Formal Verification of Connections at SoC-level

Penny Yang[2], Prasun Das[3], Yuya Kao[1], Mingchu Kuo[1]

[1]MediaTek Inc., Hsinchu, Taiwan
[2]Synopsys Taiwan Co., Ltd., Hsinchu, Taiwan
[3]Synopsys India Pvt Ltd., Bengaluru, India

***Abstract*-Verification of connections at SoC (System on Chip) level is a fundamental requirement to ensure correct operation. It is a significant challenge for verification engineers to cover every scenario because simulation patterns in whole chip environment are usually fewer and thus corner case bugs are very difficult to find. Formal verification, on the other hand, exhaustively explores the mathematical representation of the design to uncover all possible incorrect behaviors. However, the state space explosion issue caused by design complexity becomes the most challenging problem in formal world. Since an SoC consists of several millions of sequential logics, the complexity issue is more critical for convergence. Therefore, months of manual efforts are needed to do abstraction to verify connectivity problems. This paper describes the experience of using connectivity checking (CC) application in a productized formal verification tool, VC Formal, to reduce manual efforts and save run time. Our results indicate that formal connectivity checking is feasible at SoC level, without manual abstraction, in proving reset and padmux connections within hours (from original days). It has advantages of exhaustiveness of formal verification and scalability of structural analysis. Moreover, CC application has customized features for setup, use and debug especially for connectivity problems, reduces overall turn-around time. MediaTek has successfully used CC solution in the regular verification flow for more than 10 projects.**

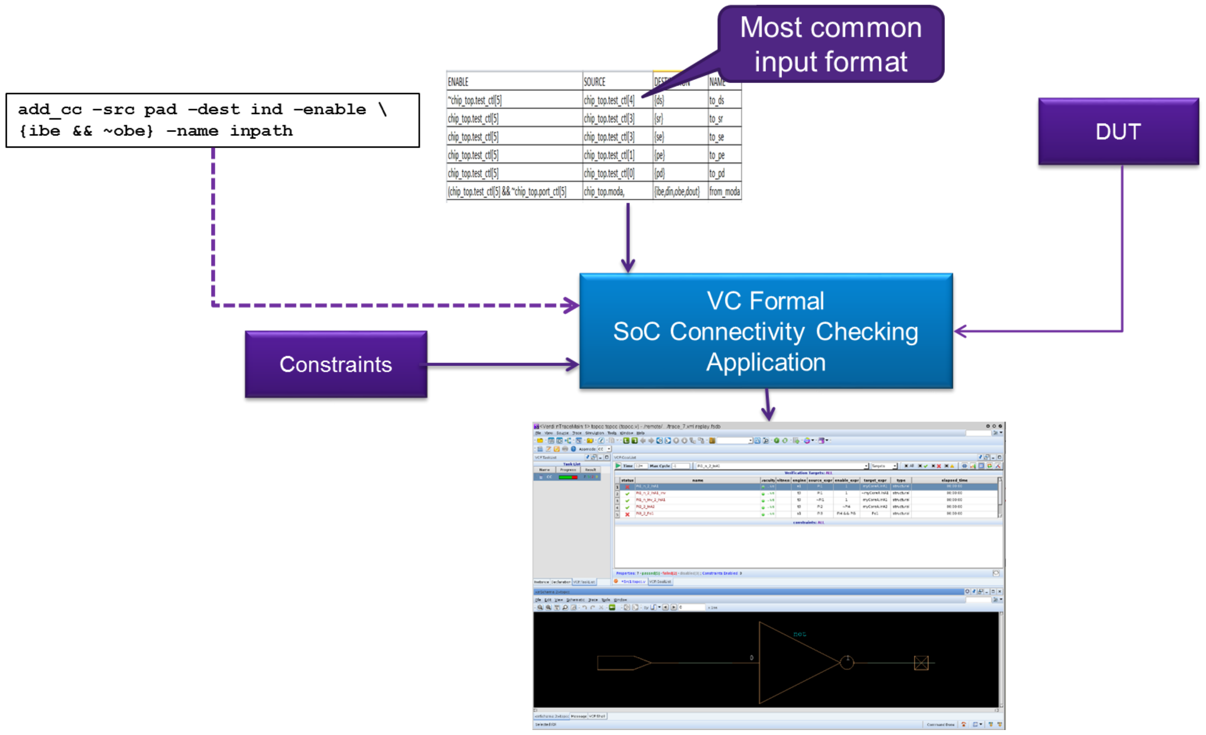
Keywords: Formal Verification, VC Formal, Connectivity Checking, SoC Reset Verification, SoC Padmux Verification.

