Improve emulator test quality by applying synthesizable functional coverage

Hoyeon Hwang[1], Taeseong Kim[2], Sanghyun Park[3], Yong-Kwan Cho[4],
Dohyung Kim[5], Wonil Cho[6], Sanggyu Park[7]

Samsung Electronics Co., Ltd.
([1]hy1518.hwang; [2]t1.kim; [3]sh94.park; [4]yongkwan81.cho; [5]dh758.kim; [6]wonil.cho; [7]sg78.park@samsung.com)
1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do 18448, Republic of Korea

***Abstract* – Emulation significantly helps to reduce turnaround time and it can cover the corner cases of failure in the design, which is hard to find in the simulation. However, it is more difficult to check the design functionality comparing with simulation. For that reason, it is needed to apply the functional coverage in emulation. Each vendors have already provided their own solution but, it is difficult to reuse coverage code used in the simulation and there are differences in the syntax supported by each emulator. In this paper, we present common rules for writing synthesizable functional coverage models that can be widely used for both simulation and emulation. Moreover, we show benefits of measuring functional coverage through case studies, which applied in emulations.**

Keywords – functional coverage; emulation; system verilog; testbench acceleration

I. INTRODUCTION

Emulator is introduced when some issues are found with (very) low probability or huge test turn-around-time (TAT) is required because emulator is much faster than simulator generally. However, it is difficult to confirm how much design intent has been verified for emulation based than simulation. In the simulation, the functional coverage has been used to verify the design functional intents. Functional verification also can be closed by adding test vectors for uncovered area. We referred to the experience conducted by simulation, and decided to apply functional coverage to the emulation.

There are benefits when the functional coverage is used in the emulation. First, like simulation, a percentage of functional verification progress can be checked in the emulation. Second, by defining a functionality for long TAT test scenario that are hard to finish in the simulation, a functional coverage metrics data for the test scenario can be measured and the test can be performed within a limited period. Third, by applying functional coverage for full combination test scenario, missing cases can be found easily and test vectors for the missing cases can be made quickly. Therefore, it is helpful to choose the minimum test vectors to verify all combinations. Lastly, emulation and simulation can share functional coverage code and each coverage metrics data can be merged. Comparing with simulator run only, it takes less time to complete verification and do coverage closure as emulator and simulator run all test scenarios together. As a result, it has the effect of reducing the test TAT.

We used C-vendor and M-vendor emulator. Each EDA vendor already had its own solution with its user guide documents [1] [2]. However, it is challenging to reuse functional coverage code applied in the simulation when the coverage are implemented in the hardware due to the emulator capacity and some functional coverage syntax supported in the emulator only. In addition, there are some syntax difference between both emulators. To solve the problem, we propose common rules for writing synthesizable functional coverage models that can be implement in less capacity and can be used for both emulation and simulation.

This paper helps emulator users to accelerate functional verification and finish the verification by measuring functional coverage and adding test vectors for uncovered area.

II. CONSIDERATIONS

For writing common functional coverage, we should take into account three considerations: 1) Capacity, 2) Speed, and 3) Functional coverage syntax.

A. Capacity

Unlike simulation, emulators have limitations on the number of bins that can be implemented due to resource constraints. C-vendor emulator manages the bin count of the entire coverage data by setting the internal memory counter (Max 2,147,483,647). M-vendor emulator cannot make more than 4096 bins per coverpoint and the total bin count is limited by AVB board's capacity. Therefore, more efficiently we use the given capacity, more coverage we can implement coverage in the emulator. EDA vendors are also trying to improve the emulator capacity. However, Design Under Verification (DUVs) also will become more complex to meet market demands. Therefore, it is always important to consider the capacity limitation when we use emulator.

B. Speed

While running test, C-vender and M-vendor emulator store coverage data into memory in the background at the same time and execute coverage data dump at once after all test cases are finished. For this reason, there is very little difference in run-time depending on whether functional coverage is applied or not. With 191,797 bins, the coverage data dump only takes additional 1 to 2 minutes. Table I shows a run-time between applying functional coverage or not. Therefore, if emulator capacity is sufficient, the emulator is more useful for measuring coverage than simulation, because it is inevitable to get lower speed to do simulation with heavy functional coverage.

