2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Connectivity and Beyond

Shahid Ikram[1], Joseph DErrico[1], Yasmin Farhan[1], Jim Ellis[1], Tushar Parikh[2]
[1]Marvell Semiconductor, Inc.
600 Nickerson Road,
Marlborough, MA 01752
[2]Synopsys
690 Middlefield Road,
Mountain View, CA 94043

***Abstract*-This paper presents an innovative workflow to deploy connectivity tools in various phases of SOC design. The process starts with designers creating connectivity specifications at the full-chip and partition level. These specifications are used to auto-generate connectivity checks on the evolving RTL (register transfer language). A weekly regression test-suite based on formal tools ensures that as chip design evolves, the connectivity remains intact. Furthermore, the workflow also verifies the completeness of the connectivity specification through fault injection verification. Next, the formal connectivity results are used to generate toggle coverage. This saves time during the integration of blocks to the full-chip closure. Finally, we process these specifications to generate higher level connectivity checks. These checks include circularity absence, one-to-many, many-to-one, and many-to-many connections. The designers review these derived high-level checks for any unexpected surprises. These high-level specifications are verified using Static tools. A number of bugs were found and fixed. The flow is now a regular part of our SOC-design process.**

I. Introduction

The complexity of SOCs built today is far beyond human intellectual capabilities. The only hope to create a predictable and reliable system is to keep building new tools and flows. Any tool flow that can ensure our understanding of the system and saves the time to closure adds great value to the SOC design process. All industrial strength SOCs are built using an integration of a well-defined set of blocks. Some of these blocks are IPs from other vendors and some of these are homegrown. One key aspect of the integration of these components is their connectivity. At the simplest level, connectivity describes connections between two RTL ports but there is more to it than meets the eye. A typical SOC contains many clocks, reset, and power domains [8] as shown in Figure 1.

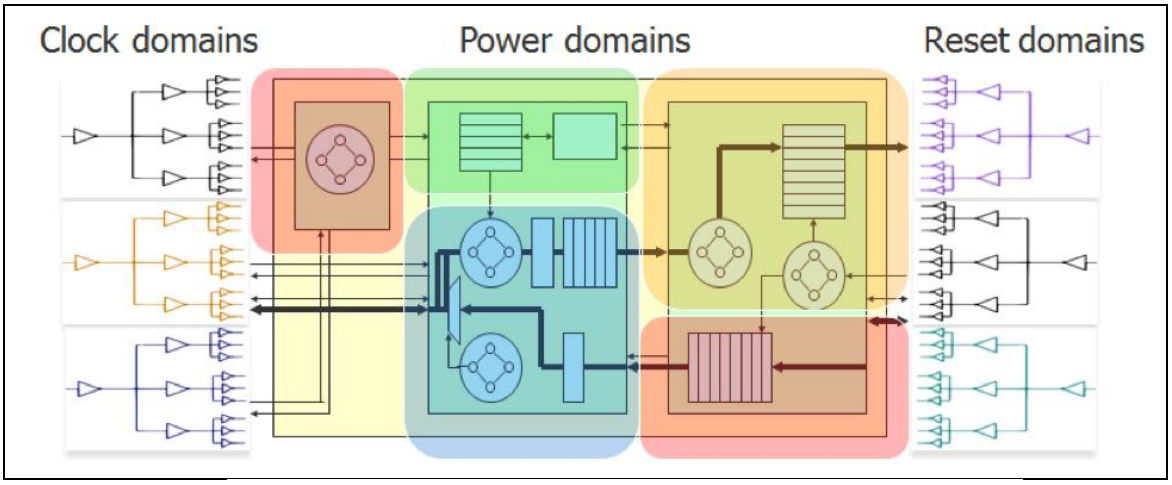

Figure 1: Source: **Hardware and Software: Verification and Testing, 2015**

Consider the case of a reset system. A reset signal is applied to multiple flops and hence a reset system design requires a one-to-many connectivity specification. Moreover, there are multiple resets like cold, warm, and soft [8]. There may be multiple reset domains as well. We may need to ensure that the exclusivity property stating that "a *reset is connected to the proper reset ports only and nothing else*" is satisfied. The verification of these (and other similar)

facts is impossible in simulation and hard in formal verification as well. As a result, this effort started with this simple question: can we ensure that reset signals are properly connected? This question spawned multiple sub-questions like:

1. All the reset signals are connected?
2. The reset signals are not connected to non-reset ports?
3. What kind of one-to-many relationship does the SOC have for reset signals?

