2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Smart Formal for Scalable Verification

Ashish Darbari
Axiomise Ltd.
71-75 Shelton Street, Covent Garden, London WC2H 9JQ
ashish.darbari@axiomise.com

Abstract- **Verifying serial designs is a formidable challenge for both simulation and formal. The complexity of such a design stems from the serialised nature where the state of each packet depends upon the history of all the previous packets in flight. In this paper, we show a smart, scalable formal verification methodology for verifying serial designs. Our smart solution using abstraction and assume-guarantee reasoning can find bugs in multi-packet serial designs with about 140K flops ($10^{42,248}$ states) in under 40 minutes, and exhaustively proves bug absence in designs with about 24K flops (10^{7397} states) in 35 minutes on a tablet PC (two Intel i5 cores with 6 GB memory). When we were not limited by compute resources (128 GB memory, 18 CPU cores), we could find random reordering bugs in under 20 minutes of proof times for a design configuration with half-a-million flip-flops ($10^{164,228}$ states).**

I. INTRODUCTION

Verification of serial designs is a known challenge for both simulation and formal verification. Whereas constrained random simulation-based verification has the usual shortcomings of not being able to apply enough input stimulus to exercise all corner cases, formal verification can very often suffer from the state-space explosion. While with formal verification in principle one can exhaustively prove correctness – in practice, this can often be challenging if not impossible unless good methodologies are used. In this paper, we show a smart, formal verification methodology based on abstractions and assume-guarantee reasoning for verifying serial multi-packet designs using limited computing resources.

II. PROBLEM DOMAIN

Serial multi-packet data designs are everywhere across the system-on-chip (SoC). Examples include but are not limited to buses (AMBA, OCP, PCIe), bus bridges, network-on-chip interconnects (NoCs), video packers and unpackers, networking devices, SoC peripherals (I^2C, I^2S, UART, USB, Bluetooth, and Ethernet), load-store units in CPUs, and memory sub-systems. Modern-day machine learning architectures have serialised behaviour where deep learning and prediction hardware typically employs looking up long sequences of historical sequential data.

The basic characteristic of multi-packet designs is that one or more packets may be written, but there is no requirement that a fixed number of packets is always written on every new write. When the number of packets is a variable, it introduces new challenges for verification of serial designs increasing the number of combinations to be verified. If the number of packets accepted every cycle is a fixed number, the problem is slightly easier to solve with formal especially using our methodology.

Typically, in such designs, there is no requirement that all packet locations are written at a given buffer index, but once they have been, the packets must *not be lost, corrupted, reordered and duplicated*. Given the non-determinism introduced in these serial designs due to an arbitrary number of packets being transported, it makes verification a lot harder as we track all possible combinations of packet writes and reads for the entire path of data transport.

For our purpose, we have three dimensions to this problem. Packets can have different sizes (defined by the **Width**), and they are stored in buffers that are defined by **Depth**. The maximum number that can be transported is defined by the parameter called **Packets**.

III. SMART TRACKER SOLUTION

In the recent past [1-2], we showed how a transaction counting methodology could be used to verify huge designs. In this section, we present the key aspects of our solution we call the smart tracker solution. The basic idea of this solution is based on observing a pre-determined symbolic data value namely a watched data value as it enters the DUT and leaves it. By using a counter to count how many non-watched data values are ahead of the watched data,

we can predict when we expect to see the watched data value appear on the output port. The formal tool chooses all possible data values as an instantiation of the watched data value and chooses to insert the watched data at all possible locations within the DUT thereby allowing an exhaustive coverage of the entire design space. This method of observing just one symbolic data value to verify that all data values remain correctly ordered in the design and do not get duplicated or dropped is a data and temporal abstraction technique. It is data abstraction because the symbolic data value encodes all possible data values composed of 0s and 1s. It is temporal abstraction because the time at which the watched data value is captured in the design is left completely arbitrary allowing multiple time points to be used for capturing the data in the DUT, but the verification engineer doesn't have to manually control when the watched data are read into the design. By leaving the exact timing completely arbitrary, the formal tool makes use of non-determinism. Without the need for tracking the state of all possible data values explicitly and at all possible time points the formal tool can obtain massive scalability in proof times and can generate converging results on properties both for cases when the DUT may have a bug (bug hunting) and when there is no bug in the design (exhaustive proof). This is the reason why we call our solution a smart tracker solution. By employing just one tracking counter to count how many data values are ahead of the watched data and employing only two sampling registers to detect the entrance and exit of the watched data we were able to verify FIFOs as deep as 8192 carrying 32-bit data payload exhaustively [1-2].