I. Introduction

Complete verification of connections at SoC level is a prerequisite to ensure correct operation. Thus, MediaTek has applied formal verification on various projects for last 10 years. Before using formal verification, chip level simulation was used to verify the connections at SoC-level. Since the patterns in chip level simulation environment are usually fewer than the ones in block-level environment, corner case bugs sometimes appeared in uncovered codes. These bugs reflect the challenges we face with the traditional simulation-based verification methodologies used in the design flow. Besides, the integration of chip level test-bench often comes late in a project cycle because the building blocks for all the sub-systems must be completed and tested. Therefore, to be able to shift left, formal verification is adopted due to its exhaustiveness and easy setup, i.e., no stimulus generator is needed. Formal verification is a systematic process of ensuring (through exhaustive algorithmic techniques) that a design implementation satisfies the requirements of its specification [1]. These characteristics of being systematic and exhaustive, perfectly solve the problems faced in simulation.

Formal property verification (FPV) has been proven to be a reliable method for different kinds of designs' verification signoff. However, it is unrealistic to achieve full proof without any manual abstraction strategies as designs are becoming more complex. Capacity and complexity has always been the biggest limitation for deployment of formal techniques. Before the availability of CC application in formal verification, only engineers with design knowledge could run FPV. They needed to partition the SoC into several different sub-systems, blackbox the modules that were not used, constrain the designs with constants and assumptions. It took several months to work on these settings from the beginning till the end of the project since the run time could be several days and hence the iterations with debugging was very slow. Trial and error was time consuming and manual abstraction was prone to false alarms too.

Formal Applications are customized for easy setup, use, and debug. It is perfect for beginners because there is no need to have formal background or knowledge to write SystemVerilog Assertions (SVA). For example, tcl commands, CSV formats and Excel files are all supported as the input format in VC Formal CC Application. Verdi is its debugging interface that user can use to list the logic in the path using text or schematics as shown in Figure 1.



```
add_cc -src pad -dest ind -enable \
{ibe && ~obe} -name inpath
```

ENABLE	SOURCE	DESTINATION	NAME
~chip_top.test_ctl[5]	chip_top.test_ctl[4]	{ds}	to_ds
chip_top.test_ctl[5]	chip_top.test_ctl[3]	{sr}	to_sr
chip_top.test_ctl[5]	chip_top.test_ctl[3]	{se}	to_se
chip_top.test_ctl[5]	chip_top.test_ctl[1]	{pe}	to_pe
chip_top.test_ctl[5]	chip_top.test_ctl[0]	{pd}	to_pd
(chip_top.test_ctl[5] && ~chip_top.port_ctl[5]	chip_top.moda,	{ibe,din,obe,dout}	from_moda

Figure 1. CC Application Flow in VC Formal

This paper describes the experience of using automatic abstraction flow in CC in a productized formal verification tool to reduce manual efforts and save run time. SoC padmux connectivity checking and Reset connectivity checking are covered in the cases of this paper.

