Coverage Driven Signoff with Formal Verification on Power Management IPs

Baosheng Wang
Advanced Micro Devices, Inc.
1 AMD Place
Sunnyvale, CA 94088

Xiaolin Chen
Synopsys, Inc.
690 East Middlefield Rd.
Mountain View, CA 94043

***Abstract*- Identifying the right balance between simulation and formal verification has always been a challenge. We have learned that in some cases where formal verification is the only viable approach to verify critical features in SoCs because they require exhaustive coverage. These critical features included reset, clock, fuse, power management controller, and so on. Using a couple of power management IPs as an example, this paper describes our experience working with formal technology from verification planning, scoping, to signing off on multiple power management blocks completely. The tools we used are VC Formal and Certitude from Synopsys. We feel the outcome is very valuable and it leads us to establish the flow in the verification process across projects deploying formal verification to fully validate all possible reset states and their potential transition scenarios. Coverage measurements are used as the guidance throughout the flow from beginning to signoff. We hope you can benefit from what we have learned in our experience.**

I. Introduction

In today's verification environments, formal verification is playing an increasingly important role in the overall improvement of the design quality and verification efficiency. However, identifying the right balance between simulation and formal verification has always been a challenge. At AMD, we have learned that in some cases where formal verification is the only viable approach to verify critical features in SoCs because they require exhaustive coverage. These critical features included reset, clock, fuse, power management controller, and so on. Using a couple of power management IPs as examples, this paper describes our experience working with formal technology from verification planning, scoping, to signing off on multiple power management blocks completely. The outcome is so valuable that it leads us to establish the flow in the verification process across projects deploying formal verification to fully validate all possible reset states and their potential transition scenarios. Coverage measurements are used as the guidance throughout the flow from beginning to signoff. The results from this paper were obtained using VC Formal and Certitude from Synopsys.

II. Verifying Power Management IPs

A. *The Designs*

We will describe a couple of power management IPs in this section.

The first IP is a power management interrupt controller IP. This IP is a block that can be added to arbitrate multiple interrupt requests and then send the final request to external system through AXI4 interface.

The second IP we will describe in more details in this paper is the reset IP block. To illustrate the formal verification process, a simplified state diagram with its legal states and transitions is shown in Figure 1.

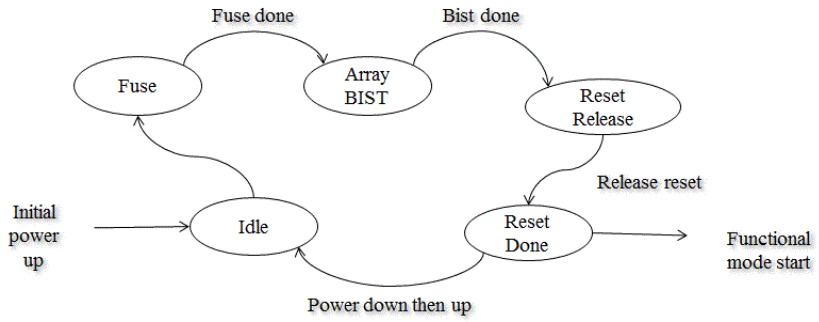

Figure 1. Power management IP state machine diagram

The reset IP is a block that can be added to any unit that controls the reset and initialization sequences in a CPU core or wherever a clock generator or clock controller block is used. It is also used to facilitate the initialization and run of DFT tests.

The verification goal for this IP is to check that only legal states and transitions are permitted and all others should be forbidden. Otherwise it can lead to catastrophic device failures. The power management IP is not a very big block, but it controls the overall operations before it can function. We must ensure the quality with exhaustive verification coverage.

B. Verification flow from yesterday

Before we introduced formal flow, power management IPs were verified at the SoC level using simulation as shown in Figure 2. It is difficult to control and debug, and impossible to verify thoroughly. Corner cases that might have catastrophic effect could have been missed.

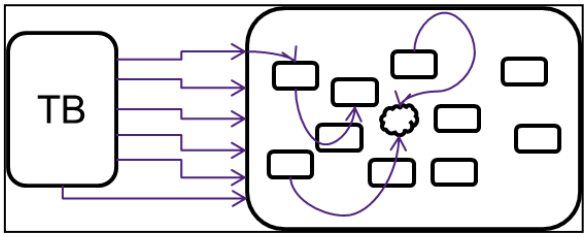

Figure 2. Simulation based power management IP verification

C. Power management IP design statistics

Formal design statistics of this power management IP mainly contained 15 input ports, one Finite State Machine (FSM), 155 sequential elements and quite a few output control signals. TABLE I shows more detailed information of the design statistics.

