Formal Sign-off Methodology for IP Blocks

Anna Chang, Chia-An Hsu
Google Inc., New Taipei City, Taiwan
annacha@google.com, chiaan@google.com

***Abstract* — Coverage driven simulation-based verification is used as the primary verification methodology for modern SoC designs. It requires a lot of effort to build the test environments, from IP blocks, to sub-systems and finally the SoC system. Implementing comprehensive coverage metrics is critical, but also error-prone and time-consuming. Moreover, functional coverage metrics are often incomplete when design features change dynamically, resulting in coverage holes. We augment the UVM simulation-based sign-off process with formal verification to deliver high quality devices and we apply formal techniques where it improves effectiveness and productivity. We use several formal methods, and block level formal sign-off in the SoC verification process. This paper discusses the formal sign-off methodology for IP blocks in our SoC designs.**

Keywords— formal verification, soc verification

I. INTRODUCTION

Modern design tools enable us to produce larger and better systems year over year and on top of the increasing complexity, the systems have more requirements in terms of power, performance, and area. These designs need to be fully verified before going to silicon and verification strategy needs to evolve alongside the design productivity.

Simulation is the most widely used technique to ensure correct design functionalities. Dynamic simulation uses randomization to generate stimuli and exercise targeted features of the DUT. This methodology relies on comprehensive coverage metrics as a cornerstone, and missing test vectors leave the targeted functionality unchecked. As design complexity grows, test vectors also grow and implementing comprehensive coverage metrics is increasingly difficult.

With formal verification, the tool does all the heavy lifting to explore all reachable states. It converts the RTL design, constraints, checkers into equations and mathematically computes all possible sequences of states to prove the properties. Once the design is formally proven the coverage is comprehensive, so there is no lengthy coverage closure phase. The formal properties can be reused in simulation to either validate the constraints, or safeguard the use cases. Moreover, formal verification has the advantage over simulation especially when the DUT has complex corner cases that require a large variety of stimuli. The formal checks can be deployed early to shift left the long verification process and reduce the risk of schedule delay. Therefore, the formal methods have become indispensable in modern SoC verification methodology where it makes up the weakness of simulation.

The formal model checking helps us handle the SoC verification problem in a different way than simulation. This paper discusses how formal methods enable us to sign-off design blocks with complex corner cases.

The rest of the paper is organized as follows:

- *Background and Motivation* introduces the design and what motivated us to use formal sign-off.
- *Challenges and Strategy* explains the challenges and how using formal verification helps us meet our verification goals.
- *Formal Sign-off Methodology* describes key areas where formal is used and how to set up the formal testbench in formal verification flow.
- *Analysis of Formal Coverage* introduces coverage analysis and compare the coverage metrics from both formal and simulation perspectives. We explain the applications of formal coverage metrics in the sign-off process and share how formal coverage failures can be resolved.

- *Results* compares the data collected from the formal and simulation-based verification using the example IP block as a case study.
- *Conclusion* sums up the takeaways from this paper.

II. Background and Motivation

Modern SoC design begins with architecture development and design spec, which is elaborated into micro architectural spec and verification plan. Then design team start putting together RTL blocks and verification team start implementing the test environment. During the implementation phase, we go through the iterations of design reviews, test planning, coverage closure to guarantee the quality of the verification.

The coverage metrics are critical to assess the extent to which the verification environment stress the DUT, since insufficient coverage indicates holes where bugs can hide. The coverage metrics include structural coverage metrics and functional coverage metrics, that are used to measure the controllability and observability of the stimuli and checkers. They are either automatically generated from the EDA tools, or manually implemented, such as the functional properties and covergroups.

Among all the components in an SoC, the bus fabrics are ubiquitous and the bus controllers are required to guarantee protocol compliance. Verification on these bus protocol controllers is mission critical as any bug escaped can be a show-stopper.

