UVM for practical SoC verification", In proceedings of SoC Design Conference (ISOCC), **(2011)**, November 17-18; Jeju
[13] B. Stohr, M. Simmons and J. Geishauser, "FlexBench: reuse of verification IP to increase productivity", In Proceeding of Design, Automation and Test in Europe Conference and Exhibition, **(2002)** March 04-08; Paris.
[14] Y. Guo, W. Qu, T. Li and S. Li, "Coverage Driven Test Generation Framework for RTL Functional Verification", In Proc. 10th IEEE International Conf. Computer-Aided Design and Computer Graphics, **(2007)** October 15-18; Beijing, China.

Author

Lirong Qiu. She received his M.Sc. in Computer Sciences (2004) and PhD in Information Sciences (2007) from Chinese Academy of Science. Now she is full professor of computer sciences at Inforamation Engineering Department, Minzu University of China. Her current research interests include different aspects of Computer-Aided Design, Artificial Intelligence and Distributed Systems.