The RTL designers generated connectivity specifications. Several tools were tried in order to overcome the various challenges. A comprehensive flow was created that combined a set of homegrown tools and selected vendor tools.

We will start with an introduction to the connectivity specifications. It will be followed by a short description of the tools tried and used. Next, we will provide a description of the various stages of the flow. Finally, we will present our results.

II. Formal Specification

A. *SOC connectivity*

The development of a complete connectivity description of any SOC is a huge undertaking. There are multiple IP blocks with potentially different owners. Each block has tens, if not hundreds, of ports to support. In our case, the smallest partition had around 250 ports and the largest one had around 2500 ports. Luckily, the designers' team already had a method in place, where every block owner maintains his/her blocks port-list using YAML [5]. The YAML specifications are rolled into a system-wide generation of RTL shells, with everyone involved in the development of the SOC using these shells. The problem with this method, however, is that these YAML specifications are manually managed. A missing signal will be spotted right away through the compilation process but a redundant signal may lurk around forever. Moreover, a bad connection, like the wrong reset, may go undetected.

We collaborated with the designers' team to generate formal connectivity specifications from this system. The formal specification format chosen was a comma-separated connectivity description [1] as shown in Figure 2. Each line in the table specifies a one-to-one connection between a source and destination signal with optional more information about clocks, enables, delays, etc. The designers' team provided us with one specification table for each partition of the SOC, enlisting all the connections, one per line.

```
Name,Clock,Enable,Enable_hold,Source,Destination,Path_delay
hier1_in1_to_hier2_in1,clk,~hier1.sel&en&c_en,2,hier1.in2,hier1.inst.in1,0
hier1_in2_to_hier2_in1,clk,hier1.sel&en&c_en,2,hier1.in2,hier1.inst.in1,1
hier2_outv_to_hier1_outv,clk,hier1.en,1,hier1.inst.outv,hier1.outv,1
```

Figure 2: CSV Connectivity Specification

The job of the formal team was to create a tool flow that could automate the connectivity verification. The effort includes verification of the RTL against this specification, as well as validation of the specification itself.

B. *The Connectivity Checks*

Generally, a connectivity check has two parts [1]:

1) *Structural Check*: It checks for the structural path from the source to the destination. A structural path exists if:
   a. There is a physical directional path from the source to the destination.
   b. The bit width of the source and the destination is equal.
   c. The number of flip-flops between the source and the destination is exactly equal to the specified delay.
2) *Functional Check*: It checks if the source and the destination ports behave in a coherent way. The following SVA[4] assertion captures this check:

```
assert property (@(posedge clk) enable_expr[*hold_time] |-> dest == $past(src, path_delay);
```

The formal connectivity application tools infer these checks from the table format shown in Figure 2 and no explicit conversion is needed. The whole process of specification generation and validation can be run with weekly regression checks.

C. The Derived Connectivity Checks

We mined the connectivity specification tables to discover interesting patterns. The captured patterns were used to derive high-level connectivity specifications. It enabled the extension of the connectivity checkers portfolio in more than one way. The list and definitions of the checks added to the connectivity verification flow based upon pattern identification is as follows:

1) *One-to-One*: A one-to-one connectivity is an enhancement of the simple connectivity check. These are mostly intra-module/block connections. It checks:
   a) A Structural path between the source and destination.
   b) Functional coherence between the source and destination as described above.
   c) That the source and destination are not connected to any other port.

A formal connectivity tool can verify the first two components. The checking for the third is non-trivial and cannot be accomplished by the VC Formal Connectivity Application. Our research to this end led us to two possible options:

   a) We can use VC Formal Security Verification (FSV) [6]. FSV defines and verifies data propagation absence among certain pins (hence no connection). The following property specifies that there is no direct connection from the input jtag to the output cold_reset:

```
fsv_generate -name jtag_cold -src jtag  -dest cold_reset
```

   b) We can use a static verification tool like SpyGlass [2] to show absent connectivity among certain ports.

In our work we tried both tools. The following sample SpyGlass property specifies connection absence among certain ports:

```
Illegal_path –from {“alpha”} –except_to {“beta”}
```

This property specifies that the port alpha is connected only to beta and nothing else.