The purpose of padmux connectivity verification is to ensure the correctness of the complex MUX connection between sub-system signals and pad ports as Figure 2 and Figure 3 shows. Padmux Connectivity Checking has challenges in SoC because it has large gate count, but limited I/O pins leading to shared I/O pinmux to control access requires significant interconnect wiring and introduces significant possibility of errors. Besides, padmux design is rather hard to do ECO (Engineering Change Order) because combinational logic is often optimized after synthesis. In padmux connectivity verification, the sources and destinations are extracted from a csv table delivered by each module owner to describe their signals from/to the pads. The enable conditions are the different mode settings for connections. Each connection becomes a checker in formal verification. Since the global control logics are implemented for different modes using several MUXes in between the path, they should be checked both structurally and functionally to ensure paths are not blocked by different values of MUXes' select pins.

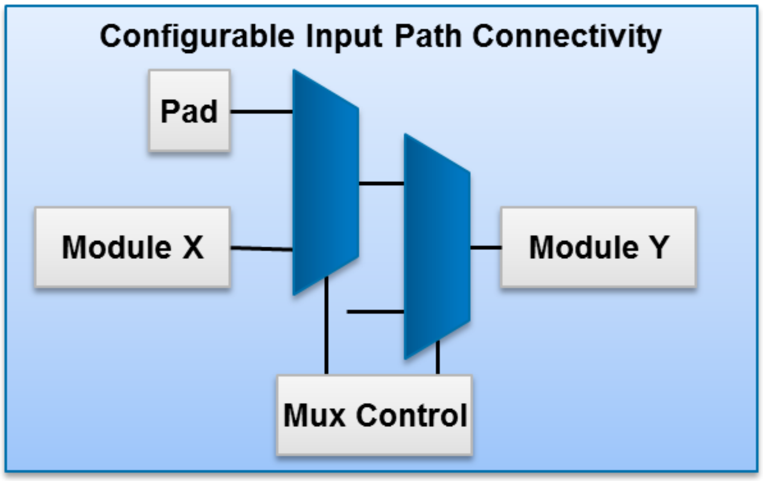


Figure 2. Configurable input path connectivity in SoC padmux

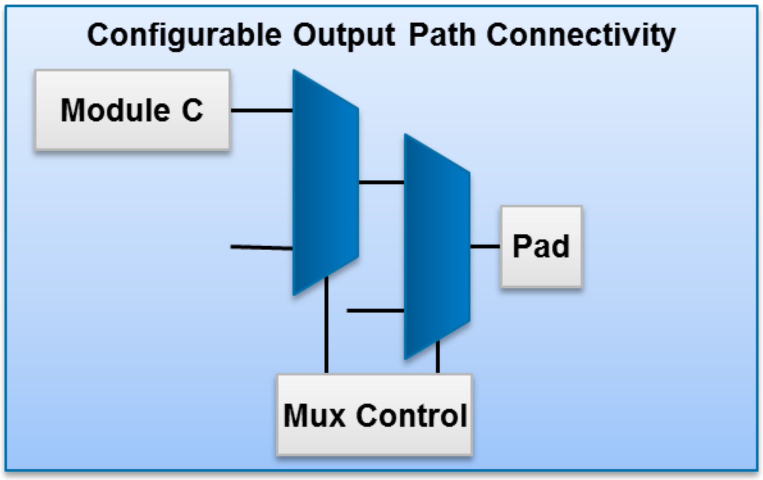


Figure 3. Configurable output path connectivity in SoC padmux

SoC designs have multiple sources of global reset, such as power-on reset, hardware reset, software reset and watchdog timer reset. These are top-level signals that should be connected to all asynchronous resets in the design, i.e., all asynchronous resets in the design should be asserted when the global reset is asserted. Unfortunately, a missing reset connection, or one that is blocked from propagating due to the mode of operation of the design, can easily go undetected. As our previous paper [3] discussed, it is hard for simulation to verify all reset scenarios and that every source of reset propagates to all the intended storage elements in the design under all the proper conditions. Taking watchdog reset verification as an example, a directed test created to verify if the watchdog reset operation works correctly must trigger the watchdog reset condition in the middle of the simulation to check if the watchdog reset has propagated to all the intended flip-flops in the entire SoC. Thus, a passing test does not mean that the watchdog reset is verified. Without completely verifying that the watchdog reset has propagated to every intended flip-flop, simulation-based watchdog reset verification is incomplete, and bugs may still sneak into the design, causing system failures in lab testing and require a silicon re-spin to fix.

