Predictable and Scalable End-to-End Formal Verification

by Ashish Darbari, Axiomise

In this article, we discuss why formal verification adoption has been limited in industry, and how abstraction-based methodology in formal verification can help DV engineers become successful in adopting formal property checking more widely. Abstraction is the key to obtaining scalability and predictability. It provides an efficient bug-hunting technique and helps in establishing exhaustive proof convergence. We illustrate our methodology on a FIFO in this article, but similar methods are used in the verification of a range of designs ranging from a RISC-V processor to multi-million gate designs.

INTRODUCTION

Despite its rich history, formal verification adoption is growing mainly through the usage of automated apps, but its full potential is hardly exploited with only 35-40% projects using formal property checking, according to Harry Foster's *2020 Wilson Research report*. Formal is not the main functional verification technology of choice in the industry, and we believe that this affects the cost of verification, the overall time-to-market as well as the quality of results.

FORMAL IS ECONOMICAL

Yes, formal verification when used properly can be economical – bugs are caught sooner, corner-case bugs are picked up in the first hour of design verification (DV), and in many cases one can establish through mathematical proofs that the design implementation does exactly what the requirement mandates. In some cases the exercise of using formal helps discover the requirements, refine them, and makes it easier to find validation flaws e.g., inconsistent requirements that can never be implemented in a design. For many problems, formal verification is the only solution, e.g., proving absence of deadlocks, robust floating-point correctness, proving security violations do not occur. In general, whenever we need to establish that a bad behavior must never occur – formal is your friend!

SO, WHAT'S THE PROBLEM?

If the benefits of formal are so well defined, what is preventing the industry from adopting it and running with it? The answer is methodology. While writing properties using SVA and PSL may not be hard, writing them for efficiency to ensure high coverage, predictable proof convergence and scalability with design size is a significant blocker in formal methods adoption.

CONQUERING DESIGN COMPLEXITY

As designs become complex due to requirements of power, performance, area, and now safety & security, the best way of overcoming design complexity is to view the designs from an abstraction lens. As an example, I²C, I²S, UART, USB, Ethernet, PCIe, a video unpacker, a pipelined processor, a packet-based networking block, a GPU memory sub-system, a cache-coherent system and a NoC may not seem to have much in common at first sight, but on a closer look it would appear that they may have a lot in common. All these designs are very hard to verify exhaustively due to the deep sequential nature. If we understand how to conquer sequential design complexity in formal, perhaps it can open up a way to verify a range of these – may be all of them?

What about a humble FIFO? How hard can it be to verify a FIFO?

What is interesting to note is that, precisely due to the sequential nature of these designs the verification timescale becomes unpredictable.

FIFO VERIFICATION

We assume that the reader is already familiar with how a FIFO is implemented in digital hardware. A FIFO is a first-in-first-out design component that stores data maintaining ordering between the data

values. Common ports on a FIFO include a clock (*clk*), an active-low reset (*resetn*), a data input (*data_i*) and a data output (*data_o*) port. We also have signals to push (*push_i)* the data into the FIFO and to pop (*pop_i*) the data out. Last but not the least, we also have two important flags that define when the FIFO is empty (*empty_o*) and when it is full (*full_o*).

VERIFICATION REQUIREMENTS FOR VERIFYING A FIFO

For verification we need to ensure that a FIFO keeps the data ordered, does not duplicate data internally, does not lose data, and is empty and full when it should be. Ordering is a key property that causes a lot of pain for exhaustive verification, especially when FIFOs become deeper and carries much wider data payload. The complexity arises because the entry and exit of a given data value in the FIFO depends on all the prior values ahead of this value. For modelling the ordering property, it doesn't matter how a FIFO is actually implemented in the design, what matters for us in verification is what it does.

ORDERING DEFINITION

We define ordering using any two arbitrary values, let's call them d_1 and d_2. For any two elements d_1 and d_2, if d_1 is sampled in the FIFO before d_2, then d_1 is always sampled out before d_2. So long as this property is preserved by the FIFO design implementation for any two values d_1 and d_2 we can deduce that the FIFO does not violate the ordering property.

