"Bounded Proof" sign-off with formal coverage

Abhishek Anand, Intel Technology India Pvt Ltd, India**,** **abhishek1.anand@intel.com**
Chinyu Chen, Intel Technology**,** USA**,** **chinyu.chen@intel.com**
Bathri Narayanan Subramanian, Mentor, A Siemens Business**,** **bathri_subramanian@mentor.com**
Joe Hupcey, Mentor, A Siemens Business**,** **joe_hupcey@mentor.com**

***Abstract* - Due to the ever-growing demand for faster and exhaustive verification, many design companies have started adopting Formal Verification [1]. With Formal, we start with the complete input state space and then work our way down into a focused, carefully chosen subset of the input state space. This helps us make sure that we do not miss any important input scenarios. One of the challenges of employing formal verification is that increase in design size can lead to an exponential increase in design and verification complexity. As verification complexity increases we are more likely to encounter technology limitations leading to inconclusive results. However, even an inconclusive result provides value since it proves true within a constrained time space; this is called a "bounded proof". In the Barefoot Switch division of Intel, while verifying a packet processing design we have encountered such bounded proofs. In this paper, we talk about our methodical approach to signoff functionalities where only bounded proofs could be achieved due to design complexity. We showcase how we defined latency requirements, interesting verification scenarios, and correlation of the formal coverage result during signoff for better confidence.**

Keywords: Formal verification signoff, Bounded proof, Formal coverage

I. INTRODUCTION

Our team has been using formal verification for various blocks given it delivers exhaustive analysis that helps in finding corner-case bugs during our verification. As the complexity for design under test increases, assertions written for some of the design functionality tend to give inconclusive results, instead of giving a pass or a fail. In formal verification terminology, these are called "bounded proofs". Although the given assertion or property is not completely proven, a bounded proof is still valuable. It conveys that there was no error in DUT behavior up until the time where the analysis was halted. Before adopting the discussed approach, we relied only on manual analysis of assertion bounds to make sure that we were covering all the interesting scenarios. This manual analysis was very involved and daunting, especially for bigger design with many corner-case scenarios. Furthermore, this was also prone to errors, with no way to cross-verify it. Hence, without the right metrics to measure our progress along with a good methodology, we lacked confidence in signing off those functionalities.

II. OUR FORMAL VERIFICATION METHODOLOGY

Our formal verification methodology involves the following steps.

1. Identify a design suitable for formal verification
2. Gain design understanding by going over the micro-architecture and having discussions with the designers
3. Implement cover properties to explore design latency for outputs and micro-architectures such as FIFOs and arbiters
4. Create a verification plan which would include a design block diagram as following:

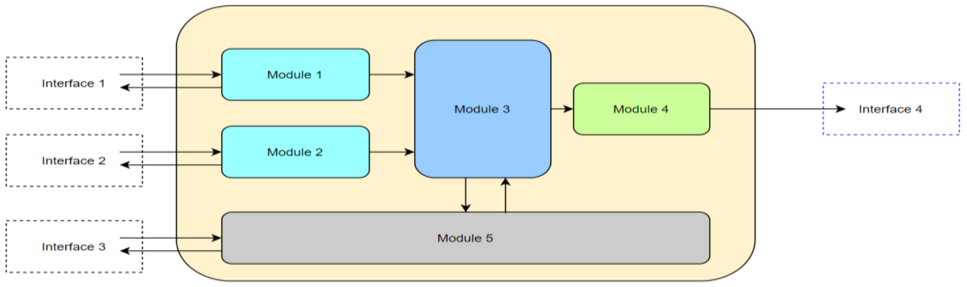


Figure: 1

And further include subsequent details like:

