The Process and Proof for Formal Sign-off A Live Case Study

IPSHITA TRIPATHI   ANKIT SAXENA   ANANT VERMA   PRASHANT AGGARWAL

Oski Technology
Gurgaon, India and Mountain View, CA, USA

Abstract- **With the advances in formal technology and the introduction of Formal Sign-off Methodology in recent years, it has become possible to create a formal testbench that can find all bugs in a design. However, this is a very new concept that most design and verification teams do not know well enough to adopt and leverage for sign-off. The paper explains in detail the process to achieve Formal Sign-off, focusing on End-to-End checkers, constraints, complexity and formal coverage. In addition, we present a live case study as proof that Formal Sign-off is possible.**

Keywords—formal verification; Formal Sign-off Methodology; End-to-End formal checkers; abstractions; Required Proof Depth; bounded proof; formal coverage.

I. Introduction

Simulation has been used for verification sign-off in the industry for several decades. It is relatively easy to understand the simulation processes, track metrics and measure progress to achieve sign-off, which is often defined as the point when all the pre-determined coverage goals are reached. However, it is also well known that reaching the simulation sign-off point does not guarantee that there is no bug left in the design. Often corner case bugs are discovered after tapeout, simply because it is not possible to simulate today's complex IC designs exhaustively.

Formal technology has increasingly been adopted in the verification flow in recent years to complement simulation, because of its ability to exhaustively verify the design-under-test (DUT). Based on state space search algorithms, formal can cover all state transitions, and can thus exhaustively prove the functional correctness of designs, by finding complex corner case scenarios that are often harder to cover with simulation. However, formal technology is not without challenges. Earlier attempts to apply formal verification to real world designs have been hindered by the size and complexity of the designs formal tools can handle. Without a sound methodology to overcome complexity challenges and offer a way for sign-off, formal application is limited to automatic formal check, formal apps and assertion-based-verification (ABV) [1] for bug hunting.

End-to-End formal verification [2] and the introduction of Formal Sign-off Methodology brought about a turning point in formal application – formal can now be used for sign-off. Designs signed off using formal verification offer much higher quality assurance compared to those that used simulation sign-off, due to formal's exhaustive nature. However, the process of Formal Sign-off is complex and highly iterative. It takes dedicated effort and requires specific skills to make an executable formal test plan, implement the formal testbench, resolve complexity, and measure formal progress to reach sign-off. The benefit of the extra effort to achieve Formal Sign-off is the guarantee that, when done properly, a formal testbench catches all bugs in a design. To provide a live case study that formal sign-off is indeed possible, we hosted the "Break the Testbench" challenge at DAC 2015 where we invited DAC 2015 attendees to insert functional bugs in a design, chosen beforehand, and watch the bugs being caught by a formal testbench developed earlier by the Oski team. The results demonstrated that it is possible to achieve Formal Sign-off guaranteeing zero bugs in a design. For this paper, we use the term "DAC challenge" to refer to the "Break the Testbench" challenge at DAC 2015.

In this paper, we describe the process of achieving Formal Sign-off using the DAC challenge design and formal testbench as a case study. The paper is organized as follows. Section II presents the Formal Sign-off Methodology and discusses the requirements for Formal Sign-off. Section III describes the design we used in the challenge. Section IV discusses the formal testbench implementation including End-to-End checkers, constraints used and the complexity solution. Section V includes some bugs that were introduced during the DAC challenge. Section VI offers concluding remarks.

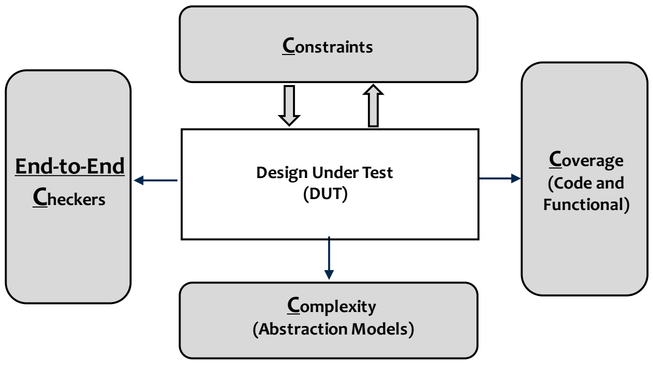

Constraints
End-to-End Checkers
Design Under Test (DUT)
Coverage (Code and Functional)
Complexity (Abstraction Models)

Figure 1. Oski Formal Sign-off Methodology

II. Formal Sign-off Methodology

A methodology is a set of system of methods, principles, and rules for regulating a given discipline. The Formal Sign-off Methodology specifically is a set of systematic methods and procedures applying formal technology to verify the functional correctness of integrated circuit design blocks in order to achieve sign-off.

