Fast Track Formal Verification Signoff

Mandar Munishwar
Qualcomm Inc.
1700 Technology Drive, San Jose, CA 95110, U.S. A.
Xiaolin Chen, Arunava Saha
Synopsys Inc.
690 East Middlefield Road, Mountain View, CA 94043, U.S. A.
Sandeep Jana
Synopsys India Pvt Ltd.
Tower A, RMZ Infinity, Bengaluru, 560016, Karnataka, India

***Abstract*- Regardless what technology is used, whether simulation or formal, when tasked with functional verification project, verification engineers will need qualitative and quantitative measurement of the verification progress. Such metrics are to provide measurement regarding the robustness, completeness, and effectiveness of the verification environment. Fault qualification is an orthogonal mechanism to FormalCore coverage. In this paper, we will discuss how fault qualification provides an additional metric to further improve the formal verification environment to prevent bugs from escaping. It complements other coverage metrics and helps to fast track the formal signoff on the design verification.**

I. Introduction

For any verification engineer, the most dreaded scenario is bugs in the design escaping from the testbench and getting into the silicon. What more can we do to prevent this from happening?

Regardless of what technology is used, whether simulation or formal, when tasked with functional verification project, verification engineers will need qualitative and quantitative measurement of the verification progress. Such metrics are to provide measurement regarding the robustness, completeness, and effectiveness of the verification environment. We can start by looking at coverage models for functional verification in the following section.

.

II. Coverage Models for functional verification

A. Coverage models for simulation verification flow

For over two decades, the coverage metrics used in simulation functional verification flow are:

- Code coverage
  - Measurement used to describe the degree to which the RTL has been exercised when a test suite runs.
- Functional coverage
  - Measurement that serves the execution of a test plan. It is used to track whether important events, sets of values or sequences of values that correspond to design or interface requirements, features, or boundary conditions have been exercised.
- Assertion coverage
  - Measurement used to describe the degree to which the properties written have been exercised.

These coverage models are used to measure the verification progress as well as to evaluate the quality of the verification test environment. When coverage percentage goals are met, design verification project can be signed off as completed. In the simulation verification flow, these coverage models are clearly defined and well understood.

B. Coverage models for formal verification flow

Tasked with using formal technology to verify a design block, formal verification engineers are faced with these questions:

1. What part of the logic is not in the fan-in cone of the any assertions in the formal testbench?
2. What part of logic is uncoverable with the constraints in the formal testbench?
3. What part of logic is required to obtain full proofs or bounded proofs of the assertions?

For quite some time, there was not clear measurable progress indicator in the formal verification flow. It had been difficult to get visibility into the verification progress to monitor and predict the progress. Most engineers relied on creating formal verification environment by writing constraints, assertions and cover properties based on the design

specification to see whether specific scenarios are covered by the testbench environment. It was not always well understood how exactly the constraints were impacting the verification environment since there was not a systematic way to measure. Also, if assertions were proven, there was not an easy way to provide visualization to what part of the logic was used to obtain those proofs, and what part of the logic fell outside of the verification environment.

In recent years, drawing parallel to what has been done in the simulation world, coverage models have been adopted in the formal verification flow as well. The following coverage models start to emerge to address the three questions mentioned above:

- Property density coverage
  - The property density is used to assess how the assert or cover properties test the design or a certain scope. For example, the list of uncovered code coverage targets indicates that the assert or cover properties cannot test that part of the design. **Figure 1** shows property density coverage which measures the percentage of logic in the fan-in cone of all the assertion properties.
- Over-constraint coverage
  - Over-constraint coverage analysis is used to determine if there are certain regions of the RTL which are unreachable because of some constraints in the testbench. **Figure 2** shows over-constraint coverage which measures code coverage of the formal testbench.
- FormalCore coverage
  - The formal core of any proven (full proof or bounded) property consists of a list of code coverage targets in the design, a set of constraints as well as a set of inputs that are involved in proving the property. **Figure 3** shows FormalCore coverage which measures percentage of logic used to produce full proofs and bounded proofs.