- Verification boundaries based on anticipated complexity and clean interfaces
  - For example in the above figure 1, for a complex end-to-end assertion from interface 1 to interface 4, we may decide to break up a "big" assertion into several "smaller" assertions that effectively daisy chain together – e.g. one assertion would "end" at the output of Module 1, the next assertion in the group would then "start" at the input of Module 3, and so forth. (This both makes it easier for the formal engines to process, and simplifies debug.)
- Assertion groups to be implemented
  - Identify types of assertions to be implemented for each of the interfaces
- Abstraction models to reduce complexity
  - For example in the above figure 1, we may decide to verify module 5 separately and replace it with an abstraction model during verification of the complete block

After reviewing the verification plan, we would move on to execution:

5. Implement properties including both assertions and assumptions; add abstraction models
6. Use assume-guarantee and simulation integration to ensure that there are no over-constraints
7. Write covers for interesting design scenarios; calculate required bounds for inconclusive properties
8. If the cover fails to hit, then it points to a possible over-constraint. Analyze the uncovered item, and make corrections as required
9. If the cover bound is higher than the achieved assertion bound, then
   - Go back and implement further abstractions
   - Use divide and conquer approach to break down assertion into smaller assertions

10. Earlier, we only relied on the cover-items identified in step (7) to ensure that we do not miss any corner-case scenarios. However, with the use of Bounded Proof Coverage, we are now able to achieve higher confidence as the tool provides additional information on the portion of the code exercised during our verification (with the following additional steps).
11. Run Bounded Proof Coverage and analyze uncovered items. If the uncovered item is expected to be covered, then go back to step (9).
12. Sign-off based on cover-items in step (7) and results from Bounded Proof Coverage.

III. CHALLENGES WITH INCONCLUSIVE PROPERTIES SIGN-OFF

After getting full proofs for most of the easy-to-verify design logic, we start focusing on more complex scenarios where we tend to get Bounded proofs. Bounded proofs can happen because of various factors:

- large sequential logic in the property Cone Of Influence (COI),
- complex micro-architectures such as, arbiters, counters, memories, FIFOs, etc.,
- assumptions models that can introduce additional complexity,
- assertion models

Our first focus is to look out for any of those standard bottlenecks and apply widely available solutions[2] to achieve full proofs. The class of designs on which we applied this methodology is the packet processing design used in the switching device. The primary complexity is due to the intricate microarchitecture and long latencies to observe interesting scenarios. More detail on the use cases will be described below, but as formal coverage is an evolving area in formal verification, it will be good to define it first before going into the details of how we can leverage it for signoff.

IV. Introduction to Formal Coverage

Formal coverage is a capability using which we can understand the RTL code that is exercised by the engine when proving properties. We can also check which are the functional coverbins exercised while proving the assertion. It provides coverage metrics to formal verification and helps in increasing the confidence in the sign-off decision. The tool reports various metrics like design reachability for the given formal environment, design observability from the written assertion, assertion coverage, and bounded reachability. One can leverage each of these metrics at various stages of the verification for quickly fixing the verification holes. (Note that Formal coverage metrics can also be leveraged for proven assertions -- this paper only focuses on how to leverage it for Bounded proofs.)

Formal coverage is calculated based on code coverage items and coverbins which are observable from the property and also reachability of the same in the given formal environment.

