Catching the low hanging fruits on intel® Graphics Designs

Arbiter FV

M, Achutha KiranKumar V, (achutha.kirankumar.v.m@intel.com)

Aarti Gupta (aarti.gupta@intel.com)

Bindumadhava S S, (bindumadhava.ss@intel.com)

Savitha Manojna (savitha.manojna@intel.com)

Abhijith A Bharadwaj (abhijith.a.bharadwaj@intel.com)

Intel India Technologies Pvt Ltd, Bangalore, India.

***Abstract*—Any digital electronic design comprising of shared resources guarantees the presence of arbiters, which skillfully delegates resource access. Discrepancies in the intended behavior of such components would result in contention and starvation, which stresses the need for robust validation. The inadvertent design space growth restricts traditional simulation methods from exploring all the scenarios that complex arbiters would present. The dependable solution is exhaustive verification of the arbiter using Formal techniques. Formal Verification (FV) is widely acknowledged for improving validation effectiveness. FV being a significant left shift from the norm, the hype about the barrier to entry into formal and difficulty in coding the appurtenant set of properties is profuse. Consequently, Intel® graphics team has taken a goal to embrace FV in most of its verification efforts. To lower the seemingly high barrier, a 'cookie cutter' library set of formal verification friendly properties has been developed for one of the low hanging areas of formal application – the Arbiters. All the arbiters in our Graphics Technology were analyzed and the property set was developed and integrated into a GUI for ease of use. This paper summarizes the fruitful efforts of our formal methodologies in shaping up a stronger verification environment for Intel® graphics.**

Keywords—Formal, Formal Property Verification, Arbiters, Assertion, Coverage, FPV application, Library

I. Introduction

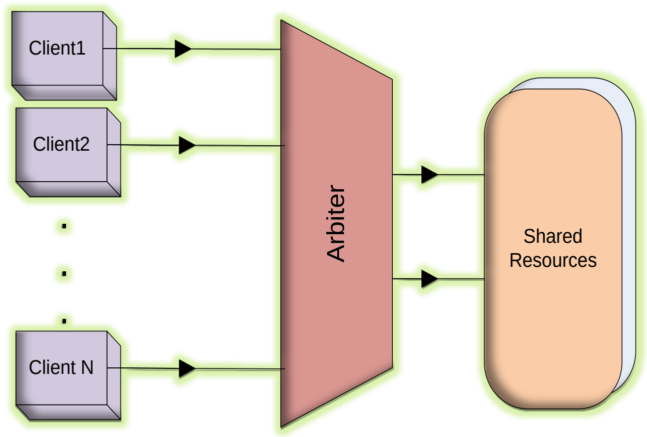


Figure 1. Arbiter block diagram

Arbiters are one of the main components in digital electronic designs, used mainly in the presence of shared resources to avoid contentions and starvation by intelligently restricting access by the contenders. It wouldn't be surprising to note that there wouldn't be any digital design without the use of arbiters. While maximizing the utilization of the common resources, the arbiters also help in maintaining constant throughput and fairness in the system. Millions of simulation cycles wouldn't guarantee the complete functional correctness of the arbiter. Formal verification has proved its mettle in this kind of arbitration schemes and has been the most preferred choice to verify. The basis of the formal property verification lies in defining the correct property set, which would need to specify the behavior of the arbitration. Due to a wide variety of arbitration mechanisms available in practice, it is also challenging to come up with a common set of properties for all. Hence a reusable property library needs to be defined where the designer can choose a set that is applicable for their designs.

II. APPLICATION

The Intel team recognized a plethora of arbitration schemes while analyzing a couple of hundreds of arbiters in Intel® Graphics Technology (GT). Regardless of the schemes and their complexity, some properties were found to be consistent across, with scheme pertinent properties differentiating the designs. This knowledge allowed the property set being developed to be contrasted into generic and arbitration specific properties. Along with the generic properties, targeted properties to exhaustively verify some ubiquitous arbitration schemes i.e. round robin based and priority based arbitration schemes were also included.

