Ironic But Effective: How Formal Analysis Can Perfect Your Simulation Constraints

Penny Yang[1], Jin Hou[2], Yuya Kao[1], Nan-Sheng Huang[1], Ping Yeung[2], Joe Hupcey[2]

[1]Mediatek Inc., Hsinchu, Taiwan
[2]Mentor Graphics Corp., Fremont, CA, U.S.A

***Abstract*-This paper describes the experience of using formal analysis to verify the correctness of the Verification IP (VIP) configurations that are used in constrained random simulation environment. We describe the methodology involving collecting the simulation VIP configurations, formulating System Verilog Assertion (SVA) code, setting up a formal run environment, and collecting and analyzing results. We illustrate this with a real project example, where VIPs are used to replace AXI and AHB design modules to verify a bus fabric that connects AXI and AHB interfaces. We have to make sure that VIPs are correctly configured to behave the same as their substituted design modules. We present SVA examples for checking the configurations. The cumulative result is that we show how formal verification techniques can be applied to exhaustively validate the proper scope of the constraints used in a constrained-random simulation environment.**

I. Introduction

Constrained random simulation is commonly used to verify complex SoC designs. In such a verification environment, users can employ verification IPs (VIPs) or constraint blocks to generate various transactions or values. Users are responsible for constraining the types of transactions and the set of values to be used. These constraints should be as realistic as possible reflecting the actual operating conditions of the DUT.

In Mediatek, we have complicated bus fabrics that interconnect the master and slave modules of AXI protocol or AHB protocol. When we verify these fabrics for interconnecting the master and the slave modules in simulation, we use Master VIPs and Slave VIPs to substitute the master and slave design modules in the simulation environment. This approach can isolate the issues of the bus fabric with respect to the master and the slave modules. It can also verify the bus fabric behaviors when the master or the slave modules are not ready. Besides, VIPs are more flexible to be configured to test many different scenarios and the run time using VIPs is much faster than using all the design modules. That said, it's easy to get into trouble by either over or under-constraining the simulation environment, and thus in this paper we will show how formal verification techniques can be applied to exhaustively avoid such issues and validate the proper scope of the constraints.

II. Problem Details

VIPs should behave the same as their substituted design modules, where the designer provides the configuration information (parameters and functions) to the verification team for configuring said VIPs. The constrained VIPs replace the master and slave design modules in verifying the Bus Fabric. The problem is that if the designer provides wrong information, the constrained VIPs may not behave correctly, or execute either excessive or insufficient functionalities than the "real" associated design modules would. As a result, the verification results and coverage on the bus fabric may be incomplete. Here are the possible outcomes:

1. DUT has MORE functionalities than VIP:
   In this case the VIP over-constrains the environment of the bus fabric, and may hide extra functionalities and bugs in the bus fabric

2. DUT has LESS functionalities than VIP
   In this scenario, the VIP under-constrains the environment of the bus fabric, resulting in false firings in simulation runs that only waste simulation and debug time

Unfortunately, in the real world design modules can be from different departments, from different companies (IP bought from other companies), or the design module owners may change over time. Then no one knows the complete functionality of the design modules. When VIPs are used to substitute design modules in simulation environment, the information provided to configure VIPs is normally not accurate. Do we have a way to verify the correctness of the configuration information for VIPs? Do the constrained VIPs have the same functionalities as the design modules?