One of such bus controllers in our SoC design connects managers and subordinates with the same handshaking protocol. After code review, regression, coverage closure, we still find issues because the controller can have long sequences of events and large number of random timing patterns. Moreover, the IP design is highly configurable and it is time-consuming to implement a large coverage set to close the complex functional holes. Therefore, we look outside of simulation methodology in search of more effective solutions. We will discuss formal sign-off flow based on this simple bus controller with one manager and one subordinate.

The example DUT has two sets of handshaking interfaces. On one end, it is connected to the protocol manager that issues requests and on the other end, it is connected to the protocol subordinate that responds to these requests. The DUT is responsible to transfer the requests from the manager to the subordinate and the responses from the subordinate to the manager following the handshaking protocol.

Furthermore, the DUT can get a request from the reset controller at any time and it has to respond with an acknowledge signal to the reset controller after it takes over the handshaking between the manager and subordinate. Once the reset controller sends a request to the DUT, it will reset either the manager or the subordinate after a fixed amount of clock cycles. The DUT has to maintain the protocol compliance on the non-resettable side of the interface, so the devices on the non-resettable bus interface won't hang.

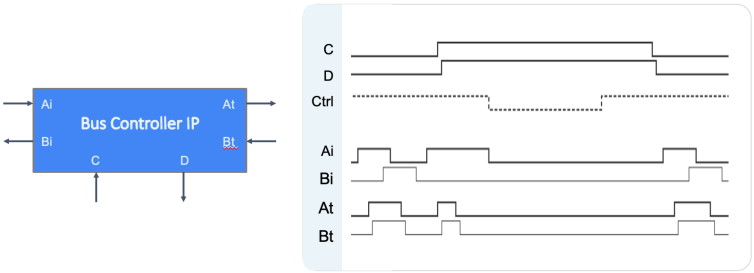

Figure 1. Design block diagram and waveform.

To cover all possible scenarios in simulation, we had to create all possible timing patterns of when the signals can be asserted or de-asserted for random period of time. We also had to implement the large set of functional coverage to measure the results. This depends on a comprehensive design specification, and requires manual work to convert the scenarios to test sequences and coverage metrics correctly.

III. CHALLENGES AND STRATEGY

The erroneous handshaking with reset described above is only one of many test scenarios for bus fabrics. Complex protocol verification needs to exercise many more cases, such as the normal handshaking with FSM transitions, sequence ID insertion, agent size randomization in shared bus, data re-ordering and forwarding, re-try mechanism, dropped packets, error responses, arbitration and flow control, bus matrix connectivity, bus fairness and starvation, minimum response requirement, maximum outstanding transactions, and so on. All of the behaviors have to be accurately specified by designers, and test components implemented by verifiers precisely.

Some of the IP designs can change on dynamic project needs and as a consequence, there often was not enough time to document all functional changes. The IP standalone simulation testbench is hard to maintain as every change requires new sequences and coverage metrics. Accordingly stressing the DUT comprehensively using traditional simulation testcases is challenging and achieving 100% coverage through dynamic simulation is difficult.

The RTL design can be seen as many states in a space and the target is to verify all the valid states and transactions. With dynamic UVM simulation we verify a sub-set of the state-space and with formal verification we verify all possible states exhaustively. Therefore, formal checking can discover interesting corner cases that are hard to imagine/hit with the simulation approach, and the overall turnaround time is much less using formal sign-off because it finds bugs in the shortest way. In comparison, the formal testbench took a lot less to build than that for simulation testbench. Formal provides us all possible input vectors that can help us gauge the completeness of the functional coverage. Therefore, using formal sign-off is more productive.

Since the complexity level of the bus controller is within the capacity of the formal tool, we decided to sign it off with the formal techniques for comprehensive functional coverage.

Our verification goals for the block level design sign-off are:

- Comprehensive functional coverage.
- Maximum Reusability.
- Minimum manual work to reduce errors.
- Schedule.

Using formal to sign off the bus controllers can not only prove the specified functionalities, but also catch undefined behaviors as it searches all possible cases if not over-constrained, so we left no stone unturned because of incomplete documentation. This allows us to gain more confidence that the design has been fully verified. Once it is proven there is no need to close the coverage and the properties are reusable in simulation and future projects.