Another issue to be considered is when an arbiter is carved out from an environment to be solely verified, there would be a need of certain tie-offs to simulate the environment. The environment would certainly differ with every arbiter, but here too some generic tie-offs were identified, such as generic expectations about the client behavior. These were also added to the library.

A well-established fact in any formal verification environment is the importance given to cover points. Over and above these properties (assumptions and assertions), certain set of covers were added to the library. These covers were strategically developed to toggle a significant chunk of the design being verified.

There has been a similar effort that went during OVL timeframe, which is not completely formally friendly. We explored our methodology and as a mark of completion, we embarked on looking at alternative options and found out the OVL library set, the properties that were defined for a similar purpose.

This effort enabled the team to strike gold with a 'cookie cutter' set of properties resulting in an exhaustive arbiter property library. The designers would be given freedom over choosing the best set of properties germane to their verification plan. The verification library comprises of the following list of properties:

Table 1. Arbiter library properties

Arbiter library properties		
Sl.no	***Name***	***Description***
	Generic assertions	
1	ASSERT_ONE_REQ_GNT	When there is only one request, it should be provided with a grant
2	ASSERT_GNT_ONLY_ON_REQ	There should be a grant only if the corresponding request has been placed
3	ASSERT_ONE_GNT_PER_REQ	There should be only one grant issued per client
4	ASSERT_ONE_GNT_AT_A_TIME	Only one grant is given at any point of time, even when there are multiple requests
5	ASSERT_GNT_PRESERVED_TILL_DATA_XFER	Grant preservation till data transfer
6	ASSERT_LIVENESS	Liveness Check: For each request, a grant must be eventually received
	Multiple output ports arbiter properties	
7	ASSERT_ONLY_ONE_GNT_PER_CLIENT_ACROSS_PORTS	There should be only one grant per client across all output ports
8	ASSERT_GNT_ON_ALL_PORTS	If there are more requests than the number of grant ports, all ports should have a grant
	Round Robin Properties	
1	ASSERT_RR_GNT_WITHIN_N_CYCLES	Grant should be provided within N cycles
2	ASSERT_RR_FAIRNESS	Round Robin Fairness Check
3	ASSERT_RR_ARBITRATION	Round robin arbitration grant check
	Priority Arbiter Properties	
1	ASSERT_PRI_ARBITRATION	Higher priority client should always get the grant
2	ASSERT_PRI_REQ_GNT	If there are no requests of priority higher

		than the current client, the grant should be provided to the current client
	Constraints/assumptions	
1	ASSUME_REQ_VALID_TILL_GNT	Request needs to stay high till it is acknowledged with a grant.
2	ASSUME_ARBPARAMS_VALID_TILL_GNT	Any arbitration parameter that has to be valid till grant can be used here
3	ASSUME_NO_REQ_ON_HOLD	There should be no request when hold is high
4	ASSUME_WINNER_AS_PREV_CYCLE_GNT	This property is useful for a round robin arbiter designed reusing priority arbiter design and needs input from another module indicating the last winner.
	Covers	
1	COVER_INPUT_REQUESTS	Cover for different number of input requests
2	COVER_EACH_REQUEST_TOGGLES	Cover to check each input is toggling
3	COVER_EACH_GNT_SEEN	Cover to check each output is toggling

III. Enhancing the Usage

The visibility of the library developed was then enhanced by integrating it into a GUI. The users now would only have to select the required property and feed in the necessary information such as the signals on which the properties would act. The properties chosen would be added to a separate file which would be bound to the design. The GUI was given abilities to, at the click of a button, initiate the formal verification setup, automatic addition of the properties selected and binding to the design, all of which would be run in the back-ground.