A Formal Sign-off Methodology centers on 4 Cs – Checkers, Constrains, Complexity and Coverage [4], as shown in Figure 1. All formal sign-off projects follow three distinct stages – formal test planning, formal testbench implementation and formal sign-off. Each stage involves all four components in different ways.

A. Formal test planning

This is the first and a critical step in the methodology. The formal test planning processes include 1) IDENTIFYing the right blocks to apply formal on, 2) EVALUATing the design metrics to determine formal verification effort on the chosen blocks, and 3) PLANning English list of important checkers and constraints to write, with estimates of rough target for the Required Proof Depth [3], while identifying where one might need to use complexity resolving techniques to overcome complexity challenges, and how one will measure coverage to achieve sign-off. Each of the IDENTIFY, EVALUATE and PLAN processes require not just formal skills, but also intimate design knowledge, and most importantly experiences from past Formal Sign-off applications. A good formal test plan leads the subsequent testbench execution on a path that uses the least amount of time & resources, avoids potential problems, and obtains the best return-on-investment (ROI) for the project. The effort spent in thorough and thoughtful formal test planning cannot be overestimated.

B. Formal Testbench Implementation

This is the execution phase of the defined formal test plan and requires the bulk of the formal verification effort. It involves implementing the End-to-End checkers and corresponding constraints, debugging counter-examples, refining the Required Proof Depth based on observed failures and coverage data, crafting complexity-solving techniques if needed, and tracking formal progress. This is where good formal skills in writing clean and efficient formal test bench codes (in Verilog, SystemVerilog and SystemVerilog Assertions [5]), understanding in-use formal tool features, capabilities and limitations, and knowing where certain complexity techniques can be applicable become extremely helpful.

A formal testbench implementation process is very iterative in nature, much more so than simulation because of two factors: 1) Formal verification is exhaustive and one checker can find more than one failure scenario in the design; 2) Formal reports shortest-distance failures first so debugging effort can be reduced and focused. As a result, it is not uncommon for one specific checker failure to be fixed before another failure is reported for the same checker, but at a deeper depth. However, the debugging process is invaluable as it enables a deeper understanding of the design, revealing all corner-case bugs and sometimes leading to a refinement of checkers and constraints.

On a weekly basis, formal progress should be tracked to record the number of checkers and constraints implemented, as well as the percentage of those implemented to the planned list of checkers and constraints; the passing, bounded-pass (those reached the Required Proof Depth and those that did not) and failing checker percentage; the number of over-constraints; and formal coverage. While all the tracked data will demonstrate the trend of formal testbench development, it is important to note that many factors will affect the trend, especially if formal testbench development parallels RTL development effort. Such factors include:

- RTL code changes, such as when new features are added, or when an RTL bug is fixed;
- Formal testbench changes, such as when new checkers are implemented, or certain over-constraints are relaxed, or when an Abstraction Model is added;
- Changes in formal run setup, such as using a different proof engine, running for longer time.

As a result, it is not uncommon to see dips in a generally upward trend. Despite these factors, the tracking of formal progress is important to gauge how far away the formal testbench is from sign-off.

C. *Formal sign-off*

This is the final stage that ties everything together to determine if the formal testbench has reached sign-off status. For formal testbench to be complete, one must answer three basic questions:

1. Is my list of End-to-End checkers complete?
2. Do I have unintentional over-constraints?
3. Have all my checkers reached the Required Proof Depth?

The answers to the three questions should have been gathered throughout the previous stages. Ensuring a complete list of End-to-End checkers involves reviewing the list with designers, making sure each output has a corresponding End-to-End checker, reaching 100% formal coverage and making use of negative testing to ensure any functional changes can be caught with the existing set of the checkers. Ensuring no unintentional over-constraint involves reviewing the list of constraints with designers, as well as using techniques such as cross proof with neighboring blocks, instantiating constraints in simulation and formal coverage. Ensuring all checkers reached the Required Proof Depth involves calculating the correct Required Proof Depth based on a 6-step methodology (this number should have been refined throughout the previous stages) and solving complexity so all checkers either pass, or reach, a bounded pass with bound greater than the Required Proof Depth. If the list of End-to-End checkers is complete, there are no un-intentional over-constraints, and all checkers have reached the Required Proof Depth, then the formal testbench has reached sign-off quality.

