How to Avoid the Pitfalls of Mixing Formal and Simulation Coverage

Mark Eslinger, Siemens, 46871 Bayside Pkwy, Fremont, CA 94538, mark.eslinger@siemens.com
Joseph V Hupcey III, Siemens, 46871 Bayside Pkwy, Fremont, CA 94538, joe.hupcey@siemens.com
Nicolae Tusinschi, Siemens, Nymphenburgerstr. 20A, München, 80335, nicolae.tusinschi@siemens.com

***Abstract*-Driven by the need to objectively measure the progress of their verification efforts, and the relative contributions of different verification techniques, customers have adopted "coverage" as a metric. However, what exactly is being measured is different depending on underlying verification technology in use. Consequently, simply merging coverage measurements from different sources -- in particular, blindly combining functional coverage from constrained-random simulations and formal analysis -- can fool the end-user into believing that they have made more progress, and/or they have observed behaviors of importance, when neither is the case. In this paper we will first review what these forms of "coverage" are telling the user, and how to merge them together in a manner that accurately reports status and expected behaviors.**

I. Introduction

The most effective functional verification environments employ multiple analysis technologies, where the strengths of each are combined to reinforce each other to help ensure that the Device Under Test (DUT) behaves as specified. However, this creates an inherent challenge of properly comparing and combining the results from each source to give a succinct, accurate picture of the verification effort's true status. The most common problem we see is when customers look to merge the results from formal analysis with the RTL code and functional coverage from their UVM testbench. There is (A) a lack of awareness of the information that "formal coverage" is providing, and (B) a proper way to interleave this information with simulation results so it is clear to the end-user what circuit pathways and behaviors have/have not been addressed by the combined verification flows. We will start with a brief re-cap of simulation-generated code and functional coverage.

II. Simulation-generated Code and Functional Coverage Defined

Code coverage is simply the percentage of RTL code measuring the number of statements in a body of code that have been executed through a test run, and which statements have not. While it is important that the testbench can exercise all of the RTL code – i.e. there is no "dead code", implying a bug in the DUT – the design can still be missing important functionality and/or the paths to key functions are in violation of the specification.

Functional coverage from RTL simulation is the metric of how much design functionality has been exercised – a/k/a "covered" by the testbench or verification environment – which is explicitly defined by the verification engineer in the form of a functional coverage model. In its basic form, it is user-defined mapping of each functional feature to be tested to a "cover point", and these coverage points have certain conditions (ranges, defined transitions or cross etc.) to fulfill before it is reported as 100% covered during simulation. All these conditions for a cover point are defined in form of 'bins'. A number of cover points can be captured under one 'covergroup', and a collection of number of cover groups is usually called a "functional coverage model".

During simulation, when certain conditions of a cover point are "hit", those bins (conditions) are getting covered and thus it gives a measurement of verification progress. After executing a number of testcases, a graphical report may be generated to analyze the functional coverage report and plans can be made to "cover up" the "holes" by creating new tests that will traverse the as-yet unexercised areas of the DUT code.[1]

Note that it is important to appreciate that the coverage score is also reflection of the "quality" of the testbench – i.e. whether the testbench is addressing all the DUT circuitry that you need to verify. Indeed, the coverage score and holes are a reflection of the quality of the test plan and its alignment with the original specification itself.

III. Formal-based Functional Coverage Defined

Formal analysis delivers the following types of coverage not seen with simulation:

- Reachability: Is there any combination of input signals that will bring the state of the circuit to a specified node of importance? If there are none, the point is "unreachable". Hence, this is yields similar information about the presence of dead code as does simulation code coverage described above.

- Observability: What are all possible circuit / state space paths from a selected node to signals specified in an assertion; and are these the paths expected and/or meet the design's specification.
- Structural Cone of Influence (COI): Working forwards or backwards from a selected point in the circuit, what is all the logic that could possibly affect this node. This is a rough form of coverage and generally not useful for final signoff analysis.
- Model-based mutation coverage: Measures how well checkers cover design by inserting mutations into the model, making the approach RTL non-intrusive, and checking if any checker/assertion detects it. Mutation coverage gives clear guidance on where more assertions are needed and represents a stronger metric than any other type of coverage since it measures error detection, not just exercise of design.