The GUI road was taken in order to achieve two important things. Firstly, automating the property addition reduces syntactical and contextual errors. The GUI would present each property chosen with a succinct description, avoiding the mishap of faultily choosing a property.

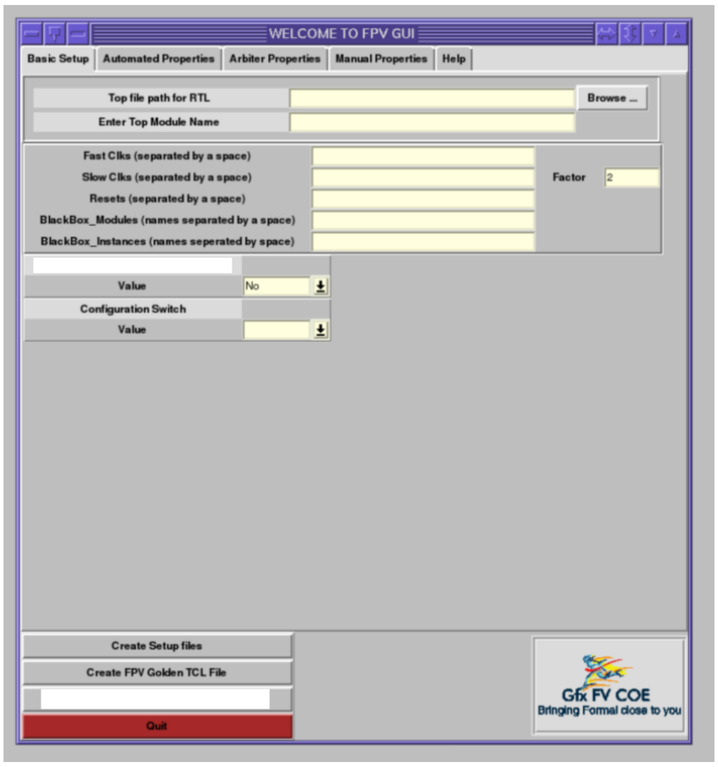
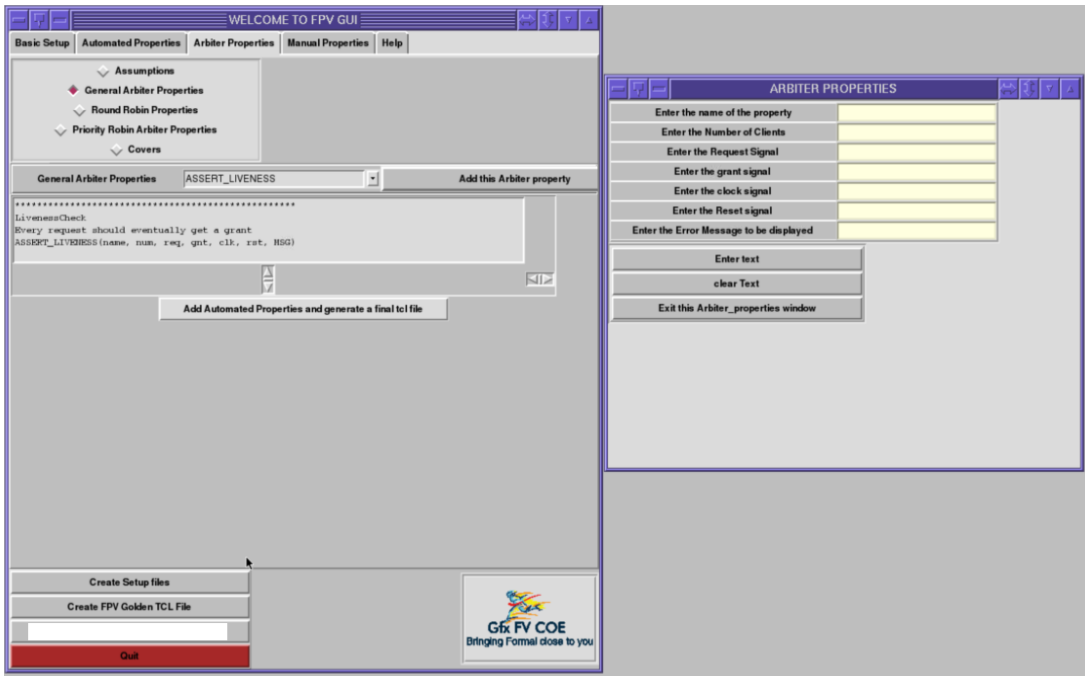