TABLE I
RESET IP DESIGN STATISTICS

Design structure	Counters	2
	Complex operators	2
End points	Primary inputs	15
	Constants	2
Clock/Resets	Primary defined clocks	1
	Primary defined resets	1
	Missing defined clocks	1
Sequentials	Flipflops	236
	Latches	5
Misc	Undriven nets	9
Property complexity	Flipflops	856

In this design, each FSM state is responsible for triggering the start and monitoring the end of specific task(s). About one-third of the sequential elements are asynchronous set-reset flip flops and verification of those asynchronous behaviors is generally more complex.

We selected IPs like this one as the targets for formal analysis not only because it is not possible to verify exhaustively at system level, but also because of the size of the block is small, therefore it may be possible to completely verify the block without stretching the formal capability limit. This way, we can reduce the need for a formal expert due to proof convergence problems. Through well-defined process and guidance from coverage metrics, we can utilize verification resources without prior experience using formal technology to complete the verification task.

As another reference point, TABLE II shows the design statistics for the power management IP design.

TABLE II
POWER MANAGEMENT IP DESIGN STATISTICS

Design Structure	Complex operators	1
End points	Primary inputs	31
	Constants	2
Clock/Resets	Primary defined clocks	1
Sequentials	Flipflops	132
	Latches	26
Property complexity	Flipflops	424
	Complex operators	1

III. FORMAL VERIFICATION FLOW

We have been using formal verification technology in our verification flow for many years. Although most formal tools have helped us solve many verification problems and locate corner-case bugs, we found that they lack signoff measurement utilities. With new tool features developed at Synopsys [1], e.g., automatic FSM state and transition checking, line, condition and toggle coverage report, livelock and deadlock verification, etc., we were able to create a formal verification flow that was completely coverage driven, similar to that of simulation verification.

Those formal coverage utilities enable us to identify holes in both functional coverage and structural coverage so that we can quantitatively measure the completeness of functional formal verification. We can use these metrics to measure progress and eventually sign off when the coverage items are reviewed and targets are reached, just like in simulation verification process.

We refined formal verification flow using the formal coverage closure utilities, as shown in the Figure. 3.

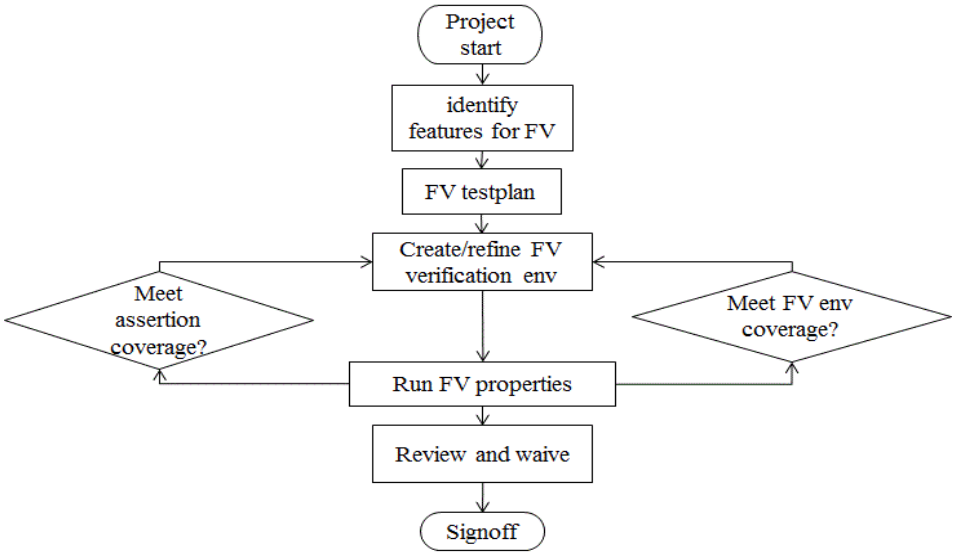

Figure 3. Formal verification flow chart

C. Verification Plan

We allocated one dedicated junior engineer who was a new college graduate to verify this IP. The verification task was required to complete in three weeks. We chose Verification Compiler (VC) formal tool from Synopsys to complete our IP-level verification. Since those assertions would be integrated along with the IP itself at upper-level block for both simulation and emulation, we tried to write assertions in OVL format as much as possible.