IV. DESIGN DESCRIPTION

FIFOs are a special case of multi-packet designs where at most one packet can be read or written in any location. When the number of packets becomes a variable, the problem of verification becomes significantly harder. The reason is that now there is more choice in the design execution – any number of packets can be written or read into

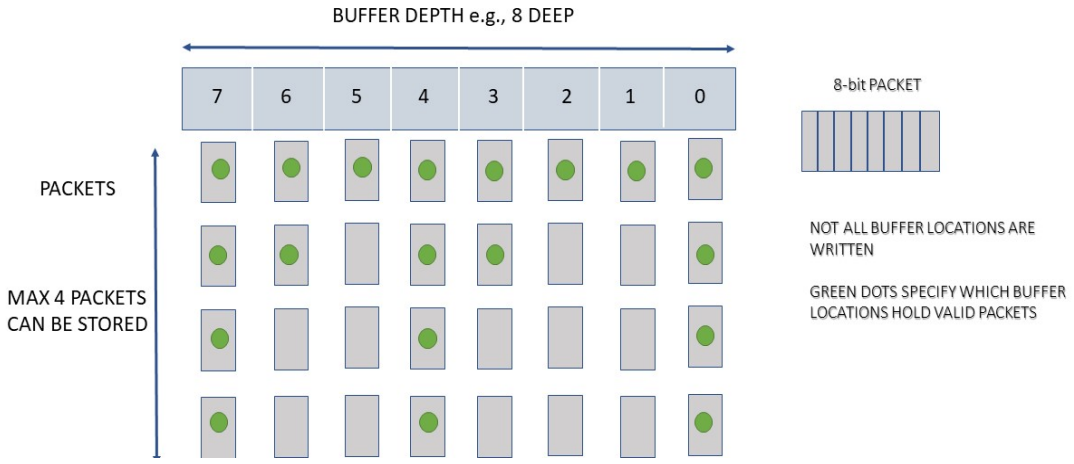

Figure 1: Multiple variable number of data packets can be written in the design. The green dots indicate locations where valid data packets are stored. Non-green locations indicate invalid data (no data has been written into these locations).

the design at any given clock cycle bounded only by a maximum pre-configured design parameter. In Figure 1, we show a configuration where up to four packets can be stored in any of the locations from 0 to 7. The packet size itself can be any number of bits defined by a top-level design parameter. A three-dimensional data structure is used.

We show the top-level design interface used in the case study in Figure 2. Input data is transferred in packets where each input packet is written into `data_i` on an input handshake `hsk_i` (`valid_i && enable_o`). Output data (`data_o`) is read out on an output handshake `hsk_o` (`valid_o && enable_i`). It is worth mentioning here that it takes multiple clock cycles to write and read multiple packets of defined size. For example, to write/read four one-byte packets we need four clock cycles and the 32-bit word would be written and read out from a location defined by the write index (`wptr`) and read index respectively (`rptr`). Validation and invalidation logic determine which data entries are legally written and must be therefore read out.

The design might come across as not very complex – but five finite-state machines control the read and the write of multiple packets. The Write FSM controls how the data is written while the Read FSM ensures that without a valid write, reads do not occur, but when a read is requested, a valid number of packets are read out in the expected number of clock cycles.

Exactly how many packets would be written is defined by the input `pkt_len` which is registered into the design on the very first beat of the input handshake of a new write transaction. Subsequent values of `pkt_len` (which is `PKT_BITS` wide) are disregarded until another new packet stream is seen. The read and write of packets is controlled by buffer read/write FSM which is a function of read/write pointers indexing the buffer depth and a packet read/write FSM which is a function of packet read/write pointers indexing the packet locations.

A new valid write starts when the write pointer index is pointing to 0 and there is an input handshake. In this clock cycle, the value of `pkt_len` is registered in the design and this is how many packets would be written at the buffer index pointed to by `wptr`. So, the `wptr` walks along the buffer depth, whilst `pkt_wptr` indexes the individual packets at a given buffer location. Similarly, `rptr` reads along the buffer depth, while `pkt_rptr` reads the individual packets from the buffer indexed by `rptr`. The design also has flags `empty_o` and `full_o` to indicate when it is empty and full respectively, and, is driven to reset by an active low `resetn`. Since this design is multi-packets, we need an array of watched values not just one.