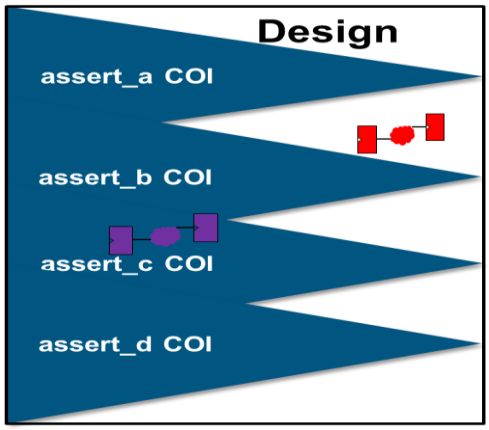

Figure 1. Property density coverage

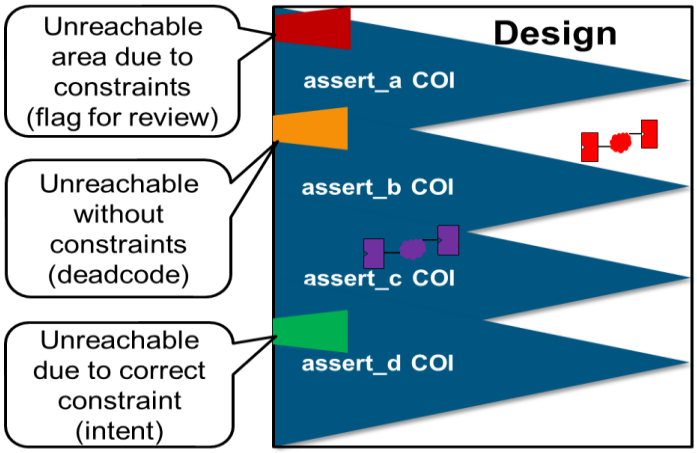

Figure 2. Over constraint coverage

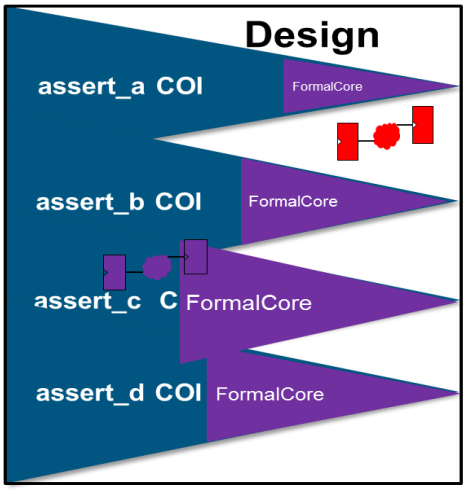

Figure 3. FormalCore coverage

With these coverage metrics, it is now possible to measure the progress and quality of the formal testbench. Just like in the simulation verification flow, when the percentage of the coverage metrics reached pre-determined goal, formal verification project signoff can be completed.

III. FAULT QUALIFICATION

C. Signoff with FormalCore coverage

Since the availability of coverage models for formal verification flow, formal verification projects started to follow these coverage models leading to signoff. In one of the control IP formal verification testbench, we did the analysis of the constraint coverage and concluded there was not over constraining the formal verification environment, and all the properties were either proven or with bounded proof results. The FormalCore coverage also showed that 100% of the RTL logic was covered in the proof bound of all the properties. In most of the cases in the past, this would have been good enough to signoff the verification project.

However, could we assert with absolute confidence we had averted the dreaded scenario of bugs escaping? What more could we have done to ensure catching hidden bugs before signoff? This presented a new challenge. This challenge lead us to start exploring other opportunities to stress our formal verification environment even further.

D. Fault coverage for functional verification

As it is with new challenges, innovative technology often rises to overcome the challenges. Fault-injection tool for hardware emerges as the technology that can provide a new type of coverage metric: fault coverage. Hardware design fault qualification works this way: Behavioral faults are systematically injected into the design code; existing test suites are run on the fault-injected design to determine whether any of the injected faults can escape from the testbench undetected. **Figure 4** shows how fault injection works in the verification environment.