The first step of developing formal verification plan was to declare the key verification points. For this particular reset IP, the list of key verification points were:

- Verify the correct functioning of the FSM
- Verify deadlock, uncoverable states, invalid/valid transitions for the FSM

- Verify that no X's exist on the inputs/outputs
- Verify that the reset IP stays in the same state once the "GO" flag for that state is asserted until the "DONE" flag is asserted.
- Verify that the reset IP pauses in the desired state when "pause" signal is asserted during silicon debug
- Verify that the “Ready” signal is asserted once the reset completes
- Verify that the outputs from the reset IP is generated properly as expected.

Once the key verification points were identified, the next step was to define the verification strategies, such as modeling tricks, reset methods, constraint development and coverage closure. As we mentioned earlier, in this case, since the power management IP was small in size regarding the number of sequential elements, no formal modeling techniques were required. Figure 4 describes the work flow for our formal verification of this IP.

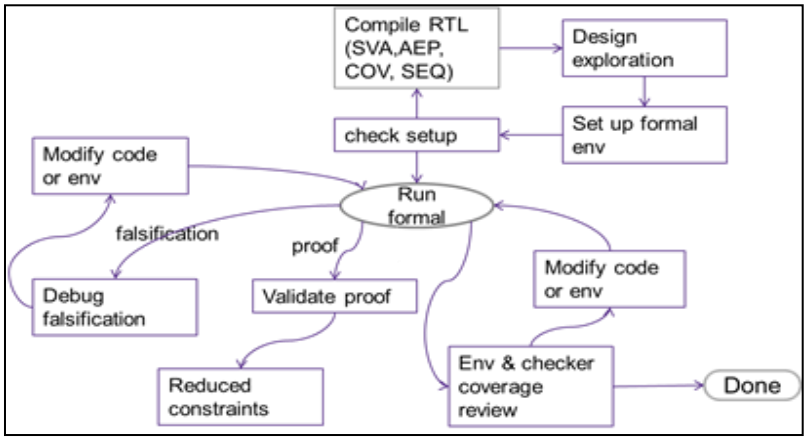

Figure 4. Formal verification work flow

Formal coverage closure mentioned here includes functional cover properties, code coverage and assertion coverage. The details of formal coverage closure will be discussed in the next several sections.

D. Develop Formal Verification Environment

Since we had a clearly defined verification plan and work flow, it was easy for the verification engineer to follow and get started. The following sections described the process.

1) Construct OVL/SVA checkers

We started out with writing OVL/SVA assertions to describe legal/illegal states and valid/invalid state transitions. Next, we developed dedicated assertions for each output. Lastly, we created several functional cover properties to check for deadlock states.

In total, 27 OVL/SVA assertions and 15 cover properties were created to cover all the functionalities described in the specification and test plan. A few examples of the assertions are shown in Figure. 5.

Debugging of properties was straightforward since we could ask the tool to provide a witness trace of the property in a waveform so we could visually inspect the behavior of the properties during the development.

```
//IP_FuseSel goes high once the IP enters the fuse statefusesel_chk:
assert property (@(posedge CLK) disable iff (disable_chk)
((!BypassFuse && !DftWarmReset) ##1 (!$stable(IP_RstState) &&
(IP_RstState == `IP_FUSE) && !BypassFuse && !DftWarmReset
&& !WarmResetFlag)) |=> (IP_FuseSel));

//IP_BistGo goes high once the IP enters the bist statebistgo_chk:
assert property (@(posedge CLK) disable iff (disable_chk)
(!$stable(IP_RstState) && (IP_RstState == `IP_BIST)) |=> (IP_BistGo));
```

Figure 5. SVA assertions

2) Set up initial state for formal

When the assertions were completed, we were ready to fire up the formal run. The first step was to set up the Device Under Verification (DUV) to the correct initial state. The reset requirements included two parts:

i) Sequential element reset using IP-level reset control signals;

ii) Certain flops requiring to be initialized with dedicated reset values before formal reset, i.e., right after power up. The second portion of the reset required special feature support from vendor tools.

For example, the VC formal TCL command to reset flops during "reset" phase is:

```
fvassume –expr { expressions } –reset
```

There were other mechanisms that allowed us to do more manipulation of the initial state. Debugging of reset states was also straightforward either using waveform or querying the reset value using TCL commands.