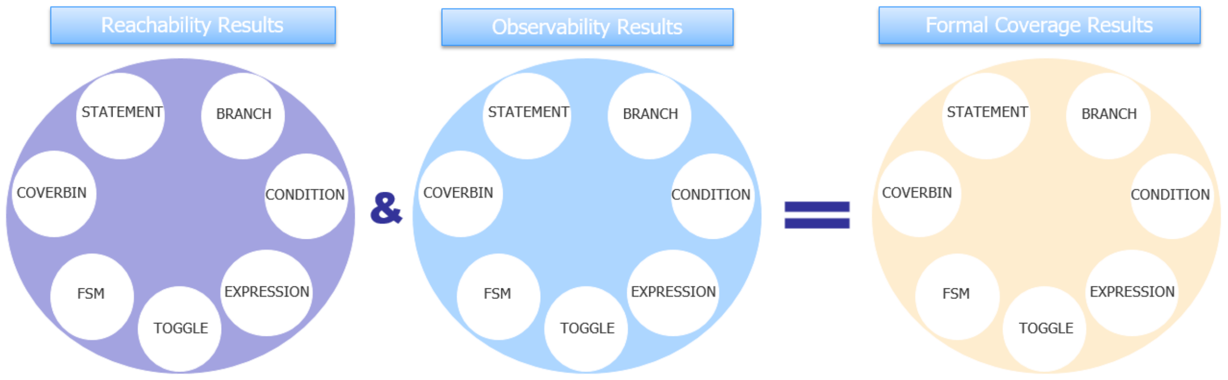
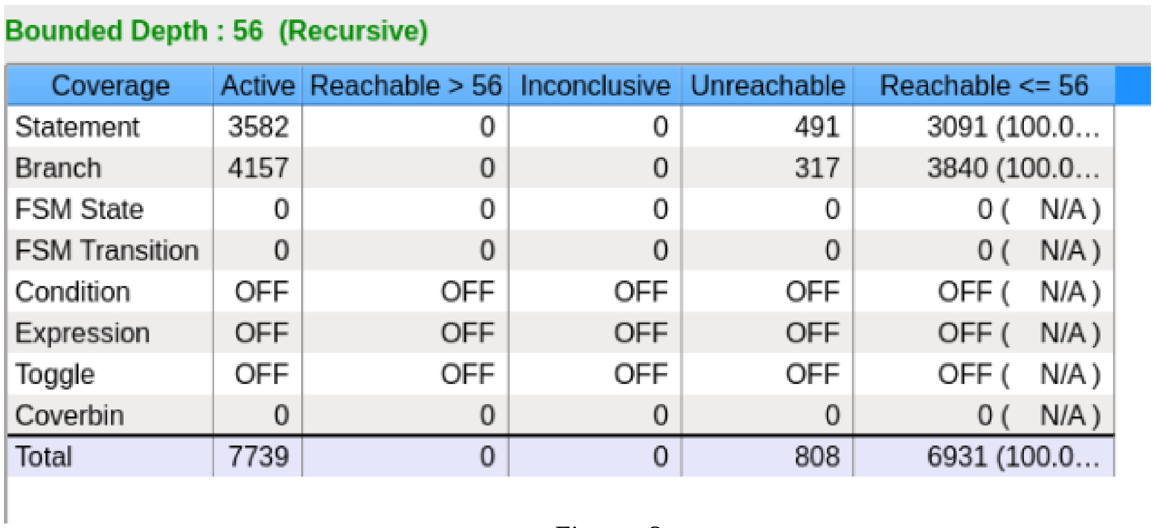


Figure : 2

Formal coverage can be computed with various degrees of accuracy, depending on where the measurement is focused. For example, when formal coverage is computed from the structural COI of the property it gives an "optimistic" coverage view. When it is done on the Proof Core, it will be a more "aggressive" coverage view. In the formal verification context, the "aggressive" adjective refers to the concept that the engine level view of the verification problem is the most optimized and accurate view. When extracting coverage from proof core, it shows only the portion of the code used by the engine, which is a subset of structural COI. At the early stage of verification, one can leverage more coarse coverage (Structural COI) data to fill the assertion coverage holes faster. Later, the focus can be shifted to Proof Core-based analysis during sign-off.

In the case of bounded proofs, when the design is explored only to a certain depth, it is not sufficient to look at the code coverage items which were reached -- it is also important to analyze how much of the code coverage items are reachable above the given bounds for the given property. The tool provided us with coverage data for both within the achieved bound and outside the achieved bound. This information helps in confirming the latency expectation which was defined based on design understanding.