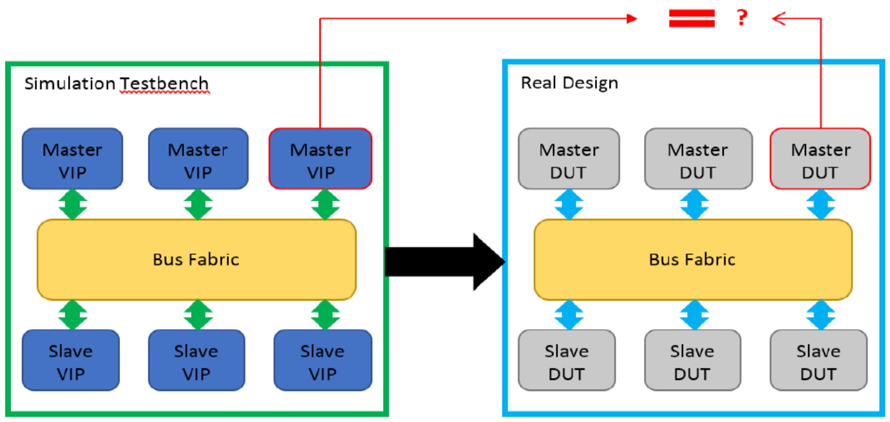


Figure 1: Simulation VIP and Its Substituted Design Module

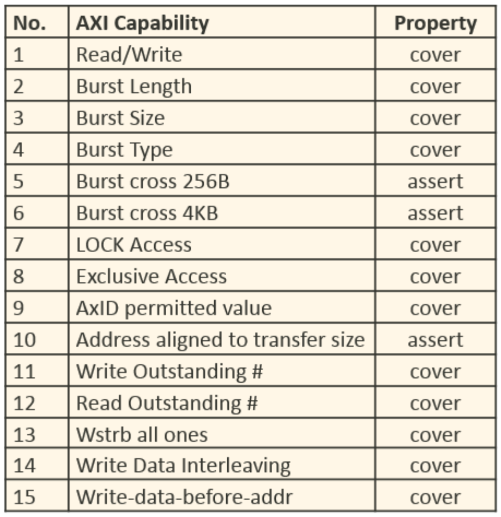
Formal verification is the technology that can help us find out the correctness of the configuration and answer if the constrained VIPs behave the same as the associated design modules. Especially when VIPs are configured to have more functions than the design modules, only Formal analysis can find these situations since only formal can prove unreachable states. For example, the cover property representing the extra function should not be coverable in the design module. We know simulation cannot prove that a cover property is not coverable, only formal verification can. Formal is the only choice for us. Another benefit of using formal verification is that we don't need to write a testbench which saves a lot of time.

The VIPs used in our random simulation environment are not formal consumable. Thus we cannot run them against the design modules with formal tools to verify the correctness of their configurations. To verify the correctness of the configurations of the VIPs, first we list all of them in a table. Second we use SVA cover/assert properties to implement them. Third we use Questa Formal tool (PropCheck App) to validate the SVA properties with the Master and Slave design modules– the results tell us the actual behaviors of the design modules. Fourth we record the results in a table to compare the real behaviors of the design modules with the configurations for setting up the VIPs, we then know if the constrained VIPs are capable of the same functionalities as the design modules, or more or less than the design modules. If mismatches happen, we discuss the mismatches with the designers, and fix the VIP setting errors accordingly.

III. VIP Configurations And SVA Implementation

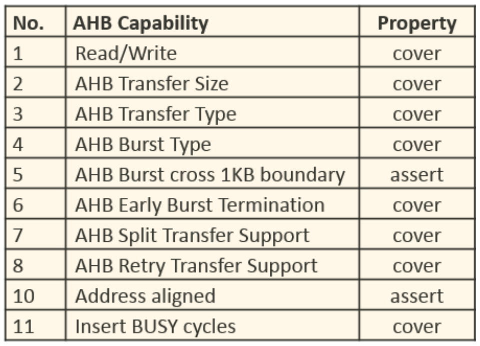
As noted above, for each VIP used in the simulation environment we list all its configurations in a table. These configuration specifications are the ones we want to verify against the design modules. We analyze the configurations and decide how to implement them using either cover property or assert property, then we use SVA to implement them. For legal behavior, we want to check that it does exist, we use cover property for it. For illegal behavior that shouldn't happen, we use assert property for it. Here are the examples of the tables listing AXI VIP configurations and AHB VIPs configurations used in our project.