The following sections explain the Formal Sign-off process using a multicast crossbar design for the DAC challenge.

III. Challenge Design Overview

The design used during the DAC challenge is a parameterized multicast crossbar. Data transport design blocks like this one are very common in real world IC designs and are good candidates for formal verification. To achieve faster turnaround time live at DAC 2015, we parameterized the design to have 8 clients and 8 targets, as shown in Figure 2.

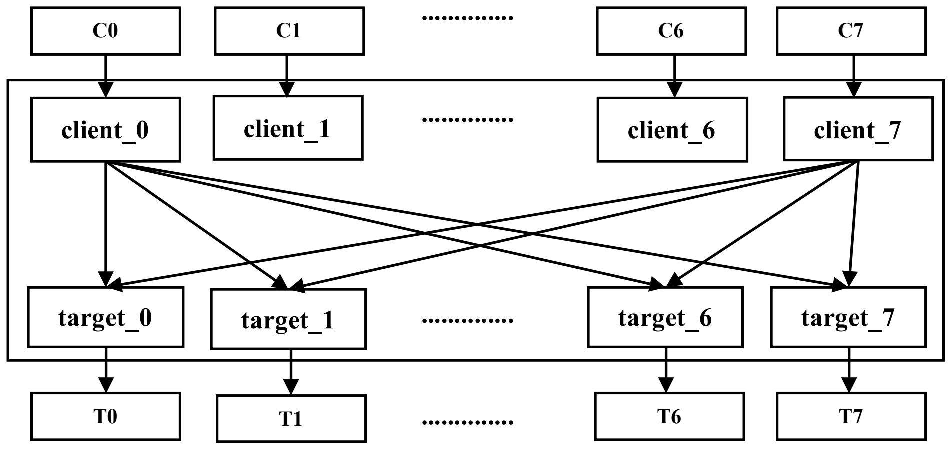


Figure 2. Multicast Crossbar Block Diagram

The requests at each client have one of three priorities attached to it – strict, high or normal. These priorities indicate that given two requests with different priorities for the same target, which request is expected to be served before the other. Based on the priority of the request, the appropriate client is processed and corresponding data is transferred to the desired target(s). Strict priority requests are serviced first, followed by high priority and normal priority requests are served the last.

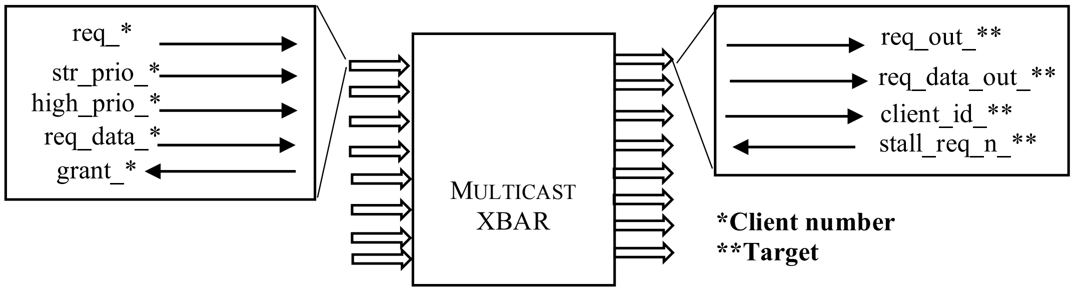


Figure 3. Multicast Crossbar Port List

There can be only 1 strict priority request at a time for a target. There can be multiple high priority requests from different clients, which are served in a round robin fashion. If the request from a client is neither strict priority nor high priority, then it is a normal priority request. Similar to high priority requests, multiple normal priority requests are granted in a round-robin fashion.

The latency of the crossbar is 1 cycle i.e. a granted request is seen at the desired target 1 cycle after it was seen at the client.

A grant is given to each client after the request has been transferred to all the desired target(s). Thus only one request can be outstanding for a target from a given client at a time. The upstream blocks can apply backpressure to stall data transfer to target(s).

