A Coverage-Driven Formal Methodology for Verification Sign-off

Ang Li, Hao Chen, Jason K Yu, Ee Loon Teoh, Iswerya Prem Anand
NVM Solutions Group
Intel Corporation

***Abstract*- Formal Verification (FV) has become a mainstream verification methodology with widely recognized advantages in finding corner case bugs in complex designs. However, given that most verification teams use and interpret coverage from constrained-random, dynamic simulation for verification signoff, the lack of an equivalent signoff metric in FV is one major barrier to adopting FV exclusively for verification signoff. This paper promotes a coverage-driven FV methodology that can be used for verification signoff. The methodology allows FV work to be reported in the same language (i.e. coverage) as dynamic simulation. This enhances confidence of FV work by providing deterministic measure of the state space that has been verified. It also enables more seamless integration and co-verification of FV and dynamic simulation with merged coverage results. Our results show significant Return on Investment (ROI) in quality of verification and productivity.**

I. Introduction

Formal Verification (FV) has become a mainstream verification methodology, and has been deployed in many companies to verify complex SoCs. Compared to dynamic simulation, some major advantages of FV are the confidence of exhaustive proofs to cover 100% state space, simpler testbench structure, and faster time to find potential bugs (counter-examples). In the past, we had already successfully used FV in targeted areas to find high-quality bugs missed in simulation, and in early design to improve quality of design to verification team handoff. However, an obstacle to adopting FV as the exclusive methodology to verify a design block has been the lack of a functional coverage metric, which non-FV verification engineers (and managers) are used to seeing, as a measure of verification completeness. FV tools have recently added support for various kinds of coverage [1][3][8], but it is sometimes difficult to translate them to equivalent coverage metrics derived from dynamic simulation.

To address this limitation, we have developed a coverage-driven FV flow that mimics dynamic verification methodologies. It allows up-front planning of verification goals, and clear verification signoff criteria (detailed signoff criteria is defined in section II). This flow seamlessly integrates with our mainstream constrained-random dynamic simulation environment and coverage metrics, such that we can derive a single coverage number that combines both dynamic simulation and FV coverage, if a design block incorporates both methodologies. Furthermore the flow enhances FV work itself by ensuring complete functional and code coverage.

With this flow, we were able to achieve verification signoff for a complex micro-sequencer block that was verified with FV only, with measurable, and equivalent or better quality, than if the block had been verified with dynamic simulation. Moreover, this flow enabled us to utilize both FV and simulation methodologies together towards the project-level signoff goal, to choose the best methodology for each item of the test plan, and report the progress (coverage) from both methodologies in a single metric. Using a FV/simulation mixed approach, we also verified another sequence-based design block, which gave us higher confidence in the completeness of verification.

This paper describes our coverage-driven formal verification signoff flow with collected ROI using this flow. Section II of the paper explains the formal coverage metrics that we used, how we wrote our FV test plan, and the formal signoff flow. Section III explains how we used FV and simulation together for verification signoff. Section IV presents the results of using this formal signoff flow and ROI collected in terms of design quality and project productivity. In section V we discuss some future work we are planning to further enhance the flow.

II. Coverage-Driven Formal Verification Flow

A. Formal Coverage

The ultimate goal of verification signoff is to provide confidence that the design under test (DUT) has been fully verified in a deterministic, and measurable way. Coverage-driven verification methodology has been widely adopted as an industry standard for dynamic simulation signoff. Usage of both functional and code coverage allows verification engineers to set clear verification goals and measure the state space covered as they verify the design.

In theory, FV promises full state space coverage because a formal proof is exhaustive. As long as we ensure the appropriate assertions are implemented to check the complete set of design behaviors (a common challenge to any verification methodology), we should be guaranteed to exhaustively verify every feature of the DUT. However, this promise frequently evaporates when we apply FV on modern complex ASIC design blocks. The two major challenges specifically for FV are [1]:

1) Legal design state space becomes unreachable due to (usually unintentional) over-constraints in the FV environment. As shown in Figure 1, the yellow area illustrates the legal state space not exercised/checked by FV due to over-constraints. It is not uncommon to have unintentional over-constraints in a FV environment since they are difficult to uncover. Without proper coverage analysis, bugs in over-constrained legal state space may be missed by FV.
2) Legal design state space cannot be exhaustively reached leading to inconclusive formal analysis results, i.e. bounded proofs. As shown in Figure 1, certain portion of the state space in green may not be fully proven due to design complexity and/or tool/resource limitations. More coverage analysis is required to ensure all bounded proofs are justified.