Table I: Configurable Functions of AXI VIP



No.	AXI Capability	Property
1	Read/Write	cover
2	Burst Length	cover
3	Burst Size	cover
4	Burst Type	cover
5	Burst cross 256B	assert
6	Burst cross 4KB	assert
7	LOCK Access	cover
8	Exclusive Access	cover
9	AxID permitted value	cover
10	Address aligned to transfer size	assert
11	Write Outstanding #	cover
12	Read Outstanding #	cover
13	Wstrb all ones	cover
14	Write Data Interleaving	cover
15	Write-data-before-addr	cover

Table II: Configurable Functions of AHB VIP



No.	AHB Capability	Property
1	Read/Write	cover
2	AHB Transfer Size	cover
3	AHB Transfer Type	cover
4	AHB Burst Type	cover
5	AHB Burst cross 1KB boundary	assert
6	AHB Early Burst Termination	cover
7	AHB Split Transfer Support	cover
8	AHB Retry Transfer Support	cover
10	Address aligned	assert
11	Insert BUSY cycles	cover

After we have the lists of the configurable functions, we implement them using the SVA language. For testing the configurations of AXI VIP and AHB VIP, most SVA properties are cover properties. Here are some examples of the SVA assertions.

1. AXI burst length: AXI burst length can be from 1 to 16 defined by *awlen* from 0 to 15

```
  default clocking @(posedge clock); endclocking
  default disable iff (~reset);
  generate
    for (genvar i=0; i<16; i++) begin
      Cover_burst_size: cover property (awvalid && awlen==i);
    end
 endgenerate
```

2. AXI write outstanding: use RTL code to define a counter *pend_bresp_cnt* that counts the number of outstanding write: pend_bresp_cnt = # of aw - # of bresp

```
  generate
    for (genvar i=1; i<=64; i++) begin
      Cover_wr_outstanding: cover property (pend_bresp_cnt==i);
    end
 endgenerate
```

3. Burst cross boundary: check if burst cannot cross specific boundary

```
  Assert_cross_boundary: assert property (awvalid && (awburst == INCR) |->
AwAddrIncr[ADDR_MAX:ADDR_BOUND] == awaddr[ADDR_MAX:ADDR_BOUND]);
```

4. Wstrb has all ones

```
Cover_wstrb_allones: cover property (wvalid && wready && (wstrb == {WSTRB_WIDTH{1'b1}}));
```

5. AHB can insert BUSY cycles:

```
Cover_busy_cycle: cover property ((hready && htrans==BUSY) ##1 (hready && htrans==SEQ));
```

6. Address aligned:

```
Assert_addr_aligned: assert property (htrans==NSEQ |-> addr_align);
```

When we test the VIP configurations represented by SVA against the design modules, we run the individual design module alone. We have to add appropriate constraints to the inputs of the design module. In addition, some design modules are programmed by software to set the default values of burst size, burst type and etc. When we run formal, we have to add SVA assumptions to constrain those variables. Here are the examples.

1. Burst size is less than double word

```
Assume_burst_size: assume property (burst_size <DOUBLEWORD);
```

2. Burst type is only INCR

```
Assume_burst_type: assume property (burst_type == INCR);
```

The SVA assertions are in separate files and connected to the design modules using SVA bind command.

```
 module AXI_assertions (clock, resetn,…);
 …
endmodule

bind AXI_design1 AXI_assertions AXI_inst(.clock(aclk), .resetn(aresetn);
```

IV. RUN FORMALVERIFICATION

After we implement the SVA assertions, we create the script to run Questa Formal. Here is an example of Makefile for submitting formal jobs.

```
Run: compile formal
compile:
            vlog –f flist;
formal:
            zin_qformalq { -J shortjob} –c –od log –do “ \
            do directives.tcl; \
            formal compile –d dut; \
            formal verify –init init_file; \
            exit”
```

The *flist* file defines the RTL files and SVA files needed for the run. The *directives.tcl* file defines the clocks and reset pins, constant signals, cut points and etc. The *init_file* file defines the initial sequence. The tool doesn't require users to define clocks, resets and initial sequence. Without users' definition, the tool can extract clock pins and reset pins from the design, generate clocks of posedge 50% duty cycle, and minimal number of cycles needed to reset the design.

