Static Checking for Correctness of Functional Coverage Models

Wael Mahmoud
Agenda

• Introduction
• Functional coverage closure problems
• Static enhancements of functional coverage models
  – Part A: Enhancements of input/output functional coverage
  – Part B: Enhancements of design-centric functional coverage
• Results and conclusion
Introduction

• Today’s designs are getting more bigger and more complex (SoC and ASIC)
• Achieving fully verified SoC is an arduous task.
• Recent industry studies, shows that the average total project time spent in verification was 57%.
• Number of projects that spent more than 80% of time in verification has been increased from the past.
Motivation

• The intent of verifying “SoC” is to ensure that the design is an accurate representation of the specification.
• Functional coverage provides visibility into the verification process.
• Writing a complete, correct, and concise functional coverage models, that conform design functionality to specs.
• Accelerate functional coverage closure.
• Assist verification teams with techniques to write concise functional coverage models.
Agenda

• Introduction

• Functional coverage closure problems

• Static enhancements of functional coverage models
  – Part A: Enhancements of input/output functional coverage
  – Part B: Enhancements of design-centric functional coverage

• Results and conclusion
Functional coverage closure problems

• Functional coverage closure can’t be achieved due to many problems, like:
  1. Problems with input stimuli, like: incomplete, insufficient, and/or redundant stimuli
  2. Incorrect implementation of functional coverage model.
1- Incomplete/redundant input stimuli

• Write more directed tests to cover specific corner case scenarios.
• Run test cases multiple times with different random seeds, and hope more interesting scenarios are covered.
• Alternatively, try out other methodologies (e.g. intelligent test-bench automation “iTBA” tools) when applicable.
2- Incorrect implementation of functional coverage model

• Functional coverage model is contradicting with test-bench’s or design’s constraints.
• The proposed methodology will shows that there are no possible solutions.

```verilog
class rand logic unsigned [0:3] a;

constraint C {
  a inside {[10:15]};
}

cp_a: coverpoint a {
  bins b1[] = {[0:9]};
}
```

No input stimuli can achieve coverage closure
3- Non-optimized forms of functional coverage

Functional coverage model is not written in an optimized form (i.e. it is not considering unreachable bins).

Input functional coverage

```verilog
rand bit [3:0] A;
constraint A_constr {
    A < 8;
}
...
covergroup cov;
    A_cp: coverpoint A;
endgroup
```

Coverage of A_cp is 50 %

Design-centric functional coverage

```verilog
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)
                    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
covergroup sm_cvg @(posedge pins.clk);
    coverpoint int_state;
endgroup
```

Coverage of wait_idle is 0 %
Agenda

• Introduction
• Functional coverage closure problems
  • Static enhancements of functional coverage models
    – Part A: Enhancements of input/output functional coverage
    – Part B: Enhancements of design-centric functional coverage
• Results and conclusion
This paper proposes a complete framework to enhance functional coverage models of both “input/output” and “design-centric”

“Part A”
 Intelligent test-bench automation (iTBA) tool, which internally use constraint solver technologies, is used to enhance “input/output” functional coverage model

“Part B”
 Formal-based coverage analysis tool, which internally use formal-based analysis, is used to enhance “design-centric” functional coverage model
Intelligent test-bench automation (iTBA) tools

- iTBA tools achieves input coverage 10-100x faster than random stimulus.
- More than 100x productive than directed test
  - It provides an efficient description of stimulus scenarios
  - It reduces time spent in writing testbenches
- More than 10X efficient than constrained random tests
  - No redundant tests
  - It helps to find tough corner case bugs easier and earlier
- This paper is using iTBA tool to enhance input/output functional coverage models.
Part A: Enhancements of input/output functional coverage (1/3)

1. Import the test-bench into Questa inFact
2. Questa inFact automatically extracts test-bench’s variables, constraints of stimulus class and functional coverage model
3. Internally solving all variables contributing in functional coverage item against the test-bench’s constraint
4. Generate an enhanced functional coverage model
Part A: Enhancements of input/output functional coverage (2/3)

Original F.C.

```vhdl
rand bit [3:0] A;
constraint A_constr {
    A < 8;
}
covergroup cov;
    A_cp: coverpoint A;
endgroup
```

Enhanced F.C.

```vhdl
A_cp : coverpoint A {
    option.weight = 8;
    bins cfg_item_inst_A[] = {[64'd0:64'd7]};
}
```
Part A: Enhancements of input/output functional coverage (3/3)

```
rand logic unsigned [0:3] A, B;
constraint add_constr {
    A + B >= 0;
    A + B <= 10;
}

covergroup cov;
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})));
```

**Original F.C.**

**Enhanced F.C.**
Agenda

• Introduction
• Functional coverage closure problems
  • Static enhancements of functional coverage models
    – Part A: Enhancements of input/output functional coverage
    – Part B: Enhancements of design-centric functional coverage
• Results and conclusion
Manual coverage closure (design-centric)

- Coverage verification is to verify that coverage goal is achieved in simulation
- Testing all possible scenarios and states are generally so hard
- Coverage holes indicate:
  - Some blocks, states and transactions in the design are unreachable
  - Some coverage items are reachable with complex test scenarios
- Huge effort and time are consumed to determine unreachable code and to create complex tests
Coverage closure using formal-based analysis (design-centric)

- Save time that would been spent for manually analyzing coverage holes
- CoverCheck provides an automatic solutions for the Coverage Closure challenges
  - Automatically exclude coverage items for unreachable code
  - Automatically generate Witness waveforms for reachable code
- Customers can easily improve the code and the tests for better coverage metrics
Part B: Enhancements of design-centric functional coverage

Run Questa CoverCheck on DUT and pass the UCDB generated from a simulation run.