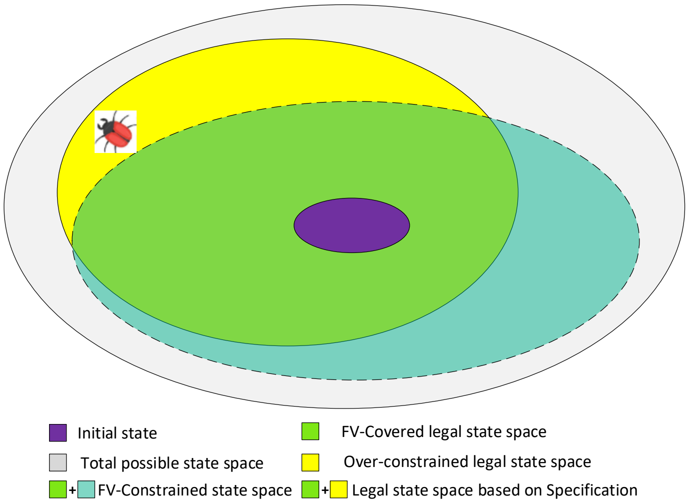
🟪 Initial state
🟩 FV-Covered legal state space
⬜ Total possible state space
🟨 Over-constrained legal state space
🟩+🟦 FV-Constrained state space
🟩+🟨 Legal state space based on Specification

Figure 1 Design State Space Coverage

To confidently signoff with FV, the challenges mentioned above have to be carefully examined and properly addressed. Current commercial formal verification tools have evolved to provide several automated code coverage metrics to help overcome these challenges [3] [8]. But based on our experience, they are not enough to gain signoff confidence because these coverage metrics cannot completely address the two mentioned limitations. Therefore some improvements need to be made.

First, to ensure there are appropriate assertions in a FV testbench to cover all design behaviors, we require that the complete functional requirements to be captured in a verification plan (vplan). More details on how to write a deterministic vplan will be covered in the next subsection. During the verification execution phase, we use assertion COI (Cone of Influence) coverage [8] to ensure the entire logic of the DUT, except dead code, is within the union of COIs from all assertions.

Second, to eliminate unintentional over-constraint, we use stimuli reachability coverage to ensure no branch/statement/expression in the RTL is unreachable due to incorrect constraints. We also use functional coverage, as we had encountered some complex interdependent over-constraints that could not be uncovered by stimuli coverage only. If a simulation environment is available, we also try to validate our FV constraints in simulation.

Third, in case of bounded proofs, formal techniques such as abstractions are first applied to improve convergence. However, it is not uncommon to sign off verification with a small percentage of bounded proofs. In such case, we use assertion bound coverage and some other design-dependent metrics [4] to determine if the proof bounds are sufficient. Moreover, current FV tools support using SystemVerilog covergroup to define complex functional coverage. Tools can also use the covergroup to explore deeper design state space beyond the assertion bounds, and evaluate the associated assertions in any functional coverage traces. Semi-formal techniques such as state swarm or cycle swarm are often required to conduct sparse, but deep, search of the state space, to help hit deep functional covers or run deep state space bug hunting [3]. This step is illustrated in Figure 2.

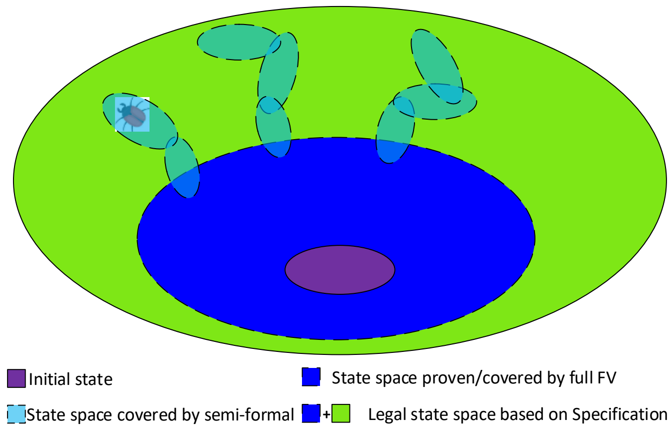
🟪 Initial state
🟦 State space proven/covered by full FV
▨ State space covered by semi-formal
🟦+🟩 Legal state space based on Specification

