2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Static Checking for Correctness of Functional Coverage Models

Wael Mahmoud, Mentor Graphics Corporation , Cairo, Egypt (*wael_mahmoud@mentor.com*)

***Abstract*— Writing a complete and correct functional verification plan is always recognized as one of the top challenging tasks during the verification cycle. This paper proposes a framework to improve the correctness of functional coverage models by reducing the gap between the written functional coverage specifications and the actual coverage model that is inherited in the test-bench implementation of design under verification (DUV). The proposed work identifies the coverage holes that arise from the conflict between written coverage specification and the actual design constraints. It utilizes formal verification as well as constraints solvers techniques to statically conclude a concise form of functional coverage model. The final coverage model is deduced from the original specification but is still respecting the identified and extracted design and test-bench constraints. Our experimental results demonstrate the effectiveness of the proposed approach in enhancing the functional coverage model description for a set of today's RTL designs and illustrates how this framework is useful in suppressing some of the coverage items that are irrelevant to actual design implementation and verification constraints.**

Keywords*—*Verification; Functional Coverage; SoC; ASIC; Constraint Solvers; Formal Verification

I. INTRODUCTION

Nowadays, designs are getting bigger and more complex, especially for SoC (system-on-chip) and ASIC (application specific integrated circuits) designs, many complex sub-systems are integrated into one or more chips. It is a very challenging task to make sure that RTL (register-transfer level) designs for SoC and ASIC are working correctly.

Recent industry studies done in 2014 [1], shows that the average total project time spent in verification was 57%. Also, the number of project that spent more than 80% of their time in verification has been increased from the past, and in sometimes this becomes the bottleneck to tape-out.

Currently, Verification engineers spend a lot of their time to translate the requirements into test plans, and implement them. One of the most important questions while writing tests, did we achieve the required coverage goal? How many tests required to achieve the required coverage goal? How do we write more tests to cover the missed items? But the important question, did we write the correct functional coverage model?

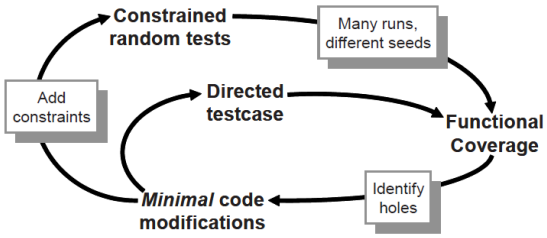
This paper is proposing unified approach to statically improve functional coverage for DUV, and to exclude some parts of the coverage that will never be hit due to either design and/or test-bench constraints.

This paper is organized as follows. Section II introduces the concept of verification. Section III shows the previous work done to automatically generate functional coverage and the related work. Section IV introduces the problem formulation and the proposed framework to solve this problem. Section V shows the experimental results. Finally Section VI concludes the work done.

II. PRELIMINARIES

As, current designs getting bigger and more complex, one of the effective ways to verify them is to use CRT (constrained-random testing). Functional coverage is a way to check or measure the design features that have been exercised by the applied tests [2]. "Figure 1", describes the life cycle of verifying the designs, it shows both directed tests to hit some specific design areas and CRT to thoroughly test the DUV, and there is a feedback loop to analyze the coverage results, and this action is repeated till the functional coverage achieves the coverage goal [3].

Figure 1: Designs Verification lifecycle



One more objective of using functional verification, is to assure that the specification and the implementation of the DUV are preserved, and the results of my proposed approach has verified that preservation.

III. RELATED WORK

Most of recent researches are targeting automatic generation of coverage models that can be used by verification engineers to verify their designs. Some of them proposed approaches to automatically generate the coverage models by statically analyze the DUV like researches presented in [4] and [5], while other researches propose dynamic approaches to mine the simulation traces and extract the candidate coverage models [6].

In this paper, a framework will be presented to use both formal-based analysis techniques and constraint solver technologies to statically analyze SystemVerilog (SV) [7] functional coverage models, and to identify coverage items that will never be hit, due to test-bench's input stimulus and constraints and RTL design's conditions.

