How to Avoid the Pitfalls of Mixing Formal and Simulation Coverage

Mark Eslinger, Siemens
Joseph V Hupcey III, Siemens
Nicolae Tusinschi, Siemens
Sound Familiar?

“I want to see coverage data from all sources combined into a single report, so we can see our progress at-a-glance.”
Introduction

• The goal: Use multiple verification strategies to ensure the Device Under Test (DUT) behaves as specified

• The challenge: comparing and combining the results from each verification strategy to the verification plan

• The most common request: merging simulation and formal coverage

• The most common problem(s): understanding what formal coverage is, proper merging of formal and simulation coverage data
Re-Cap: Simulation Coverage

• Code coverage
  • The % of RTL code that have been executed by test(s)
  • “Dead” / “Unreachable” code could imply a bug
  • Says nothing about DUT conformance to the specification

• Functional coverage
  • Metric of how much design functionality has been exercised
  • Spec./functional feature mapped to a “cover point”
  • Goal is 100% coverage conformance to specification
Brief Digression:
Formal Results Are Valid for All Inputs & All Time

Analogy

Finding solutions to $ax^2 + bx + c = 0$

• Constrained-random simulation approach:
  Randomly plug-in numbers in the hope you eventually satisfy the equation

• Formal approach:

  $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ for all values of $t$

✓ The formal solution is valid for all inputs and all time → it is exhaustive
Formal Coverage

Reachability
• A sequence of input signals which can reach the coverage element

Observability
• All possible state space paths from a selected node to signals in an assertion

Structural Cone of Influence (COI)
• All logic from a specified node back to the primary inputs

Mutation
• Automatically inserted “mutations” in the DUT cause an assertion failure
Pitfalls of Merging Sim & Formal Coverage

• #1 Caveat: just because something is “covered” doesn’t mean it’s properly verified

• Simulation coverage only reflects specific forward paths the simulation has traversed from the inputs through the state space, for a specific set of stimuli

• Some types of formal coverage also reflect a “forward traversal” from the inputs, but often the amount of logic “covered” is greater than simulation

• Other types of formal coverage “works backwards” from an output

• Code coverage from simulation represents end-to-end cluster/SOC level testing, while formal is typically run at the block level
Example 1: Basic Sim. Vs. Formal Code Coverage

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>OUTPUTS</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sel</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Focusing on output B requirement:
B output is a pulse

Property: B => !B
Example 1: Basic Sim. Vs. Formal Code Coverage

Simulation

Coverage includes logic not related to satisfying the requirement

Formal

Coverage only includes logic related to satisfying the requirement
Example 2: Sim. Vs. Formal FSM Code Coverage

The following property was run in simulation and formal against a Verilog model of this state machine:

```
a_mout_mutex: assert property (@(posedge clk) $onehot0(mout) );
```

Simulation was run with one value of the select signal used which exercised the output

The property passed in simulation and was proven in formal
Example 2: Sim. Vs. Formal FSM Code Coverage

Simulation
```
always @(posedge clk or negedge rstn)
if (!rstn) cnt <= 3'b000;
else cnt <= cnt + 1;
assign done = (cnt == 3'b111) ? 1'b1 : 1'b0;
always @(posedge clk or negedge rstn)
if (!rstn) cstate <= ST1;
else cstate <= nstate;
always @*
case(cstate)
ST1: if (start) nstate <= ST2;
else nstate <= ST1;
ST2: if (sel == 2'b01) nstate <= ST3;
else if (sel == 2'b10) nstate <= ST4;
else nstate <= ST2;
ST3: if (done) nstate <= ST1;
else nstate <= ST3;
ST4: if (done) nstate <= ST1;
else nstate <= ST4;
default: nstate <= ST1;
endcase
always @*
if (cstate == ST3)
  mout <= 2'b01;
else if (cstate == ST4)
  mout <= 2'b10;
else mout <= 2'b00;
```

Formal
```
always @(posedge clk or negedge rstn)
if (!rstn) cnt <= 3'b000;
else cnt <= cnt + 1;
assign done = (cnt == 3'b111) ? 1'b1 : 1'b0;
always @(posedge clk or negedge rstn)
if (!rstn) cstate <= ST1;
else cstate <= nstate;
always @*
case(cstate)
ST1: if (start) nstate <= ST2;
else nstate <= ST1;
ST2: if (sel == 2'b01) nstate <= ST3;
else if (sel == 2'b10) nstate <= ST4;
else nstate <= ST2;
ST3: if (done) nstate <= ST1;
else nstate <= ST3;
ST4: if (done) nstate <= ST1;
else nstate <= ST4;
default: nstate <= ST1;
endcase
always @*
if (cstate == ST3)
  mout <= 2'b01;
else if (cstate == ST4)
  mout <= 2'b10;
else mout <= 2'b00;
```

Simulation run with sel = 1, uses 3 states of the FSM and more

Formal only needs the final logic for the full proof, no FSM needed
Example 3: Closing Code Coverage Holes

**Simulation**

```verilog
always @*
if (in1) n1 <= 1'b1;
else n1 <= 1'b0;

always @*
if (in2 && !in1) n2 <= 1'b1;
else n2 <= 1'b0;

always @*
if (in3 && !in2 && !in1) n3 <= 1'b1;
else n3 <= 1'b0;

always @(posedge clk or negedge rstn)
if (!rstn) A <= 1'b0;
else A <= n1;

always @(posedge clk or negedge rstn)
if (!rstn) B <= 1'b0;
else B <= n2;

always @(posedge clk or negedge rstn)
if (!rstn) C <= 1'b0;
else C <= n3;
```

**Specification:**
Outputs are mutex-based on an encoding of the 3 inputs

**Good news:**
Only one coverage hole remains after simulation
Example 3: Let’s Quickly Close This ...

```verilog
a_bogus: assert property (@(posedge clk) in1 |-> n1 );
```

Formal coverage shows the first two lines are covered.