2) *One-to-Many*: Not all the connections in an SOC are one-to-one. There are connections such as clocks, reset that require one-to-many specification. These kinds of specification check for:
   a) A structural path between the source and destinations.
   b) The functional coherence between the source and destinations.
   c) The source is not connected to any other port but the specified ones.

A sample SpyGlass check for the reset connectivity is shown below:

```
illegal_path -from {“Reset_Driver.resets"} -except_to {"*resets“}
```

It verifies that the reset driver is connected only to reset ports and nothing else.

3) *Many-to-One*: SOCs also have bus-like signals that require many-to-one specification. To specify a bus connectivity, we will check that the bus-drivers are only driving the bus and nothing else. A sample property capturing this notion is shown below:

```
Illegal_path –from {“*bus_drivers”} –except_to {“bus”}
```

4) *Many-to-Many*: We have found many-to-many connections as well though it was rare. Again, a SpyGlass check can capture this notion as shown below:

5)

```
Illegal_path –from {“a”,”b”,”c”} –except_to {“d”,”e”,”f”}
```

All these checks are automatically derived from our specification tables. We wrote Python scripts that processed the specification tables and generated a list of checks in a format that would allow tools like SpyGlass and FSV to then process these checks.

D. User-Defined Connectivity Checks

These are the checks that cannot be derived from general connectivity specifications, and are obtained instead by asking designers to specify any of their special concerns. One of the very successful applications for this scenario was SOC integration checking. SOC designs are built by dividing the design into partitions and blocks. Near the end of the design cycle, these blocks are integrated into full SOC. One of the biggest challenges is to make sure all the block-level ports are connected to full-chip level ports. Generally, verifiers write signal-toggling tests to verify these connections. Given that there are thousands of signals, it is an extremely tedious task. By using connectivity specifications, however, this task may become trivial and allow efforts to shift to formal proof convergence.

Here a few other example cases where user-defined connectivity can potentially be highly effective [1]:

1) SOC integration checking.
2) No interference among the virtual channels.
3) No data propagation from any primary input to any primary output.
4) No data propagation from any register to the rd_data output.

III. SPECIFICATION'S VALIDATION

An important aspect of any specification is its quality. For instance, incompleteness, redundancies, or undesired patterns all negatively impact quality. We devised a number of checks to validate our specifications.

A. *Undesired patterns*

The generated one-to-one, one-to-many, many-to-one, and many-to-many checks are the patterns present in the connectivity specifications. A review of these patterns by the designers' team is highly desirable and recommended. Furthermore, the designers' teams may suggest some modification of these patterns for future regressions.

B. *Completeness*

The sources of the specifications are YAML files manually created by the designers and hence prone to human error, with potential redundant ports as well as missing specifications. We therefore need to validate that the generated specification is completely covering all the ports of the RTL. The two options at our disposal to carry out this validation are the toggle coverage and the mutation analysis methods [7]. Toggle coverage is a simple but a lengthy process to check that all the signals are connected. It verifies if the toggling of the source signal of a connection toggles the destination signal and hence not stuck at zero or stuck at one. Toggle coverage does not check if the destination signal is inverted, delayed by a wrong value, enabled properly, etc.

The mutation analysis methods are fault-injection-based testing techniques that introduce small changes into a program or specification to check the quality of the testing procedures. The types of changes introduced in the programs are called faults. The types of faults depend upon the application domain. In the case of SOC RTL, these changes can be stuck-at faults, operation swapping like replacing & operator with |, inversion, etc. A mutation analysis tool introduces these changes to the program systematically one-by-one and validates that if there is a test in the test-suite that captures the change (artificial fault) and fails. If every artificially introduced fault is captured by some test in the test-suite, then the program is in good shape in terms of verification quality.

In our scenario, the test cases are the connectivity specifications, and the program is SOC RTL. The fault types chosen are *stuck-at-0*, *stuck-at-1*, and *inversion*. We generate SVA checkers [4] from the connectivity specifications. These checkers encompass all the connectivity related information. We bind these checkers with RTL and run the two through a mutation analysis tool called VC Formal - Formal Test Bench Analyzer (FTA) [1]. FTA systematically introduces *stuck-at* and *inversion* faults one by one to all the ports of RTL and see if any of the checkers fail. If no checker fails for a particular fault, then a checker is missing in the connectivity specification. If connectivity specification checkers can capture every possible fault introduced, our specification is complete, and our formal verification exhaustively covers all the outcomes.