To the best of my knowledge, it is the first time that this framework put together into a research work.

IV. PROBLEM FORMULATION

In this section, the problem verification engineers face while developing functional coverage models, will be illustrated and a proposed methodology to enhance the functional coverage models will be introduced. This section introduces a static methodology to achieve a precise form of input/output functional coverage, as well as a way to exclude un-reachable functional coverage items from coverage calculations, which leads to a good amount of enhancements to the overall functional coverage results.

This section is divided into two sub-sections, first one shows the problem from input/output functional coverage perspective, and the other sub-section shows it from design-centric functional coverage perspective, design-centric functional coverage, means the functional coverage models defined indie the DUT itself to get the coverage info related to DUT's variables.

A. Input and Output Functional Coverage

SystemVerilog has introduced some constructs to define functional coverage models, which are widely used by verification engineers. "Example 1" is a part of a SystemVerilog test-bench, which contains input functional coverage for variables "A" and "B".

In "Example 1", there is a definition of 2 input stimulus variables, "A" which has range of values from 0 to 15, and "B" which has range of values from -32,768 to 32,767. Also, the test-bench is constraining these 2 inputs to some legal values using constraint constructs, "A" is constrained to values' range from 0 to 7 using "A_constr" constraint, while "B" is constrained to values' range from 0 to 15 using "B_constr" constraint.

"Example 1" shows a simple input functional coverage model written to measure the coverage of input stimulus variables, and according to SystemVerilog terminologies, "A_cp" cover-point has 16 distinct bins, and "B" cover-point has 64 distinct bins (this is the default auto-bin-max in SystemVerilog).

Example 1: Part of SV test-bench with simple cover-group

```
rand bit [3:0] A;
rand shortint B;

constraint A_constr {
        A < 8;
}
constraint B_constr {
        B inside {[0:15]};
}
…
covergroup cov;
        A_cp: coverpoint A;
        B_cp: coverpoint B;
endgroup
```

After running simulation for this test-bench and collecting the coverage results using the simple defined cover-group in "Example 1", the maximum coverage results can be achieved for "A_cp" is 50%, since due to "A_constr", "A" will have values from "0" to "7" (i.e. 8 values/bins), and at the same time "A_cp" requires 16 distinct values/bins to be generated for "A" variable to achieve 100% coverage. Similarly, the maximum coverage results achieved for "B_cp" is *1.5%*, since there are 64 bins automatically generated to cover the whole values range of "B" variable, and according to "B_constr", constraint, only 16 values will be generated for "B" variable and all of these 16 values are grouped into only one bin out of the 64 bins, which equals to 1.5%. Finally, the maximum coverage percentage for "cov" cover-group will be 25.7%. This is due to the existence of "A_constr" and "B_constr" which are limiting the whole range of "A" and "B" values to be generated, and hence the 100% functional coverage cannot be achieved.

In "Example 2", the written cover-point for input stimulus variable "A" has cover-point bins, although there are some bins definition added for "A_cp" cover-point but these bins are not written in a correct way to achieve the 100% coverage. In "Example 2", "A_bins2" bin will never be hit because of "A_constr" constraint.

Example 2: Part of SV test-bench with cover-point with bins

```
rand bit [3:0] A;
constraint A_constr {
        A < 8;
}
…
covergroup cov;
        A_cp: coverpoint A {
                bins A_bins1[] = {[0:10]};
                bins A_bins2 = {[11:$]}
        }
endgroup
```

B. Design-centric Functional Coverage

Now, it's time to check the other part of the problem, which is the design-centric functional coverage. In some cases, developers may define some extra design states that are not reachable by the design itself, like for example, defining a state-machine with some states that are unreachable due to design constraints or defined conditions in the RTL, like using if/else or ternary statements that defines some constraints to RTL, which may lead to some unreachable states or areas in design under test.

If there are some design-centric functional coverage constructs used to provide visibility into the verification of the written RTL, then there is a possibility to not achieve 100% coverage due to some unreachable design states.