Bounded Depth : 56 (Recursive)

Coverage	Active	Reachable > 56	Inconclusive	Unreachable	Reachable <= 56
Statement	3582	0	0	491	3091 (100.0…
Branch	4157	0	0	317	3840 (100.0…
FSM State	0	0	0	0	0 (N/A)
FSM Transition	0	0	0	0	0 (N/A)
Condition	OFF	OFF	OFF	OFF	OFF (N/A)
Expression	OFF	OFF	OFF	OFF	OFF (N/A)
Toggle	OFF	OFF	OFF	OFF	OFF (N/A)
Coverbin	0	0	0	0	0 (N/A)
Total	7739	0	0	808	6931 (100.0…

Figure : 3

As shown in figure 3 above, the coverage summary highlights the number of cover items which are reachable above the given bounds, inconclusive, and unreachable. This table also corresponds to a source code viewer that shows more information on these items in order to more closely analyse the scenario and take corrective steps.

V. Case Studies

Case study 1

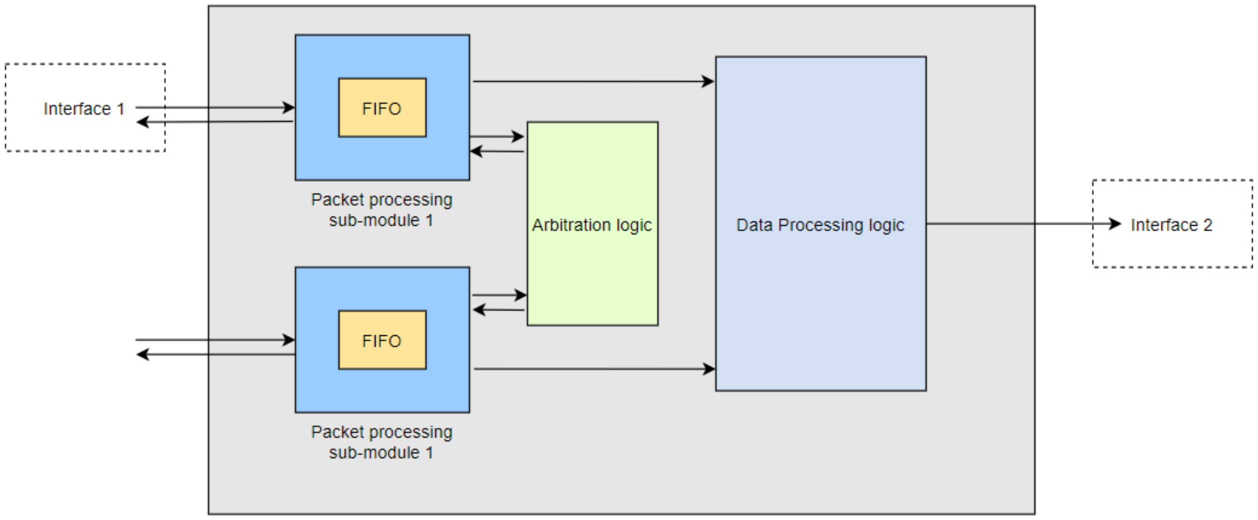


Figure : 4

Figure 4 above shows a simplified block diagram of one of our packet processing module DUTs. The input packets come in on multiple interfaces and are stored in FIFOs. Upon being picked up by arbitration logic for processing, these packets are de-queued from a FIFO, and then processed further, before being sent on to the output.

At the outset of the project, we had employed common abstraction techniques to reduce the state-space the formal analysis had to digest. We had performed black-boxing of FIFO memories, and other design elements that were outside the scope of the given test plan. We had also verified certain sub-modules separately -- including the arbiters

-- and a few of the other packet-processing modules. These individually verified sub-modules were replaced by abstraction models.

Still, even after applying abstractions, we had ~100,000 flops in the "cone of influence" (COI) of the formal analysis. This large COI translated into huge state space as the assertion went deeper, making it difficult for the end-to-end checkers to get full, "unbounded" proofs.

In the particular checker picked for this case study, we wanted to verify correctness of a particular data-field from interface 1 until interface 2.

