Traversing the Abyss: Formal Exploration of Intricate State Space

Tanishq Sharma *tanishq@cadence.com*, Sakthivel Ramaiah sramaiah@cadence.com,
Craig Deaton *cdeaton@cadence.com*
Cadence Design Systems

Abstract **- PCI express is a widely used high-speed serial computer expansion bus standard and verifying the serial interface exhaustively is always challenging as complexity increases. One of the critical features of PCIe is Credit-Based Flow Control (CBFC), which enables the requester to monitor the available queue/buffer space in the agent across the Link. This paper explores the model checking approach to verify the Credit-Based Flow Control and ensure the reliable communication in PCIe based systems. CBFC performs the internal calculation of the credits and displays the updated credits. The complexity of the CBFC can be explored by reaching deeper state space in "Formal". However, exploring the complexity of CBFC through traditional formal verification methods becomes impractical due to the huge state space, which refers to the vast number of reachable states, leading to the computational complexity and prolonged run time that often yield undetermined results. As a result, specialized techniques are employed to cope with these challenges, we make use of abstraction techniques, helper assertions and State Space Tunneling (SST) [4].**

Introduction

Formal verification is a stringent method used in software and hardware engineering to ensure that a system behaves correctly according to its specifications. In the context of data transmission, Credit-Based Flow Control is orthogonal to the data integrity mechanisms that ensure reliable information exchange between the transmitter and receiver. Since these mechanisms correct any corrupted or lost Transaction Layer Packets (TLPs) through retransmission, Flow Control can assume that the flow of TLP information from the transmitter to the receiver is flawless. The Transaction Layer, in collaboration with the Data Link Layer, manages Flow Control to ensure efficient and reliable data transfer.

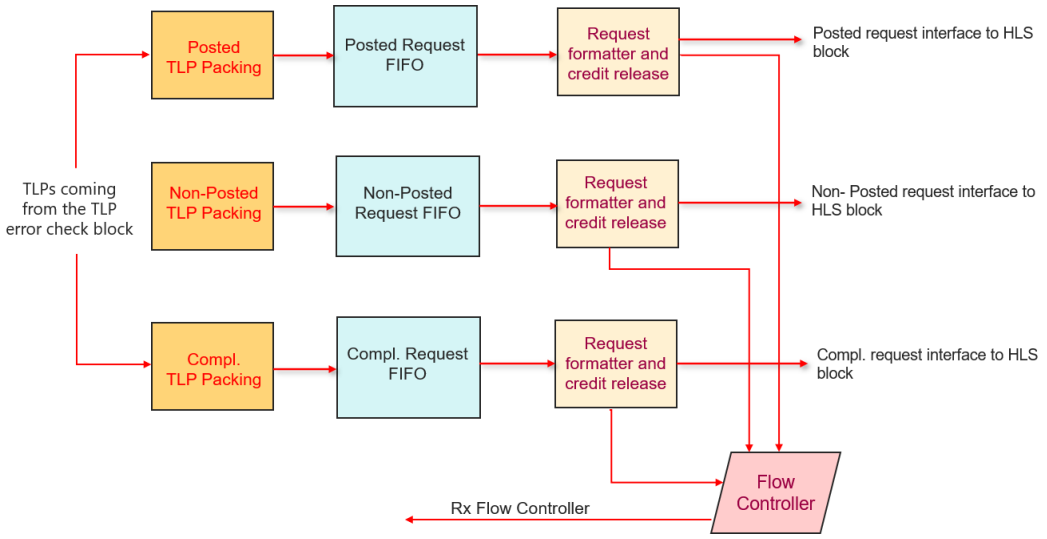

Fig.1(a) CBFC Block Diagram

As shown in *Fig.1(a)*, TLPs coming from the TLP error check block is first feed into the packing module which packs the TLP according to the TLP packing rules (i.e., DW alignment) for posted, non-posted, completion. The packed TLPs are then stored in their respective transaction FIFOs. Subsequently, the request formatter transmits the TLPs to the HLS block and credit is released accordingly.

Features of flow controller:

- **Monitor credit updates**: The CBFC continuously monitor credit updates exchanged between PCIe devices to keep track of changes in available credits. This allows the flow controller to stay informed about the current credit status and adjust as needed.
- **Maintain credit status**: It maintains a record of the current credit status for each device connected to the PCIe link. This information is essential for managing credit allocation and ensuring that each device has sufficient credits to transmit data.
- **Implement credit management algorithm**: It performs the credit calculation to allocate credits fairly and efficiently among the resources. This algorithm ensures that credits are distributed in a way that optimizes data flow and minimizes congestion.
- **Handle congestion**: It handles congestion and credit starvation by taking appropriate actions such as pausing credit updates or adjusting credit allocation. This helps to maintain a smooth flow of data and prevents devices from running out of credits.