For running formal, we often set some input ports to constant values. Note that the compilation process deletes the DUT logic that is made irrelevant by the constant signal definitions and formal model becomes much smaller which can improve run convergence. We usually disable test mode logic and scan logic during formal runs. Here is an example of *directives.tcl*.

```
netlist clock aclk –period 10
netlist constraint aresetn –value 1’b1 –after_init
netlist constant test_mode 1’b0
netlist constant scan_mode 1’b0
```

Here is an example of initial sequence. For our runs, the initial sequence is simple. However you can define complicated initial sequences. The tool supports multiple ways of defining initial sequences.

```
$default_input_value 0
aresetn = 0
##4
```

After run, the tool reports proven or fired for assert properties, and covered or uncoverable for cover properties. We can analyze the waveforms of the counterexamples of fired properties and check the sequences of covered properties in GUI. Here is an example of the GUI. The waveform window shows that the burst length 9 (awlen==8) is covered. The *Start* signal marks the start of the trace, and the *Cover* signal marks the time when the cover property is covered. At the time marked by the *Cover* signal, (*awvalid && awlen==8*) representing burst length 9 occurs.

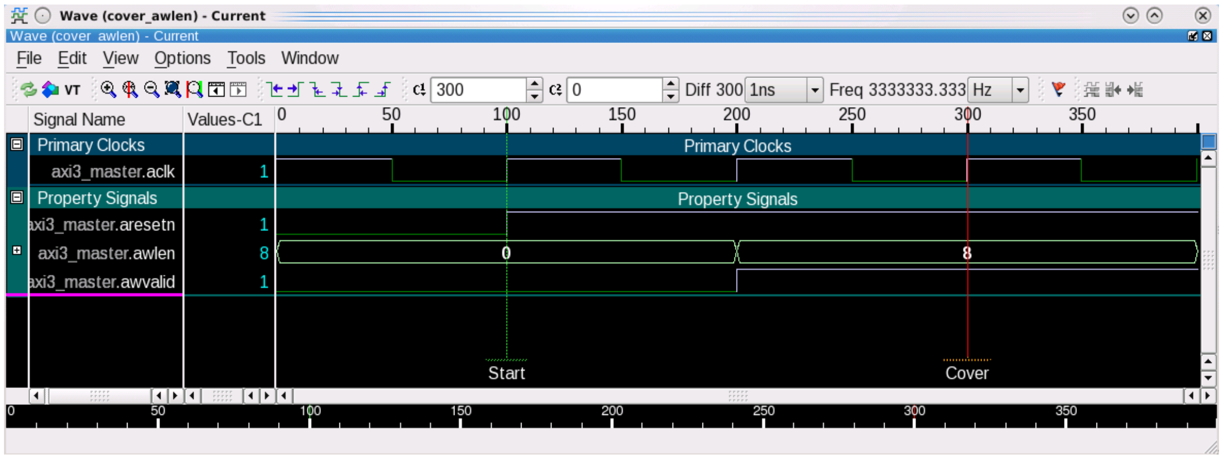


Figure 2: Questa Formal GUI

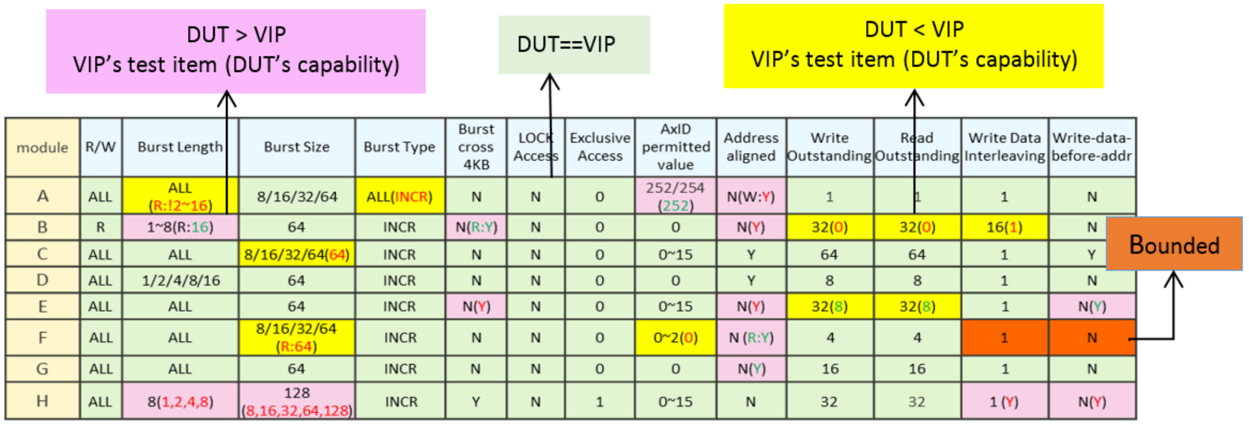
We discuss the fired properties and uncoverable sequences with the designers to figure out which ones can be ignored and which ones are the real errors that have to be fixed.