While these measurements are fundamentally different than simulation coverage, the overall application is the same at a high-level: measure progress and simultaneously evaluate the quality of the formal testbench (comprised of constraint and functional verification properties)

IV. Pitfalls of Merging of Simulation and Formal Coverage Objectives

The most common customer request is to "merge" simulation and formal-generated coverage metrics corresponding to each specified coverage point; where often the objective is to enable verification teams to select either formal or simulation to exclusively verify a sub-set of a design. For example, imagine a given IP contained a design element that was very well suited to formal analysis – an arbiter is one example – and the rest of the design could be easily verified by either technology. Naturally, customers want to enable formal to "take credit" for the arbiter verification, and seamlessly "merge" this with the simulation results on the rest of the circuit. This is where the trouble can start! The key points to be aware of are:

- First, regardless of the chosen technology, just because something is "covered" doesn't mean it is properly verified. Again, consider the relationship between code coverage vs. functional coverage – just because tests can traverse 100% of the code doesn't mean that the code is also functionally correct.
- 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 do reflect a "forward traversal", but just because the formal analysis might traverse some of the same states as a simulation, the logic "covered" is usually greater. Additionally, the formal analysis has free-floating input stimulus, and is valid for all time.[2]
- Other types of formal coverage report how logic and signaling "works backwards" from an output
- Simulation is run at the cluster/SOC level, while formal is typically run at the block level. Hence code coverage represents end-to-end testing vs. a more localized function. How the coverage data was generated can be lost once the results are logged in the coverage database.
- Mutation coverage enables users to measure the quality of either their simulation or formal testbenches vs. the testplan and specification.[3]
- Bottom-line: if you are careful about understanding the differences discussed above, you can roll up these metrics into a master, testplan level coverage metric that effectively unifies the results of the underlying verification technologies; giving actionable feedback on the progress of the verification project.

With all the above points in-mind, consider the following illustrative examples that employ the "simplest" form of coverage – RTL code coverage – to make clear the differences between simulation and formal analyses.

V. Example 1: Basic Illustration of Simulation vs. Formal Code Coverage

The following simple design will be used to illustrate the different code coverage results that simulation and formal analyses deliver when run on the same DUT. This design has a single clock and reset. There is a 2-bit input: 'sel'. There are three single bit outputs: 'A', 'B', and 'C'; and each output can be active for only one clock cycle. The 'sel' signal specifies which outputs to drive per the following truth table:

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

To simplify and focus the discussion of results, we will confine attention to output ‘B’. (Obviously, in a real verification flow all requirements and outputs would be tested.) There is a property to verify the requirement that output ‘B’ can only be active for 1 clock cycle. A simulation test is run to stimulate the ‘B’ output and formal is run on this property. Code coverage results for this circuit written in Verilog are shown below for simulation (left-hand-side of Figure 1) and formal (right-hand-side). (Note that VHDL designs would exhibit similar coverage metrics in this example, as well as the subsequent examples below.)

