2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

How to Avoid the Pitfalls of Mixing Formal and Simulation Coverage

Mark Eslinger, Siemens
Joseph V Hupcey III, Siemens
Nicolae Tusinschi, Siemens

SIEMENS

accellera SYSTEMS INITIATIVE

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 <u>functionality</u> 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

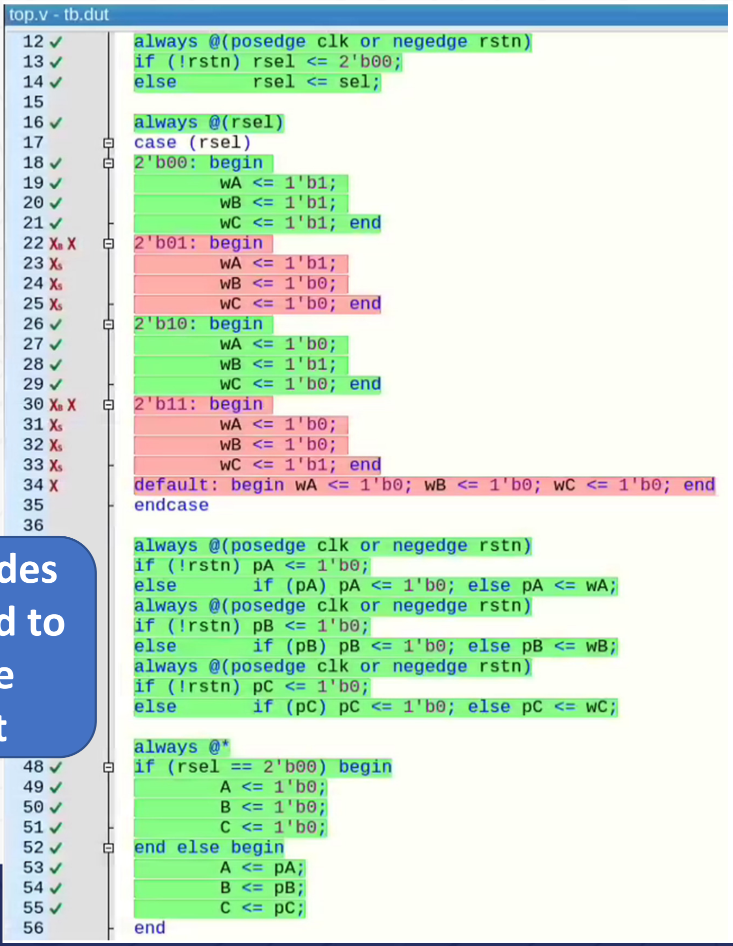
INPUTS	OUTPUTS		
Sel	A	B	C
00	0	0	0
01	1	0	0
10	0	1	0
11	0	0	1

