2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Fully Automated Functional Coverage Closure

Manohar Kodi, +917675020009, Nvidia Graphics Pvt Ltd, Hyderabad, India (mkodi@nvidia.com)
Sagar Sudam Patil, +918123172587, Nvidia Graphics Pvt Ltd, Hyderabad, India (sagarsudamp@nvidia.com)
Ranjith Nair, +917893955664, Nvidia Graphics Pvt Ltd, Hyderabad, India (ranjithn@nvidia.com)

Abstract— Functional coverage is the main metric for measuring stimulus quality in a metric driven verification (MDV). Verification engineers use it to sign off the complex design verification features. Constraint random tests are more comprehensive than traditional directed tests. To get coverage closure, large numbers of simulations are required. It is then necessary to merge the coverage data from those simulations which can take a significant amount of time and effort, only to realize that total coverage is still not 100%. This becomes an iterative process where coverage is analyzed, constraints updated, and tests rerun till the target is achieved.

This paper covers automated functional coverage closure by regenerating constraints based on uncovered functional coverage points from previous functional coverage runs. This is achieved using system functions to get the merged coverage numbers from each run. Coverage points are also written in a systematic way which will be further explained in detail in this paper.

Keywords— Functional Coverage, UVM, Coverage Convergence, System Verilog.

I. INTRODUCTION

Today's RTL is growing in complexity but due to faster time to market needs verification cycle is shrinking. Functional coverage is a very important and key metric in verification closure. Verification engineers write constraint random sequences to drive the design and collect coverage to ensure early convergence. Despite domain specific language support, advanced analysis tools and over 15 years of functional coverage history in HVLs, many engineers still find it difficult to develop functional coverage models effectively for today's projects. Even using methodologies such as the UVM, and the built-in System Verilog features for functional coverage, implementation can be inconsistent and error-prone. Sometimes, little or no consideration is given to results analysis. This results in un planned project delays due to delay in coverage closure.

The solution is not simply to add, or fix, functional coverage features in whatever language is being used, nor is it to develop new tools. The real solution is for our developers to better understand the actual requirements and come up with techniques needed for the correct modeling of functional coverage. They must recognize and understand the different kinds of functional coverage, from basic coverage of operational states, through complex use-cases, all the way to the mapping of results to a plan/specification. Given the trust we put in functional coverage results for tape out decisions, this is an oddly overlooked requirement. An essential component of modeling is to make the review and accurate analysis of results as painless as possible.

Implementing coverage has probably never been more straightforward. Cover groups allow a high degree of automated bin generation, and we're even offered essentially free coverage from register packages and the like. The reality, though, is that behind every coverage definition there is a cost. It takes time and effort to design coverage, collection, reach coverage goals, as well as to check and analyze the results.

This paper outlines the method to automate functional coverage closure with fast convergence, efficient verification resources (LSF slots, simulation time and disk space) and controllability. The focus of this paper is on a method to reduce the number of random test runs required to hit the 100% functional coverage.

II. AUTOMATIC FUNCTIONAL COVERAGE CLOSURE METHOD

Getting functional coverage closure in a short time is hard. There are several challenges in getting functional coverage closure. We have created many constraint random tests to cover a more comprehensive verification than traditional directed tests. To reach 100% functional coverage with a large regression, we must run each test with several different seeds (the number is usually difficult to quantify). This will lead to a huge number of test threads in the regression, and then it needs more server LSF slots, regression time, disk space and will generate huge coverage databases which will slow down database. In most cases, the total functional coverage still might not reach 100%. We must override the constraints and rerun a specific test to cover the holes based on manual coverage analysis. And finally, we have to merge the coverage again. This definitely wastes a lot of time, LSF slots and additional effort, all of which are costly. In this paper, to address this challenge, we propose an automatic functional coverage closure method which will speed up the process.