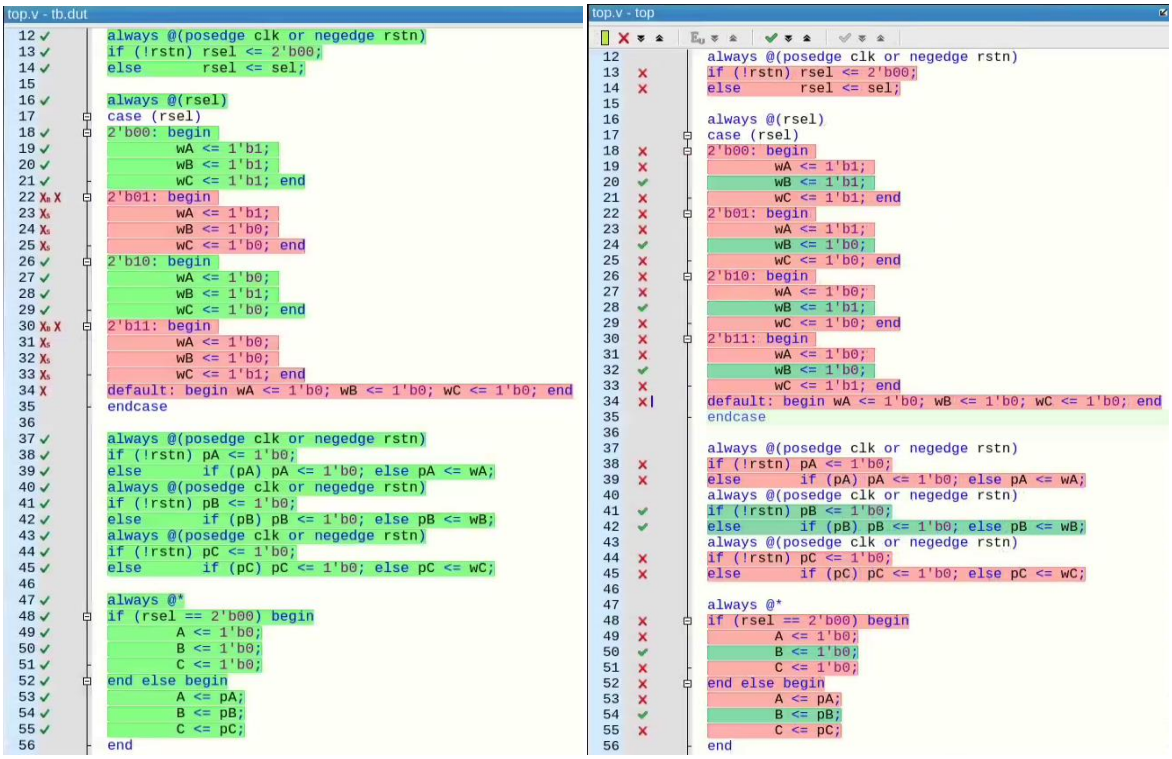


Figure 1: Basic logic example code coverage results from simulation (left) and formal (right).
Lines with green checkmarks and green shading are “covered”.
Lines with red ‘Xs’ and red shading are not covered.

Referring to the simulation results on the left of Figure 1 above, you can see that most of the design is covered, even though only one of the outputs was driven. The only thing not covered was related to the select input not being driven with “01” or “11”. This shows the generous nature of simulation code coverage in that logic unrelated to what is being tested can be covered too. For example, all statements of a block are covered. Why is this? In short, the simulation stimulus (a/k/a/ input vectors) are driven

into the inputs of the Verilog model of the by the simulation tool, with the analysis progressing from the inputs to the outputs of the design being verified.

Clearly, the formal code coverage results are quite different, reflecting the nature of the formal analysis that effectively works backwards from the given DUT output being verified by writing a System Verilog Assertion (SVA) property that specifies the behavior of the node(s) designated therein. Specifically, the way that formal proof algorithms work is they start by looking at a small amount of logic that is in the immediate fan-in cone of the output signal specified by the SVA property. The property may be proven right at that point, and the analysis would be complete. But if the property isn't proven in this first step, more logic is "pulled-in" just ahead of this area. If this is enough logic to obtain a proof, you are done. If not, the process continues until primary inputs and, potentially, "assumptions" – i.e. constraints – are used in the analysis. Again, you can think of the formal code coverage analysis as proceeding from the node of interest – output 'B' in this example – towards the inputs of the design. Consequently, the only lines of code declared "covered" by the formal analysis are those that were needed to obtain the proof of the property; in this case those that directly control output 'B'.

Given these very different approaches – and the subsequent very different results – it is critical to keep the simulation and formal results "parallel and separate" from each other when exporting this data to a common verification results database so users can accurately assess progress and make use of downstream verification management techniques for tracking and testplan completion.

VI. Example 2: Simulation vs. Formal Code Coverage of a Finite State Machine (FSM)

The following simple FSM will be used for another illustration of the different code coverage results simulation and formal analyses deliver when run on the same DUT.