WELCOME TO FPV GUI
Basic Setup
Automated Properties
Arbiter Properties
Manual Properties
Help
Top file path for RTL
Browse ...
Enter Top Module Name
Fast Clks (separated by a space)
Slow Clks (separated by a space)
Factor
2
Resets (separated by a space)
BlackBox_Modules (names separated by a space)
BlackBox_Instances (names seperated by space)
Value
No
Configuration Switch
Value
Create Setup files
Create FPV Golden TCL File
Quit
Gfx FV COE
Bringing Formal close to you

Figure 2. GUI Design Select Page

Secondly, the GUI would make the initial stages of a formal environment setup more approachable to beginners, while giving a clear sense of direction on initiating a formal environment set up for any future projects.

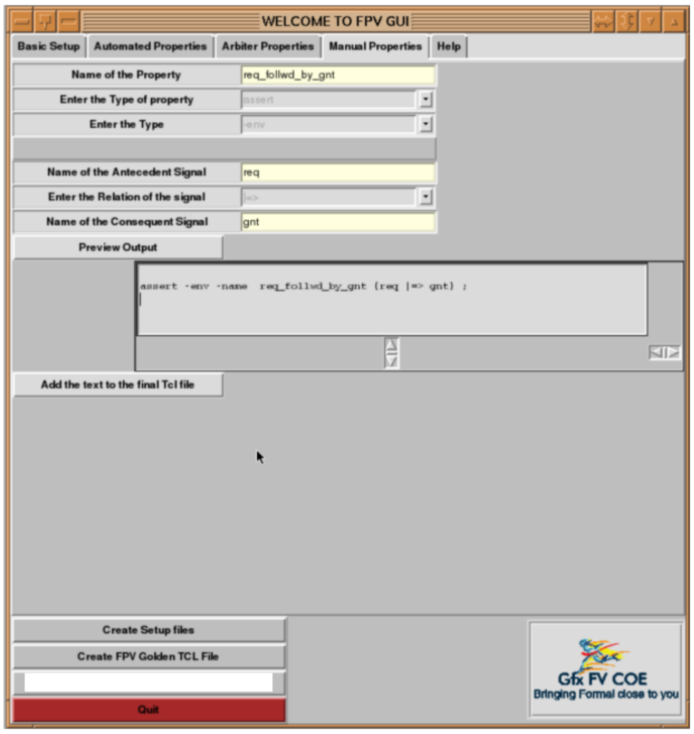

WELCOME TO FPV GUI
Basic Setup
Automated Properties
Arbiter Properties
Manual Properties
Help
Assumptions
General Arbiter Properties
Round Robin Properties
Priority Robin Arbiter Properties
Covers
Add this Arbiter property
Add Automated Properties and generate a final tcl file
Create Setup files
Create FPV Golden TCL File
Quit
Gfx FV COE
Bringing Formal close to you
ARBITER PROPERTIES
Enter the name of the property
Enter the Number of Clients
Enter the Request Signal
Enter the grant signal
Enter the clock signal
Enter the Reset signal
Enter the Error Message to be displayed
Enter text
clear Text
Exit this Arbiter_properties window

General Arbiter Properties: ASSERT_LIVENESS

```
***************************************************
LivenessCheck
Every request should eventually get a grant
ASSERT_LIVENESS(name, num, req, gnt, clk, rst, MSG)
```