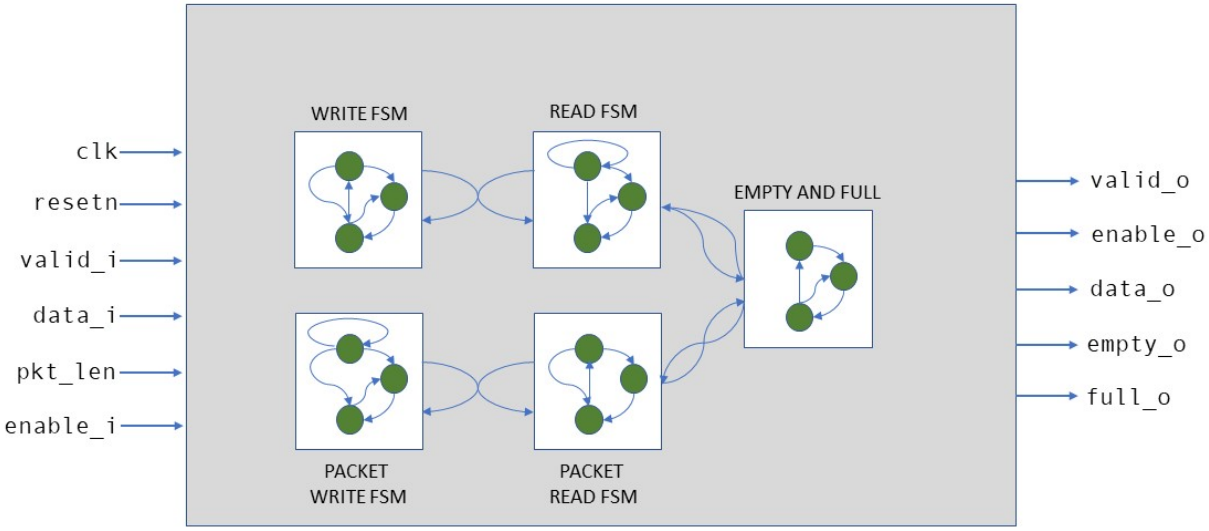

Figure 2: Top-level design interface shown. Five interacting finite-state machines control the read and write of multiple packets. The number of packets to be written in any given clock cycle is determined by `pkt_len`, and the packets themselves are written and read from `data_i` and `data_o` respectively. Handshake signals determine when the data will be written or read from the design, while empty and full denote when the buffer is empty and full respectively.

V. SMART TRACKER SOLUTION FOR MULTI-PACKET DESIGN

In this section, we show a smart solution that exhaustively verifies serial multi-packet designs to ensure that they have no data loss, no data reordering, and no data duplication. There are two fundamental components of our smart solution. We have taken our foundational smart tracker abstraction from our previous work [1-2] and adapted it to verify the multi-packet behaviour of serial designs. The basic architecture of our formal testbench is a collection of auxiliary glue logic in Verilog and formal properties (assertions, covers and constraints) in a bound SVA module. Other than this there is nothing else to be done, in contrast to simulation-based verification.

The basic principle of our abstraction is to watch an arbitrary symbolic data (an array of them for multi-packets) called watched data on the input interface of the design and count how many data values are ahead of this watched data. We use a transaction counter that is incremented on every write until the watched data appears on the input

interface. We decrement the counter on every read, and when the counter has reached the value of one, we expect the watched data to appear on the output of the design in the same clock cycle or multiple clock cycles (in case of multi-packets). We describe the details of the solution now. We start by defining a set of watched values by using the logic datatype in SystemVerilog. As this design is multi-packet, at any given time we can have more than one watched packet in the design. Thus, we need a whole set of watched values bound by the maximum number of packets defined by `PKT_LEN`.

```
logic [DATA_WIDTH-1:0] wd [PKT_LEN-1:0];
```

where `PKT_LEN` is a function of the input `pkt_len` and is defined as `1<<PKT_BITS`. We constrain these watched values to be stable after reset. This allows the formal tool to keep these values stable for each run that it executes, but the value in each run is chosen to be unique.