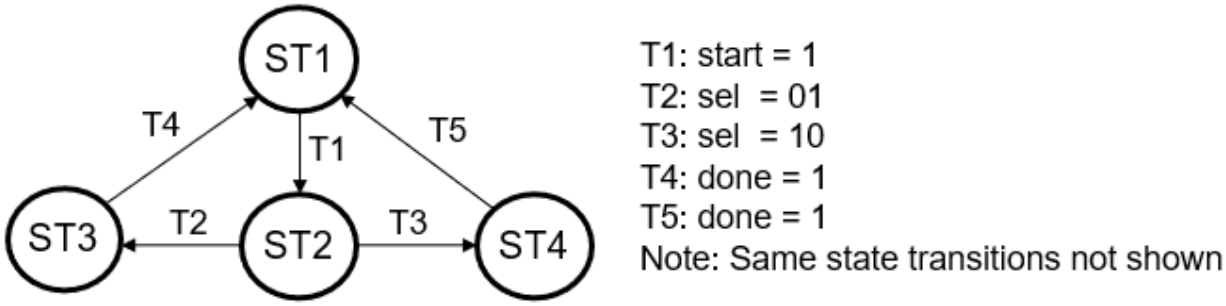


Figure 2: State diagram of the example FSM

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. The coverage for both is shown below in Figure 3 below:

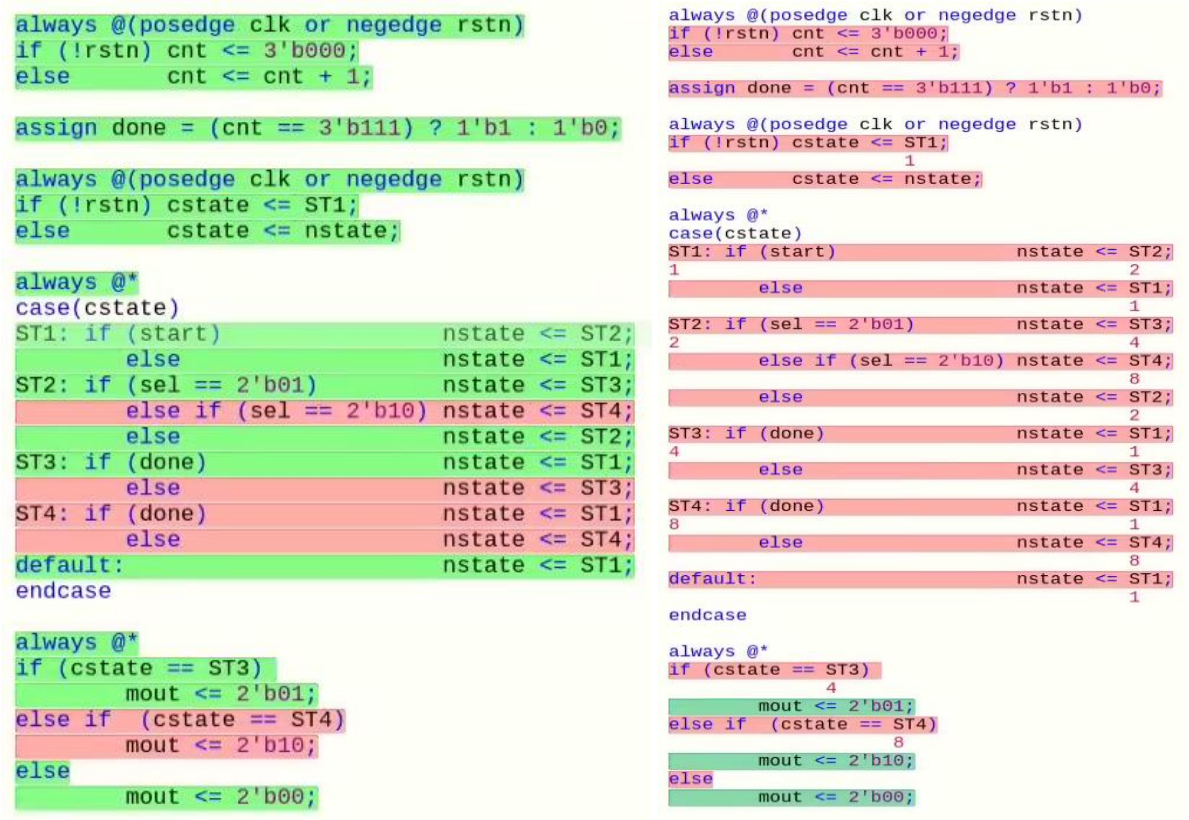


Figure 3: FSM code coverage results from simulation (left) and formal (right).
Lines with green checkmarks and green shading are "covered".
Lines with red 'Xs' and red shading are not covered.

Referring to the simulation results, as with the prior example you can see that most of the design is covered. Perhaps the uncovered lines of code suggest that more vectors or "unreachability" analysis is needed; but all the "green" in this result gives the impression this FSM design is in good shape.

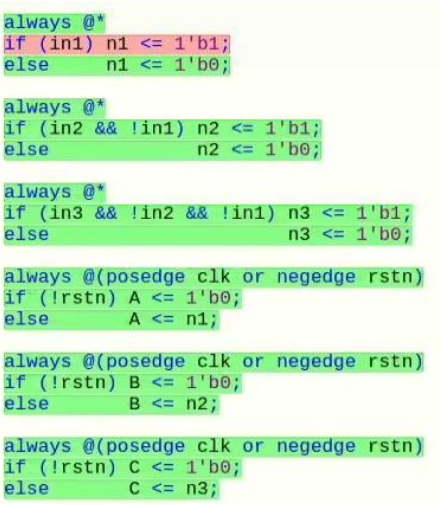
Clearly the formal analysis results paint a different picture. In this case, the formal analysis was focused on proving that the 'mout' output is mutex, per spec. Hence, formal only used the logic that was needed for the proof -- in this case NONE of the FSM state bits were needed – just some of the signal assignments! While this mutex requirement is needed, its correctness doesn't need much of the design. As such, the FSM design itself obviously needs a reexamination and further testing. From a formal verification perspective, more properties are needed, or a property which tests higher order requirements that makes fuller use of the FSM should be applied.

It should also be noted that the formal coverage shown here is based on the logic needed to prove the property, commonly referred to as a "proof core". (A similar result would be achieved using mutation coverage analysis outlined in Section VIII below). A crude form of formal code coverage is based on the structural COI of the property, which would show coverage from the property all the way back to the inputs, closer to what is shown above for the simulation result. Clearly, this is not enough for signoff coverage, and hence only useful very early in the verification process when adding properties (and in many cases, this step can be ignored completely).

VII. Example 3: Closing Code Coverage Holes

In this next example, imagine the user is looking to close coverage by using formal analysis to fill a coverage "hole" found in simulation. For purposes of illustration, we will show an initial failed attempt at this – then the corrected version.

First, consider the following simple design coded in Verilog, where outputs are again mutex-based on an encoding of the 3 inputs. The simulation code coverage results for the following Verilog design are almost perfect – only one hole remains!



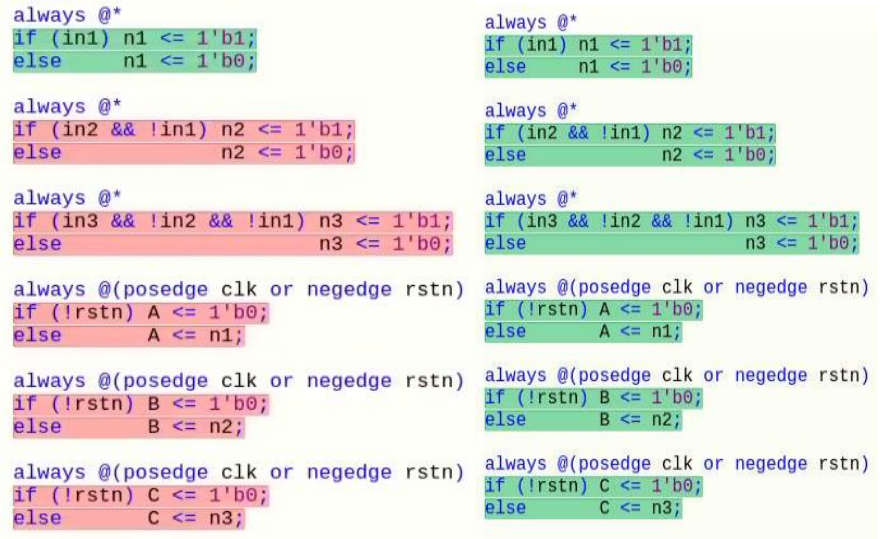
```
always @*
if (in1) n1 <= 1'b1;
else     n1 <= 1'b0;

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;
```