IV. FORMAL SIGN-OFF METHODOLOGY

The followings are the key areas where we use formal apps and techniques in the SoC verification process:

- Connectivity check
- Automatic formal linting
- X-propagation check
- Formal coverage analysis
- Unreachability analysis
- Sequential equivalence check
- FPV in block level design sign-off

Automatic formal checks or formal linting can sanitize design code by detecting dead code, arithmetic overflow/underflow, out of bound indexing, state machine deadlocks, et cetera. The formal connectivity check is more efficient than running pin wiggling simulations. Sequential Equivalence Check verifies if two design models

are functionally equivalent even when their state elements are different, and it is widely used to verify clock gating schemes, design re-pipelining and incremental design changes. X-propagation verification identifies paths that can hide bugs or create discrepancies between the RTL and gate simulation. Unreachability analysis detects the dead code that can be bugs, or redundant code.

For connectivity check, formal linting, x-propagation check, these apps require no knowledge of SVA, or formal bench. They can be up and running with a simple tcl script and design specific constraints. They are quick to bring up and catch low hanging bugs right off the bat. These applications make it easy to start the verification early.

The end-to-end formal checks offer the most benefits to prove a design completely, but it is also the most difficult to build, and converge. The result of each formal check can be a failure, an unbounded proof, or a bounded proof by a depth of cycles from reset. Just like in simulation-based flow, we need meaningful metrics to help us answer: if a property is proven, or it achieves a bounded proof of N cycles on a given design, how much of the design functionality have I verified? Is it complete or is there still risk for any error? Are there sufficient checkers for all the design features? At the end of the formal regression, we analyze formal coverage to decide how complete the formal verification is. Formal sign-off flow has been developed to allow for measurement of such quantifiable metrics on the quality of a formal environment.

To meet our goals for the bus controller verification, we need to make sure all the specified behaviors are understood and their checkers are implemented. The verification targets and metrics were set during the verification planning stage to facilitate execution. In simulation, the progress can be tracked by the testcases, regression results and the coverage from the regression. In formal, test items are listed by design feature functionality and coverage is also collected to determine the quality of the proofs. Each test item for a feature can be implemented with several different assertions that link to this feature. The feature is deemed to pass only when all the assertions linked to the feature pass and the progress is measured on the basis of design features.

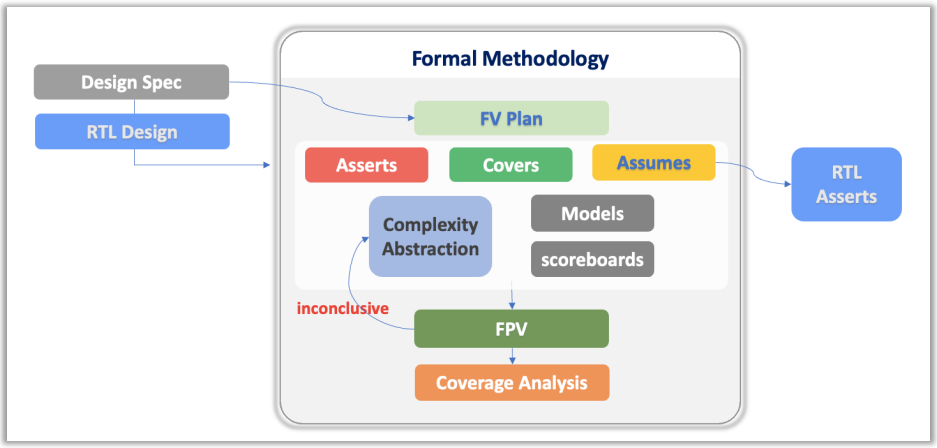

Figure 2. Formal Methodology

To bring up a full-blown FPV formal verification environment, we need the formal testbench, the formal properties and the tool specific tcl command file. Constraints and assertions are two main components in the formal testbench. The constraints are responsible for determining stimulus for design sensitization and the assertions are responsible for providing checking capability.