Verification challenges: Traversing the vast design space and managing runtime complexity is a significant challenge in formal methods, necessitating meticulous planning to overcome these challenges. Credit-Based Flow Control (CBFC) is particularly susceptible to this issue, as it requires tracking multiple parameters, including available credits, updated credits, and buffer size, across various components of the PCIe architecture, resulting in an extensive state space.

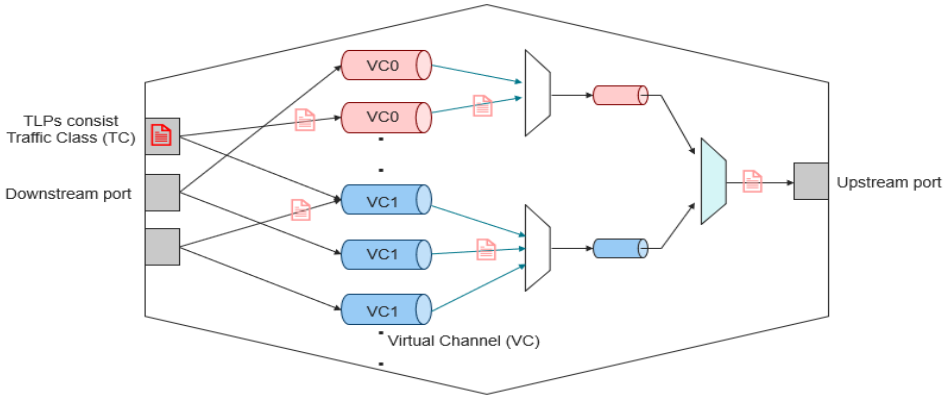

Fig.1(b). Depiction of transactions through the virtual channels.

There are multi-VC transactions and each virtual channel associated with a specific Traffic Class (TC). Transaction Layer Packets (TLPs) contain the TC number which can be any number from 0 to 7 (max).

Design Complexity: There are two sets of resources (FIFO) associated with each virtual channel:

- A typically small pool of "Dedicated resources" associated independently with each virtual channel (VC).
- A typically large pool of "Shared resources" which is shared among the virtual channels, allowing for more efficient utilization of resources.

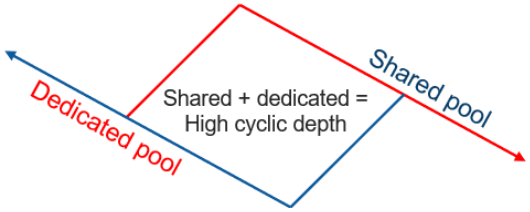

Fig.1(c). Depiction of resource complexity.

- Shared and Dedicated resources add high cyclic depth in the design, resulting in high sequential complexity.

Complexity techniques to overcome these challenges: -

Parameter reduction: Model reduction techniques attempt to identify subset of logic which can be replaced by equivalent, smaller pieces of logic [3]. This reduction can make verification more efficient by simplifying the analysis and reducing computational complexity.

- Flow controller generates the Fc_update_request to DLL (Data Link layer). One of the parameters to generate the request is *MAX CREDIT INTERVAL*,
- The timer value is set internal to the block as a local parameter. As there are multiple counters in the COI (Cone of Influence) of *credit update timer* that counts the value.
- Method 1: Reducing the parameter during elaboration.
    I. *elaborate -parameter i_tl_rx_flow_ctrl.MAX_CREDIT_INTERVAL 30***.**
- Method 2: Using cut point and assume statement.
    I. Apply cut point on "*i_tl_rx_flow_ctrl.MAX_CREDIT_INTERVAL"***.**
    II. *assume i_tl_rx_flow_ctrl.MAX_CREDIT_INTERVAL == 30***.**

Initial Value Abstraction:

- Initial Value Abstractions (IVA's) [3] are a unique technique in formal verification.
- IVA is helpful in design like flow controller which is having a large counter that need to reach a specific value before reaching an interesting state, where reaching a specific Counterexample (CEX) value is unfeasible.
- In Flow controller, TLPs are forwarded, and absolute credits are counted by the intermediate credit limit counter. Credit limit reaches the max number after specific number of TLPs being send to HLS block.
- IVA helps to initialise the counter with warm state rather than reset state and checks whether the update credits are roll over after reaching the max credit value.