C. *Circular Connections*

Circular connection checker verifies if there a circular path from a port to back itself with a zero delay. For instance, consider the case when connectivity specification includes a check from port A to B with a zero delay, a check from port B to C with a zero delay, and finally a check from C to A with a zero delay. The success of these checks is dependent upon the completeness of the connectivity specification.

Table 1: SpyGlass vs. VCFormal Memory and Run Times

Partition	Specs	SpyGlass	VCFormal
A	Total=641 Passed=630 Failed=11	Total Time(S) :6131.36 CPU Time(S) :5960 Peak Memory(MB):23469	Total Time(S) :934.35 CPU Time(S) :216.85 Peak Memory(MB):2676
B	Total=670 passed=670	Total Time(S) :5233.59 CPU Time(S) :5167 Peak Memory(MB):16358	Total Time(S) :544.81 CPU Time(S) :420.78 Peak Memory(MB):7770
C	Total=247 Passed=247	Total Time(S) :256169 CPU Time(S) :251781 Peak Memory(MB):166524	Total Time(S) :11007.50 CPU Time(S) :8527.24 Peak Memory(MB):59224
D	Total=2400 Passed=x Failed = x Undecided=x	Total Time(S) :246990 CPU Time(S) :246236 Peak Memory(MB):102461	No convergence.

IV. The Tools

The two key tools we selected to design our flow are VCFormal and Spyglass [2, 7]. VCFormal is a formal verification tool. It builds a model of the design and verifies it against a set of checks while observing certain assumptions provided by the user. It is exhaustive in nature which is an advantage and disadvantage at the same time. The advantage is that if it is able to verify a check, the results are good for all possible inputs with the given assumptions. The disadvantage is that large design exploration space can lead to memory blowup and lack of convergence.

SpyGlass is primarily a static verification tool, though it has some built-in FV engines as well. It analyzes the design and finds design patterns like clocks, flip-flops, DFT, etc. There is a large set of design rules available, so it is up to the user to pick the rule-set appropriate for verifying the different aspects of the design. We found it useful to specify and verify high-level properties like exclusives one-to-one, etc.

The choice between VCFormal and SpyGlass depends upon a number of factors like the time available, design size, specifications size and specification type. Table 1 provides comparative data for four different designs. For small and moderate designs, VCFormal is a good choice in terms of speed and time. However, for larger designs like the full-chip, SpyGlass is the only option. Another important aspect is functional connectivity verification. For instance, no unwanted inversion should exist between the source and destination. In this case, it is VCFormal and not SpyGlass which is capable of verifying that information.

SpyGlass and VCFormal both provide a set of applications that cater to specific verification needs. In the case of SpyGlass, there is a connectivity application that lets the user specify connections positively (by asserting "require_path") as well as negatively (by asserting "illegal_path"). SpyGlass provides a mechanism to specify one-to-many, many-to-one and many-to-many checks. SpyGlass also provides the ability to specify connection types like:

a) *Direct connection*: It specifies a wired connection between two ports, with no RTL artifact (flip-flops, muxes, etc.) between them.
b) *Buffered connection*: A one-to-one connection with some flip-flops in the path. It only fails if there is a gate other than a buffer or inverter along with the path.
c) *Conditional connection*: A sensitizable or sensitized connection between two ports. It fails if there is a buffer or a black-boxed module between the two ports.
d) *Topological connection*: This check only fails if there is a black-boxed module between two ports.

In the case of VC Formal, we have used four applications from the VC Formal Tool Suite:

1) Connectivity Checker (CC) [1] is the formal connectivity checking application. It provides multiple ways to specify design connectivity including CSV (commas separated values). An aggressive use of black-boxing and

other aggressive design reduction strategies allows it to work efficiently on large designs. We have used it to verify basic connectivity connections.

2) Formal Test Bench Analyzer (FTA) [1] is a key tool for a formal signoff of a design. It provides fault injection analysis capabilities [7] i.e. if there a bug in the design, do we have enough checkers to capture that bug? We used it to verify the completeness of the connectivity specifications.
3) We used Connectivity toggle coverage in VC Formal CC [3] to generate toggle coverage from the proven assertions.
4) Formal Security Verification App (FSV) [1] is a security verification tool. We used it to prove negative connectivity checks i.e. absence of the connections.