Figure 3 shows the port list of the multicast crossbar. The inputs are:

- Client inputs (* refers to client number and varies from 7 to 0):
  - req_*: vector indicating target(s) to which client requests to send data; vector bit-width is 8 i.e. number of targets
  - str_prio_*: indicates whether client has strict priority or not; only one client can have strict priority at a time
  - high_prio_*: indicates whether client has higher priority or not; more than one client can have higher priority at a time
  - req_data_*: data associated with the client. It is vector and has parameterized bit-width.
- Target input (** refers to client number and varies from 7 to 0):
  - stall_req_n_**: active low external backpressure from upstream blocks for stalling data to a target

The outputs are:

- Target outputs (** refers to target number and varies from 7 to 0):
  - req_out_**: indicates target has valid data
  - req_data_out_**: data at the target interface; corresponding to req_out_**
  - client_id_**: indicates the client id which has sent the data
- Client output (* refers to client number and varies from 7 to 0):
  - grant_*: indicates completion of request(s) i.e. all the targets for which the request was initiated have received the data

IV. FOMAL TESTBENCH IMPLEMENTATION

We followed the three stage process for Formal Sign-off the multicast crossbar design, explained in Section II. During the planning stage, we identified that the important design functionalities are:

- Arbitration correctness – When multiple clients are requesting to send data to the same target, we need to verify that different priority requests are granted in accordance to the arbitration scheme.
- Grant correctness – We need to verify that a grant for a client is only generated once all the desired targets receive the data. Also, we need to prove that no (spurious) grant is generated for a client, unless the client has a pending request.

- Data correctness – We need to verify that the data is correctly transferred to the desired target i.e. data is not modified while it is transferred from a client to a target. And also we need to ensure that no (spurious) data is transferred to the target unless there is a request for a target.

A. End-to-End Formal Checkers

In order to verify the identified functionalities, we implemented the following list of End-to-End checkers:

1. *Arbitration checker:* The checker was implemented as a combination of two checkers which verified the arbitration scheme of the crossbar:
   - The first checker verifies that among multiple requests for a target, a request (from a client) with highest priority type (following the order - strict, high, normal) should be seen at the target. This checker ensures that there is never a scenario where a lower priority request gets served before a (pending) higher priority request.
   - The second arbitration checker verifies that among multiple requests of same priority for a target, a given client should not get grant twice before other clients have been granted. This checker ensures that the arbitration is fair and no request is starved of grant.

2. *Consistency checker on req_out and client_id, grant:* In order to verify that no spurious data is sent to a target and no spurious grant is given to client, we have the following checkers:
   - A data valid at a target (req_out_**) should have a corresponding request at client side.
   - A grant at a client (grant_*) should have corresponding request at the client

   An SVA implementation for consistency checker for grant is shown in Appendix.

3. *Consistency checker on req_data_out:* The checker verifies that the data is correctly transferred from the client to the desired target i.e. data is not corrupted, duplicated, reordered or dropped. Data consistency checker can be implemented using the following two methods:
   1. *FIFO based method*: In the FIFO based method, as shown in Figure 4, the data corresponding to a request for a watched target is stored in a FIFO in the testbench. When the output is seen at the watched target, the FIFO is read and the data is compared against the DUT data. If there has been any data corruption, drop or duplication inside the crossbar, the outputs will not match.