This checker was implemented using the coloring technique, where we colored an incoming tracked packet at interface 1. For the tracked packet, we stored value of the incoming data-field. Further, when the packet was observed at interface 2, we compared the output data-field value with the saved value from input. Since this checker covers the complete end-to-end path from interface 1 to interface 2, the associated complexity was quite high and the formal tool was unable to provide a full-proof. Instead, this checker was bounded proven until a master clock cycle depth of 28.

We started implementing design covers, trying to calculate the required bound for this checker. We focused on the following parameters for our analysis:

a. An end-to-end latency calculation of the DUT logic
b. Requirements for the number of "interesting" events to be observed
c. Defining the finer, second-order requirements of the microarchitecture to be verified

The starting point of this analysis was the fact that the design had an end-to-end latency of 15 cycles. Also, while the packets were sent serially at the input, they could appear in parallel (up to 2 packets) at the output. So, 1st cell at the output was seen in cycle#16, where-as, first 2 cells were seen together in cycle#17.

Furthermore, in our abstractions, we had reduced the FIFO depth (figure 1) from 16 to 4. In order for neighbouring block to know if this FIFO had an empty slot available, design maintained credits, which were equivalent to number of empty FIFO slots available. So, these design credits were also reduced from 16 to 4. Thus, to view all transitions of the credit update mechanism we wanted to see at least 5 packets at the output. With help of the cover, we found this depth to be 21 cycles. With 1 extra cycle for safety, we calculated our required bound for this checker to be 22 cycles.

Since we were able to achieve the manually analyzed bound, we moved on to running Bounded Proof Coverage for this checker.

In the first coverage run, we observed the coverage to be at 99%. After analyzing the uncovered items, we found a formal test bench issue with the reset abstraction applied to FIFO pointers. To allow FIFO pointers to start from any depth (rather than default = 0), we had applied reset abstraction. However, in the 2nd cycle of the run, an internal reset mechanism was defaulting the pointers back to 0, essentially removing our abstraction.

Once the testbench issue was corrected, and we re-ran the coverage, we were able to observe 100% coverage. Below is the final coverage report for the given checker.

Coverage – 100% (6,931 cover items)

- Active – 7,739
- Unreachable - 808

In this context, “Unreachable” means that cover-point will not be reached even if this checker is unbounded proven. Upon analysis of these cover-points using the formal tool’s GUI, these were found to be reset conditions for flops that do not occur (by design) after cycle#0 onwards; thus we were okay to waive them.

Finally, with 6,931 cover-points being achieved within the specified bound, in addition to the manual analysis, this result gave us enough confidence to sign-off on this checker.

CASE STUDY 2:

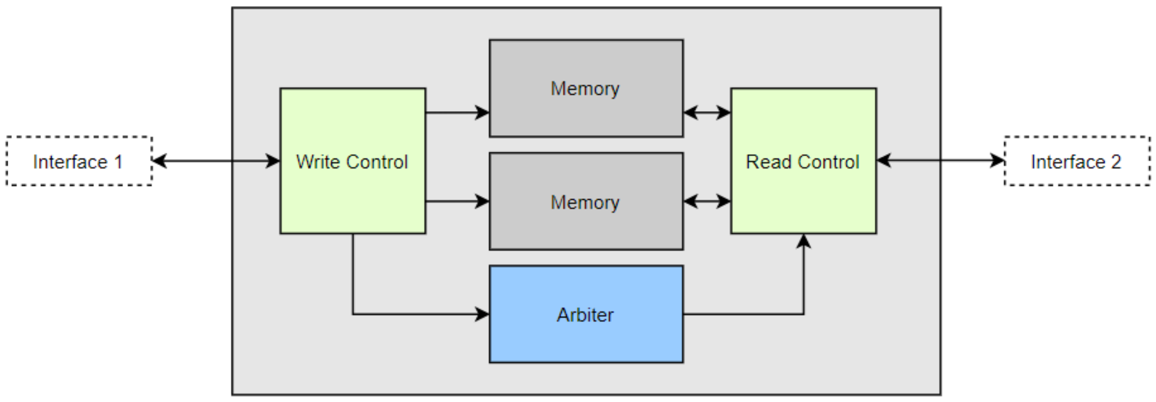


Figure : 5

Our second case study is also on a packet processing design, but here the focus of verification is on the read-write control logic.