Advance Complexity techniques

Helper Assertions:

- Helper assertions are assertions that assist in proving other assertions in formal verification.
- Helper assertions can help each other during the proof process.
- Helper assertions do not change the reachable states of the system being verified.
- Formal analysis tools use assumptions expressed through assertions to constrain the inputs and test all valid combinations to find errors.
- All modern formal tools support the use of helper assertions.

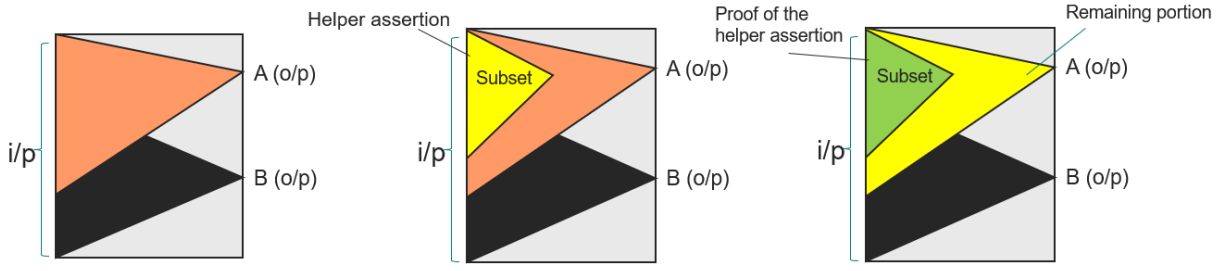

Fig.2(a) Helper Assertion Usage

- Write helper assertion on the subset of the COI of target assertion.
- Then make a use of proven helper assertion to prove the target assertion.

Note - Proved helper assertions are assumed by future proofs [1].

Tracking the header credit:

- Below statement is one of the features that checks the header credit are updated correctly as we forwarded the TLPs with desired virtual channel.

In non-flit mode if scaled flow control is low and tlps are forwarded, then vc with "non-infinite" credits should show the updated header credits when the update fc req and ready is high.

Target property:

```
genvar var.
generate.
for (var = 0; var< MAX_VC_NUM; var = var+1) begin
assert property @ (posedge clk)
(! es_flit_mode_i && vc_en_active[var] &&
 ! es_scaled_flow_control_active_i && tl2dl_rx_posted_fc_credit_update_ready_i &&
lm_tl_rx_posted_credit_header_i[var*8+:8]! = 'd0 &&
tl2dl_rx_posted_fc_send_credit_update_req_o == 2'b01 && posted_req_vc_num == var
        |-> ## [0:3] (tl2dl_rx_posted_fc_header_credit_o == tl2dl_posted_header_credit[var])).
end
endgenerate
```

S.no	Signal Name	Direction	Description
1	*es_flit_mode*	i/p	Indicates the two modes. 0 (Flit Mode), 1 Non-Flit Mode
2	*vc_en_active*	i/p	Indicates which VC is enabled and active.
3	*es_scaled_flow_active_i*	i/p	CBFC updates credits in two ways, either with scaled flow enable (multiple of 4) or without scaled flow enable.
4	*lm_tl_rx_posted_credit_header_i*	i/p	Shows the header credit. (If it is "0" means infinite credits).
5	*tl2dl_rx_posted_fc_credit_update_ready_i*	i/p	Data link layer asserts ready to indicate that it's ready to accept updated credits.
6	*tl2dl_rx_posted_fc_send_credit_update_req_o*	o/p	TL assert request only when the credits are updated.
7	*tl2dl_rx_posted_fc_header_credit_o*	o/p	Shows the updated credit after forwarding the TLPs.

Description:

There are multiple virtual channels, and the transaction is going on in a random manner and only single channel output reflects the updated credits, this can lead to a large fan-in cone for the target assertion, making it challenging for the property to converge. To address this issue, a helper assertion can be used to simplify the proof process. By analyzing the COI and extracting the property information from the modern formal tool, we can identify specific signals that drive all computations and influence the update of header credits. However, this process requires a deep understanding of the design, and manual extraction of the helper assertion from the target assertion is necessary.