Table 1 shows the key wires used in the testbench. The `ready_to*` signals are used to model the conditions that detect the start and finish of sampling of watched data values both at the input and the output data ports. These signals are used in modelling the sampling registers which are used for tracking the watched data values. The increment (`incr`) and decrement (`decr`) wires (also shown in Table 1) are used for our tracking counter that calculates "*how many values are ahead*" of the watched data.

ready_to_start_sampling_in	sample_in_started_cond || sample_in_started	wire
ready_to_finish_sampling_in	sample_in_finished_cond || sample_in_finished	wire
ready_to_start_sampling_out	sample_out_started_cond || sample_out_started	wire
ready_to_finish_sampling_out	sample_out_finished_cond || sample_out_started	wire
inp_sampled_completely	sample_in_started && sample_in_finished	wire
not_sampled_out_completely	sample_out_started && !sample_out_finished	wire

Table 1: Key signals used in our testbench.

We use four sampling registers – one pair for detecting when the watched data packets have entered the DUT and another to detect when the sampled in watched packets leave the DUT. These registers are cleared at reset and subsequently take on the values of the `ready_to*` signals. Table 2 shows how they are defined.

	Value at Reset	Otherwise	
sample_in_started	1'b0	ready_to_start_sampling_in	reg
sample_in_finished	1'b0	ready_to_finish_sampling_in	reg
sample_out_started	1'b0	ready_to_start_sampling_out	reg
sample_out_finished	1'b0	ready_to_finish_sampling_out	reg

Table 2: Sampling registers used in our testbench.

We now define the smart tracker counter below. We increment (`incr`) the counter so long as we do not see the watched data stream on the input, and decrement (`decr`) the counter every time there is read for non-watched data.

```
always @(posedge clk or negedge resetn)
      if (!resetn)
        tracking_counter <= 'h0;
      else
        tracking_counter <= tracking_counter + incr - decr;
```

We also need to define another counter that simply counts how may watched data packets appear on the output data port once we have read in all the packets but not read them all out. This counter (`counter_out`) is used to establish a check for each watched data packet. We now define two additional conditions that capture when we have sampled in the watched packets completely and when we have started to sample out the watched packets but not completely sampled them out.

```
assign input_sampled_completely    =  sample_in_started    &&  sample_in_finished;
assign not_sampled_out_completely  =  sample_out_started   &&  !sample_out_finished;
```

We then use two fairness assumptions to force a stream of `valid_i` and `enable_i` signals to allow write and read handshakes. The property that establishes that all packets received at the input interface are delivered to the output at the correct time without a loss, reorder or duplication is shown below. We use a generate loop to model these, where the loop itself is sensitive to `PKT_LEN`.

```
generate
for (i=0;i<PKT_LEN-1;i=i+1) begin:all_packets
     but_last: assert property (input_sampled_completely && not_sampled_out_completely &&
                                (counter_out==i)
                                     |=>
                                data_o==wd[i]);
               end
endgenerate
```

Note, that the loop above ranges up to `PKT_LEN-2`, and for the final packet `PKT_LEN-1`, we write a separate assertion. This is due to the design artefact and the way we have modelled our testbench registers that track the data. When `sample_out_finished` goes high then we see the watched data stored at index `PKT_LEN-1` or the watched packet at location 0 is seen if the watched packet was of length 0 at the time of input sampling.

The property that checks that the very last packet has been delivered is shown below:

```
last_packet: assert property (inp_sampled_completely  &&  sample_out_started  &&
                              $rose(sample_out_finished)
                                        |->
                             (data_o==wd[PKT_LEN-1] || data_o==wd[0]));
```

The two assertions shown above are enough to check for reordering, data loss and duplication bugs. We also wrote additional checks to validate that empty and full work correctly and that validation/invalidation is correctly implemented.

VI. RESULTS

When we ran our checks against the design, we started finding bugs in the design. One classic bug that we caught was in the packet pointer calculation where we were wrapping the pointers too soon and were losing the data captured. Once we started to get exhaustive proofs, we moved on to the next step to obtain design coverage. We used two methods – first was to run the formal coverage analysis in the tool (where available), and the second was manual fault injection. Manual fault injection is cheap to run, and we can inject interesting bugs to assess the vulnerability in our testbench to see if the properties that previously passed are now failing.