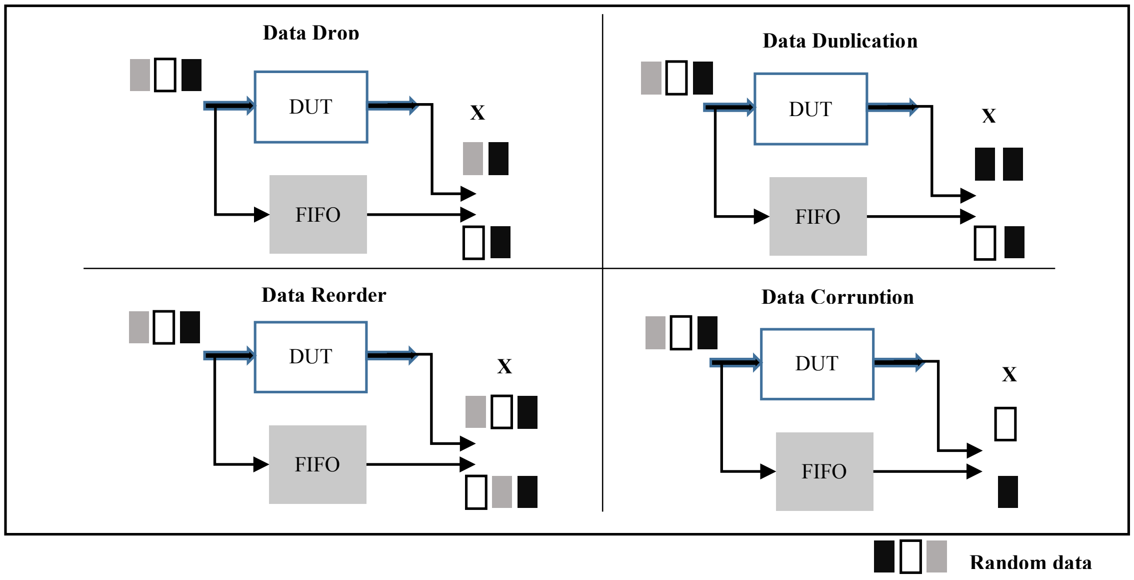


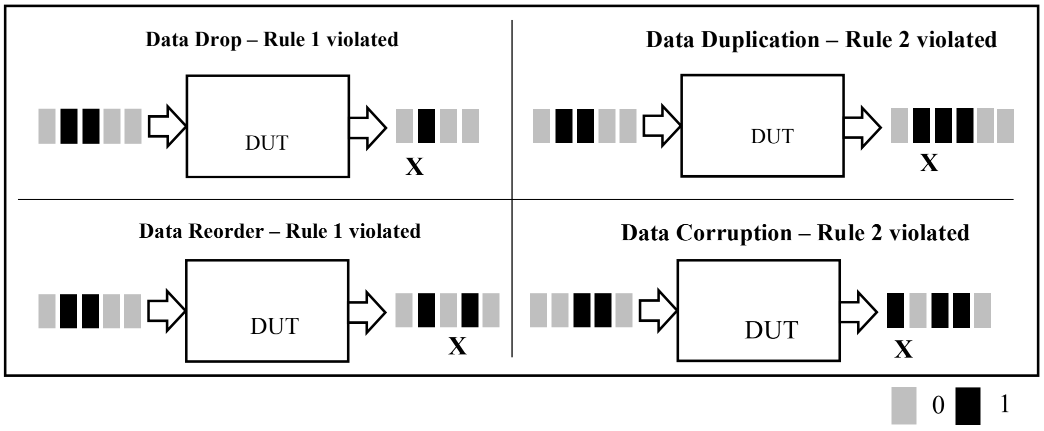
Figure 4: Data Consistency Checker using FIFO

   2. *Wolper Coloring Technique:* While the FIFO based method is very useful in verifying data consistency, it can add to formal complexity by increasing sequential depth and state–space because of FIFO depth and registers for storing data. In such situations, we can use another

method, which adds very little complexity to the formal testbench, termed as Wolper Coloring technique. This method uses a coloring technique proposed by Wolper [6]. In this technique, input data at DUT is "colored" such that a sequence of 0^*110^ω or 1^*001^ω is sent. At the outputs of DUT, we verify if we received the same sequence that is been sent at input. The rules for generating and verifying 0^*110^ω sequence are:

- Rule 1: If the first 1 is seen, next input/output should be 1
- Rule 2: If two 1's are seen, only 0's should be seen

By replacing 1 by 0 and 0 by 1 in the above rules, we can generate and verify 1^*001^ω sequence. Figure 5 below illustrates the Wolper Coloring technique for 0^*110^ω sequence. The SVA implementation of contraints and checkers required for Wolper Coloring technique is in the Appendix.



Data Drop – Rule 1 violated
DUT
X
Data Duplication – Rule 2 violated
DUT
X
Data Reorder – Rule 1 violated
DUT
X
Data Corruption – Rule 2 violated
DUT
X
0
1

Figure 5: Data Consistency Checker using Wolper Coloring technique

Apart from the above mentioned End-to-End checker, we had two forward progress checkers to ensure the design does not hang or in a deadlock.

1. *Forward progress checker on req_out:* The checker verifies that a request is seen at the desired target output within finite time.
2. *Forward progress checker on grant:* This checker verifies that a request gets a grant within a finite time after the data has been transferred to the desired target. This check ensures that once the data is transferred to the target output, a grant is correctly generated at the client side so that the crossbar is ready to accept the next input from that client.

B. Constraints

In order to ensure that we allow legal inputs to the design, we added the following constraints:

1. If a strict priority request is seen from a client, then there should be a request from that client
2. If a high priority request is seen from a client, then there should be a request from that client
3. If a request is waiting for grant, then request and corresponding priority (strict & high priority) and data should hold their value
4. For a target, no more than one client should send strict priority request at a time

C. Required Proof Depth

After implementing the above set of checkers and constraints, we need to ensure that our checkers are reaching the Required Proof Depth in order to achieve Formal Sign-off.