V. Result Analysis

To compare the configurations of each VIP instantiation with the substituted design module, we create a table as follows. This example table does not contain our whole results. The first row lists all configuration types. The first column lists all design modules. The values outside of brackets are the VIP configurations, the values inside brackets are the actual functions of the designs. When VIP configurations are matching the design functions, there is no need to show the brackets. More details about the table include:

- Items with green background are the matching ones between the design module and its substitute VIP.
- Items with pink background are the ones that the design module has more functionalities than its substitute VIP. This is high priority since the VIP is over-constrained and simulation coverage for the bus fabric is not enough that might hide bugs in the bus fabric.
- Items with yellow background are the ones that the design module has less functionalities than its substitute VIP. This is low priority since simulation tests more for the bus fabric that might run into false alarms.
- Items with orange background are the ones that formal cannot finish run.
- The values outside the brackets are the spec for VIP settings, the values inside the brackets are the mismatching results of the design module. E.g., the item of module A's burst length is 'All (R:!2~16)', which means that the VIP is set to support all burst length for both read and write, but the read operation of module A doesn't support burst length 2 to 16. The item of module A's burst type is 'ALL(INCR)', which means that the VIP is set to support all burst types, but module A only supports type INCR.
- Red colored ones inside brackets are real errors, green colored ones inside brackets are the mismatches that can be ignored.

Table III: Comparison of DUT and VIP


DUT > VIP
VIP's test item (DUT's capability)
DUT==VIP
DUT < VIP
VIP's test item (DUT's capability)
Bounded

module	R/W	Burst Length	Burst Size	Burst Type	Burst cross 4KB	LOCK Access	Exclusive Access	AxID permitted value	Address aligned	Write Outstanding	Read Outstanding	Write Data Interleaving	Write-data-before-addr
A	ALL	ALL (R:!2~16)	8/16/32/64	ALL(INCR)	N	N	0	252/254 (252)	N(W:Y)	1	1	1	N
B	R	1~8(R:16)	64	INCR	N(R:Y)	N	0	0	N(Y)	32(0)	32(0)	16(1)	N
C	ALL	ALL	8/16/32/64(64)	INCR	N	N	0	0~15	Y	64	64	1	Y
D	ALL	1/2/4/8/16	64	INCR	N	N	0	0	Y	8	8	1	N
E	ALL	ALL	64	INCR	N(Y)	N	0	0~15	N(Y)	32(8)	32(8)	1	N(Y)
F	ALL	ALL	8/16/32/64 (R:64)	INCR	N	N	0	0~2(0)	N (R:Y)	4	4	1	N
G	ALL	ALL	64	INCR	N	N	0	0	N(Y)	16	16	1	N
H	ALL	8(1,2,4,8)	128 (8,16,32,64,128)	INCR	Y	N	1	0~15	N	32	32	1 (Y)	N(Y)

When we see a VIP configuration is a subset of the functions supported by the standard IP protocol, we need to add additional properties for the uncovered functions by the configuration to prevent that the VIP is over-constrained. Here are the few examples we did in our project.