Focusing on output B requirement:
B output is a pulse

Property: B => !B

Example 1: Basic Sim. Vs. Formal Code Coverage

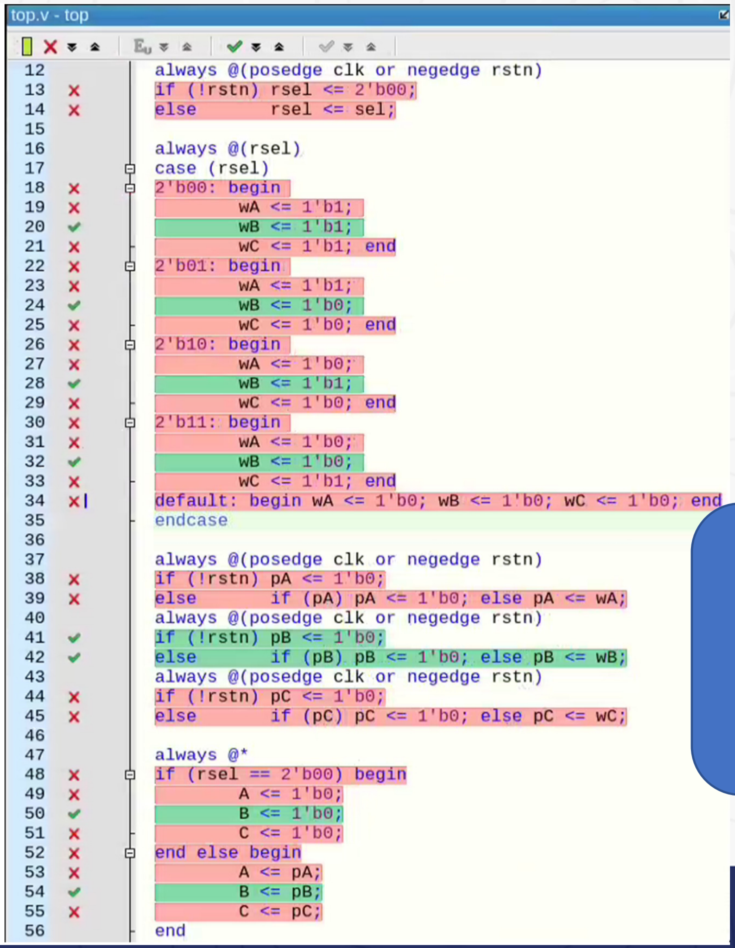
Simulation



top.v - tb.dut

```
always @(posedge clk or negedge rstn)
if (!rstn) rsel <= 2'b00;
else       rsel <= sel;

always @(rsel)
case (rsel)
2'b00: begin
          wA <= 1'b1;
          wB <= 1'b1;
          wC <= 1'b1; end
2'b01: begin
          wA <= 1'b1;
          wB <= 1'b0;
          wC <= 1'b0; end
2'b10: begin
          wA <= 1'b0;
          wB <= 1'b1;
          wC <= 1'b0; end
2'b11: begin
          wA <= 1'b0;
          wB <= 1'b0;
          wC <= 1'b1; end
default: begin wA <= 1'b0; wB <= 1'b0; wC <= 1'b0; end
endcase

always @(posedge clk or negedge rstn)
if (!rstn) pA <= 1'b0;
else       if (pA) pA <= 1'b0; else pA <= wA;
always @(posedge clk or negedge rstn)
if (!rstn) pB <= 1'b0;
else       if (pB) pB <= 1'b0; else pB <= wB;
always @(posedge clk or negedge rstn)
if (!rstn) pC <= 1'b0;
else       if (pC) pC <= 1'b0; else pC <= wC;

always @*
if (rsel == 2'b00) begin
         A <= 1'b0;
         B <= 1'b0;
         C <= 1'b0;
end else begin
         A <= pA;
         B <= pB;
         C <= pC;
end
```

Formal



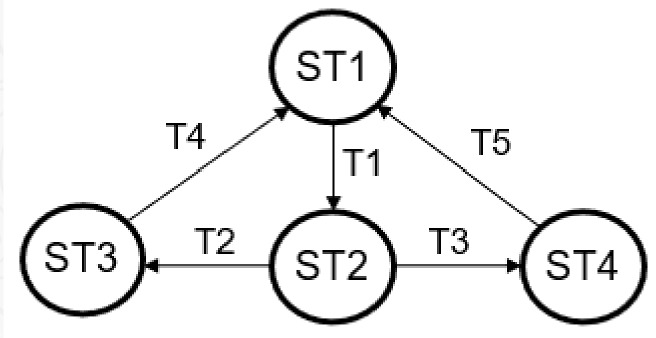
top.v - top

```
always @(posedge clk or negedge rstn)
if (!rstn) rsel <= 2'b00;
else       rsel <= sel;

always @(rsel)
case (rsel)
2'b00: begin
          wA <= 1'b1;
          wB <= 1'b1;
          wC <= 1'b1; end
2'b01: begin
          wA <= 1'b1;
          wB <= 1'b0;
          wC <= 1'b0; end
2'b10: begin
          wA <= 1'b0;
          wB <= 1'b1;
          wC <= 1'b0; end
2'b11: begin
          wA <= 1'b0;
          wB <= 1'b0;
          wC <= 1'b1; end
default: begin wA <= 1'b0; wB <= 1'b0; wC <= 1'b0; end
endcase

always @(posedge clk or negedge rstn)
if (!rstn) pA <= 1'b0;
else       if (pA) pA <= 1'b0; else pA <= wA;
always @(posedge clk or negedge rstn)
if (!rstn) pB <= 1'b0;
else       if (pB) pB <= 1'b0; else pB <= wB;
always @(posedge clk or negedge rstn)
if (!rstn) pC <= 1'b0;
else       if (pC) pC <= 1'b0; else pC <= wC;

always @*
if (rsel == 2'b00) begin
         A <= 1'b0;
         B <= 1'b0;
         C <= 1'b0;
end else begin
         A <= pA;
         B <= pB;
         C <= pC;
end
```