In “Example 3” below, there is a RTL for a finite-state machine (FSM), this FSM has some defined states, and the transition from each state is controlled by the use of SystemVerilog ‘case’ statement.

“Example 3” shows a state machine, with some defined states, and from this example, “wait_idle” state is an ureachable state, and hence the coverage results of the bins created for this state will always stuck at 0%, when measuring the coverage results of “sm_cvg” cover-group.

Example 3: RTL for FSM and a cover-group

```
covergroup sm_cvg @(posedge pins.clk);
  coverpoint int_state;
endgroup
always @(posedge fsm_clk or negedge fsm_reset_n)
            if(!fsm_reset_n)
               int_state <= idle;
            else
               int_state <= nxt_state;

always @(*)
begin
  nxt_state = int_state;
  case (int_state)
    idle:
      if(in_hs)
        nxt_state = send_bypass;
      else
        nxt_state = idle;
    send_bypass:
      if(out_hs)
       if(enable)
         nxt_state = load_bypass;
       else
         nxt_state = idle;
    load_bypass:
      if(in_hs)
        nxt_state = send_bypass;
    wait_idle:
      if(out_hs)
        nxt_state = idle;
  endcase
end
```

So, as shown in “Example 3”, there is a possibility of not achieving 100% design-centric functional coverage due to some RTL conditions, which causes some of design areas and states unreachable.

V. PROPOSED METHODOLOGY

As shown in above section, the problem of not achieving 100% coverage for input/output and/or design-centric functional coverage due to design and/or test-bench constraints or conditions has been illustrated. In this section, a framework to enhance functional coverage models is introduced. The proposed framework is addressing the problem of having some unreachable functional coverage targets and shows a methodology to fix them and finally enhancing the overall functional coverage results.

The proposed framework uses both constraint solver algorithms and formal-based analysis techniques to statically identify the unreachable functional coverage items, and exclude them, which enhances the overall functional coverage of DUV.

To achieve the proposed framework, Mentor Graphics company’s tools are used to propose a complete static methodology workflow for enhancing functional coverage models, the first tool is called Questa inFact [8], which

internally deploys a highly speed constraint solvers algorithms, which can help us in enhancing input and output functional coverage. The second tool is called Questa CoverCheck [9], which deploys formal-bases analysis techniques to analyze the reachability of design-centric coverage items defined within RTL design.

A. Static Enhancement of Input and Output Functional Coverage

In "Example 1", the input stimulus variable "A", has a range of 16 values [0..15] (this representation means the values range is from 0 to 15), also, there is a constraint "A_constr" which constraint the values range of input stimulus "A" to be less than 8, i.e. the allowed range of values is [0..7].

Also, "Example 1", defines "A_cp" cover-point which covers the whole values range of "A", so this cover-point is waiting for 16 distinct values/hits to be generated from simulation, in order to achieve 100% functional coverage for this specific cover-point, although according to test-bench's constraint, the generated values of input stimulus variable "A" should be in [0..7] range, i.e. only 8 values or to be more specific only the first 8 bins for "A_cp" will be hit, and the rest of the bins will never be hit, and this will cause the coverage results to be 50%.

So, next we use constraint solver techniques to generate an enhanced version of functional coverage, in order to achieve 100% coverage goal or getting better coverage results.

Below are the steps to statically enhance input/output functional coverage models:

1. Import the test-bench into Questa inFact to extract test-bench's variables, constraints and the existing un-enhanced functional coverage model (i.e. cover-group(s)), Questa inFact automatically extracts the variables, and constraints defined in the specified stimulus class, also it automatically extracts the coverage constructs (i.e. cover-points and cross) defined in cover-group(s).
2. For each coverage item defined in the functional coverage model, solve all the target variables ranges contributing in this coverage item against the test-bench's constraint.
3. Generate an enhanced functional coverage model.