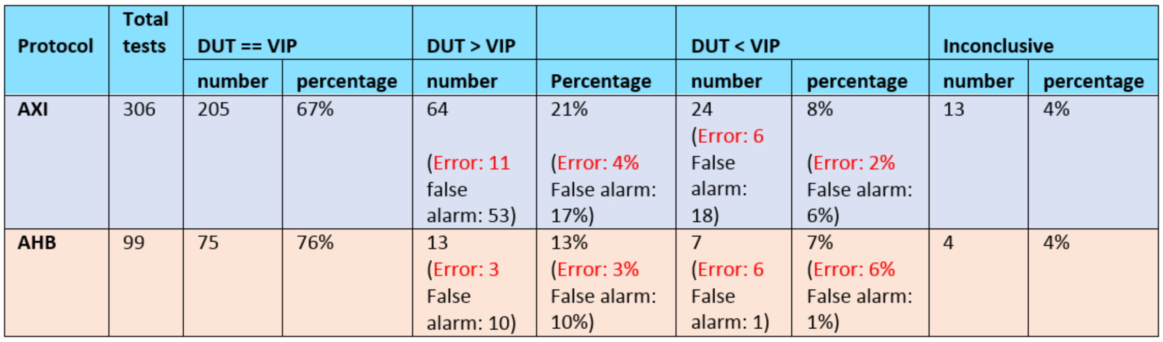
Example 1: The designer said that that module H only supported burst size 128. The substitute VIP for module H was setup to only support burst size 128, which was different from AXI standard supporting burst size 1/2/4/8/16/32/64/128. To prevent that the VIP was not over-constrained, we wrote extra cover properties to cover burst size 1/2/4/8/16/32/64 in addition to the cover property of burst size 128. We applied the SVA assertions to module H and ran formal tool. Formal proved that module H supported burst size 8/16/32/64/128. The substitute VIP only generated busrt size 128 and might hide bugs in the Bus Fabric that only occurred when burst size was not 128. We fixed the VIP and let it allow burst size 8/16/32/64/128.

Example 2: The designer said Module A could have all burst types. The substitute VIP was set to have all burst types in simulation environment. We wrote the SVA cover properties to cover the burst type 'FIXED', 'INCR', 'WRAP'. We applied the SVA assertions to Module A and ran formal tool. Formal tool proved that burst type 'INCR' was covered, but 'FIXED' and 'WRAP' were not coverable. The substitute VIP had more functionalities than the design module A and might result in false firing in the Bus Fabric. Then we fixed VIP settings to only have burst type 'INCR'.

Example 3: The designer said that module E didn't support burst crossing 4kb boundary. The substitute VIP of module E was set not to support burst crossing 4kb boundary. We wrote SVA assert property to check it. Formal tool proved that module E supported burst crossing 4kb boundary. The error in substitute VIP was fixed by setting it to support burst crossing 4kb boundary.

We applied this methodology on a smart phone project for 18 AXI Designs and 9 AHB Designs. We tested 17 configurations of AXI VIPs for 18 design modules. The total tests for AXI were 17x18=306. We tested 11 configurations of AHB VIPs for 9 design modules. The total tests for AHB were 11x9=99. The results were as follows. The real errors were marked by red color.

Table IV: Test Results for AXI and AHB VIPs



Protocol	Total tests	DUT == VIP		DUT > VIP		DUT < VIP		Inconclusive	
		number	percentage	number	Percentage	number	percentage	number	percentage
AXI	306	205	67%	64 (Error: 11 false alarm: 53)	21% (Error: 4% False alarm: 17%)	24 (Error: 6 False alarm: 18)	8% (Error: 2% False alarm: 6%)	13	4%
AHB	99	75	76%	13 (Error: 3 False alarm: 10)	13% (Error: 3% False alarm: 10%)	7 (Error: 6 False alarm: 1)	7% (Error: 6% False alarm: 1%)	4	4%

For the AXI VIPs, 205 configurations (67% of total AXI tests) matched design behaviors. 64 configurations (21% of total AXI tests) were over-constraining the VIPs where the VIPs had less functions than the design modules. However after diagnosing the 64 mismatching ones and discussing them with the designers, we narrowed down the real errors to 11, and the other 53 were false alarms due to lack of constraints for the design modules. The real errors of over-constraining the VIPs might hide bugs in the bus fabric. 24 configurations (8% of total AXI tests) were under-constraining the VIPs where the VIPs had more functions than the design modules. The designers confirmed that 6 were real errors and the rest were false alarms. The real errors of under-constraining the VIPs might result in false firing during bus fabric verification. The total errors were 17 and 6% of the total AXI tests. The total false alarms were 71 and 23% of the total AXI tests. There were 13 inconclusive (bounded proof) results that was 4% of the total AXI tests. It was OK for us to have some inconclusive tests. We asked the designers to double check the information they provided for the inconclusive ones.