Therefore, the verification plan needs to define the design features under verification, the design constraints and the associated assertions to be implemented. Other than the constraints and assertions, we also need to specify the design configuration cases, critical case coverages, abstraction strategy, and the comparison mechanisms.

Formal friendly modeling logic in synthesizable SVA are commonly used to constrain the logic outside of the design, or generate reference behaviors for assertions. For instance, proof accelerated scoreboards and assertion-based VIP are developed with symbolic variables and formal friendly modeling techniques to avoid state space

explosion. Using these models provided by tool vendors can greatly improve convergence. Project specific properties need to be developed case by case to check custom features such as reset sequence, FSM transitions, et cetera.

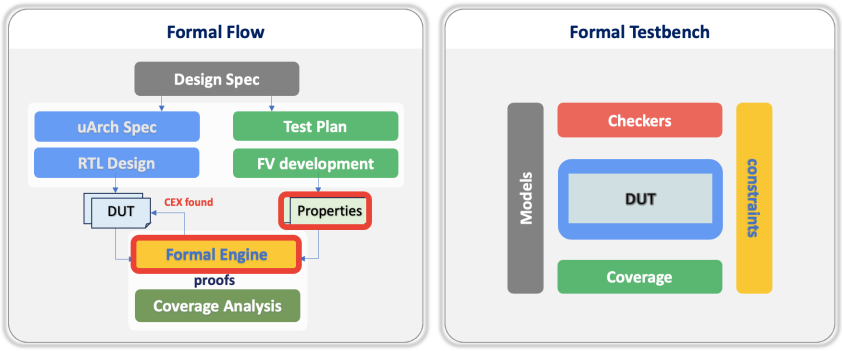

Figure 3. Formal flow and testbench diagram

Our formal bench uses formal friendly assertion based VIP for standard protocol verification, and tool-specific scoreboard on data integrity checks. To reduce the state space, abstraction techniques are used, such as over-constraining part of the wide data bus. To speed up convergence, we use techniques such as case-splitting, initial value abstraction, assume-guarantee.

Our formal verification strategy includes:

- Assertion based, coverage driven
- Target full and bounded proofs of entire IP block
- Target functionality of entire design
- Target interesting scenarios
- Target one level up end-to-end features
- Bug hunt to search for difficult scenarios
- Use coverage as feedback for targets
- Blocks from lower level are formally proven first and they become assumes in higher level verification.
- Incremental changes are verified using sequential equivalence check.
- Automate the run management.

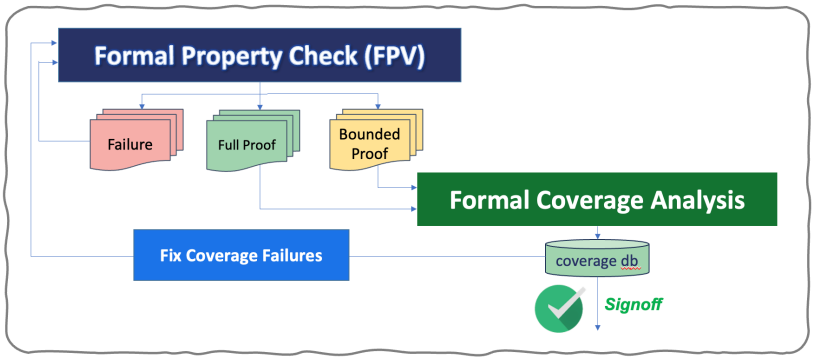

Figure 4. Formal sign-off flow

Once the formal bench is built, the formal verification process begins. At the end of the formal analysis, we collect formal coverage to decide how complete the formal verification is.

Formal sign-off flow has been developed to allow for measurement of such quantifiable metrics on the quality of a formal environment. Although the aspects in the formal coverage methodology may be different than those in simulation methodology, the concept is the same. We set the test goals and measure the completeness of work by meeting the goals, which are the coverage targets hit with the testbench. This allows us to determine how close we are to a tape-out worthy design.

There are many formal coverage metrics that can be used to provide insight into the formal environment quality.