TABLE I
COMPARE RUN-TIME W/ AND W/O FUNCTIONAL COVERAGE

	C-vendor emulator		M-vendor emulator	
	w/o Func. Cov.	w/ Func. Cov.	w/o Func. Cov. (Throughput: 90.47%)	w/ Func. Cov. (Throughput: 90.59%)
Run-time Average	12min 25sec	13min 13sec	26min 31sec	24min 27sec
Coverage DB Dump Time	N/A	1min 2sec	N/A	1min 20sec

C. Functional coverage syntax

According to C-vendor and M-vendor's functional coverage user guide documents [1] [2], there are functional coverage syntax allowed only in each emulator. Furthermore, some syntax used in RTL simulation are not synthesizable so it can not be applied in emulator. Therefore, it is not ideal to support functional coverage, which a specific emulator is only supported. To overcome the difference, a common coverage design pattern is important.

In the section III, we introduced functional coverage coding guidelines considered three factors above.

III. SUGGESTED FUNCTIONAL COVERAGE CODING GUIDELINES

In this section, we present common rules for synthesizable functional coverage: 1) Bin count limitation, 2-3)Cross coverage; "*ignore_bins*" usage and separating into covergroups, 4) Sampling, 5) Multiple instantiate of covergroup using parameter, 6) Coverpoint guard and 7) Appling "*set_inst_name*" option

A. Limit the number of bins

In case of M-vendor emulator, the number of bins per one coverpoint must do not exceed 4,096. Therefore, a coverpoint with large number bins should be divided into multiple coverpoints that covers different slices. Cross coverage of those slices are equivalent to the original coverage as shown in Table II.

TABLE II
GUIDELINE FOR BIN COUNT LIMITATION CASE

RTL simulation general usage	M-vendor emulator (recommend)

covergroup CG_IMG_SIZE@(posedge clk);
  P: coverpoint pre_img_size[12:0]; // Max size: 8192
  S: coverpoint scaled_img_size[12:0];
endgroup
	covergroup CG_PRE_IMG_SIZE@(posedge clk);
	  P_0: coverpoint pre_img_size[11:0];
	  P_1: coverpoint pre_img_size[12];
	  P: cross P_1, P_0;
	endgroup
	
	covergroup CG_SCL_IMG_SIZE@(posedge clk);
	  S_0: coverpoint scaled_img_size[11:0];
	  S_1: coverpoint scaled_img_size[12];
	  S: cross S_1, S_0;
	endgroup

B. Resource reduction of cross coverage; "ignore_bins" usage

In case of C-vendor emulator, instead of using "*ignore_bins*" in cross coverage, it is better to apply the "*ignore_bins*" at coverpoint included in the cross coverage to use less emulator resources. Table III shows a comparison of resources between two cases. Even if the number of bins is reduced by using "*ignore_bins*" within the cross coverage, the emulator internal memory requires space to store all bins, which does not applied "*ignore_bins*". For that reason, we recommend that "*ignore_bins*" apply to the coverpoint instead of the cross coverage.

TABLE III
GUIDELINE TO USE MINIMUM RESOURCE OF CROSS COVERAGE USING "*ignore_bins*"

RTL simulation general usage	C-vendor emulator (recommend)

covergroup CG @(posedge clk);
  A: coverpoint alpha; // 4 bins
  B: coverpoint bravo; // 4 bins
  odd_combinations cross A, B { // 4 bins
  ignore_bins odd_A = binsof(A) intersect {0, 2};
  ignore_bins odd_B = binsof(B) intersect {0, 2};
  }
endgroup
	covergroup CG @(posedge clk);
	  A: coverpoint alpha {
	  ignore_bins odd = {0, 2};
	  }
	  B: coverpoint bravo {
	  ignore_bins odd = {0, 2};
	  }
	  odd_combinations: cross A, B;
	endgroup

Cross coverage bin count: 4	Cross coverage bin count: 4
Emulator internal memory: 16	Emulator internal memory: 4

C. Resource reduction of cross coverage; separating cross coverage

If a coverpoint is included into multiple cross coverages, it is better to separate the cross coverages into different covergroups. Table IV shows an example how separate the cross coverages.