Figure 2 Increased State Space Coverage with Semi-Formal

In summary, we use the same code and functional coverage metrics as dynamic simulation to measure the completeness of our FV environment and work to achieve verification signoff. With these quantitative metrics, FV engineers are able to report FV progress in a common language as simulation, and verification managers are able to plan FV work the same way as simulation. This has greatly increased our team's confidence in using FV exclusively as a signoff verification methodology. In the next two subsections, we will explain in more detail how we define the coverage metrics during verification planning phase and how we use them during the verification execution phase.

B. Formal Verification Plan

At the verification planning phase, we need to first identify whether the whole target design block, or a subset of features in the target design block, are suitable for FV signoff, based on characteristics such as sequential depth, design type, block size, etc. [1][2]

If we determine FV is suitable for signoff, we will develop a formal verification plan (vplan). A vplan contains a prioritized list of functional requirements (FR). We use three common types of requirements in our vplan:

1. Generate requirements: defines stimulus generation. In FV, this type of FR defines proper constraints on input stimuli.
2. Check requirements: defines behavioral checks. In FV, this is implemented as assert properties in SystemVerilog Assertions (SVA).
3. Cover requirements: defines design behaviors that need to be covered during verification. In FV, we use functional cover requirements to show that certain design behaviors have been exercised and checked. They are implemented in SVA cover properties or SV covergroups based on the requirements, and sampled when the associated check occurs or is guaranteed to occur. Automatic witness covers can sometimes be used. However, frequently they are inadequate, as each witness cover only covers one single, usually less desired, design behavior.

Check and cover requirements are both needed to verify a FR to ensure the check has been exercised on the desired condition (defined in the cover). The association between check and cover requirements is furthermore ensured by 1) proper mapping and association in the vplan, 2) consistency between the sampling condition of a covergroup and the precondition of the corresponding assertion(s). Code coverage is usually defined as a separate requirement.

Table 1 explains how we implement each type of requirement in FV. Functional requirements in general are tool/methodology agnostic. We can choose the most efficient methodology to implement them.

TABLE 1
FORMAL FR TYPES

Requirement Type	Corresponding FV Implementation
Generate	assume property
Check	assert property
Cover	cover property or covergroup

Requirements should also be written in strong language which is clear, specific, and verifiable. Some examples are provided in Table 2. Note that EXAMPLE.COVER.01 is the corresponding cover to EXAMPLE.CHECK.01. Together they ensure the design behavior has been exercised and checked.

TABLE 2
EXAMPLE REQUIREMENTS

TYPE	DESCRIPTION
EXAMPLE.GENERATE.01	Assume that when the sequencer is not idle (**`seq_idle`** = 0), Sequencer reset shall remain de-asserted (**`seq_rst_n`** = 1).
EXAMPLE.CHECK.01	When multiple triggers are active and there is one or more active trigger(s) with priority 3 (highest), check that any one trigger with priority 3 is chosen in the same clock cycle.
EXAMPLE.COVER.01	Cover cases where multiple simultaneous trigger inputs are active. The bins shall be: - 1 to MAX triggers are active with priority 3, the rest of triggers are either not active or active with lower priority; This cover shall be sampled when EXAMPLE.CHECK.01 is triggered.

C. *Coverage-Driven Formal Verification Flow*

The execution phase includes implementing requirements in a FV testbench, running regressions to prove all properties, and releasing coverage results to our verification reporting infrastructure. Our complete signoff flow is illustrated in Figure 3.