To determine the Required Proof Depth for the 8x8 configuration, we followed a number of steps as outlined in [2], which includes micro-architectural analysis, latency analysis, and interesting corner case analysis etc. The following is the analysis we did for our DUT which are collected during formal testbench implementation to determine the Required Proof Depth:

- *Latency Analysis*: The latency for DUT is 1 cycle

- *Micro-architectural Analysis:* The important micro-architectural structure is the arbiter. In order to see that the arbiter is functioning correctly, we need to see at least 8 requests of the same priority being processed by the arbiter.
- *Interesting Corner Cases:* This step involves brain-storming the interesting scenarios to exercise different corner cases of RTL. However, we need to carefully filter out the corner-cases not relevant to the RTL. For the crossbar design, we would like to see that the different types of requests are correctly granted as per priority. Thus the interesting corner cases that would cover all the above scenarios are:
    1. 1 strict priority request, 8 high priority request, 1 normal priority request
    2. 1 strict priority request, 8 normal priority request, 1 high priority request

Adding 1 cycle for backpressure and 1 cycle for safety net, our Required Proof Depth was 13 cycles.

The cycle counts above refer to the minimum number of cycles that it takes for the sequence of inputs to be seen at the target. Thus, in the above case, the strict priority request will be seen at the target after 2 cycles, and the last request (normal priority for corner case 1 and high priority for corner case 2) will be seen at the target at 11 cycles, after all the other requests have been processed and seen at the output. It is also important to note that even though additional requests can be seen at the target beyond 13 cycles, these input sequences will not represent any new scenario which cannot already be covered within the 13 cycles.

Since formal will report the shortest possible path to failure, any failure associated with such inputs sequences will already have been seen at a lower depth, and no new interesting case can be covered beyond this depth. Thus 13 cycles is sufficient to cover all interesting scenarios for the crossbar.

In order to ensure formal sign-off, it is critical for all our checkers to either reach or cross the Required Proof Depth. It is important to note that if the configuration is changed to allow different numbers of clients or targets, then we would again need to estimate Required Proof Depth. For 8x8 configuration, we found that 13 cycles sufficient and necessary to ensure Formal Sign-off.

D. Complexity & Abstraction Models

We used symbolic variable to reduce the formal testbench implementation effort. In the formal verification setup, we used symbolic variables to select a client and target. We implemented all of checkers for the symbolic client and target pair. Since formal tool assigns random non-deterministic values to symbolic variable while making sure that none of checkers show failure, all client and target pairs got verified. If we had not used symbolic variables, we would have written checkers for 64 different pairs of client and targets, which could have been substantial testbench effort and prone to errors.

In order to reach the Required Proof Depth for all checkers, the temptation is to increase run time. However, often in real designs we observe that increasing run time does not help in proof depth of checkers, as formal analysis hits the exponential curve. In such scenarios we add Abstraction Models, or use other complexity solving techniques to reduce the state space of the design. Since all of our checkers for the crossbar design were reaching the Required Proof depth in reasonable time, we were not required to add Abstraction Models.

E. Coverage

During formal verification of the crossbar design, we ran coverage analysis to track progress as well as to measure Formal Sign-off. Formal coverage is a relatively new feature added in some formal tools in recent years. The reachability analysis from formal coverage gives us a measure of the code coverage targets that are hit. The following are the results of the coverage analysis run for the crossbar when the formal testbench is completed:

- Line Coverage
    - Total lines: 288
    - Covered lines: 288
    - Deepest line cover point: 5
- Condition Coverage
    - Total lines: 88

- Covered lines: 88
- Deepest condition cover point: 5

Since all the cover points are below our Required Proof Depth, it serves as a validation that our Required Proof Depth is not optimistic. From the above results we can see that we have met the goal of 100% coverage for the DUT, also indicating that there are likely no over-constraint in the testbench.

Throughout the process of formal testbench building, we evaluate our testbench and formal results in the light of the 3 questions required for sign-off. We know our list of checkers is complete, there are no over-constraints and all our checkers reached the Required Proof Depth. When formal coverage analysis shows 100% coverage, as a final measure we know that we have reached sign-off and that our testbench can face any functional design mutations (at DAC 2015 in this case).