3) Incrementally add constraints

With the correct initial condition, some of the properties still falsified. As formal engines identified failure scenarios and presented in replay waveforms, we debugged the failures and realized that additional constraints were required. Constraints or assumptions were developed manually by the verification engineer. This process of debugging and refinement of the properties and constraints was iterated over and over again until we felt the minimum constraints required to verify the full functionality were complete.

During the formal prove process, we developed 11 SVA constraint properties. Some constraint examples are in Figure. 6.

```
assume2 assume property (
  @(posedge CCLK) disable iff (disable_chk) ((IP_RstState==`FUSE ||
  (IP_RstState==`IP_WAIT)) |-> (!Reset));

//BistDone doesn't assert in any state before BIST state
assume3 assume property (
  @(posedge CCLK) disable iff (disable_chk) ((IP_RstState==`IP_BIST|
   && BistDone)) |=> (BistDone));
```

Figure 6. SVA constraints

With the qualified constraints in place, we were able to prove all assertions and cover properties in about 15 minutes.

Later in the project, the formal tool we were using was enhanced to have a capability of automatically extracting FSM from RTL and creating coverage goals for all FSM states and transitions in our formal environment. Another enhancement was the auto-generation of livelock and deadlock goals for both toggle and FSMs.

We find these two features very beneficial in terms of time/effort saving since we no longer need to write SVA assertions to start for the complete FSM verification. The new features not only enabled us start running the tool and reviewing results right away, but also demonstrated corner scenarios and forced us to scrutinize more thoroughly for both the IP design and our formal environment.

E. Measure progress by coverage

As is with all formal verifications, there are three questions one must answer to throughout the verification process in order to determine when to sign off:

- "Have I written enough assertions?"
- "Have I over-constrained my formal TB"?
- "How effective are my checkers at catching design bugs?"

Similar to simulation coverage signoff, we also established coverage signoff process for formal verification applications. This includes the following:

- Cover properties and assertions to ensure all functional points in the specification are being verified
- Coverage measurement to check for missing assertions
- Coverage measurement to check for over-constraint scenarios

The first coverage goal is straightforward. We just need to make sure we have verified what is in the functional specification. In the following sub sections, we will discuss in detail how we achieve the last two coverage requirements.

1) Coverage to check for missing assertions

The answer to the first question "How do we know if we have written enough assertions?" is the measurement of assertion coverage.

We find that the "Cone Of Influence" (COI) logic analysis feature beneficial. As illustrated in Figure 7, the colored portion of the logic is in the COI of properties. We know that a property cannot possibly verify any logic outside of the transitive fan-in of the signals it is monitoring. By computing the union of the fan-in logic for all of our properties, the tool can determine if there is any logic that is not in the cone of influence of any property. With the feedback provided by the tool, we measure assertion coverage by reviewing the area not covered by any

assertions at all. If there is, we have a coverage hole and more properties maybe needed, or we review and waive these uncovered goals.

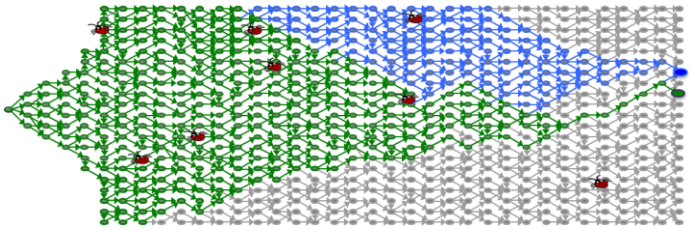

Figure 7. Cone Of Influence logic of properties

It is simple to ask the tool to provide a list of elements outside of COI of any properties. The VC Formal commands used to analyze "cone of influence" coverage are:

analyze_fv_coverage –hierdepth 100
report_fv_coverage –hierdepth 100 –list_uncovered

With these two TCL commands, we were able to identify the list of flops never covered by any assertions. Our verification engineer deemed this very useful. Initially, he was very confident that all flops were covered. However, after running these two commands, the tool still identified one flop that is never covered by any assertions. In this case, it was a duplicated flop for timing purpose and we did not need assertions associated with this flop. At least the tool brought this to our attention for review.