SOME POSSIBLE SOLUTIONS

With the above goal, let's find out what are the possible solutions. If we are verifying a FIFO in dynamic simulation, we can easily use a reference model – another data structure such as a queue and compare the states of the reference model with the FIFO design. One thing to note here is that in simulation we would be sending a trace of 0s and 1s. For example, if the width of the FIFO is two bits, we would be sending 00,01,10, and 11 and if we were aiming for an exhaustive simulation, we would send all possible permutations of these to cover the entire depth of the FIFO. Of course, in practice for bigger depths and widths for the FIFO, performing exhaustive simulation won't be feasible and typically randomization – cornerstone of UVM – would take care of ensuring that corner-case scenarios are tested.

For formal verification, can we use a similar approach?

Theoretically, we can, but in practice this approach will not be exhaustive i.e., if we could even use a queue data structure. Modern formal tools require synthesizable constructs and SystemVerilog queue is not synthesizable in formal tools. Of course, we can use another pre-verified FIFO design as a reference model and repeat the flow we described for simulation but the challenge will be scalability – formal tools will not be able to produce an exhaustive result even for very small configurations of FIFOs if we simply choose to copy the simulation-based method. When we introduce another FIFO, we simply duplicate the number of states for overall verification, and as formal exhaustive verification covers all states, we have just created more work for the formal tool.

We need to think differently about this in formal! The solution lies in understanding abstraction.

ABSTRACTION

Abstraction is a complex topic and widely addressed in formal methods literature but not as well understood amongst the formal users. Two most well-known forms of abstraction are black-boxing and cut-pointing. These are standard and somewhat better-known techniques for reducing proof complexity and are certainly valuable abstractions as they hide away the chunk of design behind the point of cut, or the design logic is simply boxed out. In either case, the goal is to reduce the connected design logic and thereby design states to make formal faster.

The abstraction we are interested is on data (data abstraction) and time (temporal abstraction). These allow us to generically verify designs end-to-end.

DATA ABSTRACTION

Data abstraction is natural to formal verification anyway, as formal relies on symbolic execution and checking is done on symbolic representations. Formal verification engineers can exploit this more widely by making use of symbols – each symbol is an abstraction of a binary value 0 and 1. For example, a 2-bit wide symbolic value d used to denote data input to the FIFO will have 4 binary values {00,01,10,11} encoded by this symbol. A 1-bit wide value would encode just 0 and 1. These symbols obtain a logarithmic reduction in data encoding when they are manipulated by the formal verification search algorithms in formal tools.

TEMPORAL ABSTRACTION

The intuition behind temporal abstraction is to not observe every clock cycle but only a few. The question of course is which ones, how many and how do we ensure that we do not abstract too much by observing too little. If we observe too little, we can miss a bug. If we observe all clock cycles, the performance penalty for exhaustive proofs is enormous to the point that formal becomes intractable. In this article, we show an example of a temporal abstraction that we use for FIFO verification.

To obtain confidence we use formal coverage. A formal coverage flow shown below in Figure 1 uses a combination of techniques such as fault injection (mutation), over-constraint analysis, checker analysis and scenario coverage, to build the evidence needed to understand if the verification was complete.

Figure 1: Formal Coverage Flow

We refer the interested reader to our recent webinar where we discuss coverage in detail.

ABSTRACTION-BASED FORMAL VERIFICATION SOLUTION FOR FIFO

We focus on verifying the ordering property. Note that from our definition of ordering above, we immediately infer that we would need to track two arbitrary values d_1 and d_2. The values need to be sized in terms of the width of the data. For WIDTH wide data ports, we will declare them in SV as:

```
logic [WIDTH-1:0] d1;
logic [WIDTH-1:0] d2;
```

During formal property checking, Questa PropCheck will ensure that all specific instances of these symbolic values are tested automatically. The only thing we need to do is to put the following constraints in place.

```
A0: assume property (@(posedge clk) ##1 $stable(d1));
A1: assume property (@(posedge clk) ##1 $stable(d2));
```

The above assumes ensure that at every positive clock edge, the values remain stable between the point of entry and exit in the FIFO, so we observe the sampled value without it getting modified in transit. By not constraining these through assumes, the formal tool is free to drive them to any value.