Example 4: Enhanced A_cp cover-point

```
A_cp : coverpoint A {
        option.weight = 8;
        bins cfg_item_inst_A[] = {[64'd0:64'd7]};
    }
```

"Example 5" shows another example for a functional coverage model, which shows the value of using constraint solver technologies to enhance the functional coverage and achieve better coverage results. In this example, there are 2 stimulus variables, each with 4-bit width (i.e. 16 values), and there is a constraint "add_constr", which limits result of the addition of the 2 stimulus variables to be from "0" to "10". Finally the example ends with a cover-group, with a "cr1" cross coverage between 2 cover-points on both "A" and "B" variables, named "A_cp" and "B_cp" respectively.

Example 5: Complex test-bench constraints with cross coverage

```
rand logic unsigned [0:3] A, B;
constraint add_constr {
    A + B >= 0;
    A + B <= 10;
 }
...
covergroup cov;
        A_cp: coverpoint A;
        B_cp: coverpoint B;
        cr1: cross A_cp, B_cp;
endgroup
```

"Example 6" shows the generated enhanced functional coverage model for "cr1" defined in "Example 5" after applying the constraint solver techniques and extracting the valid range of values and then re-writing the enhanced concise form of functional coverage.

"Example 6", shows the generated enhanced functional coverage model, the generated cover-point for both "A" and "B" variables, contains bins for the allowed ranges of both variables (i.e. from 0 to 10), and the enhanced "cr1" cross coverage, is ignoring all the out of range values for both "A" and "B" which make the summation out of the allowed range from 0 to 10 (i.e. sum (A, B)>10).

Example 6: Enhanced cr1 cross

```
A_cp : coverpoint A {
   option.weight = 0;
   bins A_bins[] = {[64'd0:64'd10]};
}
B_cp : coverpoint B {
   option.weight = 0;
   bins B_bins[] = {[64'd0:64'd10]};
}
cr1 : cross A_cp, B_cp {
   option.weight = 66;
   ignore_bins unreachable_bins = ((binsof(A_cp) intersect {64'd1} && binsof(B_cp) intersect {64'd10}) ||
(binsof(A_cp) intersect {64'd2} && binsof(B_cp) intersect {64'd9, 64'd10}) || (binsof(A_cp) intersect {64'd3} &&
binsof(B_cp) intersect {64'd8, 64'd9, 64'd10}) || (binsof(A_cp) intersect {64'd4} && binsof(B_cp) intersect
{64'd7, 64'd8, 64'd9, 64'd10}) || (binsof(A_cp) intersect {64'd5} && binsof(B_cp) intersect {64'd6, 64'd7, 64'd8,
64'd9, 64'd10}) || (binsof(A_cp) intersect {64'd6} && binsof(B_cp) intersect {64'd5, 64'd6, 64'd7, 64'd8, 64'd9,
64'd10}) || (binsof(A_cp) intersect {64'd7} && binsof(B_cp) intersect {64'd4, 64'd5, 64'd6, 64'd7, 64'd8, 64'd9,
64'd10}) || (binsof(A_cp) intersect {64'd8} && binsof(B_cp) intersect {64'd3, 64'd4, 64'd5, 64'd6, 64'd7, 64'd8,
64'd9, 64'd10}) || (binsof(A_cp) intersect {64'd9} && binsof(B_cp) intersect {64'd2, 64'd3, 64'd4, 64'd5, 64'd6,
64'd7, 64'd8, 64'd9, 64'd10}) || (binsof(A_cp) intersect {64'd10} && binsof(B_cp) intersect {64'd1, 64'd2, 64'd3,
64'd4, 64'd5, 64'd6, 64'd7, 64'd8, 64'd9}) || (binsof(A_cp) intersect {64'd10} && binsof(B_cp) intersect
{64'd10}));
}
```

B. *Static Enhancement of Design-Centric Functional Coverage*

Now, it's the turn for statically enhancing the design-centric functional coverage models to complete the proposed framework.

"Example 3" shows RTL of state machine, which has one unreachable state "wait_idle" due to design conditions, so the coverage of cover-point defined for "int_state" variable, will never achieve 100%, and the bins assigned to "wait_idle" state will never be hit, and hence the hit count of this bins will always stuck to 0.