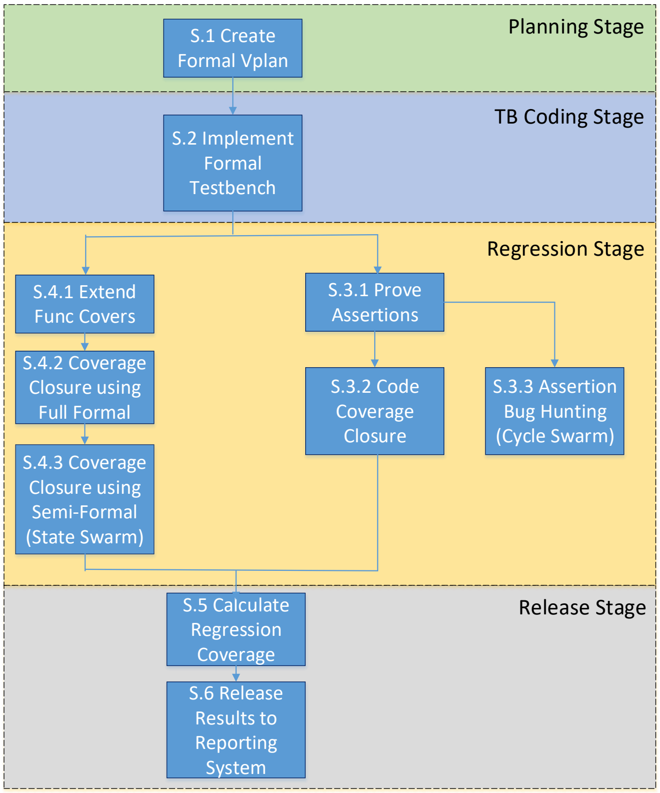
Planning Stage
S.1 Create Formal Vplan
TB Coding Stage
S.2 Implement Formal Testbench
Regression Stage
S.4.1 Extend Func Covers
S.4.2 Coverage Closure using Full Formal
S.4.3 Coverage Closure using Semi-Formal (State Swarm)
S.3.1 Prove Assertions
S.3.2 Code Coverage Closure
S.3.3 Assertion Bug Hunting (Cycle Swarm)
Release Stage
S.5 Calculate Regression Coverage
S.6 Release Results to Reporting System

Figure 3. The Coverage-Driven Formal Signoff Flow

Planning Stage:

- S.1 Create Formal Vplan: in this step we create the formal vplan based on design specification.

TB Coding Stage:

- S.2 Implement Formal Testbench: in this step we create the FV testbench. A FV testbench consists of SV modeling code, SVA properties that implement all generate/check/cover requirements, and FV environment setup files such as tool control scripts and any assertion-based VIP (ABVIP).

Regression Stage:

- S.3.1 Prove Assertions: in this step we prove all assert properties. It is an iterative process to debug assertion failures to fix any spec, RTL, or testbench issues, before assertions can be proven cleanly. In regression mode, we also enable code coverage generation and prove code cover properties with assertions. Our goal is to achieve full proof of all assertions, but usually there is a small percentage of assertions that do not converge. Complexity reduction techniques such as abstractions [1][3] are applied to improve convergence.
- S.3.2 Code Coverage Closure: in this step, code coverage, specifically stimuli reachability and assertion COI coverage, is analyzed and closed. Proof core coverage is also reviewed, but it is not required to close for signoff, as it is difficult to achieve 100% proof core coverage with bounded assertions. In case of bounded assertions, assertion bounds are analyzed based on code coverage to ensure sufficient proof depth.
- S.3.3 Assertion Bug Hunting: for assertions that are not fully proven, deep state space bug hunting is run to search beyond the assertion bound for potential bugs. Semi-formal techniques such as cycle swarm are used in this step [3].
- S.4.1 Extend Func Covers: functional covers written as SV covergroups are sampled when the associated assertions are triggered (refer to Table 2 for an example). To further utilize the cover traces, we extend them to an end-of-test condition to witness more design behaviors after the sampling condition [3]. For example, the original cover implemented for EXAMPLE.COVER.01 (described in Table 2) is sampled when multiple triggers are active on the sequencer’s input. In this step we further extend this cover to an end-of-test condition where all pending active triggers are eventually serviced.
- S.4.2 Coverage Closure using Full Formal: in this step, we try to prove all functional covers using full formal engines (traditional engines that do not incorporate semi-formal techniques). FV tools now also support evaluating that all assertions hold true along the functional cover traces [3]. This is extremely important for bounded assertions in case the functional cover traces are longer than the bounded proof depth.
- S.4.3 Coverage Closure using Semi-Formal: for deep functional covers (usually thousands of cycles), semi-formal techniques such as state swarm need to be applied to reach these cover points. Very often we need to create helper cover properties to guide semi-formal engines to reach the final desired cover point [3].