In this paper, we will present a highly automated methodology using formal connectivity checking application to completely verify SoC padmux and reset schemes without significant manual effort as opposed to simulation-based verification or formal property verification alone.

II. Connectivity Checking Methodology

Formal Connectivity Checking is one popular application with the following goals:

- Structural Check: Check if there is a structural connection/path between source and destination and is directional
- Functional Check: Check that two signals in the design have the same value

There are many applications which are ideal for formal connectivity checking, e.g.

- SoC I/O Connectivity
- Block pin muxing/demuxing
- Connectivity & constant checking of macros
- Scan mode connectivity & constant checking
- Reset and global signal connectivity
- Registers to Debug Bus
- Configurable interconnect verification

Like any formal verification environment, design read, clock-reset specification and setting constant values are first steps in CC application as well. The design will be compiled and loaded into the tool with these steps. Thereafter, verification engineer can specify the signal names corresponding to the source and destination of the connection that must be proven. If there is an enable signal under the influence of which the connection holds good, then that enable signal should also be provided. This information can be provided in different ways (also shown in Figure 4, 5 and 6):

1. Tcl commands
2. CSV format
3. MS Excel format

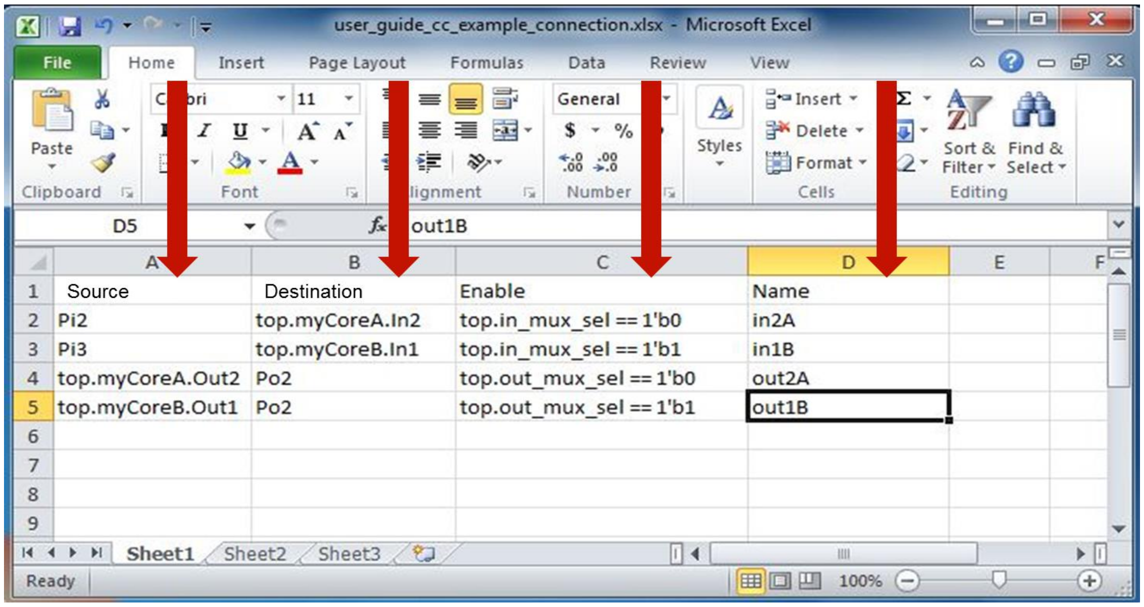
VC Formal provides the flexibility to edit, delete or add a connection without the need to recompile the design and hence saves much of time. The tool can be run on grid for faster convergence. It has an automatic blackboxing mechanism which is deployed on the fly by the tool to tackle convergence issues. This mechanism is specifically targeted for solving connectivity problems at SoC level and that is what differentiates CC application from traditional formal property verification.

```
#reset_checker.tcl
add_cc -name c1 -src 0 -dest u_cpu.resetn_i -enable $cpu_rst_en
add_cc -name c2 -src 1 -dest u_cpu.g1.reset_i -enable $cpu_rst_en
…
```