V. DAC Challenge bug examples

A final validation step for the claim of completeness of the formal testbench is to see if, empirically, all bugs have been shown to be detectable by the formal testbench. Towards this end, at DAC 2015, 73 different functional bugs were inserted by attendees. Some examples of the bugs are given below, as along with the End-to-End checkers that caught the bug:

1. Incorrect connection of inputs of different targets: While internally connecting requests from different clients to different targets, the high priority port for target 2 was left hanging. Arbitration and forward progress checkers failed.
2. Data corruption by accidental inversion of output: A bug was introduced at one of the target outputs which inverted the data output. Consistency checker on req_data_out failed.
3. Changing priority type: The high priority input of one of the targets was internally tied to 1. Arbitration checker failed.
4. Incorrect sizing of loop leading to unfair arbitration: The loop which performed round robin arbitration was starting at a higher count than minimum value. As a result a request was not given grant and was unfairly starved. Arbitration and Forward progress checkers failed.
5. Grant to client tied to 0: This bug ensured that while all the outputs were correctly transferred to the respective targets, the grant was never sent back to the client. Thus, no new input request was coming and a deadlock scenario was created. Forward progress checkers failed.

The fact that all the inserted functional bugs triggered failures of one or more checkers offered a live proof that our formal testbench was complete. Table 1 has most popular bug categories inserted in the DAC challenge, original and modified RTL and number of failing checkers.

TABLE 1
BUG CATEGORY WITH ORIGINAL AND BUGGY RTL CODES AND NUMBER OF FAILING CHECKERS

Bug Category	Original RTL	Buggy RTL	Number of failing checkers
Arbitration scheme	// arbiter.sv 60 assign high_prio_req = (|str_prio_req) ? {NUM_CLIENT{1'b0}} : (high_prio & req);	// arbiter.sv 60 assign high_prio_req = (|(str_prio_req | **high_prio_req**)) ? {NUM_CLIENT{1'b0}} : req;	4
Arbitration scheme	// rr_scheme.sv 56 assign shft_req = {req, req};	// rr_scheme.sv 56 assign shft_req = {**8'd0**, req};	2
Arbitration scheme	// rr_scheme.sv 65 for(i = 0; i < (2 * NUM_CLIENT); i = i + 1) begin	// rr_scheme.sv 65 for(i = 0; i < (2 * NUM_CLIENT **- 1**); i = i + 1) begin	2
Connectivity	// xbar_8x8.sv 274 .high_prio(high_prio_4),	// xbar_8x8.sv 274 .high_prio(**high_prio_3**),	4
Grant generation	// one_dly.sv 71 assign gnt_i = has_data ? outgoing_data : 1'b1;	// one_dly.sv 71 assign gnt_i = has_data ? outgoing_data : **1'b0;**	2
Wrong operator	// target.sv 89 assign t2c_grant = {NUM_CLIENT{ext_grant_pp}} & arb_gnt_pp;	// target.sv 89 assign t2c_grant = {NUM_CLIENT{ext_grant_pp}} && arb_gnt_pp;	8

Mask generation logic	`// client.sv`
	`65 c2t_req_mask <= c2t_req ^ t2c_grant;`
		`// client.sv`
		`65 c2t_req_mask <= c2t_req ^ {`**`t2c_grant[7:5], 1'b0, t2c_grant[3:0]`**`};`
			3

Datapath	`// one_dly.sv`
	`62 else if(incoming_data) begin`
	`63 has_data <= 1'b1;`
	`64 end`
	`65 else if(outgoing_data) begin`
	`66 has_data <= 1'b0;`
	`67 end`
		`// one_dly.sv`
		`62 else if(outgoing_data) begin`
		`63 has_data <= `**`1'b0`**`;`
		`64 end`
		`65 else if(incoming_data) begin`
		`66 has_data <= `**`1'b1`**`;`
		`67 end`
			5

Apart from these 73 functional bugs, there were two more design changes made during the challenge for which none of our checkers failed. We are considering each of these non-bugs for the following reasons.

1. One change was to modify the initial value of parameter for round robin arbiter from 0 to 1.

Original RTL code:

```
for(i = 0; i < (2 * NUM_CLIENT); i = i + 1)
```

Modified RTL code:

```
for(i = 1; i < (2 * NUM_CLIENT); i = i + 1)
```

After examining the RTL, we realized that making `i=1` makes an RTL optimization to use an n-iteration loop instead of an n+1-iteration loop. The optimization makes the design slightly better area-wise but had no functional impact. As a result, no checker failed.

2. In the second change, a counter was added to the design and a grant signal gets tied to 0 after counter reaches a certain threshold.