Release Stage:

- S.5 Calculate Regression Coverage: once a FV regression completes, we calculate coverage results and annotate them onto the vplan. Coverage score is calculated based on the percentage of cover points reached per cover requirement. If any associated check requirement (assertion property) fails or is proven with insufficient bound, its cover score is nullified.
- S.6 Release Results to Reporting System: in this step, we release coverage results and FV regression reports to our verification progress tracking tool.

Verification signoff for the design can be declared when the following criteria are met:

1. 100% functional coverage reached. No assertion failures evaluated in any functional cover trace.
2. 100% COI and Reachability coverage reached with waivers (for dead code, etc.)
3. All assertions are either fully proven or bounded proven with sufficient bounds.
4. No conflict in FV constraints.

III. Formal and Simulation co-Verification

A. Formal and Simulation Co-Verification Plan

As described in section II, requirements are general verification features. Therefore we should be able to choose any suitable methodology (simulation or formal) to implement a particular requirement. With this FV coverage methodology, we can mix and match FV and simulation methodologies to target different requirements, then merge the results, to achieve signoff confidence in a more efficient way. For example, we can use FV to target hard-to-simulate sub-blocks or design features to verify them more thoroughly, and run dynamic simulation at a higher level to test end-to-end scenarios with high sequential depth. Our vplan allows us to select FV or simulation as the verification method per group of requirements, as illustrated in Table 3.

TABLE 3
Example Formal and Simulation Co-Verification Vplan

Feature	Requirement ID	Requirement Type	Method	Coverage Score
Feature 1	FEATURE1.GEN.01	Generate	Simulation	
Feature 1	FEATURE1.CHK.01	Check	Simulation	
Feature 1	FEATURE1.COV.01	Cover	Simulation	80%
Feature 2	FEATURE2.GEN.01	Generate	Formal	
Feature 2	FEATURE2.CHK.01	Check	Formal	
Feature 2	FEATURE2.COV.01	Cover	Formal	100%

B. Merging FV and Simulation Coverage

When both FV and simulation methodologies are used to verify a design block, we need to build two separate testbenches---one simulation-based, one FV-based---then run separate regressions and close coverage in each environment respectively. Scripts can annotate coverage score per cover requirement onto the unified vplan.

IV. Results

In our project work, we applied the coverage-driven formal signoff methodology presented in this paper to two design blocks. The first block is a complex micro-sequencer design where we achieved the coverage-driven signoff criteria using FV only. The second block is a sequence-based design verified using constrained-random simulation approach, with a few control path sub-blocks verified in FV. In both cases, the proposed FV methodology demonstrates excellent ROI. In this section, ROI such as design specification improvement, quality of bugs and execution time improvement are presented based on the micro-sequencer FV experience. We also present ROI on merged FV/simulation coverage based on the other sequence-based block verification experience.

A. Design Specification Improvement

A corollary of FV being able to exhaustively prove a design property is it will also find all cases where the design property or specification is incomplete or incorrect. Therefore FV necessitates very accurate and complete design specification. In our experience with the micro-sequencer design, deployment of FV identified many ambiguities in the initial design specification. The process forced the designer to improve the specification to clarify the ambiguous requirements so the associated assertions can be proven exhaustively. Moreover, the FV-driven specification refinement process helped the designer identify several design and architecture issues even before the FV testbench was fully built. As a result, more than 70% (78 out of 104) of the requirements in the specification were improved and clarified.