Figure 4: Code coverage results from simulation showing only one remaining "hole"
Lines with green shading are "covered". Lines with red shading are not covered.

Rather than spend time manipulating the parameters of the simulation test, the verification engineer elects to run a formal analysis to try to cover this hole. Hence, the following property is written:

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

Which when run in formal yields the following results (shown on the left of Figure 5):



```
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;
```

Figure 5: Code coverage results with formal due to trivial property targeting just that code (shown on the left), and code coverage from a property that's checking the design's requirements (shown on the right)

As per the example on the left of Figure 5, in the rush to plug the simulation hole in the 'n1' signal path, the verification engineer tried to use a trivial property in formal which would plug the hole in simulation. While combining this result with simulation would give 100% coverage of this logic, this property is actually useless. It tests nothing – it is not tied to a testplan, or to the verification of any design requirements. Closing coverage this way is bogus, to use a colloquialism.

A better approach is to write and apply a property tied to the testplan and verification of design requirements. In this case the design requires the 3 outputs to be mutex, thus a more useful property which checks the requirements would be written as follows:

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

This property when proven in formal yields the coverage shown on the right of Figure 5 above. Ideally, the simulation hole would be filled by running a test which leveraged that logic; or if that logic was deemed unreachable, further work would not be needed. The hole points to a weakness in the testplan or testbench of the engine being used. If it is planned as part of the testplan and requirements needing verification that part of this design be verified with simulation and part with formal, then the above coverage from each engine could be safely merged to give an accurate, complete coverage picture for the overall verification status analysis.

VIII. Example 4: Full Code Coverage Achieved – But There is Still A Bug

The following example FSM and supporting logic shows a trap that many fall into – assuming that 100% code coverage means that they are done with verification. Instead, the sole value of code coverage is in pointing out areas of the DUT which haven't been verified, regardless of which engine is being used. In the below example, the two outputs are again mutex and a property has been written which passes in simulation and is proven in formal. In this verification scenario, the below results from simulation and formal show full coverage on the mutex check for the 'out1', 'out2' signals:

```
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;
```

```
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;
```

Figure 6: 100% code coverage of FSM from simulation (left) and formal (right)!
But could there still be a bug in the DUT?

Despite these promising results, there are two important points to consider. One is the importance of a complete testplan. Coming back the key requirement of "*the FSM to only be in state 2 for no more than 3 cycles*" – in this case all the code is traversed, BUT the functional behavior is incorrect! If this requirement was not part of the testplan, then the above coverage reports would fool the user into believing that their verification was complete while the bug slipped through! However, if the requirement made it into the testplan – which was in-turn captured in the following assertion – then it would be properly tested:

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

In this scenario, simulation could have completely missed this functional bug if there was no test to cover this case; and in fact, for the vectors run to give the above coverage that was the case as the assertion passed. However, the exhaustive formal analysis will have spotted the functional error and reported it as a counter-example to the user.

The larger point of this example is the cautionary note that coverage is mainly for seeing what isn't covered vs. the focus on achieving 100% coverage at all costs – i.e. even if you have 100% code coverage, you may still have bugs that other analyses and coverage metrics will reveal.