V. THE WORKFLOW

Figure 3 shows an abstract view of our connectivity flow. The data generally flows from left to right and feedback to the user flows from right to left. The flows marked with "Failure" are the feedback to the designers for missing or failing cases. The flows marked with "Success" feedforward the successful specifications to the next tool in the flow or to the final reports.

The regression testing flow starts with the users' provided connectivity specifications, supplied in a CSV file format with each line specifying one connection. These files are directly consumed by VCFormal Connectivity tool. The tool internally converts these files into connectivity checks, reads the RTL, performs the black boxing and formally verifies the connectivity checks against the design RTL. The designers get reports on the failed checks.

The next step is verification of the completeness of the specification. For this purpose, we use the Formal Test Bench Analyzer (FTA). In FTA, we only select connectivity faults. The passed checks are converted to SVA and bound [4] to the design to run through fault analysis process. In this case, success indicates a failure, i.e. if the insertion of fault does not make any checker in the specification to fail, then we have a hole in our specification. The hole either indicates a missing checker, i.e. a missing line in the connectivity table or a redundant signal. In either case, the result is reported back to the designers' team and they have to fix it one way or other.

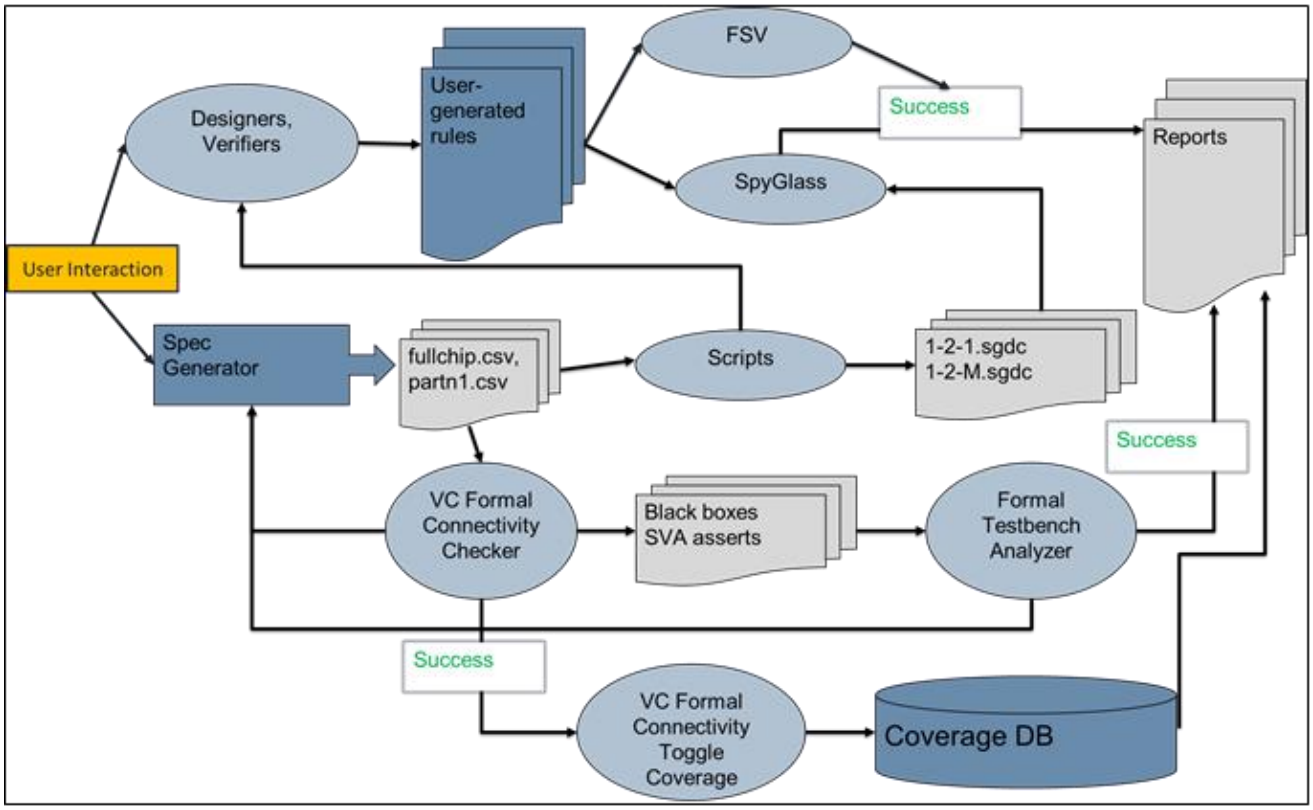

Figure 3: Connectivity Regression Flow

Table 2: A Sample Missing Check

MODULE	Results	Non-Activated	Error Signal
C	FPV_FTA ---------- > Fault - # found : 243 - # non_activated: 3 - # detected : 240 > Disabled - # found : 5226 - # assert : 5226	[1062] non_activated - c.fault_id_13 (/rtl/c.v:68) [1173] non_activated - c.fault_id_14 (/rtl/c.v:68) [1284] non_activated - c.fault_id_15 (/rtl/c.v:68)	clk_obs "Present in C.v but missing from C_vcformal.csv" Fix: C_test_fta,clk,,,C.c_clk_wrap__cclk_out,c.clk_obs,0

Another thread runs concurrently to FTA. It generates and verifies the derived connectivity specifications. Python scripts were written to check for circularity in signals. Furthermore, scripts were written to find one-to-one, one-to-many, many-to-one and many-to-many connectivity patterns in the specifications. These patterns are used to generate input specifications for SpyGlass. The failures are reported back to the designers and the verifiers. The success is reported and uploaded to the Reports database.