Coverage includes logic not related to satisfying the requirement

Coverage only includes logic related to satisfying the requirement

Example 2: Sim. Vs. Formal FSM Code Coverage



T1: start = 1
T2: sel = 01
T3: sel = 10
T4: done = 1
T5: done = 1
Note: Same state transitions not shown

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

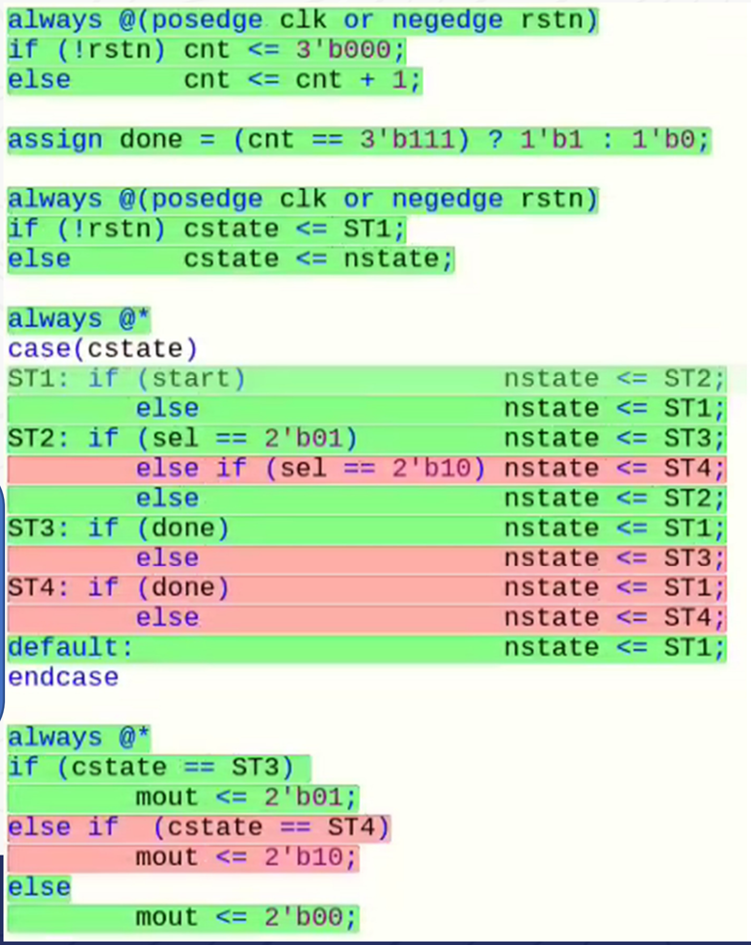
```
a_mout_mutex: assert property (@(posedge clk) $onehot0(mout) );
```