TABLE IV
GUIDELINE TO USE MINIMUM RESOURCE OF SEPERATING CROSS COVERAGE

RTL simulation general usage	C-vendor emulator (recommend)

reg [2:0] a,b,c;
 	
covergroup CG_A;
  pa: coverpoint a;
  pb: coverpoint b;
  pc: coverpoint c;
  cross pa,pb;
  cross pb,pc;
endgroup
	reg [2:0] a,b,c;
	
	covergroup CG_B_0;
	  pa: coverpoint a;
	  pb: coverpoint b;
	  cross pa,pb;
	endgroup
	
	covergroup CG_B_1;
	  pb: coverpoint b;
	  pc: coverpoint c;
	  cross pb,pc;
	endgroup

In the table IV, if the covergroup *CG_A* is applied in the C-vendor emulator, it is considerate two cross coverages "*pa, pb*" and "*pb, pc*" as one cross coverage "*pa, pb, pc*", and it causes increasing the address space required to store the internal memory counters. It also requires more gates and can affect emulation performance. If the cross coverages "*pa, pb*" and "*pb, pc*" are separated into different covergroup (*CG_B_0, CB_B_1*), the emulator need two small memory counters even though a memory for the coverpoint "*pb*" is counted multiple times in each covergroup (*CG_B_0, CB_B_1*).

D. Sampling

In the simulation, the coverage is generally sampled using the "*sample*" method. However, the "*sample*" method is not synthesizable in emulator. Instead of that, we have to define special sample event like *@(posedge clk).* Table V shows a comparison of sampling coverage between simulation and emulation.

TABLE V
SAMPLING EXAMPLE IN THE SIMULATION AND EMULATION

RTL simulation general usage

```
class c_data_path;

logic [3:0] cha_scn_data_path, chb_scn_data_path;

covergroup cov_data_path;
  pa: coverpoint cha_scn_data_path;
  pb: coverpoint chb_scn_data_path;
  cross pa, pb;
endgroup : cov_data_path

function new();
  cov_data_path = new;
endfunction : new

function void sample_cha (input path);
  this.cha_scn_data_path = path;
endfunction: sample_a

function void sample_chb (input path);
  this.chb_scn_data_path = path;
endfunction: sample_a

endclass: c_data_path


logic [3:0] cha_scn_data_path, chb_scn_data_path;

c_data_path real_path;
real_path = new();

always @(posedge cha_start_pulse)
  real_path.sample_a(cha_scn_data_path);

always @(posedge chb_start_pulse)
  real_path.sample_b(chb_scn_data_path);

always @(posedge all_scn_done)
  real_path.sample();
```

C-vendor and M-vendor emulator (recommend)

```
logic [3:0] cha_scn_data_path, chb_scn_data_path;
logic [3:0] lat_cha_scn_data_path,
lat_chb_scn_data_path;

covergroup cov_data_path @ (posedge all_scn_done);
  pa: coverpoint lat_cha_scn_data_path;
  pb: coverpoint lat_chb_scn_data_path;
  cross pa, pb;
endgroup : cov_data_path

cov_data_path my_cov = new();

always @(posedge clk) begin
  if(cha_start_pulse) begin
    lat_cha_scn_data_path <= cha_scn_data_path;
  end
end

always @(posedge clk) begin
  if(chb_start_pulse) begin
    lat_chb_scn_data_path <= chb_scn_data_path;
  end
end
```

To describe difference of sampling in the table V, we assume two virtual data path: *A*, *B*. The verification purpose of the example is checking a combination coverage for two data path in the "*cov_data_path*" covergroup. Here is a specification for the sampling example.