Figure 4. CC’s checkers in tcl format

```
#padmux_checker.csv
u_cksys.tck, u_io.PAD_MCK.O,{gpio_mode==1}, a1
u_io.PAD_MDAT.I, u_cksys.dsp_out,{gpio_mode==6}, a2
…
```

Figure 5. CC’s checkers in csv format



	A	B	C	D
1	Source	Destination	Enable	Name
2	Pi2	top.myCoreA.In2	top.in_mux_sel == 1'b0	in2A
3	Pi3	top.myCoreB.In1	top.in_mux_sel == 1'b1	in1B
4	top.myCoreA.Out2	Po2	top.out_mux_sel == 1'b0	out2A
5	top.myCoreB.Out1	Po2	top.out_mux_sel == 1'b1	out1B

Figure 6. Generate csv file from an excel table

In reset connectivity verification, the destination signal is the reset signal extracted from every register and the source is 0 or 1 depending on the active polarity of the reset destination. The reset and register list can be obtained using tool command. The enable expression is the active condition of the global reset signal. The global reset can be specified as source in tcl to verify that the global reset is correctly propagated to the intended storage elements and indeed resets them.

```
# Generate reset and register list
report_ff_reset -list
# Reset path
set cpu_rst_en u_reset.wdr
source cc_checker.tcl
```

Figure 7. Tcl commands to generate resets and specify reset source

It is efficient using CC to verify that the connection is correct structurally and functionally. Structurally disconnected errors are reported once the CC checker is loaded as Figure 8 shows. It allows to start debugging as shown in schematics in Figure 9 and in the form of textual report as shown in Figure 10 without even waiting for formal verification run to start. After the formal verification run is done, tool will give proven results for functionally connected paths and failure traces for functionally disconnected paths. Verdi is its integrated debugging interface to check results, trace code, drag failed waveforms and view schematics in VC Formal with customized features for CC as shown in Figure 9 and Figure 10.

```
[Error] CC_UID017: Connection 'InA2_2_Pi1' is structurally disconnected.
No path from source to target. Please fix the connection
```