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

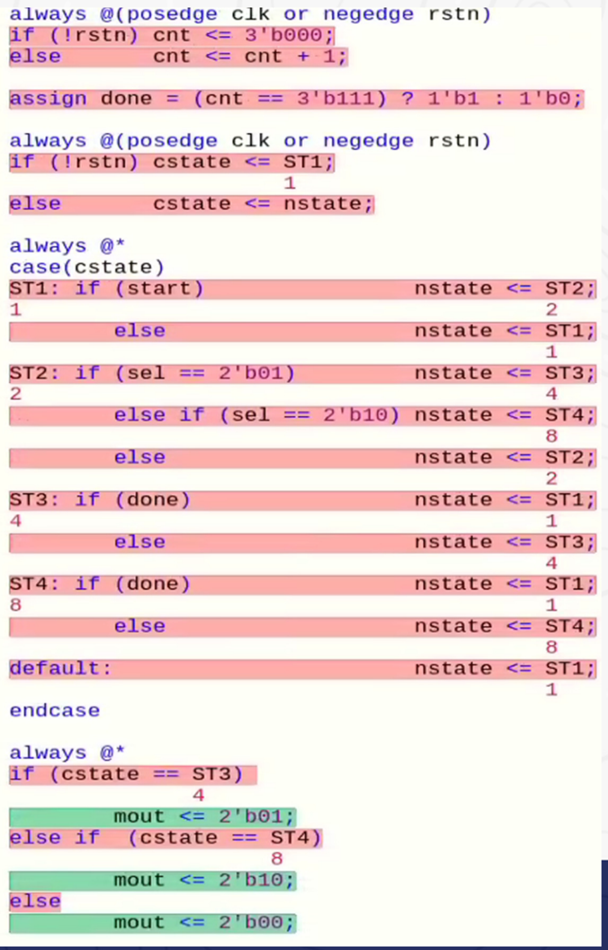
Example 2: Sim. Vs. Formal FSM Code Coverage

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;
                     1
else       cstate <= nstate;

always @*
case(cstate)
ST1: if (start)                     nstate <= ST2;
1                                             2
          else                      nstate <= ST1;
                                              1
ST2: if (sel == 2'b01)              nstate <= ST3;
2                                             4
          else if (sel == 2'b10)    nstate <= ST4;
                                              8
          else                      nstate <= ST2;
                                              2
ST3: if (done)                      nstate <= ST1;
4                                             1
          else                      nstate <= ST3;
                                              4
ST4: if (done)                      nstate <= ST1;
8                                             1
          else                      nstate <= ST4;
                                              8
default:                            nstate <= ST1;
                                              1
endcase