- *cha_scn_data_path*: Information for the *A* data path
- *chb_scn_data_path*: Information for the *B* data path
- *cha_start_pulse*, *chb_start_pulse*: Valid signal for each data path information signal
- *all_scn_done*: Operation of two data path is done
- The two data path are operated asynchronous, so the valid signals can be asserted at different time

In the simulation, the data path information can be sampled when each valid signals are asserted. Once "*all_scn_done*" signal is asserted, coverage can be measured using "*sample*" method. On the other side, because emulators do not support the "*sample*" method, the data path information should be latched until the "*all_scn_done*" signal is asserted. Figure 1 illustrates a timing diagram: latching data information signals in green

and sampling time in red. In the example, the "*lat_cha_scn_data_path*", "*lat_chb_scn_data_path*" signals are used for latching, which are not necessary for simulation. In the emulator, the latch signals should be used for cross coverage instead of the data information signals. The covergroup for emulation is defined with sample event *@(posedge all_scn_done)* to measure the coverage when the "*all_scn_done*" signal is asserted.

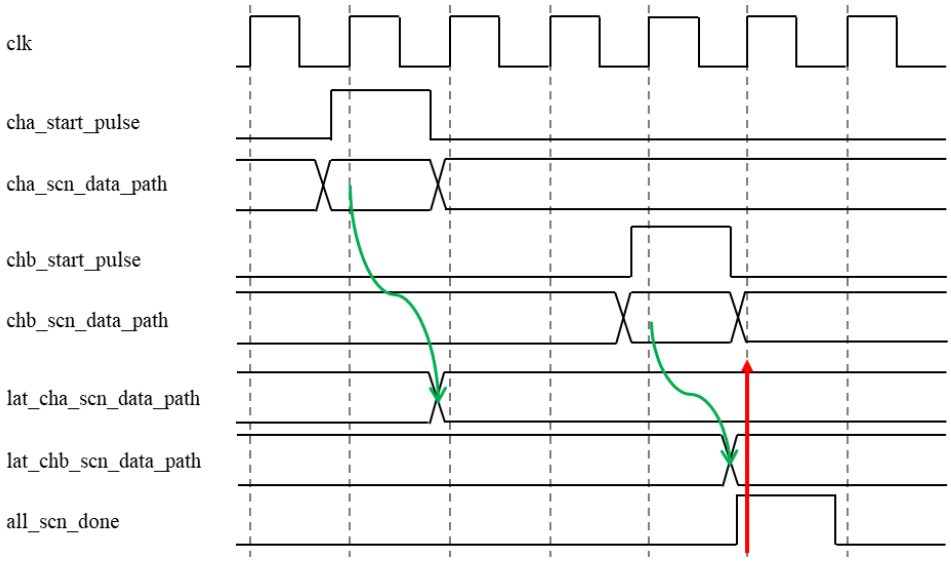

Figure 1. Latching and sampling in emulation

E. Multiple instantiate of covergroup using parameter

In the simulation, a covergroup can be defined with argument, which is used for coverpoint's specific condition at "*iff*" clause. However, the argument of covergroup is not supported in M-vendor emulator. To solve this restrict, the specific condition should use a parameter in module declaration. In the table VI, we shows a comparison between using covergroup argument in the simulation and using parameter in the M-vendor emulation.

TABLE VI
GUIDELINE FOR MULTIPLE INSTANTIATE OF COVERGROUP IN SPECIFIC CONDITION

RTL simulation general usage

```
module cov_top (
);

covergroup cov_data_path (int _block_id);
  a: coverpoint (testValue[1:0]) iff (_block_id==0)
  …
  b: coverpoint (testValue[3:2]) iff (_block_id==1)
  …
endgroup : cov_data_path

cov_data_path  inst_cov_data_path;

initial begin
  // Constructs
  cov_inst_decon_data_path = new(0);
  cov_inst_decon_data_path = new(1);
  …
end

endmodule : cov_top

module tb_top ();
  cov_top  cov_top ();
endmodule  : tb_top
```

M-vendor emulator (recommend)

```
module cov_top #(parameter p_block_id = 0); (
);

covergroup cov_data_path;
  a: coverpoint (testValue[1:0]) iff (p_block_id==0)
  …
  b: coverpoint (testValue[3:2]) iff (p_block_id==1)
  …
endgroup : cov_data_path

cov_data_path  inst_cov_data_path = new();

initial begin
  $display("block_id = %0d", p_block_id);
  // cov_inst_decon_data_path = new(block_id);
  …
end

endmodule : cov_top

module tb_top ();
  cov_top  #(.p_block_id(0)) cov_top_0 ();
  cov_top  #(.p_block_id(1)) cov_top_1 ();
endmodule : tb_top
```