A third parallel thread processes user-specified rules. Depending upon the type of the rule, either SpyGlass or FSV (Formal Security Tool) is used. A decision between two tools depends upon the size of the design and/or type of the problem. SpyGlass has more capacity than formal tools but takes a longer time for most of the medium and small designs. The security-based connectivity specifications are more intuitive to model in FSV.

The last thread of the flow is the generation of Toggle coverage from the proven connectivity properties [3]. This can be termed as a by-product of the connectivity regression flow but may save a lot of time for full-chip toggle coverage closure. We were able to generate up to 10% of very difficult to cover toggle points, with no extra effort.

VI. RESULTS

We are running the connectivity flow regression on a weekly basis.

Here is a summary of the results:

1) Found loops in the specification.
2) Found missing signals in RTL which were part of the specification.
3) Found bad-connections i.e. path does not exist or not properly specified in terms of delay etc.
4) Verified 1-1, 1-M, and M-1 rules.
5) Verified completeness of specification for multiple partitions as well as full-chip.
6) Generated toggle coverage from connectivity proofs.

A sample bug found by FTA is shown in Table 2. In this case, FTA found a signal in the RTL but no corresponding signal in the connectivity specification. The fix was to add the missing connection to the CSV file.

The designs include all the major partitions of the SOC as well as the full-chip connectivity. The user defined checks specify connectivity behavior of various design components like:

a) Clocks
b) Resets
c) Design for Testability(DFT)/Scan
d) Fuse
e) Debug Bus connections
f) Virtual channels

Finally, Figure 4 shows the toggle coverage generated for a partition from the passing connectivity specification of a partition. In most of the cases, we were able to obtain 10-15% toggle coverage. As mentioned earlier, we also used connectivity specification to replace toggle coverage tests with connectivity specifications to verify the connections after SOC integration. It is important to note that we are not replacing full-chip toggle coverage with this method but only helping it to converge faster.