always @*
if (cstate == ST3)
               4
          mout <= 2'b01;
else if  (cstate == ST4)
                     8
          mout <= 2'b10;
else
          mout <= 2'b00;
```

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

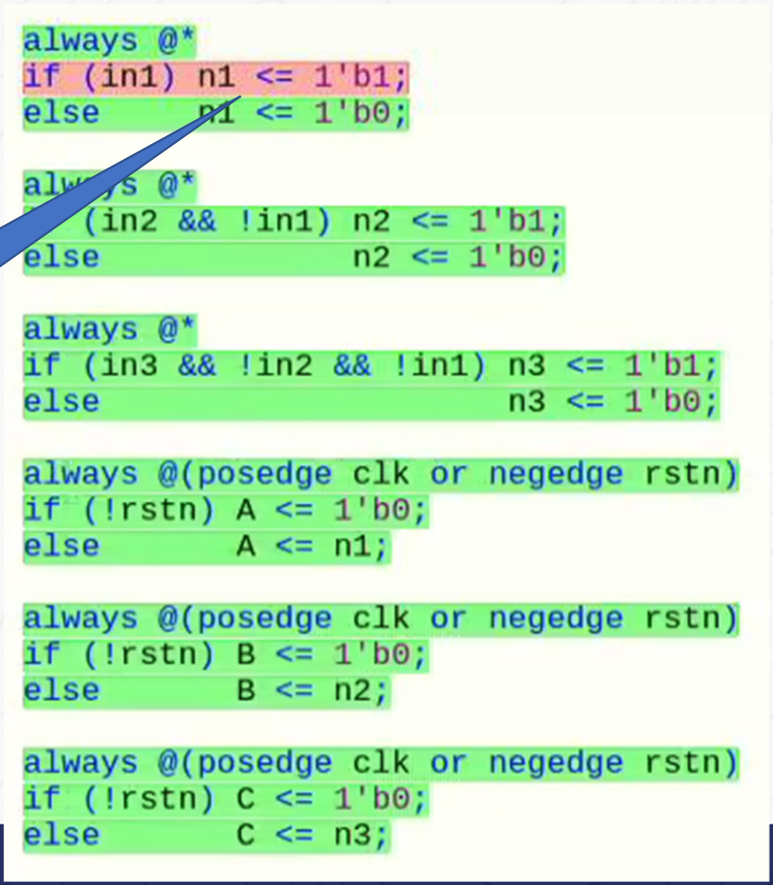
Formal only needs the final logic for the full proof, no FSM needed

Example 3: Closing Code Coverage Holes

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

<u>Good news:</u>
Only one coverage hole remains after simulation

Simulation



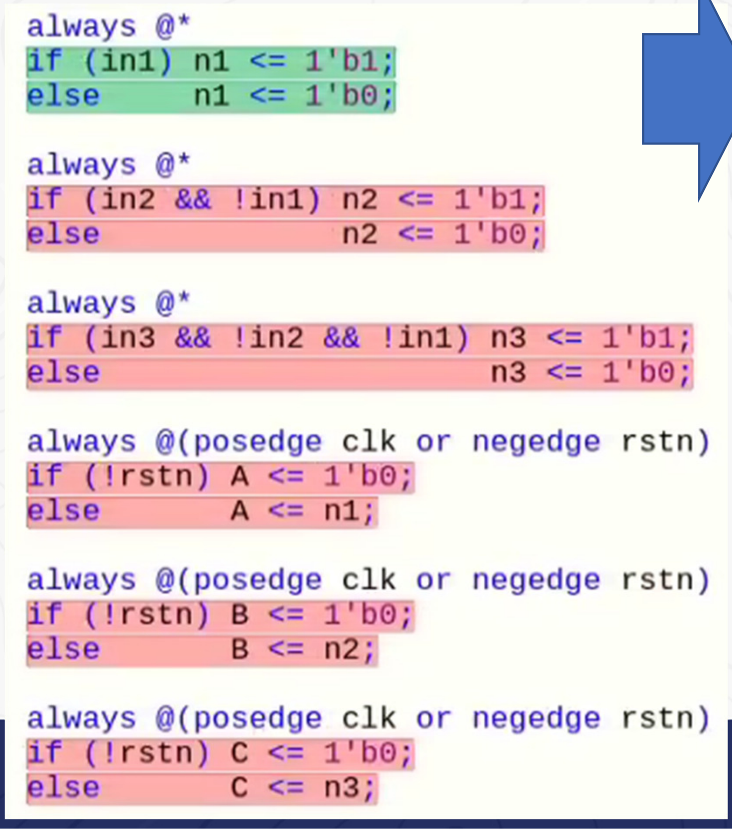
```
always @*
if (in1) n1 <= 1'b1;
else     n1 <= 1'b0;