Manually extract the Helper assertion:
*i_tl_rx_flow_ctrl.*dedicated_or_nfm.header_credit_limit* (its internal signal of the DUT). We wrote helper assertions on this signal and proved it, then marked it as helper. These helper assertions contribute to reduce the complexity of the target assertions.

	Type	Name	Engine	Bound	Traces	Time	Task
✓	Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	N (39)	Infinite	0	9.6	<embedded>
✓	Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	N (78)	Infinite	0	21.1	<embedded>
✓	Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	N (91)	Infinite	0	27.4	<embedded>
✓	Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	N (101)	Infinite	0	23.8	<embedded>
✓	Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	N (113)	Infinite	0	20.1	<embedded>
✓	Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	N (58)	Infinite	0	10.3	<embedded>
✓	Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	Mpcusto...	Infinite	0	20.3	<embedded>
✓	Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	Mpcusto...	Infinite	0	29.0	<embedded>

These helpers are for VC0 to VC7.

Fig.2(b): Helper assertions

Note: Proven helper assertions are taken as an assumption for the future proof of the complex target property.

- But these helper assertions didn't yield good results as the target property still not converging or reaching an acceptable bound. At this stage, we introduce a technique called SST (State space tunneling).
- Prior to applying SST, we applied traditional formal techniques, but the property remained undetermined with lesser bound. As a result, we transitioned to SST [4] at later stage to achieve more definitive result with good bound.

SST (State Space Tunnelling): It refers to a technique used to overcome the problem of state explosion in model checking. It is employed to systematically achieve proof convergence on hard-to-prove, critical properties with a large state space by adding helper assertions. It is way to identifying the exploiting paths through the state space that are relevant to the property being verified, while avoiding exploring the irrelevant states. Eliminates the scenarios where formal tool engines wasting their time to analysing them.

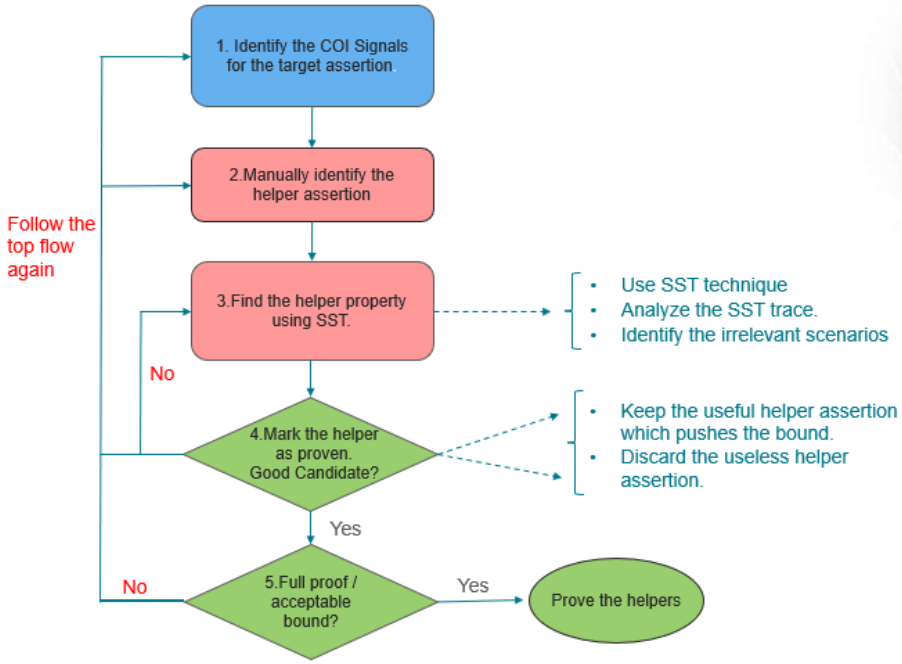

Fig.3(a) SST Flow Chart.

SST lets you examine the challenges the engines are facing in induction, where they need to prove that a proof to bound N implies the property is true at N+1. (i.e., assume P(n) is TRUE, then prove P(n+1) [4]). To aid in this process, the SST generates an example where the property holds true for a certain number of cycles but then becomes false. However, since this counterexample is not generated from the reset state, it is likely to be bogus. By analysing the signals in the property's COI and understanding why the counterexample is bogus, you can often derive a useful helper assertion that makes induction easier and reduces complexity.

The SST flow diagram is the visualization of the SST methodology, which explores the complex state space by strategically "tunneling" through the states. *Fig.3(a)* provides step-by-step overview of how SST efficiently navigates through the sequential depth of the state space while bypassing the irrelevant or non-promising areas. This structured approach offers insight into SST's ability to accelerate the proof of target property.