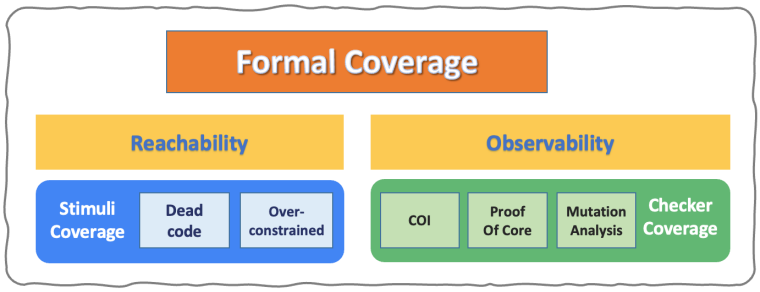

Figure 5. Formal coverage metrics

Due to the nature of the formal technology, formal not only tells what is reachable but it also proves what is unreachable. The most immediate contribution from the formal coverage analysis is the automatic exclusion file generation and using the UNR analysis with simulation to speed up coverage closure has become a standard practice.

The code coverage has been integrated as a standard feature of formal verification and the generated targets are identical to the targets used by simulation tools. The general purpose of code coverage remains the same, which is to find the reachable holes in the design, that arise from a lack of input stimulus or deadcode. In simulation, code coverage holes indicate insufficient quality and quantity of tests, whereas unreachable coverage failures in formal usually mean a presence of over-constraints in the setup or insufficient runtime of the formal proof.

V. Analysis of Formal Coverage

Simulation uses a zero-plus approach where we add tests to exercise different pieces of code and improve coverage over time. In formal, we use an infinity-minus approach, where we start from unlimited input space and add appropriate constraints to remove illegal input vectors.

The result of each formal check can be a failure, a full proof, or a bounded proof by a depth of cycles from reset.

Formal coverage analysis helps us determine:

- Are there sufficient assertions to exercise all the features?
- Are they of good quality to detect all erroneous behaviors?
- Are the depths of bounded proofs sufficient to rule out the possibility of a bug?
- Is the formal engine able to access all legal state space without over-constraints?

Using reachability analysis can examine whether any legal scenarios are left out due to over-constraint and using the observability analysis can confirm whether all the output behaviors are checked by the assertions.

A. Formal Coverage Analysis for Sign-off

Our formal coverage analysis for verification sign-off is performed in five steps.

The first coverage analysis for formal sign-off is the reachability analysis, which measures how exhaustive the formal engine explores the DUT, followed by the checker coverage analysis and bounded coverage analysis for inconclusive proofs. Each step of the methodology is realized through coverage model and type that allow us to gain accurate understanding of how much of the intended features are exercised.

1) Over-constraint analysis

The constraints we wrote to rule out illegal input stimuli can sometimes eliminate legal cases inadvertently and this can invalidate a pass. The code coverage or stimuli coverage tells us whether the verification environment has controllability or reachability issues. If the over-constraint is intentional as in the example of splitting the case, it can be waived. Otherwise, over-constraints need to be removed to allow legal state space for formal engine to explore.

On the flip side if under-constraint happens while the property is proven, this is usually not something to be concerned about and, in some cases, using stopat or free wire to under-constraint the design can help get convergence faster. When there is an issue with under-constraint, it manifests as an assertion failure more often than not.

2) COI analysis

The checker or property density analysis measures the completeness of property set applied to the DUT. When formal tool analyzes each property, it computes a network of gates and flops that influence the result of this checker and this network is called the cone-of-influence, as known as COI. Once the COI is built, the tool searches the full state space of possible values that can apply to the logic inside COI. This process takes place in parallel so if a counter example or a cover trace is found, it is always the shortest sequence.

The formal tool flags observability issues by determining the cover items in the structural COI of assertions. It finds the union of the assertion COIs and the remaining out-of-COI items indicate blind spots in the assertion set.