Questa CoverCheck automatically analyzes DUT for formal/static reachability using formal-based analysis.

Exclusions file is generated with unreachable functional coverage bins, which is applied to simulation UCDB to exclude unreachable functional coverage.
Agenda

• Introduction
• Functional coverage closure problems
• Static enhancements of functional coverage models
  – Part A: Enhancements of input/output functional coverage
  – Part B: Enhancements of design-centric functional coverage
• Results and conclusion
## Results

### Input/Output F.C.

<table>
<thead>
<tr>
<th>Coverage item name</th>
<th>Type</th>
<th>Coverage results without new approach</th>
<th>Coverage results with new approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>up_cv::upcov_data</td>
<td>Cover-point</td>
<td>0.7%</td>
<td>100%</td>
</tr>
<tr>
<td>up_cv::upcov_sync</td>
<td>Cover-point</td>
<td>40%</td>
<td>100%</td>
</tr>
<tr>
<td>up_cv::up_delay</td>
<td>Cover-point</td>
<td>95%</td>
<td>100%</td>
</tr>
</tbody>
</table>

### Design-Centric F.C.

<table>
<thead>
<tr>
<th>Coverage item name</th>
<th>Type</th>
<th>Coverage results without new approach</th>
<th>Coverage results with new approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>sm_cv::int_state</td>
<td>Cover-point</td>
<td>92.3%</td>
<td>96%</td>
</tr>
<tr>
<td>sm_cv::in_hsXint_state</td>
<td>Cross</td>
<td>46.1%</td>
<td>92.3%</td>
</tr>
<tr>
<td>sm_cv::out_hsXint_state</td>
<td>Cross</td>
<td>46.1%</td>
<td>100%</td>
</tr>
</tbody>
</table>

### Ethernet Design

<table>
<thead>
<tr>
<th>Coverage item name</th>
<th>Type</th>
<th>Coverage results without new approach</th>
<th>Coverage results with new approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>ethmac_rxtx_seq_c::tx_size</td>
<td>Cover-point</td>
<td>85.9%</td>
<td>92%</td>
</tr>
<tr>
<td>ethmac_rxtx_seq_c::rx_size</td>
<td>Cover-point</td>
<td>84.4%</td>
<td>84.6%</td>
</tr>
<tr>
<td>ethmac_rxtx_seq_c::rx_size</td>
<td>Cross</td>
<td>2.9%</td>
<td>3.1%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coverage item name</th>
<th>Type</th>
<th>Coverage results without new approach</th>
<th>Coverage results with new approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASH0_1_Cvg::BYTE2</td>
<td>Cover-point</td>
<td>0.7%</td>
<td>100%</td>
</tr>
<tr>
<td>HASH0_1_Cvg::BYTE3</td>
<td>Cover-point</td>
<td>0.7%</td>
<td>100%</td>
</tr>
<tr>
<td>HASH0_1_Cvg::BYTE4</td>
<td>Cover-point</td>
<td>0.7%</td>
<td>100%</td>
</tr>
<tr>
<td>HASH0_1_Cvg::BYTE5</td>
<td>Cover-point</td>
<td>0.7%</td>
<td>100%</td>
</tr>
<tr>
<td>EthRw_Cvg::wrXaddrXdata</td>
<td>Cross</td>
<td>25%</td>
<td>25%</td>
</tr>
</tbody>
</table>
Functional coverage development become easier

Testbench constraints
- Automatically exclude unreachable coverage bins, and provide concise forms of F.C., which leverage coverage results

Design conditions
- Automatically exclude unreachable bins, which leads to improve DUT for better coverage metrics

Detect conflicts
- Constraints and original functional coverage conflict can be easily detected

Minimize manual mistakes
- Manual writing of exclusion bins is a common source of mistakes
Conclusion

• Writing complete, correct, and concise functional coverage models to verify the correctness of SoC is a challenging task.
• The proposed methodology uses constraint solvers and formal-based analysis to enhance functional coverage models.
• The proposed methodology is helpful in writing correct and concise functional coverage models.
• The proposed methodology helps verification engineer to start writing functional coverage models, or re-calibrate existing coverage metrics.
• Proposed methodology saves effort and time to determine unreachable code or coverage bins.
Thank You!

Any questions?