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;
```

Example 3: Let’s Quickly Close This …

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

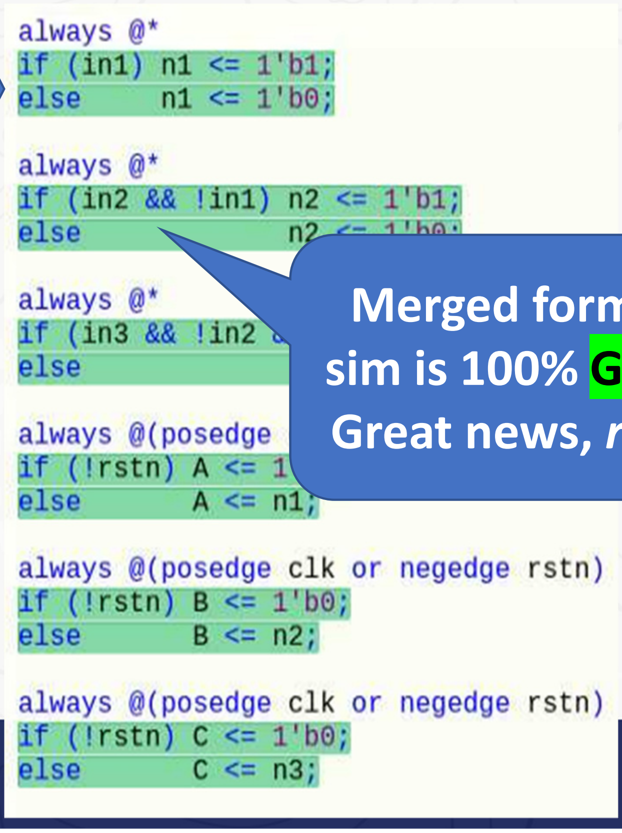
Formal coverage shows the first two lines are covered

Formal



```
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;
```

Merge Formal + Sim



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

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

always @*
if (in3 && !in2
else

always @(posedge
if (!rstn) A <= 1
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;
```

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:

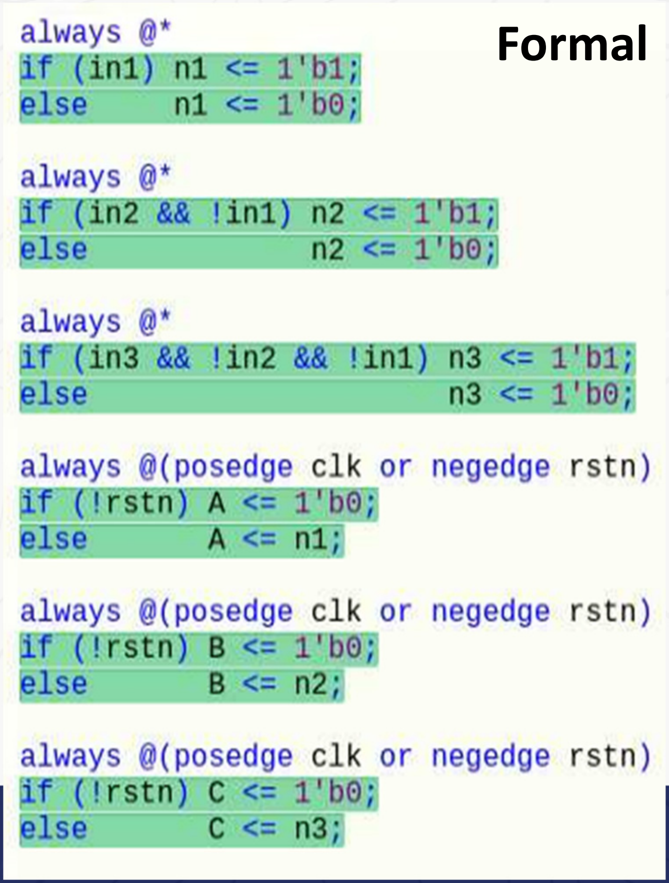
The design requires the 3 outputs to be mutex, thus a more useful property which checks this requirement is:

```
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



Formal