Figure 8. Error message of structurally checking

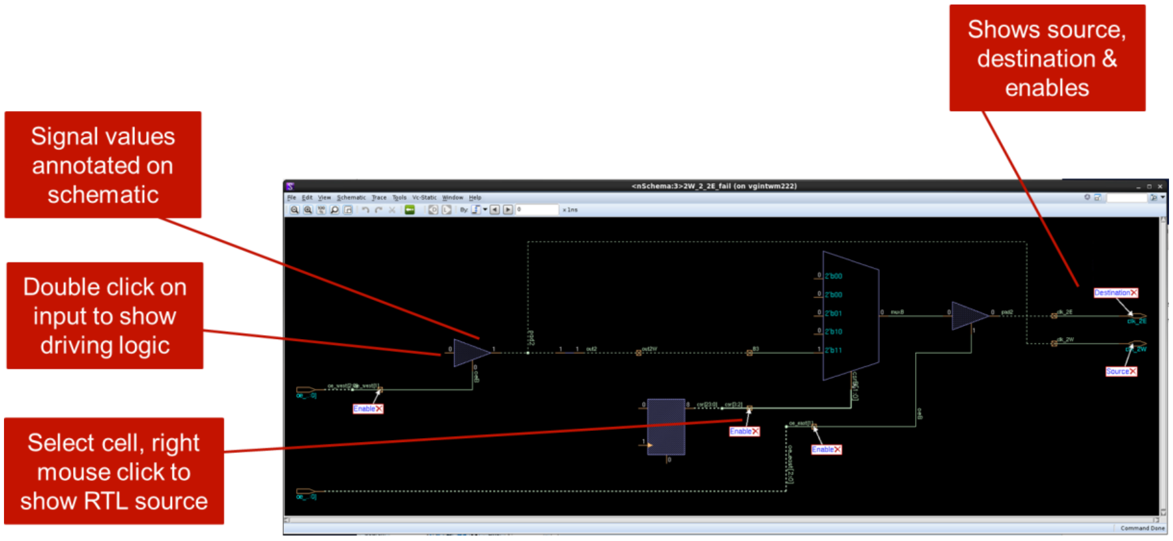
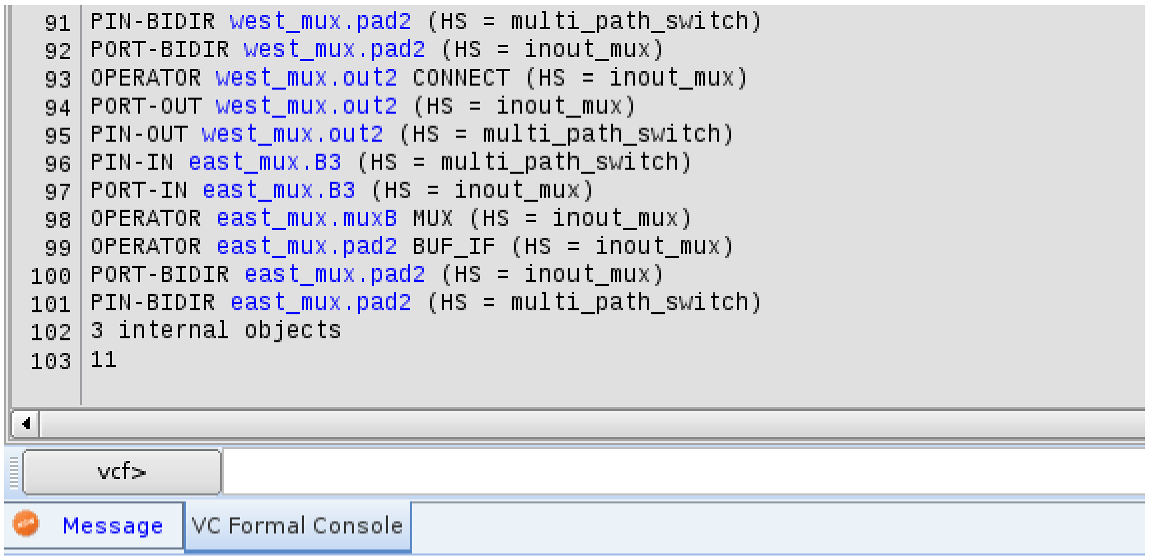

Shows source, destination & enables
Signal values annotated on schematic
Double click on input to show driving logic
Select cell, right mouse click to show RTL source
Destination
Source
Enable
Enable
Enable
Enable

Figure 9. Schematics view in CC



```
PIN-BIDIR west_mux.pad2 (HS = multi_path_switch)
PORT-BIDIR west_mux.pad2 (HS = inout_mux)
OPERATOR west_mux.out2 CONNECT (HS = inout_mux)
PORT-OUT west_mux.out2 (HS = inout_mux)
PIN-OUT west_mux.out2 (HS = multi_path_switch)
PIN-IN east_mux.B3 (HS = multi_path_switch)
PORT-IN east_mux.B3 (HS = inout_mux)
OPERATOR east_mux.muxB MUX (HS = inout_mux)
OPERATOR east_mux.pad2 BUF_IF (HS = inout_mux)
PORT-BIDIR east_mux.pad2 (HS = inout_mux)
PIN-BIDIR east_mux.pad2 (HS = multi_path_switch)
3 internal objects
11
```

Figure 10. Textual report

III. Result Analysis

The design in this formal verification is an SoC with 64,056,916 register bits. There are 5 cases in the results. 3 of the cases are for reset connectivity verification from 1 checker to 173981 checkers to verify different parts of the design. 2 of the cases are for padmux connectivity verification from 447 to 4732 checkers to verify different modes.