Merged formal + sim is 100% **Green**! Great news, right?
Example 3: Not so fast ... Look at the Property

```
a_bogus: assert property (@(posedge clk) in1 |-> n1 );

• This property is actually useless – it tests nothing
• It is not tied to a testplan, or to the verification of any design requirements

A better approach:

The design requires the 3 outputs to be mutex, thus a more useful property which checks this requirement is:
```
a_good: assert property (@(posedge clk) $onehot0({A,B,C}) );
```
Example 3: NOW the Requirement Is Proven

```
a_good: assert property (@(posedge clk) $onehot0({A,B,C}) );
```

This formal coverage result reflects requirements and can be merged with sim
Example 4: 100% Coverage – But There is Still a Bug!

**Specification:**
- Outputs are mutex-based -- check the ‘out1’, ‘out2’ signals
- The FSM to only be in state 2 for no more than 3 cycles

Great news, right?

Both sim AND formal are 100% Green!

```verilog
always @(posedge clk or negedge rstn)
  if (!rstn) cnt <= 2'b00;
  else cnt <= cnt + 1;

always @(posedge clk or negedge rstn)
  if (!rstn) cstate <= ST1;
  else cstate <= nstate;

always *
  case(cstate)
    ST1: if (sel) nstate <= ST2;
        else nstate <= ST1;
    ST2: if (&cnt) nstate <= ST3;
        else nstate <= ST2;
    ST3: nstate <= ST1;
    default: nstate <= ST1;
  endcase

assign out1 = (cstate == ST3);

always @(posedge clk or negedge rstn)
  if (rstn) out2 <= 1'b0;
  else if (out2) out2 <= 1'b0;
  else if (cnt[1]) out2 <= 1'b1;
  else out2 <= out2;
```
Example 4: Not So Fast ...

All the code is traversed, BUT the functional behavior is incorrect!

Recall the key requirement:

"the FSM to only be in state 2 for no more than 3 cycles"

**Solutions**

A) Have the forethought to manually write and include the following assertion

```verilog
a_st2_3_max: assert property (@(posedge clk) disable iff (!rstn)
$rose(st2) |-> ##[1:3] !st2 );
```

B) Use automated Mutation coverage
Example 4: Using Mutation Coverage to Reveal Bugs in 100% Covered DUTs

Mutation coverage reveals the missing requirement and related test from the testplan.
## Summary of Formal vs. Sim Coverage Differences

<table>
<thead>
<tr>
<th>Formal Code Coverage</th>
<th>Simulation Code Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Property based</strong></td>
<td>Vector based</td>
</tr>
<tr>
<td><strong>Exact: Only logic used in proof is covered</strong></td>
<td>Generous: Whatever a vector hits, is covered</td>
</tr>
<tr>
<td><strong>Covered logic only related to proof</strong></td>
<td>Covered logic may be unrelated to test</td>
</tr>
<tr>
<td><strong>Only needed statements in a block covered</strong></td>
<td>All statements in block covered by default</td>
</tr>
<tr>
<td><strong>Calculated from synthesized netlist</strong></td>
<td>Calculated from RTL (may include testbench)</td>
</tr>
<tr>
<td><strong>Formal uses abstractions – impacts coverage</strong></td>
<td>No abstractions used</td>
</tr>
<tr>
<td><strong>Qualify input constraints used in a proof</strong></td>
<td>Input constraints only impact reachability</td>
</tr>
<tr>
<td><strong>Reachability analysis used for exclusions</strong></td>
<td>Reachability analysis used for exclusions, done with formal (CoverCheck)</td>
</tr>
</tbody>
</table>
Recommendations

• Close code coverage for each verification engine separately
  • Focus on improving testbench completeness and robustness in each domain
  • In the formal domain run both proof core and mutation coverage to check testbench completeness

• Use test planning to assign which verification engine verifies which parts of the design
  • When mixing formal and sim coverage, try to keep it to instance boundaries
  • Have peer reviews of coverage to ensure short cuts are not being taken

• Know where your code coverage comes from: Formal vs. Simulation
  • Keep code coverage data from each domain separate in the main coverage database
  • The reporting must also make it clear where the coverage data came from

• Avoid adding targeted tests/properties to trivially get to 100% coverage
  • Coverage holes point to an incomplete testplan, and ultimately an incomplete/weak testbench
  • When adding properties to close code coverage holes, testplan design requirements is your guide!
Summary

• It is possible to combine the strengths of simulation and formal to ensure that your DUT behaves as specified

• Understanding how sim and formal coverage metrics work in isolation – and how they combine – provides a holistic picture of your verification

• Mutation analysis coverage is a powerful tool to make sure 100% coverage doesn’t fool you into missing bugs in your testbench
Questions?