```
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;
```

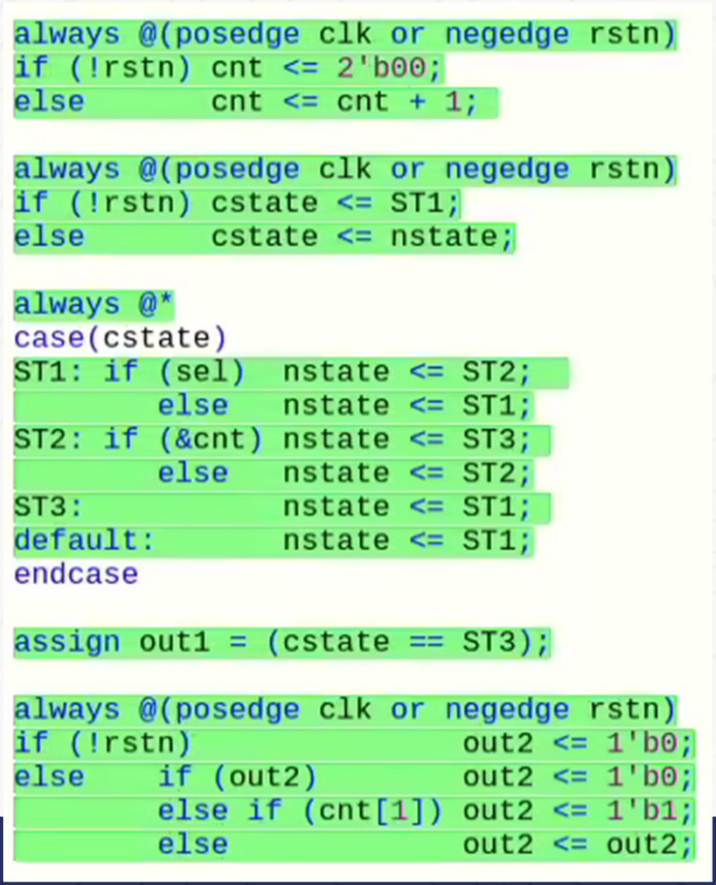
Example 4: 100% Coverage – But There is Still a Bug!

Specification:

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

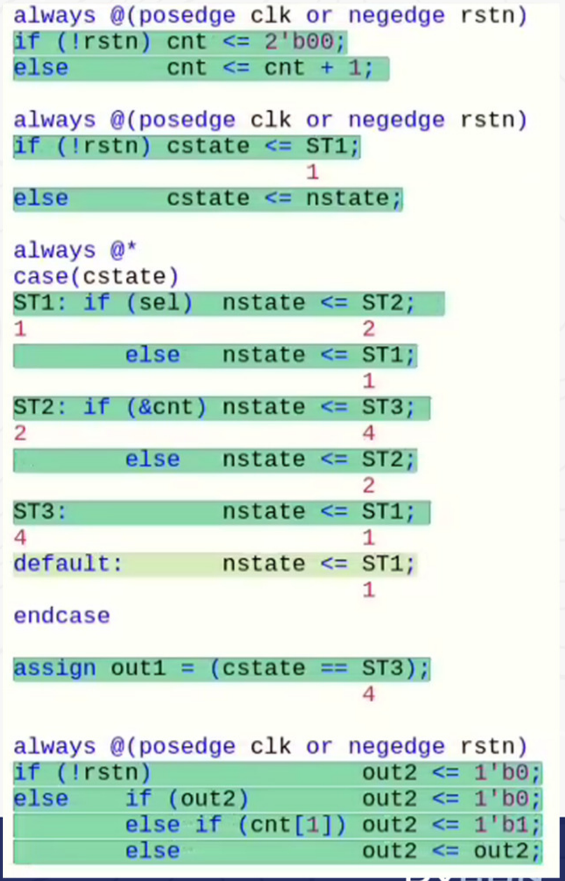
Both sim AND formal are 100% Green! Great news, *right?*

Simulation



```
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;
```

Formal



```
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;
                     1
else       cstate <= nstate;

always @*
case(cstate)
ST1: if (sel)  nstate <= ST2;
1                        2
     else      nstate <= ST1;
                         1
ST2: if (&cnt) nstate <= ST3;
2                        4
     else      nstate <= ST2;
                         2
ST3:           nstate <= ST1;
4                        1
default:       nstate <= ST1;
                         1
endcase

assign out1 = (cstate == ST3);
                         4

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”