According to the table VI, the block_id information is delivered through *_block_id* argument when the *new()* construct is called in the simulation. On the other hand, in M-vendor emulator, the "cov_top" module is defined with *p_block_id* parameter instead of covergroup argument, so the coverpoint can use the block_id information at

"*iff*" clause like the simulation. If emulator verification engineer wants to check the "*cov_data_path*" coverage according to the "*p_block_id*" parameter, the "*cov_top*" module should be instantiated as much as the type of "*testValue*" in the "*tb_top*" module.

F. Coverpoint guards

The "*with*" clause is not synthesizable in the both emulators so that the *bins* in coverpoint cannot be defined with "*with*" clause. Instead of "*with*" clause, specific condition can be applied when a coverpoint is defined with "*iff*" clause. Table VII shows an example, which defines a coverpoint according to the *pId* in the simulation and emulation.

TABLE VII
COVERPOINT "*iff*" GUARD INSTEAD OF "*with*" CLAUSE

RTL simulation general usage	C-vendor and M-vendor emulator (recommend)

`all_id_bins: coverpoint (inValue[3:0])`
`{`
`  bins p_id0 = { [00:07] } with (pID == 0);`
`  bins p_id1 = { [08:15] } with (pID == 1);`
`}`
	`id_0_bins:coverpoint (inValue[3:0]) iff (pID == 0)`
	`{`
	`  bins p_id0 = { [00:07] };`
	`}`
	
	`id_1_bins:coverpoint (inValue[3:0]) iff (pID == 1)`
	`{`
	`  bins p_id1 = { [08:15] };`
	`}`

G. Apply "set_inst_name" option

When a single covergroup is instantiated in multiple times, the "*set_inst_name*" option is used to prevent generating same instance name. The "*set_inst_name*" should be applied in the initial statement in C-vendor emulator. Table VIII shows an example for applying "*set_inst_name*" option.

TABLE VIII
APPLYING "*set_inst_name*" OPTION

RTL simulation general usage	C-vendor emulator (recommend)

`module tb_top();`
 	
`covergroup mycg;`
`  option.per_instance=1;`
`  …`
`endgroup : mycg`
 	
`mycg cg1 = new();`
`cg1.set_inst_name("inst1");`
 	
`…`
 	
`endmodule: tb_top`
	`module tb_top();`
	
	`covergroup mycg;`
	`  option.per_instance=1;`
	`  …`
	`endgroup : mycg`
	
	`mycg cg1 = new();`
	
	`initial begin`
	`  cg1.set_inst_name("inst1");`
	`end`
	
	`…`
	
	`endmodule: tb_top`

IV. REUSABLE CODING STYLES WITH SIMULATION

Unlike the syntax difference mentioned above section, there are reusable functional coverage coding style, which can be applied with both the emulation and the simulation. In the table IX, we introduce four cases that are useful for writing functional coverage based on our experience: 1) *ignore_bins* 2) defining bins with unsized array[] 3) *wildcard bins* 4) transition bins.

TABLE IX
EXAMPLES OF REUSABLE FUNCTIONAL COVERAGE SYNTAX

Reusable functional coverage syntax	Simple example

ignore_bins [1)]	`pa: coverpoint (a_value[3:0]);`
	`pb: coverpoint (b_value[3:0]);`
	`pc: coverpoint (c_value[3:0]);`
	
	`cross_ignore: cross pa, pb, pc {`
	`  ignore_bins xa = !binsof (pa) intersect {4'b0001};`
	`  ignore_bins xb = binsof (pb) intersect {4'b0010};`
	`  ignore_bins xc = binsof (pc) intersect {4'b0100};`
	`}`