We ran these cases using VC Formal from Synopsys. First, we used assertions to describe the connections and applied to FPV. Without any black-box settings in the whole chip design, FPV was not able to conclude in 24 hours due to inherent capacity limitation. This experiment shows that applying the assertion based CC approach is not suitable and does not scale to the SoC level. Then we tried structural check. Though it can finish checking in 2 hours, it is not able to point out structurally connected but functionally disconnected bugs.

Therefore, we verified the cases with the application of CC and got the results as shown in Table I, verification time w/ traditional FPV based CC flow from 4 hrs to 35 hr. Furthermore, we applied the cases using a newly optimized CC flow (with automatic abstraction feature) and got the results in Table I, verification time w/ optimized CC flow from 3 min to 3.4 hrs, with 1.3 to 160X improvements.

Table I. Test Results of the 5 testcases

Case	# of connections for verification	Verification time w/ traditional FPV based CC flow	Verification time w/ Optimized CC flow	Improvements
Reset 1	173981 (103971 proven, 70010 failed)	35 hrs	40 mins	52X
Reset 2	25000 (23241 proven, 1759 failed)	4 hrs	12 mins	20X
Reset 3	1 (1 proven)	8 hrs	3 mins	160X
Padmux 1	4732 (4731 proven, 1 failed)	4 hrs	3 hrs	1.3X
Padmux 2	447 (441 proven, 6 failed)	5.6 hr	3.4 hr	1.7X

The application of CC is optimized for connectivity checking problems. The run time for formal verification is improved significantly with the innovative automatic abstraction flow optimized in CC compared to FPV. Thus, we can easily integrate the commands into MediaTek's regular flow using either tcl or csv format.

IV. Conclusion

This paper presents highly automated methodologies using Formal techniques to verify the correctness of global reset schemes and padmux connection, without the large amount of effort required by manual abstraction in SoC. The methodologies described above have been deployed on 10 projects at MediaTek. As the results show, we have found that the strength of CC App is more efficient than pure FPV based connectivity verification methodologies. The connectivity verification flow can be completed in hours, without any inconclusive properties, on an SoC size design. To shorten the iteration time and facilitate debugging, we also recommend to use the schematic view and utilities in Verdi. In summary, the application of CC combines the advantages of exhaustiveness in formal verification and scalability in structural analysis to verify reset schemes and padmux connection. It also enables design teams to gain greater confidence in their designs with less effort.

V. Future Work

Our results indicate that formal verification of connections is feasible at the SoC level now. Therefore, the next step is to measure what part of the design is covered when doing formal connectivity checking to determine if the verification is complete. We expect to measure line, condition and toggle coverage of ports, nets and flip flops and save coverage database that can be merged with coverage database collected from simulation connectivity verification. Besides, we would like to determine if there are paths where connectivity checks are missing, including checking that all paths through muxes have been checked, e.g., if line or condition coverage goals corresponding to a mux are not covered. It helps build confidence to formally signoff on connectivity checking and allows us to close more projects in an efficient way.

ACKNOWLEDGMENT

The authors would like to thank Sean Chou from MediaTek and Alvin Chen, Xiaolin Chen, Sean Safarpour, Pratik Mahajan, Alfred Koelbl from Synopsys for their support during the development. Without their help, we could not have achieved these great results.

REFERENCES

[1] D. Perry, H. Foster, “Applied formal verification: for digital circuit design”, McGraw Hill, 2005.
[2] Synopsys. (2017) VC Formal Verification User Guide, version M-2017.03-SP2.
[3] Kaowen Liu, Penny Yang, Jeremy Levitt, Matt Berman, Mark Eslinger, “Using Formal Techniques to Verify System on Chip Reset Schemes”, DVCon 2013.