Original RTL code:

```
assign high_prio_gnt = tmp_high_prio_gnt;
```

Modified RTL code:

```
assign high_prio_gnt[0] = tmp_high_prio_gnt[0] &
                          ~(my_bug_delay == 1024'123456789101213);
always @ (posedge clk) begin
    if (rst)
      my_bug_delay <= 1024'b0;
  else
     my_bug_delay <= my_bug_delay + 1'b1;
end
```

This bug was inserted with a "malicious" intent, i.e. to specifically change the design so that the defect cannot be caught by the verification methodology; we believe that this can always be done, and defeats the purpose of true verification to find all "naturally occurring" bugs. Hence, we labelled this as a non-bug. This change increased the Required Proof Depth and it would be impossible for formal (and sometimes simulation) without using Abstraction Models to reach increased Required Proof Depth.

VI. Conclusion

The experiment conducted in the DAC 2015 challenge is significant because for the first time, a live case study showed that formal technology, when used properly and systematically, can be used for sign-off. The paper outlines the process for formal sign-off using a common data-transport design, the multicast crossbar. Different designs will require different End-to-End checkers, constraints, complexity solving techniques and a different Required Proof Depth to target for sign-off, but the process remains the same and can be learnt. It is our hope that through our work and demonstration, formal will become one of the verification techniques routinely used for verification sign-off.

Acknowledgment

The authors would like to acknowledge the support from Synopsys for using VC Formal during our live challenge at DAC 2015.

APPENDIX

In this section we have attached snippets of the code for some of our constraints and checkers. The consistency checker on grant is written in SVA as shown below. The checker tracks a single symbolic client at a time. As explained in Section III, formal tool assigns random non-deterministic values to symbolic variables and thus in this case it proves the checker for all clients in the design.

```
e2e__xbar2cl_no_gnt_at_client_if_no_req : assert property(
   @ (posedge clk) disable iff(rst)
      (!(|req[sym_client])) |-> (!grant[sym_client])
);
```

Following is SVA implementation of the constraints required for Wolper Coloring technique for 0^*110^ω sequence. first_one_seen and second_one_seen are two sticky flags which are asserted when the first and second "1" is seen at input (client) data respectively. Similar to symbolic client and target, we are coloring only one symbolic bit at input.

```
cl2xbar_1st_1_seen_next_input_should_be_1: assert property (
   @ (posedge clk) disable iff(rst)
      (first_one_seen && !second_one_seen &&
       input_data_valid) |->
      (colored_input == 1'b1)
);
cl2xbar_2nd_1_seen_at_input_0_input_forever: assert property (
   @ (posedge clk) disable iff(rst)
      (second_one_seen && input_data_valid) |->
      (colored_input == 1'b0)
);
```

Similarly, we implemented checkers for output (target) data and the SVA implementation is as follows. Please note that colored_output has same symbolic value as been assigned to colored_input.

```
e2e__xbar2tgt_1st_1_seen_implies_next_output_is_1 : assert
property (
   @ (posedge clk) disable iff(rst)
      (first_one_seen_at_output && !second_one_seen_at_output &&
       output_data_valid) |->
      (colored_output == 1'b1)
);
e2e__xbar2tgt_2nd_1_seen_implies_0_at_output_forever : assert
property (
   @ (posedge clk) disable iff(rst)
   (second_one_seen_at_output && output_data_valid) |->
   (colored_output == 1'b0)
);
```

REFERENCES

[1] D. Perry, H. Foster, "Applied formal verification: for digital circuit design", McGraw Hill, 2005.

[2] P. Aggarwal, D. Chu, V. Kadamby, and V. Singhal, "Planning for end-to-end formal with simulation-based coverage", Proc. Formal Methods in Computer-Aided Design FMCAD 2011, Austin, TX, USA, 2011, pp. 9-16.

[3] N. Kim, J. Park, H. Singh, and V. Singhal. "Sign-off with Bounded Formal Verification Proofs", DVCon 2014.

[4] V. Singhal and P. Aggarwal, "Using coverage to deploy formal verification in a simulation world", in Proc. Conf. Computer-Aided Verification CAV 2011, Snowbird, UT, USA, G. Gopalakrishnan and S. Qadeer (Eds.), Springer, 2011, pp. 44-49.

[5] F. Haque, J. Michelson, and K. Khan, "The art of verification with SystemVerilog Assertions", Verification Central, 2006.

[6] P. Wolper, "Expressing interesting properties of programs in propositional temporal logic", POPL '86 Proc. of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages", pp. 184-193