Defining bins with unsized array[]	`cp_unsize: coverpoint (test_value[255:0]) {`
	`  bins unsized_bins[] = { [0:255] };`
	`}`

wildcard bins	`cp_wcbins: coverpoint (in_value[3:0]) {`
	`  wildcard bins state1 = { 4'b???0 };`
	`  wildcard bins state2 = { 4'b0??? };`
	`}`

Transition bins	`cp_tsbins: coverpoint (state_value[1:0]) {`
	`  bins change_state = ( 2'b00 => 2'b01 );`
	`}`

[1)] In section III, we discussed the "ignore_bins" usage in cross coverage. However it does not mean that the "ignore_bins" cannot use in the cross coverage not all. We recommend that it is better to use the "ignore_bins" less in the cross coverage. Emulator supports the "ignore_bins" syntax like simulation.

V. INTEGRATION FUNCTIONAL COVERAGE

The UVM(Universal Verification Methodology) environment has UVC(UVM Verification Component) including UVM driver, monitor, scoreboard, sequencer and DUT(Design Under Test) wrapper that can be synthesizable. The functional coverage is placed in UVC scoreboard and be declared as a class member generally. However, we changed the functional coverage code's place from the UVC to the DUT wrapper. Figure 2 illustrates a block diagram of integrating synthesizable functional coverage block and Table X shows simple examples of implementation in both class and module.

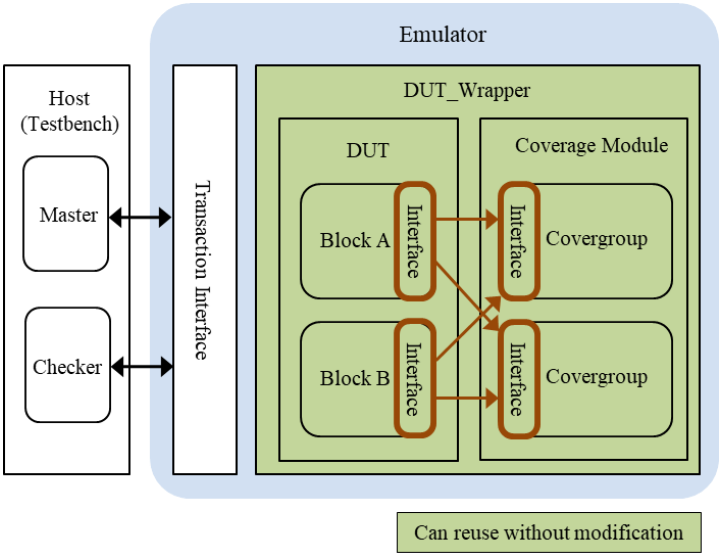

Figure 2. Integration of synthesizable functional coverage module

TABLE X
SIMPLE EXAMPLE OF IMPLEMENTATION IN A CLASS AND IN A MODULE

Implementation in a class

```
//Interface module
interface cov_intf();
  logic done;
  logic [3:0] data_path;
endinterface : cov_intf

//Coverage class
class data_path_cov;
  covergroup cov_data_path;
    option.per_instance = 1;
    data_path : coverpoint (cov_if.data_path)
  endgroup : cov_data_path

  function new();
    cov_data_path = new;
  endfunction : new

endclass : data_path_cov

//Checker
class Checker;
  cov_intf inst_cov_intf;
  data_path_cov inst_data_path;

  function void build_phase(uvm_phase phase);
    super.build_phase(name, parent);
    inst_cov_data_path = new();
  endfunction : build_phase

  task run_phase(uvm_phase phase);
    forever begin
      @(posedge cov_if.done);
      inst_data_path.sample();
    end
  endtask : run_phase

endclass : checker

//DUT Wrapper module
module DUT_wrapper ();
  cov_intf inst_cov_intf();

  //Connect DUT internal signal and interface
  assign inst_cov_intf.done = inst_DUT.done;
  assign inst_cov_intf.data_path = inst_DUT.data_path;

  //Instance DUT
  DUT inst_DUT(done, data_path);

endmodule : DUT_wrapper
```