The verification team needs to identify which design features needs functional coverage points, determine functional coverage goals, develop coding guidelines used to instrument functional coverage points, and determine a methodology for running tests and gathering functional coverage metrics to determine if their functional coverage goal has been met.

Traditional functional coverage closure involves below steps:

1. Identifying cover groups and points.
2. Coding Cover groups and bins.
3. Running regressions.
4. Generating reports and analyze.
5. Modifying constraints manually and rerun the regressions.
6. Repeat steps 4 and 5 until we hit 100%.

Fully automated functional coverage closure method involves below steps:

1. Identifying cover groups and points.
2. Generate the cover points and algorithm.
3. Running regression which makes coverage 100%.

With Fully Automated functional coverage closure method we are avoiding few steps which involves more manual intervention, LSF slots and inefficient simulation runs.

Identifying cover groups and points:

Based on the feature list, verification engineer identifies the cover groups and corresponding cover points that are to be coded as part of functional coverage. These cover groups, corresponding cover points and bins are added in an excel sheet.

Generate the cover points and algorithm:

Excel sheet containing information about cover groups, cover points and bins ~~is~~ used to code cover points such that each cover point has only one bin. We have a script which takes the excel sheet as input and generates the code in the formatted needed.

First let's see the differences in the writing the functional cover points in the traditional method vs Automated functional coverage closure method. In traditional functional coverage closure, we have to write the functional cover points in a cover group as shown in the below figure1, but in the Automated functional coverage closure method we have to write each bin corresponds to a particular variable as separate cover point in a cover group like the code shown below in the figure2.

```
class sample_coverage_monitor extends uvm_monitor;
  bit var_1;
  bit [1:0] var_2;
  bit [2:0] var_3;
covergroup TraditionalSampleCovGrp;
   option.per_instance = 1;
   cov_var_1            : coverpoint var_1 {bins    b0 = {0};
                                            bins    b1 = {1};}
   cov_var_2            : coverpoint var_2 {bins    b0 = {0};
                                            bins    b1 = {1};
                                            bins    b2 = {2};
                                            bins    b3 = {3}; }
   cov_var_3            : coverpoint var_3 {bins    b0 = {0};
                                            bins    b1 = {1};
                                            bins    b2 = {2};
                                            bins    b3 = {3};
                                            bins    b4 = {4};
                                            bins    b5 = {5};
                                            bins    b6 = {6};
                                            bins    b7 = {7};}
   cross_cov_var_1_cov_var_2 : cross cov_var_1,cov_var_2;
  endgroup
endclass
```

Figure-1: Example cover group.

In the test, coverage database is accessed using system tasks provided in system Verilog LRM. We have defined two-dimensional array of cover groups and corresponding cover point values. For normal cover points corresponding bin value is stored in the array and for cross cover points combined value of both the bin values contributing to the cross point is stored as single value in the array.

A target coverage number above which the fully automated coverage closure method is to be applied is defined by the user. Before starting any sequence run in the test, the merged coverage database is analyzed and gets the data on how much percentage has been covered. If coverage percentage is less than targeted number defined to apply this method, the configuration class is randomized among all possible values based on initially defined constraints. If analyzed coverage percentage is more than or equal to the target percentage then the algorithm of fully automated coverage closure starts.

In this algorithm each cover point is checked to see if it is hit or not using the system task. If it is hit then cover point value is deleted from the array containing the cover points values. Then configuration is randomized among the remaining uncovered cover point values in the array. This algorithm runs for each run of the random test and keeps deleting the covered bins from the array and randomizes among the remaining values.

We developed a script that generates the task which is used to modify constraints in the test based on coverage database loaded and individual bin coverage.