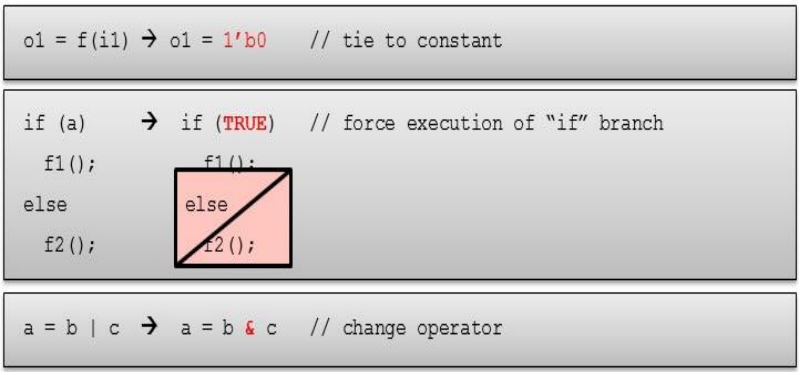

Figure 4. Behavioral fault examples

Fault coverage provides detailed information on the ability of the verification environment to activate, propagate and detect "systematic faults" that represent potential bugs in the design, exposing significant weaknesses that have gone unnoticed by other means. The system provides data to identify vulnerabilities in the stimuli, observability, as well as holes in the verification plan. With the uncertainty removed, the verification efforts will be more reliable and efficient. **Figure 5** shows how the fault qualification works in a nutshell.

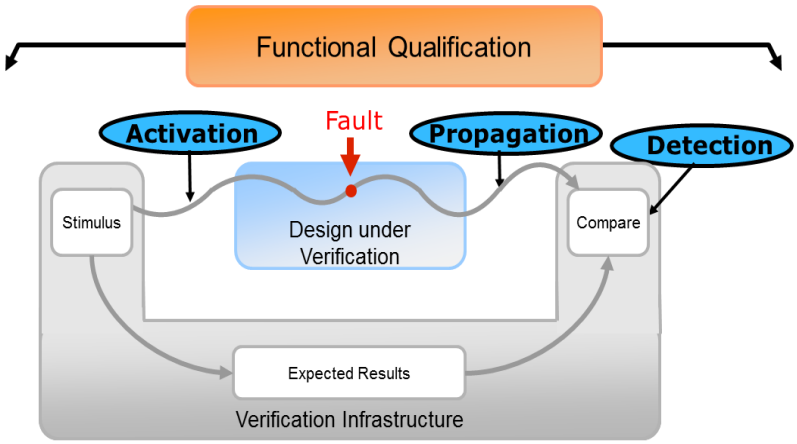

Figure 5. Fault qualification in a nutshell

How does fault coverage work in formal verification? In the formal flow, formal property verification can be used to run on fault injected design to determine if the formal verification environment can detect the injected faults. The flow is to run all proven (sometimes bounded proven as well) formal properties on the original design with one fault activated at a time to see if this fault can be detected by one of the properties getting falsified. If not, this fault is non-detected. The non-detected faults can help identify design logic where faults are not caught by any of the existing checkers therefore quantifying the quality of the formal property verification environment. If all the checkers pass in the presence of a fault, it indicates weakness in the formal verification environment, either missing checker, incomplete specification or over-constraining. This is shown in **Figure 6.**

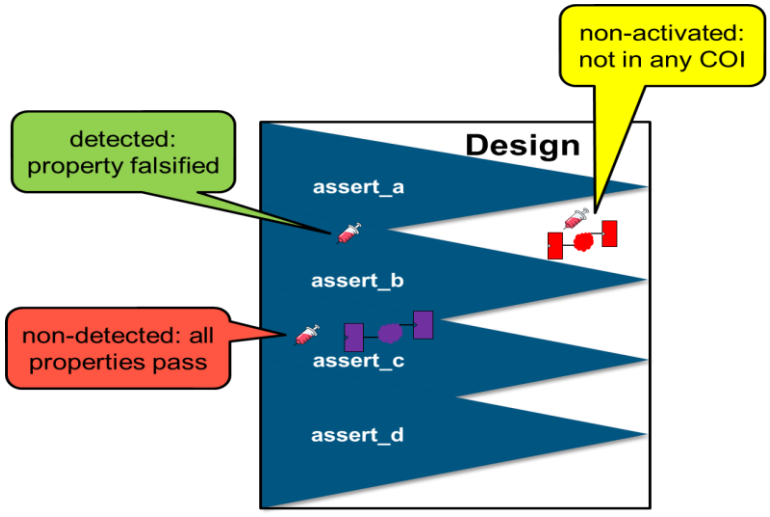

Figure 6. Formal fault qualification

IV. FAULT QUALIFICATION ON A CONTROLLER IP

For this controller IP, we applied the fault qualification technology. We correlated and qualified faults in the FormalCore as well as outside of FormalCore. For this design, we find 9 major non-detected faults, as shown in

Figure 7. We followed the guidance from these non-detected faults for further examination of our verification environment.