By using formal-based analysis techniques, DUV is analyzed for reachability, to identify all un-reachable coverage bins due to design conditions, and then promote these un-reachable bins to be excluded from the functional coverage results. Finally the coverage results of these un-reachable bins will be deducted from the overall functional coverage calculations, which accordingly enhances and increases the overall coverage percentage. For example, if the RTL contains only one cover-point as its design-centric functional coverage model, and this cover-point has 4 bins, one of these bins is un-reachable due to design conditions, and the other 3 bins are reachable, so after running simulation with a complete test-bench and measuring the coverage, the results will always stuck at 75% (3 bins out of 4 bins have been hit), and after applying formal-based analysis on this design and its cover-point, which will exclude the un-reachable bin from the calculation of functional coverage results, which will results to an overall functional coverage percentage equals to 100%.

VI. EXPERIMENTAL RESULTS

Deploying both constraint solver technologies and formal-based analysis techniques, a complete framework has been proposed to statically enhance the overall functional coverage models for DUV.

This framework has been experimented on many in-house designs, and it has been proved to show improvements and enhancements to the defined functional coverage model. When the input original functional coverage has some bins which will never be reached due to either test-bench constraints, or design conditions, we will get enhanced functional coverage model which eliminate these unreachable bins, and hence the overall coverage will be enhanced.

As more illustrative examples to results of the proposed framework, Interleaver design [10] is used to show the experiment results of this new approach by applying both constraint-solver technologies and formal-based analysis techniques to enhance the overall functional coverage models defined to verify Interleaver design. Interleaver design is used to scramble the byte order of incoming data in order to aid error detection and correction schemes such as Reed Solomon/Viterbi. "Table 1" shows the results of coverage items defined in "up_cvg" cover-group when applying constraint solver technologies to enhance the input/output functional coverage model, against the results of the same cover-group without using the approach. The results show a huge enhancements in the coverage results for both "up_cvg::upcov_data" and "up_cvg::upcov_sync" cover-points after using the newly enhanced version of functional coverage models for the selected cover-points.

"Table 2", shows the results of deploying formal-based analysis on the same Interleaver design to enhance design-centric functional coverage, and exclude the unreachable bins, and it shows the results of "sm_cvg" cover-group defined in the design's "dut" module. From the results, there are good enhancements to both "sm_cvg::in_hsXint_state" and "sm_cvg::out_hsXint_state" crosses, after applying the formal-based techniques to exclude un-reachable bins.

Table 1: Input/output functional coverage results of Interleaver design

Coverage item name	Type	Coverage results without new approach	Coverage results with new approach
up_cvg::upcov_data	Cover-point	0.7%	100%
up_cvg::upcov_sync	Cover-point	40%	100%
up_cvg::up_delay	Cover-point	95%	100%

Table 2: Design-Centric functional coverage results of Interleaver design

Coverage item name	Type	Coverage results without new approach	Coverage results with new approach
sm_cvg::int_state	Cover-point	92.3%	96%
sm_cvg::in_hsXint_state	Cross	46.1%	92.3%
sm_cvg::out_hsXint_state	Cross	46.1%	100%

"Table 3", shows the results of applying the approach on Ethernet MAC design [11], it shows the results of applying constraint solver technologies to enhance the functional coverage of "ethmac_rxtx_seq_cg" cover-group, "tx_size" cover-point has good increase in the coverage results after applying the proposed approach, and both "rx_size" cover-point and "rx_tx_size" cross coverage show small enhancements. All of the coverage items didn't achieve 100% coverage due to incomplete test-bench, but the proposed approach shows enhancements to the overall functional coverage results.