Running regression which makes coverage 100%:

We developed a script which runs regression, and analyzes the logfile of the latest test run and breaks the regression once it sees 100% functional coverage. After each set of test command line runs the script merges the coverage data base of all the previous runs.

```
class sample_coverage_monitor extends uvm_monitor;

 bit var_1;
 bit [1:0] var_2;
 bit [2:0] var_3;

 covergroup SampleCovGrp;

  option.per_instance = 1;
  cov_0_var_1              : coverpoint var_1 {bins b = {0};}
  cov_1_var_1              : coverpoint var_1 {bins b = {1};}
  cov_0_var_2              : coverpoint var_2 {bins b = {0};}
  cov_1_var_2              : coverpoint var_2 {bins b = {1};}
  cov_2_var_2              : coverpoint var_2 {bins b = {2};}
  cov_3_var_2              : coverpoint var_2 {bins b = {3};}
  cov_0_var_3              : coverpoint var_3 {bins b = {0};}
  cov_1_var_3              : coverpoint var_3 {bins b = {1};}
  cov_2_var_3              : coverpoint var_3 {bins b = {2};}
  cov_3_var_3              : coverpoint var_3 {bins b = {3};}
  cov_4_var_3              : coverpoint var_3 {bins b = {4};}
  cov_5_var_3              : coverpoint var_3 {bins b = {5};}
  cov_6_var_3              : coverpoint var_3 {bins b = {6};}
  cov_7_var_3              : coverpoint var_3 {bins b = {7};}
  cross_cov_0_var_1_cov_0_var_2 : cross cov_0_var_1,cov_0_var_2;
  cross_cov_0_var_1_cov_1_var_2 : cross cov_0_var_1,cov_1_var_2;
  cross_cov_0_var_1_cov_2_var_2 : cross cov_0_var_1,cov_2_var_2;
  cross_cov_0_var_1_cov_3_var_2 : cross cov_0_var_1,cov_3_var_2;
  cross_cov_1_var_1_cov_0_var_2 : cross cov_1_var_1,cov_0_var_2;
  cross_cov_1_var_1_cov_1_var_2 : cross cov_1_var_1,cov_1_var_2;
  cross_cov_1_var_1_cov_2_var_2 : cross cov_1_var_1,cov_2_var_2;
  cross_cov_1_var_1_cov_3_var_2 : cross cov_1_var_1,cov_3_var_2;

 endgroup

endclass
```

Figure-2: Example cover group for automated coverage closure.

The coverage system tasks used in the random test:
$load_coverage_db (name) — Load from the given filename the cumulative coverage information for all coverage group types.

$get_coverage ( ) — Returns as a real number in the range 0 to 100 the overall coverage of all coverage group types. This number is computed as described above.

get_coverage() — Calculates type coverage number (0...100) for each cover group or cover point or cross.

The sample code snippet in Figure-3 is the algorithm used in the random test to delete the covered cover bin values from the array and randomize the remaining bin values.

```
class nitro_ether_cov_test extends nitro_ether_base_test;
  `uvm_component_utils(nitro_ether_cov_test)
  class_cfg_1 cfg_1;
  int directed_t; // 0 -> complete rand , 1 --> directed random , 2 --> directed
  int array_num_point[$] ;
  int array_cov_value[1][$];
  rand bit [2:0] var_1_2;
  real target_coverage_for_automation_start;
  function new(string name, uvm_component parent = null);
     super.new(name, parent);
     cfg_1 = new("cfg_1");
     array_num_point.push_back(8);
     for(int i = 0; i < array_num_point[0] ; i++) begin
       array_cov_value[0].push_back(i);
     end
  endfunction
  task main_phase(uvm_phase phase);
      // Load coverage from DB
     $load_coverage_db("test");
     $display("coverage percentage = %f",$get_coverage());
     if(target_coverage_for_automation_start > $get_coverage()) begin
       rand_vars();
     end
     else begin
       directed_rand();
     end
     // Start sequence here
  endtask: main_phase
  virtual task rand_vars();
   cfg_1.randomize();
  endtask //}
  virtual task directed_rand();
   int idx_ar[$];
   bit [2:0] var12_val;
   //Sample code for deleting covered coverpoint from the array
   if(env.cov_10g.SampleCovGrp.cross_cov_0_var_1_cov_0_var_2.get_coverage() == 100) begin
     var12_val[2] = 0; var12_val[1:0] = 0;
     idx_ar = array_cov_value[0].find_first_index(x) with (x == var12_val);
     array_cov_value[0].delete(idx_ar[0]);
     array_num_point[0] = array_num_point[0] - 1;
     cfg_1.var_1 = 0; cfg_1.var_2 = 0;
   end
   //Randomizing amoung remaining coverpoint values
   if(array_cov_value[0].size() != 0) begin
     randomize(var_1_2) with {var_1_2 inside {array_cov_value[0]}; };
     cfg_1.var_1 = var_1_2[2];  cfg_1.var_2 = var_1_2[1:0];
   end
  endtask
endclass: nitro_ether_cov_test
```