For the AHB VIPs, there were 75 configurations (76% of the total AHB tests) matching design behaviors, 3 real errors of over-constraining the VIPs that were 3% of the total tests, and 6 real errors of under-constraining the VIPs that were 6% of the total tests. The total errors were 9 and 9% of the total tests. The total false alarms were 11 and 11%. The total inconclusive tests were 4 and 4% of the total AHB tests.

It took us 3 days to implement the SVA assertions. It only took us 2 days to setup the formal environment for 27 design modules including binding the SVA assertions to all design modules, which was 20x faster than simulation because there was no need to write testbench for using formal. Instead, we used a simple Perl script to setup the formal environment automatically. The script generated the tool directive commands and Makefiles to run the formal tool, based on the information users provided, such as design module name, clock signal name, reset signal name and constant signal name. It took us 1 week to get all the results in the table. 26 real errors were found in the configurations of the VIPs that could produce incorrect verification results of the bus fabric.

This above example indicates that formal verification is an effective way to verify the correctness of the constraints used in simulation environment. Without formal tool, it would have been impossible to find out that a VIP has more functionalities than the substituted design module since only formal could mathematically prove that the extra functions in the VIP were not coverable in the design module with any stimulus.

VI. Future Work

For the project presented in Table IV, there were some false alarms due to insufficient constraints to the design modules. Adding all constraints for the inputs is difficult since they must follow multiple interface protocols, and in general it is time consuming for us to write SVA assertions to cover the whole protocol(s). We are not sure if our assertions are accurate, complete and efficient for formal. This is the main reason we haven't used formal analysis to fully verify AXI and AHB design modules. Instead we have been using simulation to verify these design modules, and only using formal analysis to verify the configurations of simulation VIPs which is simpler than to verify the full sets of the protocol functions.

However, Mentor has formal-friendly assertion IP libraries for the AXI and AHB protocols (the Questa Formal AMBA Library). Since QFAL libraries can be used as checkers and assumptions, for running a master design module we can use QFAL slave library as assumptions since many inputs of the master module are outputs of slave module. For running a slave design module we can use QFAL master library as assumptions as well. Using QFAL may help us reduce false alarms and improve run performance in addition to save our time to code SVA assertions. Since QFAL has the complete sets of assertions covering the AXI and AHB protocols, not only it can help us verify the VIP configurations, but also the design modules themselves. From our positive experience of running formal to check VIP configurations, we want to expand formal usage to verify the whole AXI and AHB design modules in the future.

References

[1] ARM AMBA AXI Protocol Specification.
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html

[2] ARM AMBA AHB-Lite Protocol Specification.
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022b/index.html

[3] Ram Narayan, Tom Symons, "I created the Verification Gap", DVCon 2015.
http://events.dvcon.org/2015/proceedings/papers/10_1.pdf

[4] Joe Hupcey, Mark Eslinger, Doug Smith, "Back to Basics: Doing Formal the Right Way", DVCon 2016.
https://dvcon.org/content/event-details?id=199-9-T

[5] Roger Sabbagh, Mark Eslinger, Ram Narayan and et al, "Formal Verification in Practice: Technology, Methodsand Applications", DVCon 2015.
https://dvcon.org/content/event-details?id=163-11-T

[6] Blaine Hsieh, Stewart Li, Mark Eslinger, "Every Cloud - Post-Silicon Bug Spurs Formal Verification Adoption", DVCon2015.
http://events.dvcon.org/2015/proceedings/papers/11_3.pdf