We do understand that the COI analysis is an over-approximation of the property coverage. We will need more in-depth analysis to look for potential coverage holes. There are more advanced technologies that will provide more detailed analysis. We choose to run this coverage measurement because it is lightweight and quick, but can provide useful feedback in the beginning.

A more accurate way to measure assertion coverage is the measurement of the proof COI coverage. We did not have a chance to explore the usage of proof COI coverage yet. We may consider doing this in future projects.

2) Coverage to check for over-constraining

The second question we would like an answer to is "How good is my FV environment?"

This is done similar to the simulation flow. In addition to cover properties to check for essential functional events, our plan is the use of line and toggle coverage metrics to measure the quality of the formal environment.

There are two aspects when checking for over-constraining. The first aspect is to check for any uncoverable line and toggle items without any constraints to identify any deadcode. Any of the deadcode will be communicated to the IP designer for review. The second aspect is to check for additional uncoverable line and toggle items after all the constraints are enabled in the setup.

TABLE III showed an image of a list of uncoverable goals from this step. Filtering out intentional uncoverable because reset or redundant default conditions, we analyzed the rest of the items to look for the cause of the goal not coverable.

TABLE III
VIEW OF UNREACHABLE COVERAGE GOALS DUE TO CONSTRAINTS

toggle		q[0]	0->1		srdff__GRS_LongReset
toggle		srdff_q[0]	0->1		srdff__GRS_LongReset
toggle		q[0]	1->0		srdff__GlbRstStateDone_d2
toggle		srdff_q[0]	1->0		srdff__GlbRstStateDone_d2
line	IF			UNINT_UNR_INT_RCH	srdff__GlbRstState_d1[0]
line	IF			UNINT_UNR_INT_RCH	srdff__GlbRstState_d1[0]
line	IF			UNINT_UNR_INT_RCH	srdff__GlbRstState_d1[1]
line	IF			UNINT_UNR_INT_RCH	srdff__GlbRstState_d1[1]

Overall, VC formal reported 736 covered lines and toggle goals, 70 uncovered lines and 56 instances of un-toggled. Figure 8 showed the summary results of the coverage of line, toggle, as well as livelock/deadlock checks. One obvious discovery during our review is that formal tool reported the default items of case statements are uncoverable. This is expected since those x-uglified statements are useful for simulation purpose only and our Vender provides a switch "UNINT_UNR_INT_RCH" to remove those lines from our review list.

We reviewed all the uncoverable items reported by the tool, as well as the falsified toggle deadlock properties. Some were set by design, and some were due to required input constraints. We tagged all of these items to be

waived for future regressions. Although we did not find any verification holes, it increased our confidences in our environment since this completely eliminated any possibilities of over-constraints. Moreover, we can use this exercise to identify any redundant constraints that are never used.

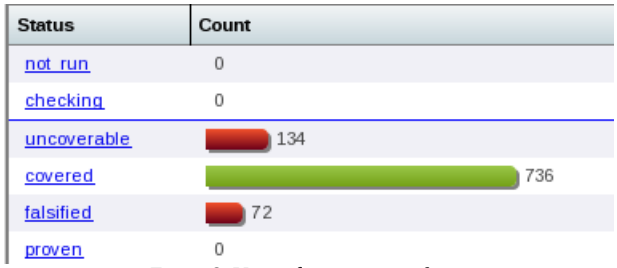

Figure 8. View of coverage results

3) Coverage through fault analysis

The third question we have to answer to was "How effective are those assertions?"

Our EDA vendor offers a tool called Certitude that can automatically insert bugs in the design. It does so by including the mutated design with functional faults injected to our formal verification environment, and then checking to see if the existing properties can detect those bugs. This process is very resource intensive, because many properties need to be checked against many different bug scenarios. In our case, since the block is small, and we feel that the detailed information of fault analysis produced is extremely high, therefore we proceeded to go through the fault analysis flow.

In addition to prevent the useless properties from on the bugger design, VC Formal working with Certitude [2-3] can also identify specific design corruptions that cannot be detected by any property. This information further guided us to create additional checkers.

Certitude injected 468 functional faults in our IP and 454 faults are analyzed after removing redundant faults. With the most basic fault injection, we found there were no non-activated faults, which was consistent with our findings of the assertion coverage results in section 1). However, the tool also reported 6 non-detected faults from output connectivity; 3 non-detected faults from combinational logic control flow, as well as 18 faults non-detected from combinational logic. The results are shown in Figure. 9.