Figure-3: Example Random test code snippet.

III. RESULTS

Fully automated coverage closure performs better than traditional coverage closure because of following reasons

- In traditional method where large number of variables are to be randomized the probability of hitting all the bins in a cover point decreases, as number of bins for that cover point increases
- Probability of hitting crosses between variables decreases as number of variables in the same cross increases.
- Verification engineer needs keep on tweaking the constraints for each run of regression based on the analysis of previous merged coverage database.

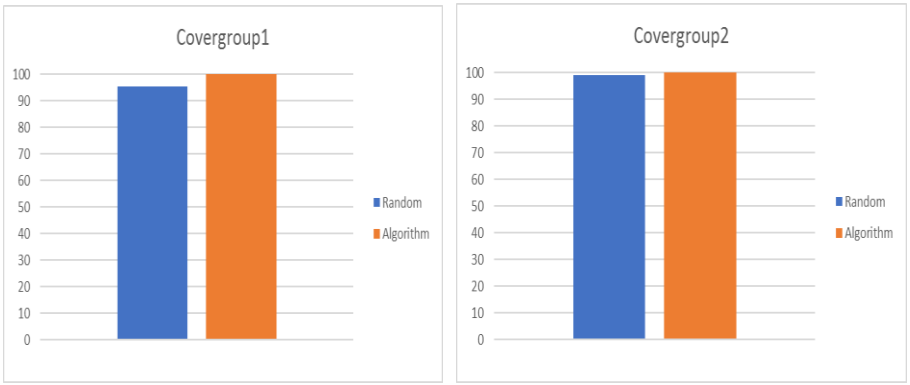
We have simulated coverage collection for traditional cover group and cover group needed for fully automated coverage closure. There was no difference in the run time in both the cases even though number of cover points present in this method are more than compared to the traditional cover group.

Cover group Name	Number of Bins	Number of Runs for 100% using algorithm
CoverGroup1	7867	1600
CoverGroup2	10410	1600
CoverGroup3	260	200
CoverGroup4	860	700
CoverGroup5	13902	6000
CoverGroup6	8017	7500

Table -1: Table with cover group and runs for coverage closure

We have implemented this fully automated coverage closure method for six different cover groups containing different number of bins. Table-1 shows number of bins in each cover group and number of runs which gave 100% coverage closure.

Below bar charts in Figure - 4 show percentage of coverage given by random runs vs fully automated coverage closure methods based on number of runs needed for 100% closure using this method.