"Table 4", shows the results of deploying formal-based analysis techniques to enhance design-centric functional coverage of Ethernet MAC design, from the results, there are huge enhancements to "HASH0_1_Cvg::BYTE2", "HASH0_1_Cvg::BYTE3", "HASH0_1_Cvg::BYTE4", and "HASH0_1_Cvg::BYTE5" cover-points, after excluding unreachable bins in all of these cover-points. Although there are some other excluded bins for "EthRw_Cvg::wrXaddrXdata" cross, but all of these exclusions are grouped into the "ignore_bin" and then there is no change to the coverage percentage.

Finally, the experimental results show good enhancements for input/output functional coverage models after applying constraints solver technologies to exclude unreachable bins due to test-bench constraints, also the results show good enhancements for design-centric functional coverage models, by using formal-based analysis techniques

to exclude the unreachable bins due to design conditions, which again enhanced the design-centric functional coverage. This approach shows excellent enhancements for the overall functional coverage of DUV.

Table 3: Input/output functional coverage results of Ethernet MAC design

Coverage item name	Type	Coverage results without new approach	Coverage results with new approach
ethmac_rxtx_seq_cg::tx_size	Cover-point	85.9%	92%
ethmac_rxtx_seq_cg::rx_size	Cover-point	84.4%	84.6%
ethmac_rxtx_seq_cg::rx_tx_size	Cross	2.9%	3.1%

Table 4: Design-Centric functional coverage results of Ethernet MAC design

Coverage item name	Type	Coverage results without new approach	Coverage results with new approach
HASH0_1_Cvg::BYTE2	Cover-point	0.7%	100%
HASH0_1_Cvg::BYTE3	Cover-point	0.7%	100%
HASH0_1_Cvg::BYTE4	Cover-point	0.7%	100%
HASH0_1_Cvg::BYTE5	Cover-point	0.7%	100%
EthRw_Cvg::wrXaddrXdata	Cross	25%	25%

Also, this methodology can be also useful for verification engineer to help them to write a concise form of the required functional coverage model by writing simple coverage constructs (i.e. cover-points and cross), then deploy this methodology to either exclude the unreachable bins from the design-centric functional coverage, or to get a more concise form of input/output functional coverage models, which is respecting design and test-bench's specification.

VII. CONCLUSION

As mentioned above, writing complete, correct, and concise functional coverage models to verify the correctness of SoC is a challenging task, and having a static methodology to enhance the written functional coverage mode is definitely helpful in writing correct and concise functional coverage models.

In this paper, a complete framework has been introduced to help verification engineer to enhance the input/output as well as design-centric functional coverage models, by using constraint solvers and formal-based techniques.

The proposed work depends on the written RTL and test-benches, and it assumes that they are correct. This framework is useful for verification engineers to enhance their existing functional coverage models or to start writing their functional coverage models, in a more consistent way and concise form with their design and test-bench's specification, and hence achieve the required coverage results.

VIII. REFERENCES

[1] H. D. Foster, "Trends in Functional Verification: A 2014 Industry Study", In Design Automation Conference (DAC) [July, 2015].

[2] C. Spear "SystemVerilog for Verification" A Guide to Learning the Testbench Language Features, 2nd edition.

[3] B. Bailey, "The Wake of the Sleeping GiantVerification." Scalable Verification Technical Publications. Internet: http://www.mentor.com [April, 2002].

[4] S. Verma, I. G. Harris and K. Ramineni," Automatic Generation of Functional Coverage from Behavioral Verilog Description", In Design Automation and Test in Europe, 2007.

[5] S. Verma, I.G. Harris and K. Ramineni, "Automatic generation of functional coverage models from CTL", In IEEE International High Level Design Validation and Test Workshop, 2007

[6] E. El Mandouh, A. G. Wassal, "Automatic Generation of Functional Coverage Models", In IEEE International Symposium on Circuits and Systems (ISCAS), 2016

[7] IEEE 1800-2012, System Verilog Unified Hardware Design, Specification and Verification Language, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[8] Questa inFact, https://www.mentor.com/products/fv/infact

[9] Questa CoverCheck, https://www.mentor.com/products/fv/questa-covercheck

[10] Shipped example with Mentor Graphics – QuestaSim tool

[11] Opencores benchmarks http://opencores.org