Figure 3. GUI Arbiter Library property selection

The GUI was also made generic enough to enable the user to code any required custom property. To achieve this, the user could go through the manual property tab and code the assertions by choosing the relevant expressions from the dropdown lists. This act helped the designers to define their own properties, both assumptions and assertions, while the syntax hurdles were easily avoided. Even with the Arbiter library in place, issues with the regular usage of the assertion library were seen as the designers were not initially aware of the full potential of the library set and its ease of usage. Being able to write the property solo raised the comfort level of the designers, as they could define their own assertions, but they would face the problem of wrong syntax. The GUI solved all those issues by providing the correct syntax and eased the property coding and lowered the barrier to entry to formal property verification.



Name of the Property	req_follwd_by_gnt
Enter the Type of property	assert
Enter the Type	-env
Name of the Antecedent Signal	req
Enter the Relation of the signal	|=>
Name of the Consequent Signal	gnt

```
assert -env -name  req_follwd_by_gnt (req |=> gnt) ;
```

Figure 4. Manual Property Entry

The arbiter library was also needed to be applicable to arbiters with data-paths. The complexity of a data-path intense logic is predominantly due to the presence of data-processing components like synchronous/asynchronous FIFOs, arbiters, serial to parallel converters and so forth. The validation data-space for such compute intensive blocks would be humongous. Formal scoreboards are optimized to address these convergence issues better than the traditional FPV approach. In lieu of these requirements, with the standard property set, support for scoreboard addition to validate data consistency was also added.

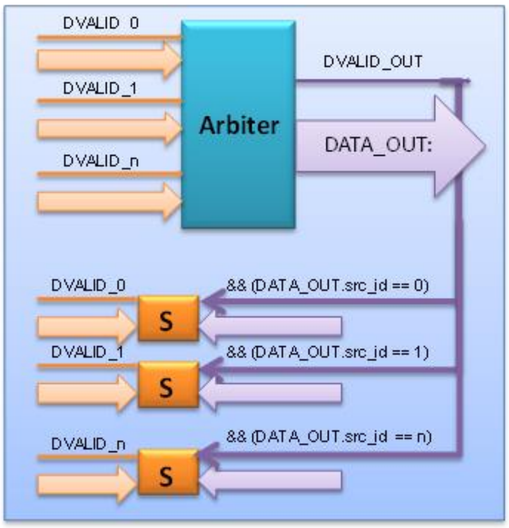


Figure 5. Scoreboards on Arbiter

Data integrity checks for buses have been optimized by exploiting its inherent symmetry. Ideally, in the formal environment, a free variable used as an index would cover the entire width of the bus. For buses of large widths, this range can be split into smaller, more manageable chunks for swift convergence. The GUI script has a provision to automatically create these ranges and define a free variable for bit selection.

IV. Results

The productiveness of the Arbiter library can be evaluated by examining two categories; Effectiveness in catching both shallow and deep bugs and design space coverage. The evaluation of both criterion are as follows.

A. *Bug catching effectiveness*

Using the library of assertions, the team was able to catch both shallow bugs and deep corner cases fairly quickly. Two scenarios are presented here, one in which a shallow bug is caught and another which shows a much deeper bug being caught.

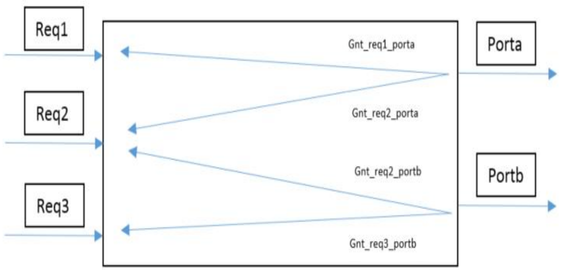


Figure 6. 3:2 Round Robin Arbiter

The arbiter under consideration has 3 requestors; Req1, Req2 and Req3. The requestors would be granted across 2 ports, only when the ports were available. Req1 would get a grant on port1 only, when p1 is available. Req3 would get a grant on port2 only, when port 2 is available. Req2 could get grant on either port1 or port2, on round robin basis. And on the top, there was a round robin scheme present in between Req1, Req2 and Req3.