The COI analysis reports the ratio of RTL line of code to number of assertions and highlights design logic that are out of any assertion's COI. To find out whether there are sufficient properties to cover the functionalities, we need to look at how much of the code is covered. The formal engine structurally traverses a cone of logic under each property, from the beginning of the assertion all the way to the point where it finds a bug, or pass. The out-of-COI cover items indicate holes in the assertion set where the bugs could exist. Therefore, all the uncovered registers, primary inputs/outputs should be carefully examined and determine whether it is a bug, to waive, to remove, or to implement more assertions to cover.

The COI coverage is only a structural check and it is an inexpensive way of checking whether a section of code is getting verified or not. In general, it is not sufficiently accurate to make the call.

3) Proof Core analysis

The Proof Core is a subset of the structural COI logic of an assertion and it contains the essential logic that needs to be exercised in order to establish the correctness of the assertion. The logic outside of the structural COI cannot influence the proof results and the logic inside of the COI may or may not influence the proof result. To compute the formal-core, the engine draws optimization on COI logic, to slice off the fan-in cone that cannot influence the proof, and that results in cut-out portions within the structural COI. The cone is not whole anymore and it shows the actual proven formal core coverage to measure property completeness. If there is a bug inside this proof core, it will be observed and trigger a counter example. Meanwhile, bugs in the uncovered area cannot be observed.

For unreachable failures from proof core analysis, we use the same approach as in Figure 6 on page 8. Intentionally uncovered portion of the design can be black-boxed or insert cut point without impacting the formal proof results. For inconclusive results, we can increase the runtime, perform bug hunting to gain more proof core coverage. We can also use cut point to check if the property is falsified, which indicates the code is within the proof

core. For signals that cannot propagate to properties or be observed, there can be bugs and should be carefully examined with cover properties.

As the proof core coverage analysis requires formal engine to mutate the regional design logic inside COI through iterations, it takes longer and greater tool effort. The union of the proof cores provides a more accurate measurement of checked verification goals.

4) Fault Injection analysis

The mutation coverage is another observability coverage metric and the tool inserts erroneous code, point by point, into DUT to measure whether the bug can be exposed by the formal testbench. The same metrics such as statement, branch, expression and toggle are used for checker analysis. Each item of the metrics is a target to apply StopAt or StuckAt mutation. This analysis checks if there is any "non-detected fault" in the DUT that escape the formal verification. The formal verification is considered complete if no bug can be injected without being detected by the assertions.

The mutation coverage is a point-based analysis and it provides the highest level of precision on the quality of a formal testbench. Since the tool has to examine each logic point in the design one by one, the mutation coverage analysis is very compute-intensive. Because the process is so time-consuming and resource-hogging, it is only run on major milestones.

5) Bounded Proof analysis

Assertion bound analysis provides analysis of the cycle bound achieved by a bounded proof. The analysis provides insight into the chance of having a bug outside of the bound cycles by reporting cover items in COI of the assertion not reachable within the bound. Covers beyond the bound cycles cannot be signed off and will require more verification using simulation or bug hunting.

B. Formal Coverage Failures and Reasons

The following tables summarize the common failures and solutions for the formal reachability, COI, proof core coverage analysis.

Unreachable / Out of COI	**How to proceed**
Inputs/signals missing checkers	Add assertions
Unused inputs	Waive or remove
Disabled or tied inputs	Waive
Connection broken by bbox	Remove bbox or add asserts to bbox inputs and add assumes to bbox outputs.
Tied inputs	Waive
Intentional default statement	Waive
Logical expression cannot happen	Could be a bug!
Over Constrained	**How to proceed**
Intentional Over-Constraint	Waive
Unintentional Over-Constraint	Fix to remove or change to intentional OC.
Inconclusive	Increase COV measure time limit Or perform bug hunting effort

Figure 6. Formal stimuli coverage analysis and solutions for failures

Out of Proof Core	How to proceed
Dead Code, Unreachable	Refer to out of COI Failures.
Inconclusive Not run long enough	Increase run time limit.
	Perform bug hunting to cover more FC.
	Use cut point to see if it causes CEX.
Signals cannot propagate to assertions	Could be bugs
	Use cover property to investigate.
Unfriendly formal code (for simulation only)	Waive or Fix the code

Figure 7. Formal checker coverage analysis and solutions for failures

VI. RESULTS

We found that using formal verification can detect the bug much faster and it requires much less effort to bring up the formal testbench. For the case in discussion, there are 3367 lines of code for the UVM testbench and it took four work weeks to build. There continues to be design issues found during the design review after the coverage closure.