Now let's go a little further and see how model-based mutation coverage can help in revealing such gaps in a verification environment. As noted above, mutation coverage is generated by the verification tool systematically injecting mutations (faults) into a formal model of the DUT, and then checking whether any assertion would detect a bug at the corresponding fault point. (The mutations are done electronically, in the mutation tool's memory – the original source code is untouched). In effect, mutation coverage measures the ability of your testbench to find bugs – i.e. if your verification environment can't find a bug that you have deliberately created, it is unlikely to find bugs you have not imagined.

As shown in Figure 7 below, running the coverage analysis with mutation indicates precisely where there is a gap covering the requirement, and thus the need to add it to the testplan in order to make it complete.

```
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;
```

Figure 7: Model-based mutation coverage (revealing the missing requirement and related test from the testplan)

IX. Summary: Using Both Formal and Simulation Code Coverage

Verification teams use coverage data from all three types of engines: formal, simulation, and emulation. All contribute to testplan signoff as well as coverage closure in all aspects. Focusing on code coverage, the following table outlines some of the differences between formal and simulation code coverage:

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
It is possible to only have some 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, need to consider when calculating coverage, may lead to coverage holes	No abstractions used
Assumptions/constraints used in a proof need qualification, potentially in simulation	Not a consideration, though input constraints can impact reachability
Reachability analysis used for exclusions, done as part of formal coverage calculation	Reachability analysis used for exclusions, done with formal (CoverCheck)

The primary concern here is that mixing code coverage from multiple verification engines can mask holes in the testbench of the other engine. There are other subtle differences which make merging code coverage from different engines difficult. The following recommendations may help.

X. Recommendations

The formal and simulation coverage flows are flexible, which allows you to make use of them in whatever capacity best fits your needs. Below are a few things to consider when using code coverage from any of the verification engines:

- Ideally, close code coverage for each verification engine separately
  - Focus on improving testbench completeness and robustness in each domain
  - Code coverage data should be kept separate in the main coverage database for this purpose
  - Additionally, in the formal domain run both proof core and mutation coverage to check testbench completeness
- Test planning is important for determining which verification engine will verify which parts of the design
  - Formal may totally verify certain modules—when mixing formal and simulation code coverage, try to keep it to instance boundaries (e.g. something that can be enabled/disabled in one domain vs. the other)
  - When mixing code coverage, plan it as part of the testplan, late in the game
- Know where your code coverage comes from: formal versus. 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
  - Only merge coverage data for final reports near the end of the project
- Avoid writing properties or adding vectors to only trivially hit some specific part of the design to get to the 100% coverage mark
  - This doesn't typically improve verification, only trivially improves coverage
  - This is why the testplan is important. Coverage holes point to an incomplete testplan and ultimately an incomplete/weak testbench
  - When adding properties or vectors to close code coverage holes, verification of features and requirements of the design is your guide and will give you the highest quality of verification

The verification team and project leaders are ultimately responsible for their results. When required to mix coverage for a given block from multiple verification engines, it is recommended to have a peer review on how the code coverage was generated to make sure valid tests tied to a testplan are used. Code coverage from various verification engines will be used to make decisions regarding meeting project milestones. The above recommendations may help in ensuring a successful tape out of the chip.

XI. CONCLUSION

With proper understanding of the nature of the coverage metrics being recorded, output from all sources can be combined to give a holistic picture of the progress and quality of the verification effort. In this paper we focus on the most common element of this need – properly merging the results from formal analysis with the coverage from a constrained-random simulation testbench such that individual contributors and team leaders will understand exactly what the data is telling them.

REFERENCES

[1] M. Singhal, "What is Functional Coverage?" https://www.learnuvmverification.com/index.php/2015/06/04/what-is-functional-coverage/, June 4, 2015

[2] J. Hupcey III, "How Can You Say That Formal Verification Is Exhaustive?" https://blogs.sw.siemens.com/verificationhorizons/2021/09/16/how-can-you-say-that-formal-verification-is-exhaustive/, September 16, 2021

[3] N. Tusinschi, V. Palfy, "Using Mutation Coverage for Advanced Bug Hunting", DVCon San Jose 2018 -- https://youtu.be/LUG_VJLXuiU, March 1, 2018