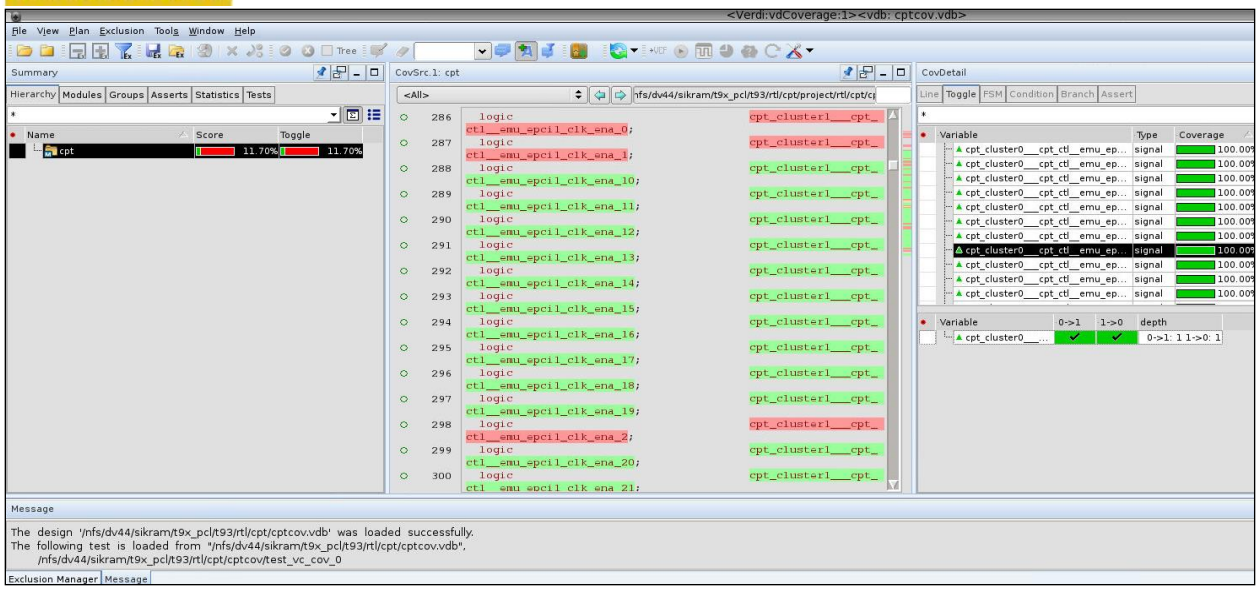

Figure 4: Toggle Coverage

VII. Conclusions And Future work

There is no single tool or solution for complex issues. SOC verification is an extremely complex problem and impossible to solve within the limited time available to verify. Every new tool or application that can find issues is an asset. We have shown here how we can orchestrate different tools like formal, static, coverage and fault injection to make faster convergence. The formal tools were used to validate connectivity. The static verification tools were used to validate restricted one-to-one, one-to-many and many-to-one connectivity. The fault injection tools were used to validate the completeness of the connectivity specification. The coverage tools were helpful in coverage convergence.

It is an ongoing effort. We are looking at multiple tools and techniques to further enhance this flow. One of the tools is formal security verification [6]. We have tried it but did not fully integrate it into our flow. This tool can be used to verify negative statements about connectivity and provides innovative ways to specify intentions. Another tool that can be incorporated to make the flow more efficient is formal unreachability tool. On one end, it can be combined with Toggle Coverage Tool to provide a more comprehensive database report. On the other end, in the case of many off-the-shelf IPs, parts of the design are configured to be unreachable. The unreachability tool can figure out these unreachable blocks and may feed this information to the connectivity flow. This will limit the false-failure cases and hence save the debugging time.

VIII. Acknowledgements

Finally, it takes a village to complete an effort like this. We would like to acknowledge the efforts put in by the design team especially Cameron Glass, Eddie Langevin, and Joel Longever. We also thankful to the support provided by Synopsys staff especially, Bill Dufresne, Gregory Milano, and Anthony Joseph.

IX. References

[1] Synopsys Inc. “VC Formal User Verification Guide” Synopsys Inc. Mountain View, CA, 2017.
[2] Joseph, Anthony. “Static Connectivity Verification/Validation” Synopsys Inc. Mountain View, CA, 2017.
[3] Nordstrom, Anders. “Formal Connectivity Checking Coverage” Synopsys Inc. Mountain View, CA, 2017.
[4] Cerny, Eduard etc., “SVA: The Power of Assertions in System Verilog,” Second Edition, Springer, 2014, ISBN: 3319071386
[5] https://searchitoperations.techtarget.com/definition/YAML-YAML-Aint-Markup-Language
[6] Synopsys Inc. “FSA: Formal Security Verification Application.” Synopsys Inc. Mountain View, CA, 2018.
[7] Okun, Vadim. “Mutation for Test Generation and Analysis.” Ph.D. Thesis, University of Maryland, 2004.
[8] Synopsys Inc. “SpyGlass User Guide” Synopsys Inc. Mountain View, CA, 2018
[9] Yeung, Ping and Mandel, Eugene. “Multi-Domain Verification of Power, Clock and Reset Domains”, Haifa Verification Conference, 2015.