<u>Scenario 1:</u> In this scenario, the failure is as follows.

- First, only Req1 comes high. It is granted immediately in porta, as shown by Gnt_req1_porta.
- In the next cycle, Req2 and Req3 comes high. On round robin basis, Req2 is granted in

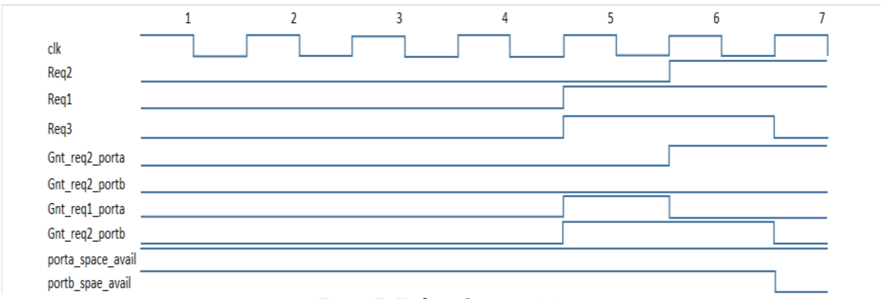


Figure7. Failure Scenario1 1

porta, as indicated by Gnt_req2_porta.

- In the next cycle, both Req2 and Req1 are asserted, while Req3 is de-asserted. Now, a fair round robin arbiter would have granted Req1, but Req2 is granted, as shown by Gnt_req2_porta. This is a valid failure.

Scenario 2:

This is a much more difficult scenario which requires one to delve much deeper into the design. This is a proper definition for 'corner case scenario', which is next to impossible for DV to catch.

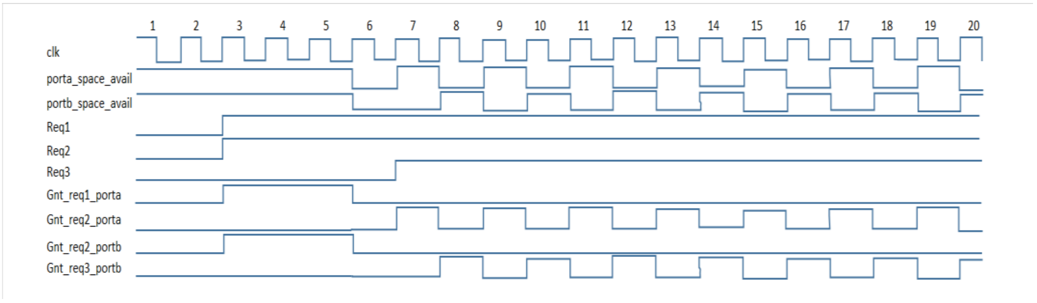


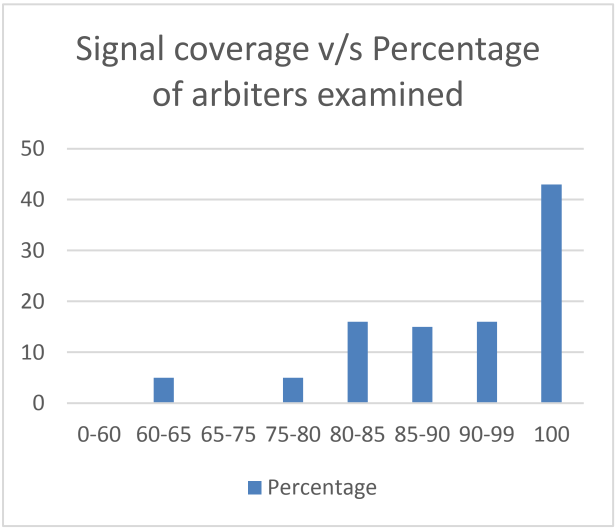
Figure 8. Failure waveform 2

The scenario was as follows:

- First few requests are serviced such that output FIFO gets full and both output ports become unavailable.
- After both ports become unavailable, all 3 clients assert request, and these requests are high continuously
- Output ports (port a & port b) start becoming available alternately (depending on output FIFO read conditions)
- Whenever porta becomes available, arbiter is giving grant to Req2 and whenever port b becomes available, arbiter gives grant to Req3. Req1 requests are totally ignored.

B. *Design space coverage*

The effectiveness of a library of assertions will also depend on the code coverage. The library should be prudently crafted in order to cover most of the code. The coverage from one arbiter to another would indubitably differ owing to the design discongruities, but at the end of the day should be substantially high. The following coverage data was obtained while exercising the arbiter library on 20% of the total arbiters in the GT.



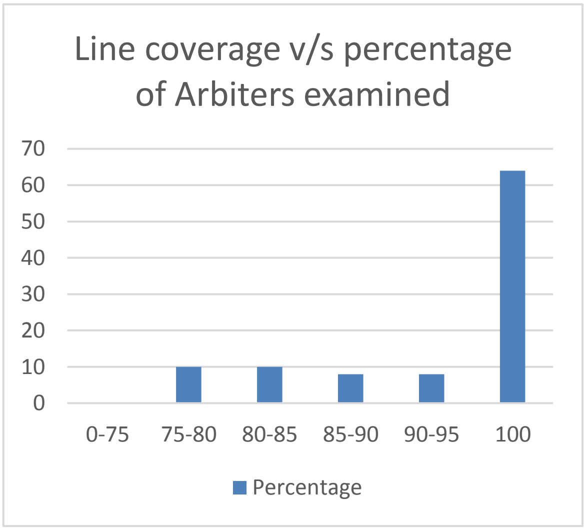


Figure 9. Coverage on Arbiters

The first graph shows the distribution of signal coverage obtained while exercising the Arbiter library over 20% of the total arbiters in the GT. It can be seen that the library proves itself in toggling most of the logic present in the design. 43 percent of the total arbiters verified gave 100 percent signal coverage.

The second graph gives the distribution of line coverage result obtained from the arbiters verified using the Arbiter library. It can be seen that total line coverage was obtained in 64% of the arbiters verified.

In most of the cases, it was seen that the arbiters gave lesser signal or line coverage due to chunks of logic in the code that were unrelated to the arbitration scheme considered, such as hold, arbitration reset, arbitration disable and/or DFT related signals which would be disabled/unused during the arbitration scheme verification.

V. Conclusion

The key take-away that the team arrived at from their experience in deploying the Arbiter library is as follows:

- The completeness guarantee offered by FV using the Arbiter library increased the overall verification confidence.
- The cookie cutter set coupled with the graphical interface lowered the barrier of formal embrace.
- The property library was holistically complete to comprehend most of the scenarios and is constantly augmented with new requirements posed.
- One year goal was taken to cover all Arbiters in the design through formal property verification.
- Future work involves proliferating these FV methodologies to more design units at Intel® and having a wider presence in all the design modules to augment the existing verification.

VI. References

[1] Achutha Kiran Kumar Madhunapantula, Aarti Gupta, and Bindumadhava Singanamalli, “RTL2RTL Formal Equivalence: Boosting the Design Confidence”, DAC 2015.

[2] Achutha Kiran Kumar Madhunapantula, Aarti Gupta, and Bindumadhava Singanamalli, “Formal Verification in Intel® Graphics designs”, DVCON 2015

[3] Erik Seligman, Tom Schubert, M V Achutha KiranKumar, “ Formal Verification: An Essential toolkit for the modern VLSI design”, Elsevier Publications, 2015