We used multiple formal tools to check if the coverage data reported made sense and once, we started to get a 100% coverage from the tools, we moved onto the manual fault injection phase. The reason we use both techniques is that formal coverage reported from tools is not yet standardised and we cannot compare like for like results. Having said that we would like to point out that we use tool-based coverage as an integrated mechanism for design bring up. This helps us identify structural defects (*reachability, toggle, deadcode, redundant code*) in the design automatically, saving us precious time later in the verification flow. This is part of our new Axiomise ADEPT FV® agile flow which is vendor agnostic and is available for anyone to use. The flow provides an agile method to avoid (A) bugs, detect (D) bugs, erase (E) bugs, prove (P) that no bugs exist and finally and tape-out (T) without bugs.

Once we fixed all the design bugs and started to get exhaustive proofs on smaller design configurations (such as a 2-deep buffer, carrying up to two packets, each packet being 32-bit wide), we decided to deliberately break the design by inserting bugs to see if they are caught by our checkers. We were able to catch all the bugs. We then decided to start increasing the design configurations to see if the ability to catch bugs scaled as well. We were indeed able to catch bugs in very big design configurations. One such configuration where we caught an ordering bug was a 512-deep buffer carrying up to 8 packets of 32-bit data i.e., nearly $10^{42,248}$ states (about 141K flip-flops). The bug was a random reordering bug. The design would work as normal until a random event happened, and when it did it will randomly choose some locations to reorder.

Once we were happy with the quality of the testbench we decided to assess how well the proofs would scale to establish bug absence for bigger design configurations. We set the timeout for an hour to see what configurations were verifiable. The results are shown in Figure 3. We found out that beyond 8-deep buffers the proofs were not

scaling for up to two 1-bit packets (i.e., 10^5 states). We noticed that we could verify up to 8 packets for a 2-deep buffer, each packet being 1-bit wide (i.e., 10^5 states). Also, we could verify for a 2-deep buffer with up to two packets where packet size can be up to 512-bit wide (i.e., 10^{616} states). It was promising to see that we could scale up to 10^{616} states with such a small test bench for verifying end-to-end properties under an hour of run time on a laptop with 6 GB RAM running two Intel i5 7300 CPU cores. However, we wanted to assess if we could scale any further in terms of the state space. One strategy was to increase the timeout to larger values and we were able to go a little further with the design sizes, but the results were not that impressive. We decided to apply assume-guarantee reasoning at this stage. To scale the proofs what we needed were a set of helper properties specified manually by the user. Helper properties speed up the state-space search by being used as guides by the formal tool. Once proven these helper properties are assumed to drive the proofs of the other properties whose proofs either don't converge or take a long time to converge. This way of leveraging helper properties first to prove, and then assume is called an assume-guarantee flow first discussed by Ken McMillan [3] and which we exploited in our earlier work [1-2].

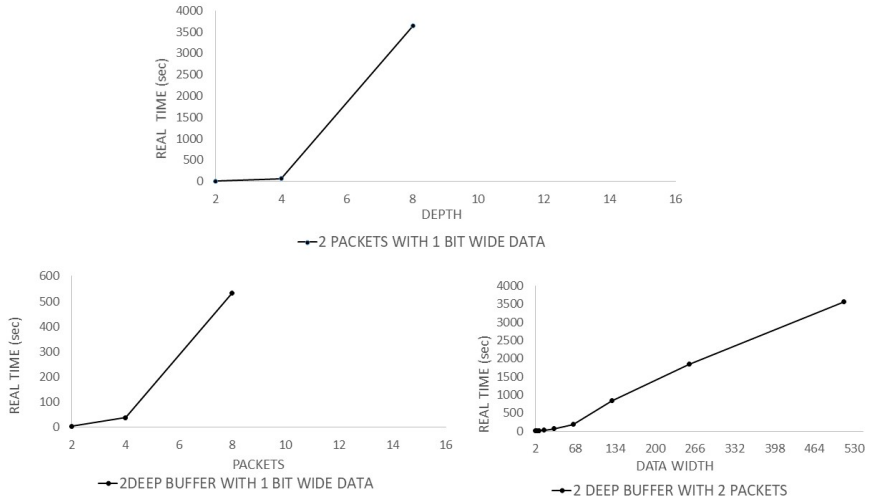

Figure 3: Runtime (real time) with varying depths, width and packet sizes using abstraction. Within an hour we could verify different design configurations ranging from 10^5 states to 10^{616} states. We used Synopsys VC Formal for proof runs.