Implementation in a module

```
//Interface module
interface cov_intf();
  logic done;
  logic [3:0] data_path;
endinterface : cov_intf

//Coverage module
module data_path_cov( cov_intf cov_if );
  covergroup cov_data_path @ (posedge cov_if.done);
    option.per_instance = 1;
    data_path : coverpoint (cov_if.data_path)
  endgroup : cov_data_path

  data_path_cov inst_data_path_cov = new();

endmodule : data_path_cov

//DUT Wrapper module
module DUT_wrapper ();
  cov_intf inst_cov_intf();

  //Connect DUT internal signal and interface
  assign inst_cov_intf.done = inst_DUT.done;
  assign inst_cov_intf.data_path = inst_DUT.data_path;

  //Instance coverage module
  data_path_cov
inst_data_path_cov(cov_if(inst_cov_intf));

  //Instance DUT
  DUT inst_DUT(done, data_path);

endmodule : DUT_wrapper
```

The functional coverage module (*data_path_cov*) is wrote in a module not a class. The interface modules (*cov_intf*) that defined for internal block's interface are re-used for connection between functional coverage module and DUT's internal signals. The functional coverage module in green can be instantiated in the simulation without modification and can be applied the C-vendor and M-vendor emulators. Furthermore, a conventional verification environment that even is not generated as UVM can re-use the functional coverage module. If the functional coverage code is a part of UVC instead of DUT wrapper in the emulation verification, it might cause a negatively performance impact by having to pull signals from the DUT to collect coverage. This is why we select the DUT wrapper for the integration in the emulation.

VI. MERGE COVERAGE METRICS DATA

If emulation and simulation use same functional coverage code, the coverage metrics data can be merged using related vendor's merge tool or script. Before merge the coverage metrics data, you have to check below.

- Each emulator's coverage data can be merged with its vendor simulator's coverage data.
- A block hierarchy including functional coverage in the emulation should be same with simulation.

It is a common case to use a different emulator comparing with a simulator because each emulator has advantages for special use-case. However, it is hard to manage verification progress and fill coverage holes when we use different vendor's simulator and emulator. Therefore, it should be better if the coverage metrics data between different vendors can be merged. The Unified Coverage Interoperability Standard (UCIS) provides an application programming interface (API) that enables the sharing of coverage data across software, hardware accelerators, formal tools or custom verification tools [3]. We are going to try to merge the emulator's coverage metrics data with simulator's data generated by different vendor using the UCIS for the next step.

VII. CASE STUDIES

Two best practices are presented to show benefits of measuring functional coverage in emulations.

A. TAT reduction of corner case hunting

We defined functional coverage with 2,632 bins including corner cases that can occur with low probability. We performed all test scenario with emulator in 2 weeks. However, with simulation, it might be took a several months because each test consumed several hours or days and needed hundreds of frames until finish a test. In the table XI, we compared the test TAT between simulation and emulation.

TABLE XI
COMPARING TAT FOR CORNER CASE HUNTING

Bin Count	Simulation	Emulation
2,632	157,920 hours [1)]	421.12 hours [2)]

[1)] 2,632 bins * 60 hours. Assume that 1test per 1bin is required.

[2)] 2,632 bins * 10 mins. Assume that 1test per 1bin is required.

B. Measure full combination coverage

By defining ″Full Combination″ as functional coverage with 4,846 bins and verifying it in the emulator, we were able to verify all combinations without missing cases, and also found a bug that is only found in specific combination in the actual project. Without the functional coverage, it was difficult to know whether the problematic combination was verified or not, even though many tests were performed repeatedly in the emulator, because test vectors for full combination scenario had been generated randomly.

VIII. CONCLUSION

In this paper, we proposed the common functional coverage coding guideline in seven cases to improve emulation test quality and applied it in C-vendor and M-vendor emulator. We discussed the integration of functional coverage module and the coverage metrics data merge. With two case studies, we showed effectiveness for TAT reduction and suitability for full combination coverage.

REFERENCES

[1] Cadence, WXE User Guide, Product Version 21.00, February 2021
[2] Siemens, Veloce Coverage and Assertion Application User Guide, Release v22.0.2, Document Revision 7.2
[3] Unified Coverage Interoperability Standard (UCIS), Version 1.0, June 2, 2012