Step 1): Identify the COI signal for the target assertion.
Step 2): Manually identify the helper assertion.

- This step requires deep understanding of the RTL to identify the focus COI signal which is adding the complexity and showstopper to verify the target property.

Step 3): Find the helper assertion using SST trace.

- This is the key step to find the good candidate that can be used as a helper assertion.
- Use State Space tunneling to extract the good helper assertion.
  - Helper properties are always included when running the property with sst.

Step 4): Mark the assertion proven.

- Initially don't need to prove the assertion just mark it as proven in the formal tool.

Step 5): Full proof / acceptable bound.

- SST is an iterative process and there can be multiple iteration until you reach a full proof / good bound.

Step 6): Prove the helper assertions.

Example:
Let's consider a small digital circuit design with a clock signal (clk) and an active-high reset signal (rst) as inputs, and two internal identical synchronous counters (i.e., counter 1 and counter 2) that are driven by the clock signal. The output of the design is generated through a logical AND operation of all the internal combinational logic blocks, which are likely implemented using Boolean algebraic expressions.

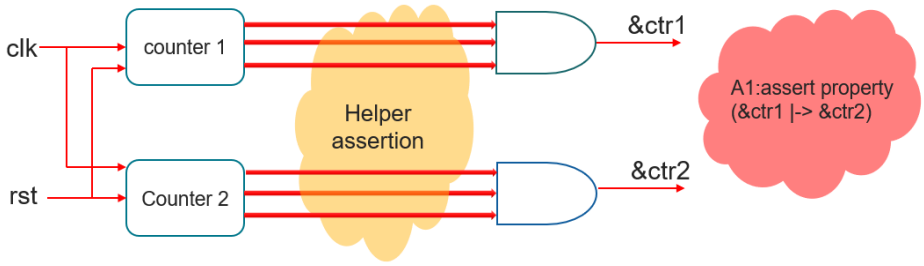

Fig 3(b) counter example

The target property A1 says &ctr1 implies &ctr2. But surprisingly it's hard to prove due to the internal logic of the counters. So, let's see how helper assertion and SST solve the issue.

1. Run the SST and analyse the SST Trace to extract good helper assertion.

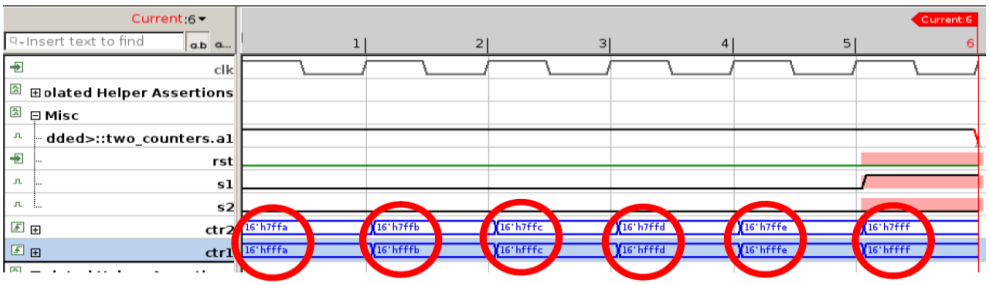

Fig 3(c) SST Trace has been found at cycle 6.

2. Waveform shows many cycles where the two counters differ.
    a. Sometime formal tool wasting time in analysing such cases.
    b. This should not happen as counters are identical.
3. To resolve this issue, create helper assertions forcing the counters to be equal (i.e., ctr1 == crt2).
    a. And mark the helper assertion as proven.
    b. Making the tool job easier.
4. Then run the property with helper assertion.

Deploying the SST in CBFC:

Following the SST flow as mentioned in Fig.3(a).

1. Assertion set as a helper (adding helper based on the understanding).
2. Prove with SST and see the SST trace.

Type	Name	Engine	Bound	Traces	Time	Task
Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	Cache	Infinite	0	0.0	SST
Assert(SST)	AST_posted_header_credit_with_vc_num	U2	130 -	1 SST	99.4	SST

Fig.4(a) SST enabled property.

SST trace has been found at 130th bound, which starts from the arbitrary state.