The key helper properties we used in our case study were:

L0: If the sampling has started and sample out finished then the tracking counter must be zero.
L1: If sample out has finished then sampled in must have started and tracking counter must be zero.
L2: If all watched data input has not been sampled in completely then the tracking counter is less than or equal to the difference between the write and the read pointers.
L3: If all the watched data input has been sampled in but sampling out of the watched data has not started then the tracking counter is less than or equal to the difference between the write and the read pointers.
L4: If all the watched data input has been sampled in and the sampling out has not started then the very first watched data that was sampled in the design must be residing at the location given by the function of read pointer and tracking counter.
L5: If all the watched data input has been sampled in and the sampling out started for the watched data values but not completed, then if the very first watched data value has been seen at the output port then the next data value to be seen on the output will be the next watched data value that was sampled in.

In Figure 4, we show runtime for different parameters with helper lemmas having been used as assumptions.

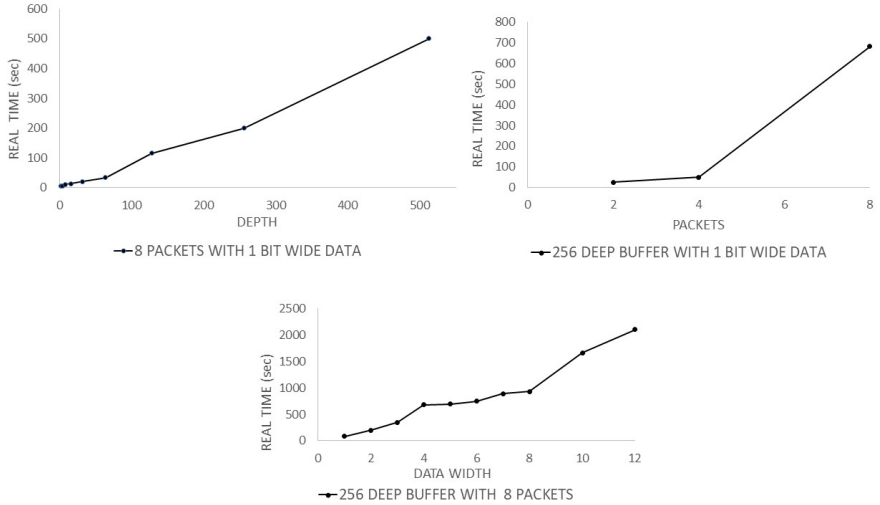

Figure 4: Runtime (real time) with varying depths, width and packet sizes using abstraction and helper lemmas as assumptions. We can exhaustively verify a design with 10^{7397} states (24K flip-flops) in 35 minutes (256-deep buffer with 8 packets, each packet being 12-bits wide).

Depth: For exhaustive proofs establishing that there is no bug in the design, runtimes scale linearly with increasing depth for 1-bit wide but a variable number of packets up to 8 (10^{1232} states). Runtime is 8.33 minutes (Figure 4) for the 512-deep buffer.

Packets: If we fix the buffer depth at 256 for packets 1-bit wide, the runtime scales nicely with the increasing number of packets (Figure 3). It takes nearly 11.33 minutes to exhaustively verify this buffer for 8 packets (10^{616} states). When we allowed up to 16 packets (power of 2 increase), the runtime jumped to 132 min 13 sec (beyond an hour).

Width: When we keep the buffer depth fixed at 256, allowing up to 8 packets to be written but the width of packets can change from 1 to 12, the runtime scales nicely (Figure 4). However, when the bit width became 13, the runtime jumped to 3 hours 13 minutes and continued to increase exponentially with further width increase. We can control this by exploiting property decomposition – splitting the word level check into individual checks on 1-bit data word (a much more tractable problem) but overall this would still make the runtime exceed an hour – in fact it took us 7 hours 54 minutes to verify 256 deep buffers with up to 8 packets where each packet is 32-bits wide on the tablet PC (two Intel i5 cores with 6 GB memory). This astronomical increase in run times with varying width is currently a challenge with all formal tools. We can confirm that the runtime decreases significantly if compute resources were not a limitation.