One Example is illustrated below. The improvements to the specification are highlighted:

Original requirement: *The micro-sequencer engine shall continuously monitor the state of all input signals defined in the registers T_Array, and use the settings in the V_Array registers to determine which, if any, inputs are currently considered Active. If an input is considered Active, the V_PENDING field of the corresponding V_Array register shall be set.*

Improved requirement: *The micro-sequencer engine shall continuously monitor the state of all input signals defined in the registers T_Array, and use the settings in the V_Array registers* ***(<input_port_name>.v_array[i])*** *to determine which, if any, inputs are currently Active.* ***An input is "Active" when the conditions specified by V_COND ((<input_port_name>.v_array[i].v_cond) and V_BHV (<input_port_name>.v_array.v_bhv) are met (v_act[i] shall be pulsed in the same clock cycle).*** *If an input is Active, the V_PENDING field of the corresponding*

V_Array register ***(<input_port_name>.v_array[i].v_pending)*** *shall be set* ***in the next cycle if the microsequencer is enabled (seq_en = 1)****.*

In above example, the highlighted updates were missing details in the specification. Different parts of the missing specification were added at different points of the verification phase. For example, the definition of "***Active***" was clarified early in the TB coding stage, while conditions such as "***in the next cycle***" and "***if the microsequencer is enabled (seq_en = 1)***" were identified while debugging assertion failures.

B. Quality of Bugs Discovered

Besides specification refinements, we found and fixed in total 32 bugs in the micro-sequencer design, among which at least 5 were deemed as critical bugs that could be hard to find in dynamic simulation. One example of a critical bug was hidden behind two seemingly unrelated logic paths: 1) a path to update conditional flags such as Carry and Zero after executing an instruction, 2) a path to pre-fetch and decode instructions. It turned out that there was a very subtle linkage where the instruction decode logic could consult the Carry/Zero flags to determine whether the pre-fetched instruction should be executed. Due to a pipeline delay of Carry/Zero flags update, the pre-fetched instruction could be mistakenly nullified when a particular sequence of instruction was executed. This complicated corner case bug required a micro-architecture redesign to correct. Since FV was able to catch this bug early in the regression stage, we provided our design architect and RTL designer sufficient time to properly re-architect the design.

A normalized timeline of design bugs found by FV in the micro-sequencer is shown in Figure 4. From this chart, we can draw two interesting observations. The first observation is the overall bug-discovery rate using FV is faster than other designs within the same project verified with dynamic simulation. As shown in Figure 4, we compare the timeline of bug fix burn up (time spent to identify and fix all design bugs) between the micro-sequencer design and two other benchmark design blocks (BLK_A and BLK_B illustrated in Figure 4) with similar complexity. The benchmark designs are chosen based on number of requirements, flop count and line of code, as shown in table 4. Our results show that the micro-sequencer verified in FV achieved 100% bug fix milestone with over 30% less time. The second observation is that we initially found only 80% of the total design bugs by proving all of our assertions (with about 5% bounded proofs) coded based on requirements. After that we found the remaining 20% design bugs during coverage closure because the coverage-driven methodology helped us identify unexpected over-constraints and missing assertions (due to missing requirements).

TABLE 4

BENCHMARK DESIGN INFO FOR COMPARISON

DESIGN INFO	**MICRO-SEQUENCER DESIGN**	**BENCHMARK BLOCK_A**	**BENCHMARK BLOCK_B**
Number of Requirements	104	100	103
Flop count	15,918	6,934	3,612
Line of RTL code	~4,000	~16,000	~6,000