With formal verification, it took 368 lines of code and four days to bring up the process and it immediately detects the bug found in the simulation bench. The runtime is also much less than running the simulations as shown below.

On the first day when the formal verification was brought up, there are two bugs found. With the help of the coverage analysis, more properties are added and constraints refined. In the third run, the coverage was greatly improved and there are two more bugs found. The rest of formal analysis did not find new bugs while we examined the over-constraints and uncovered design code.

Third Run

Types	Number
Assertions	25 (16 pass, 9 CEX)
Covered	171
Unreachable	40
Bugs	4

Formal Coverage	Stimuli Coverage	Checker Coverage
170/245 (69.39%)	210/245 (85.71%)	128/199 (64.32%)
118/166 (71.08%)	134/166 (80.72%)	118/164 (71.95%)
9/11 (81.82%)	9/11 (81.82%)	9/11 (81.82%)
6/7 (85.71%)	6/7 (85.71%)	6/7 (85.71%)
9/11 (81.82%)	9/11 (81.82%)	9/11 (81.82%)
9/11 (81.82%)	9/11 (81.82%)	9/11 (81.82%)

Formal Coverage Runtime (CPU Time in seconds)

	UVM Bench	Formal Bench	Comparison
SIM Coverage	1494 sec.	-	1494/21.53=69.39X
COI Coverage	-	3.56 sec.	
Over Constraint	-	6.34 sec.	
Formal Core	-	11.63 sec.	3.56+6.34+11.63=21.53

Bench Comparison

	UVM Bench	Formal Bench	Efficiency
Line of Code	3367	368	9.5X
Effort	4 weeks	4 days	7X
Run Time	1440 sec. (30s each)	5.04s (cpu time in sec.)	286X
Results	Bug escaped!	Found bug the first day	Invaluable!

VII. CONCLUSION

In this paper we discussed how to build a formal verification testbench and analyze the coverage to determine the completeness and quality of the formal proofs.

Using formal property verification and coverage analysis to sign off IP block is more effective and efficient especially in fast design feature changes. Many formal applications allow the verification process to start much

earlier than using simulations and with full formal proofs, there is no lengthy coverage closure phase to impact the schedule.

The formal coverage analysis provides metrics from coarse grain to fine grain down to point level on the quality of the formal verification. These metrics offer accurate assessment on the risk of malfunction after a design is signed off using the formal verification. That increases our confidence in reaching our goals of delivering high quality designs and devices.

REFERENCES

[1] B. Murphy, M Pandey and S. Safarpour , "Finding Your Way Through Formal Verification"

[2] Anna Chang, "Applying Formal Analysis in Simulation Signoff Flow," SNUG 2020, San Jose, CA

[3] Sabbagh, Saxena, Mittal, Nordstrom, "Formal Coverage for Verification Sign-off," SNUG 2017, San Jose, CA

[4] Fernanda Augusta Braga, "Bootstrap Formal Coverage Analysis," Jasper User Group 2022, San Jose, CA

[5] N. Kim, J. Park, H. Singh, V. Singhal, "Sign-off with Bounded Formal Verification Proofs," Design Verification Conference (DVCon) 2014.

[6] Abhinav Gaur, Gaurav Jain, Ruchi Singh, "Metrics Driven Sign-off for SoC Specific Logic (SSL) Using Formal Techniques," DVCON 2023, San Jose, CA

[7] "JasperGold Command Reference," Cadence Inc., Sep, 2022

[8] "VC Formal Sighoff Methodology Guide," Synopsys Inc. T-2022.06