3. Analyse the SST waveform.

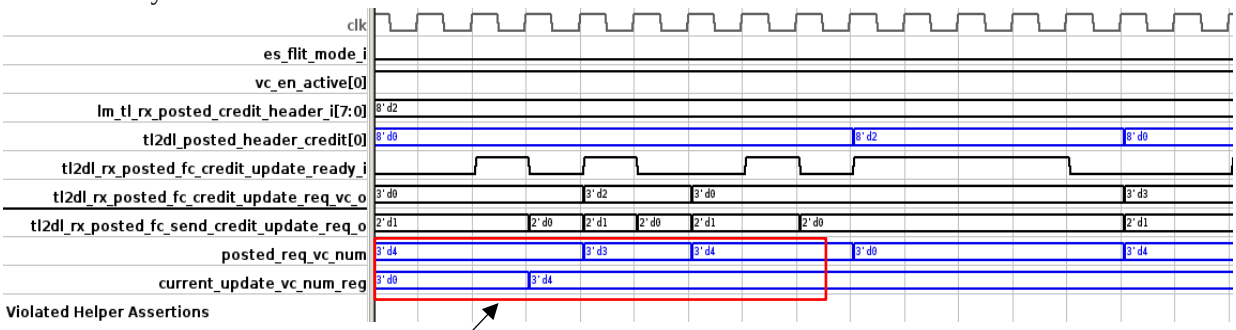

Fig.4(b) SST Trace

- *current_update_vc_num_reg* calculates the VC number through which the TLPs are forwarded.
- This signal adds complexity because it is tracking the header credit updates, rather than the VC number.
- The combination of *current_update_vc_num_reg* and *posted_req_num* are not relevant to check the VC0 header credits as shown in the waveform.

- Writing helper assertion on this and "mark as proven" is only a temporary step to evaluate the effectiveness of the helper on the target property. We must go back and prove the helper before signoff.
- Suppose we are checking for VC1, but the arbitration logic works for all the VCs, and if we control the VC number then it's easy to track the header credit.

	Type	Name	Engine	Bound	Traces	Time	Task
✓	Assert(Helper)	AST_posted_header_credit_with_vc_num_helper	User	Infinite	0	0.0	SST

Fig.4(c) Set as helper and Mark it as proven.

4. Run the property.

	Type	Name	Engine	Bound	Traces	Time	Task
✓	Assert(Helper)	AST_posted_hdr_crdt_limit_w_no_scale	Cache	Infinite	0	0.0	SST
?	Assert(SST)	AST_posted_header_credit_with_vc_num	Mpcustom2	526 -	1 SST	23484.9	SST

Significant change in the bound of the target property.

Fig 4(d). Target property

As evident from the analysis, the target property bound has been successfully increased, thereby expanding the state space. The SST algorithm plays a crucial role in identifying optimal sets of helper assertions, which enables the formal verification tools to overcome the limitations of exhaustive analysis and efficiently prune the search space, thus avoiding getting stuck in analysing infeasible scenarios.

Result

In high-speed communication systems, the flow controller is a critical component that requires thorough verification to ensure data integrity and prevent performance degradation. Formal verification is essential to guarantee that the flow controller functions as intended under all possible operational conditions, including both positive and negative scenarios. However, verifying multi-VC transactions in flow controllers poses significant challenges in formal verification, particularly due to the complexity introduced by state space explosion. As the number of virtual channels (VCs) increases, the state space grows exponentially, making it computationally intensive to explore all possible paths, especially when complex dependencies exist between channels. To overcome these challenges, "Advanced Formal Techniques" such as Helper Assertions and State Space Tunneling were employed. This technique should only be considered when all other options have been exhausted, as it demands an exceptionally high level of specialized knowledge of the design.

Conclusion

Formal verification offers a robust set of techniques for handling complex designs by abstracting intricate system behaviour into more manageable models. By leveraging State Space Tunneling (SST), we can effectively converge the undetermined results into definitive proof, thereby facilitating a more efficient analysis of state space without compromising accuracy. Formal methods provide a systematic and rigorous framework for verifying the correctness of a design and ensuring its adherence to specified requirements.

References

[1] Jonathan Bromley & Jason Sprott "*Formal Verification in the Real World*" DVCON US 2018.

[2] Ankit Garg "*Forward Progress Check in Formal Verification: Liveness vs Safety*" DVCON US 2024.

[3] M V Achutha Kiran Kumar, Erik Seligman, Tom Schubert "*Formal Verification - An essential toolkit for Modern"* Book.

[4] Varun Ramesh & Mahesh Prabhu "*Using SST for Faster Proof Convergence*". JUG US 2024.