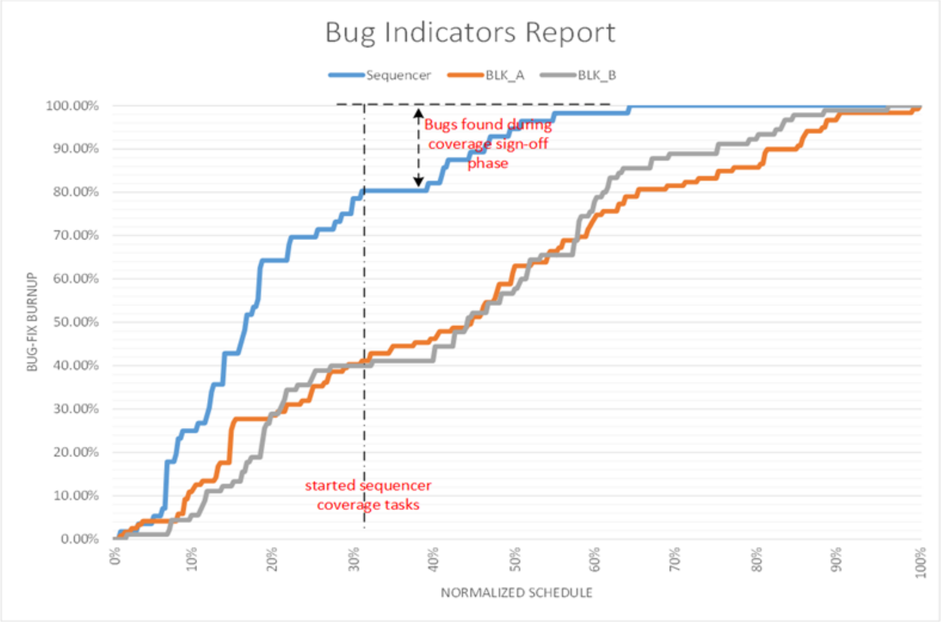

Figure 4. Design Bugs Tracking Report

C. Execution Time/Project Schedule Improvement

We also found overall execution time improvement using the FV signoff methodology. The micro-sequencer block was one of the first block to achieve verification signoff in the project, faster than many blocks of similar complexity verified using dynamic simulation. This is due to two factors:

1) FV testbench is in general simpler. Our micro-sequencer FV testbench is composed of 670 assertions and over 20,000 cover properties (most of them are auto-generated from covergroups), plus SV modeling logic and scoreboard. In total the testbench has about 4,000 lines of SV code. Compared to the other two benchmark designs, BLK_A's simulation testbench has 20,000 lines and BLK_A's simulation testbench has 13,400 lines.
2) Coverage closure is in general faster using FV. As shown in Figure 5, we compared the effort in terms of execution time to close coverage for the micro-sequencer using FV and the other two benchmark designs using simulation. Our result shows almost 2x execution time improvement for coverage closure using the FV signoff methodology.

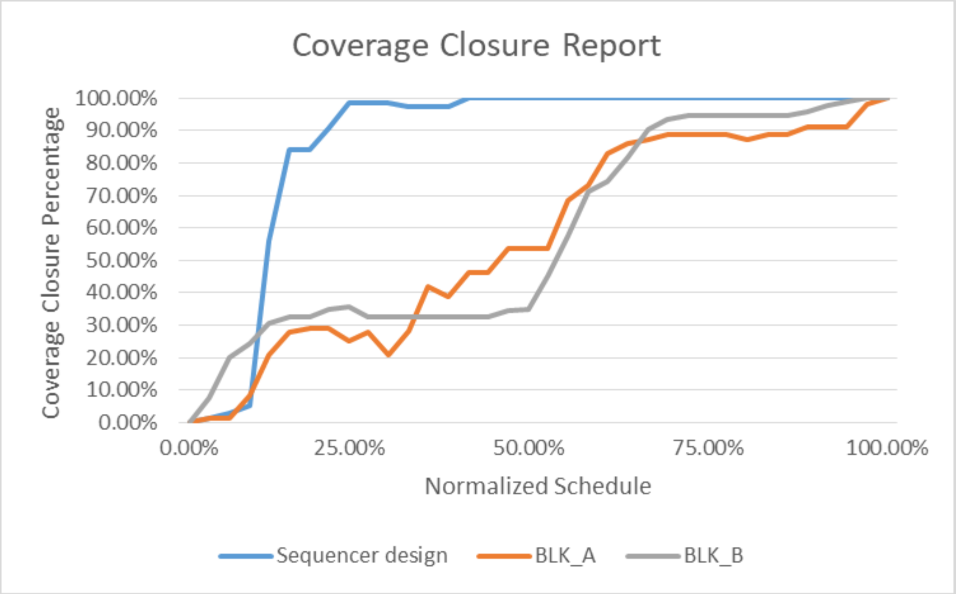

Figure 5. Coverage Closure Report

Table 5 summarizes the detailed metrics to demonstrate the values for using the FV for block-level verification signoff.