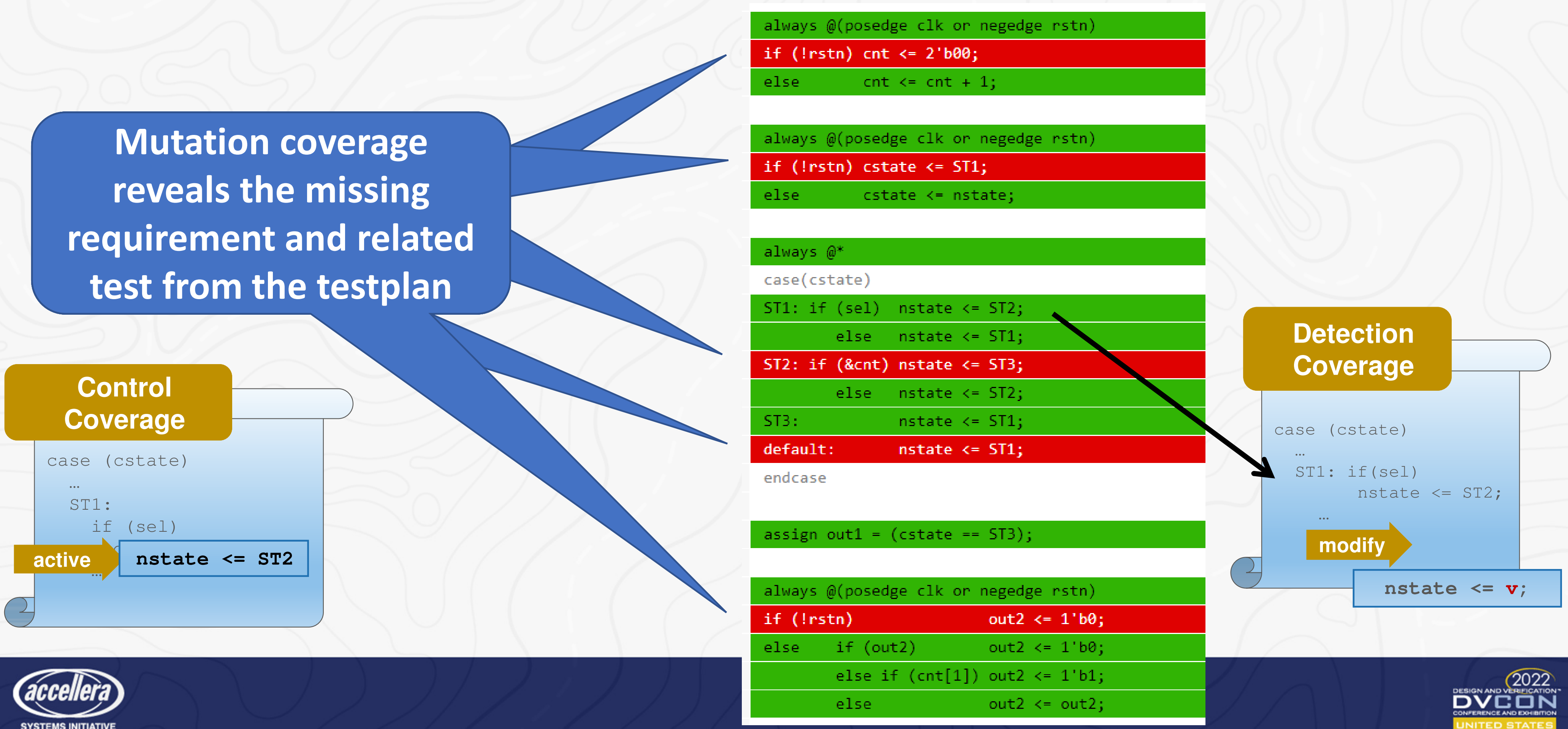
Solutions

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

```
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



```
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;
```

Mutation coverage reveals the missing requirement and related test from the testplan

Control Coverage

```
case (cstate)
  …
  ST1:
    if (sel)
      nstate <= ST2
  …
```

Detection Coverage

```
case (cstate)
  …
  ST1: if(sel)
          nstate <= ST2;
    …
nstate <= v;
```

Summary of Formal vs. Sim Coverage Differences

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

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?