To summarise our results, we evaluated our solution for both bug hunting and exhaustive proofs. We found reordering bugs in the design in less than 40 minutes for a design that was 512-deep, carrying up to 8 packets of 32-bit data (about 140K flip-flops, $10^{42,248}$ states in the design). We were able to prove bug absence in designs as big as 24K flops (10^{7397} states) in 35 minutes (using helper properties as assumptions) on a tablet PC with two Intel i5 7300 cores and 6 GB memory running inside VMware. If we were not limited by compute resources, then for a design configuration with nearly half-a-million flops ($10^{164,228}$ states), we can find random reordering bugs in under 20 minutes of proof time on a machine for example with 128 GB memory (with 18 Intel CPU cores). This specific configuration was 256 deep, with up to 64 packets each packet being 32-bit wide. We were able to prove exhaustive correctness for this design configuration in under 3 hours on this big machine using a single tool license. We were able to reproduce these results across multiple different tools when we could get the tools to compile and elaborate these configurations successfully.

VII. DISCUSSION

To summarise, the five key steps used in our overall verification reported in this paper are:

1. Developed formal verification testbench by using abstraction and end-to-end assertions, in the presence of interface constraints, and then run properties in the formal tool.

2. Pipe-cleaning the testbench and design – finding testbench issues, design bugs, and tightening constraints. Run formal coverage analysis (akin to simulation functional coverage) to visualise any blind spots – catch dead code, redundant code, over-constraints, and missing checks.
3. Manually inject bugs to assess the robustness of the formal testbench.
4. Check what is the biggest design configuration on which bugs can be found within an hour.
5. Finally, deploy invariants (helper properties) and assume-guarantee flow to improve scalability.
6. Check if proofs scale for bigger configurations.

The reader will note that there is nice linearity in the results in Figure 4 and this is on a problem that is increasing exponentially in size. This phenomenon is due to the usage of helper properties. The problem of verifying multi-packet serial designs is significantly challenging for any kind of verification, not just formal due to the variability in the number of packets. If the number of packets to be written at every clock cycle is fixed rather than variable, the verification problem is hard but much easier to solve. In both cases though, without an abstraction, it is nearly impossible to obtain any proof convergence on even the smallest of design configurations.

For abstraction-based solutions there are many choices, some are more scalable than others. It depends on what is being tracked, how much of it is tracked, and how often. In our solution, we track a 'single' symbolic transaction which in the case of multi-packet design consists of observing a single stream of watched packets. The single transaction abstraction is smart in the way that it allows end-to-end properties to be modelled and verified exhaustively for proofs as well as bugs. In the past, we have used a two-transaction based tracking solution [1-2]. The two-transaction solution makes use of tracking 'two' symbolic transactions rather than one and is similar in concept to the Wolper's colouring method [4]. Our comparisons showed [1-2] that two-transaction is worse than single transaction method, so we didn't employ that for this design.

Abstraction provides a solid foundation for laying the overall solution. The scalable proof convergence is obtained by providing helper properties in the test bench for minimising proof search for end-to-end properties. Such is the impact of these helper lemmas that once they are proven and then assumed, the proofs of end-to-end property converges very fast even on huge design configurations. Overall, the return on investment using abstraction and user-defined invariants is significant as evidenced by the results we show. Without this investment, the problem of verifying such designs with rigour is intractable with all forms of verification including formal, simulation, emulation and FPGAs.

VIII. CONCLUSION

Our smart formal verification solution scales both for computing exhaustive proofs as well as finding bugs in multi-packet serial designs with varying configurations. We believe this scalable solution is smart in the way it leverages abstraction with assume-guarantee to exhaustively verify complex multi-packet serial designs for a vast range of configurations within an hour on a tablet PC with 6 GB memory. When not limited with compute resources we could find random ordering bugs in design configurations with half-a-million flip-flops in under 20 minutes, and exhaustively prove correctness in under 3 hours.

REFERENCES

[1] A. Darbari et al., "*Exhaustive Formal Verification of Sequentially Deep Data-Transport Components*," DAC Designer Track, 2014.

[2] A. Darbari et al., "*More is Less: Exhaustive Formal Verification of Sequentially Deep Data-Transport Components*," Cadence CDN Live EMEA 2014.

[3] K. L. McMillan, "*A Methodology for Hardware Verification using Compositional Model Checking*," Science of Computer Programming, vol. 37, no. 1-3, pp. 279–309, 2000.

[4] P. Wolper, "*Expressing Interesting Properties of Programs in Propositional Temporal Logic*," in Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, 1986, pp. 184–193.

[5] E. M. Clarke, O. Grumberg, and D. E. Long, "*Model Checking and Abstraction*," ACM Trans. Program. Lang. Syst., vol. 16, no. 5.