Class Name	Faults In Design	Faults In Report	Non-Activated	Non-Propagated	Detected	Non-Detected	Disabled By Certitude	Disabled By User	Dropped	Not Yet Qualified
TopOutputsConnectivity	126	121	33	0	78	0	5	0	0	5
ResetConditionTrue	13	13	0	0	9	1	0	0	0	3
InternalConnectivity	111	111	6	0	53	1	33	0	7	11
SynchronousControlFlow	344	340	26	0	135	2	41	0	105	31
SynchronousDeadAssign	67	67	8	0	38	0	5	0	15	1
ComboLogicControlFlow	702	702	70	0	340	1	8	0	245	38
SynchronousLogic	633	633	54	0	83	0	46	0	403	47
ComboLogic	1292	1292	79	0	244	4	67	0	848	50
OtherFaults	0	0	0	0	0	0	0	0	0	0
All Fault Classes (9)	3288	3279	276	0	980	9	205	0	1623	186

Figure 7 Fault qualification report of the controller IP

Most of the time, there is a correlation between the FormalCore results and the fault detection: The non-detected faults are outside of the formal core. However, in certain cases, we observed non-detected faults even within the FormalCore. It helped to identify the area where the formal testbench needed improvement.

In one of the blocks, there was RTL coding of a FIFO, shown in **Figure 8.** The verification plan for the functionality of this part of the code was to check:

- FIFO asserts the full or empty flag are set when the count reaches ‘hf or ‘h0.
- The count should not change when there’s no push or pop.

```
(input clk,
           input rst_x,
           input push,
           input pop,
           output logic full,
           output logic empty);

   logic [3:0] cnt;

   always@(posedge clk or negedge rst_x)
     if(!rst_x)
       cnt <= '0;
     else begin
        if(push && ~pop && cnt!=4'hf)
          cnt <= cnt + 1'b1;
        else if (!push && pop && cnt!= 4'h0)
          cnt <= cnt - 1'b1;
     end

   assign full = (cnt == 4'hf);
   assign empty = (cnt == 4'h0);
```

Figure 8. FIFO RTL coding

Following the plan, three properties were written to verify the FIFO, shown in **Figure 9.**

```
check_full : assert property ( @(posedge clk) cnt == 4'hf |-> full);
check_empty: assert property ( @(posedge clk) cnt == 4'h0 |-> empty);
check_no_change: assert property ( @(posedge clk) ~push&~pop |=> $stable(cnt));
```

Figure 9. FIFO assertions

All 3 properties were proven. We then ran FormalCore coverage on the FIFO design, and the results showed 100% of the logic was covered. This gave us confidence about the design. However, did this mean the verification is done? Could there still be bugs lurking? How could we be sure?

We turned to fault qualification for some answers. Faults were injected in the design. Some of the faults are shown in red in **Figure 10**. Since these faults alter the functionality of the design, we were expecting the 3 assertions we had written to catch these faults.

```
(input clk,
           input rst_x,
           input push,
           input pop,
           output logic full,
           output logic empty);

   logic [3:0] cnt;

   always@(posedge clk or negedge rst_x)
     if(!rst_x)
       cnt <= '0;
     else begin
        if(0/*push && ~pop && cnt!=4'hf*/)      //Condition False
          cnt <= cnt + 1'b0 /*1'b1*/;           //BitFlip
        else if (!push && pop && cnt!= 4'h0)
          cnt <= cnt +/*-*/ 1'b1;               //Operator
     end

   assign full  = (cnt == 4'hf);
   assign empty = (cnt == 4'h0);
```

Figure 10. Fault injection in FIFO design

However, all 3 assertions were still proven with the presence of some of the faults in the RTL. And the results of the fault qualification showed 15 faults not detected (**Figure 11.**) The examples shown in **Figure 10** are part of the non-detected faults.