Class Name	Faults In Design	Non-Activated	Non-Propagated	Detected	Non-Detected	Disabled By Certitude	Disabled By User	Dropped	Not Yet Qualified
TopOutputsConnectivity	24	0	0	12	6	0	0	6	0
ResetConditionTrue	0	0	0	0	0	0	0	0	0
InternalConnectivity	0	0	0	0	0	0	0	0	0
SynchronousControlFlow	0	0	0	0	0	0	0	0	0
SynchronousDeadAssign	0	0	0	0	0	0	0	0	0
ComboLogicControlFlow	60	0	0	7	3	10	0	40	0
SynchronousLogic	0	0	0	0	0	0	0	0	0
ComboLogic	384	0	0	43	18	28	0	295	0
OtherFaults	0	0	0	0	0	0	0	0	0
All Fault Classes (9)	**468**	**0**	**0**	**62**	**27**	**38**	**0**	**341**	**0**

Figure 9. View of fault coverage analysis

All the non-detected faults pointed out to us that where there were no assertions at all to catch these 27 fault scenarios, i.e. all the assertions were passing with these faults in the RTL. This information is very important. It forced us to have discussion with the architect team on the ambiguity or missing specification. It also made us be more thorough and guided us to add the missing properties to make sure that these faults were caught. Information of such detail is extremely valuable in ensuring the quality of the formal verification environment, giving us the confidence we need to be able to signoff the design.

We would like to see some enhancements to this flow. For example, import the waivers already done from previous steps into Certitude. Otherwise, duplicated efforts were spent in analyze the same faults previously waived. In general, better integration of the formal and fault coverage will enhance the usability as well as performance.

F. Review and waive

As defined in the test plan discussion, we collaborate with the IP RTL designer as part of review. We also check in our waivers along with our assertions so that our regression flow can automatically waive them without user interference. Any coverage regression failures indicate there are RTL changes happening as we do detect this through a coverage miss when the IP designer re-organizes the design for better timing.

G. Validate Assumptions

It is important that we validate these assumptions used by formal setup in the simulation to ensure correctness. All the constraint properties were used as checkers in simulation at upper level of designs where the reset IP is instantiated. This will ensure that the constraints created for formal are not over-constraining the environment, otherwise formal proofs could be invalidated and bugs could escape.

H. Sign off

We submit all verification metrics to our verification database for signoff. Our verification managers periodically review those metrics to make sure they meet project milestones and tape out quality requirements.

I. Summary

Through well-defined flow and process, we were able to deploy formal technology in our verification projects without having to rely on specialized formal teams. Project teams were able to execute the verification tasks with engineers who had little or no prior experience with formal technology. We see this as a launching vehicle to develop new formal talents within the project team so they can take on more complicated formal problems.

IV. Results

Through formal verification, we have reported 9 RTL bugs on the reset IP and 14 bugs on the power management interrupt controller IP. The whole process from start to finish took 3 weeks for a brand new fresh graduate knowing nothing about formal and fault analysis to complete verification. Later on, we extended this flow to many other power management IP blocks and 50 plus RTL bugs were reported.

V. Conclusion

Before we employed formal, the reset functionality was done in simulation at very high level, it was difficult to control and debug. Corner case bugs could be missed. We were able to completely verify the functionality with formal and sign off with confidence in three weeks, using just one junior engineer.

The conclusion from our experience by using formal technology as a signoff toolset has proven to be beneficial in the following aspects:

1. High confidence in IP quality: because of the exhaustive nature of the technology, formal helped finding corner case bugs that is impossible to be caught by simulation.
2. Reduced verification cycle: the IP block is completely verified using formal, replacing the need for simulation.
3. Efficiency in resource planning: reduced overall requirement for verification resources as this freed up the simulation team as well as the computing resources.
4. Coverage closure deployed in formal verification environments for the first time: this enabled us to measure the quality and quantity of the formal verification environment and gave us the confidence to sign off on functionalities of the blocks.

Acknowledgment

We would like to thank Praveen Tiwari, Vijay Korthikanti, Khalid Siddiqi from Synopsys for their support during the initial set up phase of the first testcase.

References

[1] Synopsys, *Verification Static Formal User Manual*, 2015.09.

[2] N. Kim, Junhyuk P., HarGovind S. V. Singhal, Sign-off with Bounded Formal Verification Proofs, *DVCon*, 2014.

[3] Synopsys, *Certitude User Manual*, 2014.12