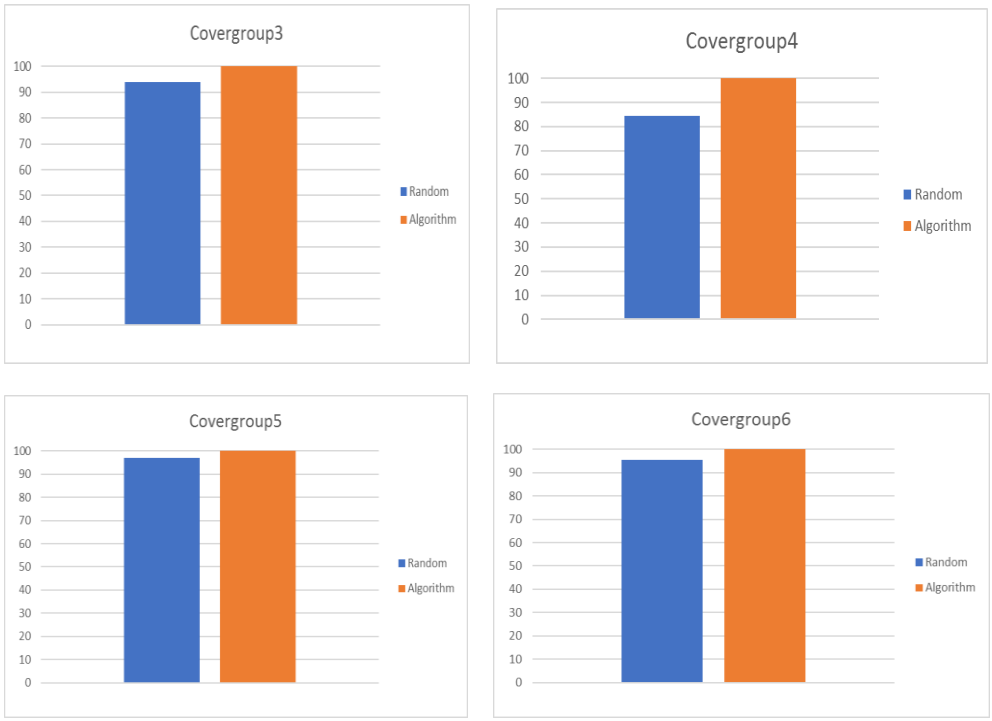


Figure-4: Figure with bar charts for 100% coverage closure

With the same number of runs which gives 100% percentage coverage in the case of fully automated functional coverage closure, traditional random runs gave coverage of less than 100%. To close this iteration of report analysis, manual constraint modification and regression rerun are needed which end up consuming LSF slots, memory and engineering effort.

The differences in percentages of different cover group runs is due to different type of crosses, number of individual bins.

When there are large number of bins for particular random variable and ~~so~~ many crosses are present in the cover group which is usually the case in our complex verification environments, then fully automated functional coverage closure gives very good results compared to traditional coverage closure method.

In Practical the traditional coverage closure takes several days as it involves lot of manual intervention, report generation, changing constraints and rerunning regressions, but in this fully automated functional coverage closure approach we will get 100% coverage in one or two days based on the complexity of the module, LSF, Queue availability and without manual intervention.

IV. CONCLUSION

Adopting the fully automated functional coverage closure method can substantially reduce the number of tests in regression. In our experiments with cover groups containing large number of bins and different type of crosses, fully automated functional coverage closure method gave better performance than traditional coverage closure for all the cases. Thus, the functional coverage closure could reach faster than ever before, and we need not to analyze the coverage reports manually and saves all the resources.

V. FUTURE WORK AND IMPROVEMENTS

Fully automated functional coverage method gives better performance over traditional coverage closure method after a certain amount of coverage is achieved. This number needs to be properly defined by verification engineer based on his previous experience of the functional coverage closure.

For fully automated coverage closure the functional cover points should be defined properly and ignore bins must be mentioned. If verification engineer is not aware of a particular bin which is illegal then this method goes into an infinite loop. We are planning to add a safeguard against this case. One solution to this problem is to break the loop of regression after certain fixed percentage of coverage is reached instead of 100%.

The algorithm for fully automated coverage closure is complex, we are working on making it simple so that it can be easily implemented by the users.

ACKNOWLEDGEMENTS

We would like to thank Suresh Atmakuri for help in developing scripts and our managers and colleagues for continued support.

REFERENCES

[1] IEEE 1800-2009 SystemVerilog

[2] http://www.accellera.org/apps/org/workgroup/uvm/