Non-Activated	Detected	Non-Detected	Disabled By User	Not Yet Qualified
0	0	0	0	0
0	0	1	0	0
0	4	3	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	4	10	0	0
0	10	0	0	0
0	0	1	0	0

Figure 11. Fault qualification results after 100% FormalCore coverage

The results indicated weakness in the verification environment. These faults should not be allowed to escape! Upon examining the non-detected faults and how they could have slipped through the current verification environment, we determined that the checkers for the counter functionality were missing! This prompted us to create 2 additional assertions (shown in **Figure 12.**)

```
check_push: assert property ( @(posedge clk) push&~pop && (cnt!=4'hf) |=> (cnt == $past(cnt) +1));
check_pop : assert property ( @(posedge clk) ~push&pop && (cnt!=4'h0) |=> (cnt == $past(cnt) -1));
```

Figure 12. Additional checkers to catch non-detected faults

The fault qualification was run again with added checkers. We were delighted to discover that this time all the injected faults in this part of the code were detected! i.e., one of the assertions failed in the presence of any fault (shown in **Figure 13.**)

Non-Activated	Detected	Non-Detected	Disabled By User	Not Yet Qualified
0	0	0	0	0
0	1	0	0	0
0	7	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	14	0	0	0
0	10	0	0	0
0	1	0	0	0

Figure 13. Fault qualification results after formal testbench enhancement

For the controller IP, when we decided to run fault qualification, the FormalCore coverage reached 100% with 208 checker properties and 215 cover properties in the verification environment, and no more bugs were found. However, the fault qualification still showed 9 major non-detected faults!

The non-detected faults provided additional information on where there was still weakness. Based on this information, we created additional properties to eliminate the non-detected faults. After the fault qualification process, we had a total of 262 checkers and 290 covers (**Table I**). We found more additional RTL bugs!

Table I
BEFORE AND AFTER FAULT QUALIFICATION

Controller IP	Before	After	Missing Properties
Checker properties	208	262	54
Cover properties	215	290	75
New RTL bugs found	0	5	

In our experience, the fault qualification process did not take major effort compared to building the whole formal testbench. However, compared to other coverage closure steps we need to go through, e.g. over-constraint coverage, bounded proof coverage, FormalCore coverage, the setup for fault analysis was slightly more involved. Also, the debug process to remove the non-detected faults required much more thorough analysis of the fault scenario and design behavior to be able to come up with the right assertions. But finding that last corner-case bugs are very important for us. In the previous generations of the IPs without fault qualification, we would sign off on the IP verification with the 5 bugs still in the design!

One of items on our wish list for further enhancing the fault qualification on formal verification was to improve the long run time and debug environment. It was somewhat cumbersome to go back and forth between two separate environments of fault qualification and formal property verification. Despite this, fault qualification gave us added confidence in signing off the verification with formal. We were convinced of the value of fault analysis provided, it now has become part of the formal signoff flow.

V. SUMMARY

Fault qualification is an orthogonal mechanism to FormalCore. It provides an additional metric to further improve the formal verification environment to prevent bugs from escaping. Our experience showed that it complements other coverage metrics. Use it in the last stage of formal verification signoff process, it can provide guidance to the weakness in the formal verification environment, and help shed light on cases of vague or incomplete specification, as well as holes in the verification environment. By refining the verification environment, more bugs may be caught. This brings closure to fast track the formal signoff on the design verification.

ACKNOWLEDGMENT

The authors wish to thank Naveed Zaman of Qualcomm, Pratik Mahajan and Sean Safarpour of Synopsys for their believing in formal verification and their support in deploying formal technology in the current functional verification flow.

REFERENCES

[1] M. Munishwar, V. Singhal, S. Safarpour and P. Mahajan, "Formal Verification Methodology: Maximizing productivity and achieving Formal closure with confidence," Section "*Coverage* Design and Verification Conference and Exhibition, San Jose, California, 2017, pp. 58-87

[2] X. Feng, and X. Chen, "Coverage models for formal verification*,"* Design and Verification Conference and Exhibition, San Jose, CA, 2017 p.1-9.

[2] B. Wang, and X. Chen, "Coverage driven signoff with formal verification on power management IPs*,"* Design and Verification Conference and Exhibition, San Jose, CA, 2016 p.7-8.