The main functionality of this module is to process an incoming packet from Interface 1 and ultimately output to Interface 2. Interface 1 can send in two packets in one cycle, while Interface 2 can only send out one packet in one cycle. The memory occupancy is controlled by credit flow from Interface 1 to the upstream module. When the packet enters the block, the write control decides which and what part of memory to write to based on the control field of the packet. Furthermore, the arbiter block decides which part of memory is read out once the memory occupancy is updated by write control.

To reduce the complexity and state space, we black-boxed the memories and arbiters and verified them separately. We also abstracted out write and read address in the write/read control, so it can start at any address of the memory out-of-reset. This abstraction ensured that we were fully exercising the minimum, maximum, and wrap around address.

In the given checker, we wanted to verify the relationship between write and read control to the memory, and ensure that when a write occurs to the memory on a certain address, this particular address will not be written again before it has been read. This checker was bounded proven for 80 cycles after 12 hours of run-time.

Moving on to bound analysis, end-to-end latency for the complete module was 23 cycles. Breaking this latency down, it took 11 cycles for a packet on the input interface to reach the memory, 7 cycles from write control to reach arbiter, 3 cycles from arbiter to memory output, and 2 cycles from read control to the output interface.

With reset abstraction in place and memory depth set to 80, we needed 80 packets to cover all scenarios for memory control logic. This translated to 40 cycles at the input (since we can see 2 packets in 1 cycle at the input). Following from this, we set our minimum proof radius for this checker at 64 (40 + 23 + 1) cycles. So, we were able to achieve 16 cycles higher bound in our runs, as compared to the minimum proof bound.

Furthermore, running bounded proof coverage on it, we observed the following result:

Coverage --- 99.3% (4057 Active item)

- Active: 3573
- Reachable(>80): 4
- Unreachable: 484

After examining the unreachable items, we were able to put them into following categories:

1. Reset phase
2. Error condition
3. Intentional over-constrained condition

After review it was agreed that all unreachable items could be waived. Additionally, we analyzed the 4 cover-points which were reachable outside the achieved bound, and it pointed to design logic which wasn't being targeted with this checker; this seemed okay to be waived as well.

In conclusion, with 3,573 cover-points being achieved within the achieved bound, in addition to the manual analysis, the results gave us enough confidence to sign-off on this checker.

VI. SUMMARY AND RESULTS

It is important to have a good methodology to signoff bounded proofs. We learned that it is not sufficient to have design knowledge and functional cover properties alone. Having the capability in the tool to provide different coverage metrics helps in building confidence. Once we established the flow for some of the checks, we found the same can be applied for derivative functionalities checking with small modifications in the checkers.

In our recommended methodology, in the initial iterations we faced reduced coverage due to the abstractions we applied in the form of black boxes and constraints. We worked with our vendor to handle these scenarios through abstraction-aware coverage computation where it can leverage the constraints when computing the coverage.

Final Results

	Active	Reachable (>N)	Inconclusive	Unreachable	Reachable (<=N)
Case study 1	7739	0	0	808	6931
Case study 2	3573	4	22	484	3547

TABLE: 1

VII. CONCLUSION

Packet processing designs are very common in our switching products, and thus we showed how to setup and employ Formal techniques to exhaustively verify our key functionalities. We also showcased how formal bounded reachability coverage increases our confidence to apply the results of bounded proof to sign-off the whole end-to-end DUT verification.

REFERENCES

[1] R. Narayan, "The future of formal model checking is NOW!" DVCON paper, March 2014.
[2] Jin Hou, "Handling Inconclusive Assertions in Formal Verification", DVCON China paper, 2018