TABLE 5
ROI SUMMARY ON MICRO-SEQUENCER FV SIGN-OFF

QUANTITATIVE METRICS	FV SIGN-OFF ROI
TB lines of code	~4000
Specification changes driven	70% (78 out of 104)
# Functional cover points	100% (18737/18737)
# Code cover points	100% (3444/3444)
Time to 50% functional cov (percentage of overall execution time)	62%
Time to 75% functional cov (percentage of overall execution time)	75% (including above)
Time to 100% functional cov (percentage of overall execution time)	100% (including above)
% of meaningful cover traces within corresponding assert bounds	93% (~1300 traces outside of assert bound)
# Assertions	670
Checker completeness: COI	100%
Proof convergence	95%
Time to 1st bug (percentage of overall execution time)	20%
# Design bugs found	32
# Bugs found by full formal / semi-formal hunt	32 / 0
# "High-quality" bugs Hard for other method to catch	~5

D. Formal and Simulation Co-Verification

For the second sequence-based design, we used a feature-based FV/simulation co-verification approach to sign off. Because this design has both logic with deep sequential depth as well as control logic with high concurrency, we chose simulation to verify the former logic and FV for the latter. This co-verification approach contains 3 major steps:

1) As described in section III, we first created over 400 methodology agnostic feature requirements in the vplan. Then based on the characteristics of each feature requirement, we selected the best methodology to solve the problem.
2) We created a simulation testbench and a FV testbench in parallel to implement respective generate/check/cover requirements. Simulation and FV regressions were run separately and the coverage results were analyzed using respective tools.
3) We used both commercial and in-house tools to calculate coverage score per cover requirement, and annotated the scores onto the vplan, as depicted in Table 3. The merged coverage results were released periodically to our coverage reporting system to track verification progress.

With this approach, we were able to parallelize the verification task to shorten project schedule and achieved higher confidence in the completeness of verification.

V. Conclusions and Future Work

Despite the increasing adoption of formal verification in the industry, using FV only for verification sign off of a design is still intimidating to many verification teams. Part of the barrier is not having a common language to describe and measure verification progress between FV and dynamic simulation to give confidence to verification engineers and managers that FV can achieve equivalent, and frequently better, verification coverage and outcome.

This paper proposed a coverage-driven formal verification methodology that is deterministic, measurable, and repeatable. The methodology collects functional (and other types of) coverage in an FV environment so it can be measured and used as an objective metric to track verification progress and declare signoff. It enables more seamless integration and co-verification of FV and dynamic simulation with merged coverage results. It also enhances confidence of FV work by using functional coverage to avoid environment over-constraint.

By applying the proposed methodology, we demonstrated significant ROI on our project in four key areas: 1) design specification improvement, 2) quality of bugs discovered, 3) measurement of FV progress, 4) execution time improvement. This pilot project was successful in convincing leaders in our team to adopt FV more widely because they are able to better understand and track FV work, and completely changed the view of how FV fits into our overall verification strategy. We intend to use this as our reference FV methodology, and apply it to more suitable designs in the future.

Acknowledgment

We would like to thank Dan Benua, Yumi Monma, and Chris Komar from Cadence Design Systems for their support during the development of our FV signoff flow.

References

[1] Erik Seligman, Tom Schubert, and M V A. Kiran Kumar, "Formal Verification, An Essential Toolkit for Modern VLSI Design," Morgan Kaufmann, 2015

[2] Douglas L. Perry, Harry Foster, "Applied Formal Verification," McGraw-Hill, 2005

[3] Anish Mathew, "Coverage-Driven Formal Verification Signoff on CCIX Design", JUG 2017.

[4] N. Kim, J. Park, H. Singh, V. Singhal, "Sign-off with Bounded Formal Verification Proofs", DVCon 2014.

[5] I.Tripathi, A. Saxena, A. Verma, P. Aggarwal, "The Process and Proof for Formal Sign-off: A Live Case Study", DVCon 2016.

[6] B.Wang, X. Chen, "Coverage Driven Signoff with Formal Verification on Power Management IPs", DVCon 2016.

[7] M A. Kiran Kumar, E. Seligman, A. Gupta, S. Bindumadhava, A. Bharadwaj, "Making Formal Property Verification Mainstream: An Intel Graphics Experience", DVCon 2017.

[8] X. Feng, X. Chen, A. Muchandikar, "Coverage Models for Formal Verification", DVCon 2017.

[9] J. R. Maas, N. Regmi, A. Kulkarni, K. Palaniswami, "End to End Formal Verification Strategies for IP Verification", DVCon 2017.

[10] M. Munishwar, X. Chen, A. Saha, S. Jana, "Fast Track Formal Verification Signoff", DVCon 2017.