All possible data values are verified at run time for free by the tool, just as it injects all legal stimulus for free without having to write any by hand unlike simulation. This is data abstraction. All we need to do is to instruct the tool what we need to verify (assertion and covers) and under which conditions (constraints). If the test environment is under-constrained, we will get illegal input stimulus injected by the formal

tool, and we will be looking at spurious failures – not real design bugs. If the environment is over-constrained, we will block legal stimulus – masking real design bugs. So long, as we constrain the environment correctly, we are half-way done! The other half is of course modelling the checker (asserts) so it verifies for all legal input combinations and on all states of the design.

The next step is to identify when to observe the chosen arbitrary values. The when is defined by identifying a start and a stop point. For the FIFO example it is not too hard to identify this. Our start point is the time point at which these values are sampled in the FIFO i.e., when they appear as inputs on the *data_i* port. Remember, the formal tool will place these values for us on the input port, we don't have to write any verification code. Moreover, these values will be placed at all possible time points: first cycle after reset, second cycle after reset, and so on. The tool ensures that all possible permutations are considered automatically. The stop point is the time at which the arbitrary values leave the FIFO on its output port (*data_o*). Also, note that these injections on the design only occur on a push and pop.

It is also worth pointing out that we need to maintain two other side conditions on the input interface of the FIFO during our verification which in practice is maintained by the driving logic feeding the FIFO when this FIFO is embedded in a bigger design. The first condition A2 defines when it is illegal to read from the empty FIFO, and the second condition A3 defines when it is illegal to write into a full FIFO.

```
A2: assume property (@(posedge clk) empty_o |-> !pop_i);
A3: assume property (@(posedge clk) full_o  |-> !push_i || pop_i);
```

Let's define the start and the stop events. The keen reader would have noted by this time, that we will need a pair of these – one for d_1 and one for d_2. We use registers to detect these events – we call them sampling registers. The sampling_in registers get set when the d_1 (resp. d_2) appear on the input ports on a push and they have not been seen before. The sampling_out registers get set when d_1 (resp. d_2) appear on the output ports on a pop and they have not been before.

The signals that define these conditions are shown below:

```
assign ready_to_sample_in_d1  = (data_i==d1) && push_i &&
                                !sampled_in_d1;

assign ready_to_sample_in_d2  = (data_i==d2) && push_i &&
                                !sampled_in_d2;

assign ready_to_sample_out_d1 = (data_o==d1) && pop_i  &&
                                !sampled_out_d1;

assign ready_to_sample_out_d2 = (data_o==d2) && pop_i  &&
                                !sampled_out_d2;
```

The sampling registers are defined for the FIFO with an active low reset.

```
always @(posedge clk or negedge resetn)
        if (!resetn) begin
            sampled_in_d1   <= 1'b0;
            sampled_in_d2   <= 1'b0;
            sampled_out_d1  <= 1'b0;
            sampled_out_d2  <= 1'b0;
        end
        else begin
            sampled_in_d1   <= sampled_in_d1
                               || ready_to_sample_in_d1;
            sampled_in_d2   <= sampled_in_d2
                               || ready_to_sample_in_d2;
            sampled_out_d1  <= sampled_out_d1
                               || ready_to_sample_out_d1;
            sampled_out_d2  <= sampled_out_d2
                               || ready_to_sample_out_d2;
        end
```

We define one last assumption A4 that d_1 is sampled in the design before d_2.

```
A4: assume property (@(posedge clk) !sampled_in_d1 |->
                                    !sampled_in_d2);
```

We now model the ordering correctness property, as noted on the following page.

```
ordering_check: assert property (@(posedge clk) sampled_in_d1 &&
                                                sampled_in_d2 &&
                                                !sampled_out_d1
                                                    |->
                                                !sampled_out_d2);
```

The assertion mirrors the ordering definition we set out earlier, which was that if d_1 is sampled in the FIFO before d_2 then d_1 must be sampled out before d_2. Our assertion detects ordering violations, data loss, and data duplication. But can it catch other bugs as well? It does indeed.

One of these for a FIFO 16,384 deep, carrying 64-bit data is shown below in Figure 2. The snapshot from Questa PropCheck shows over a million flip-flops in the structural COI tab and it took 4 minutes 56 seconds of wall-clock time to find this bug in 5 cycles.

DECODING FORMAL ASSERTION FAILURE

The waveform shown on the opposite page in Figure 3 explains why the assertion failed. The property evaluation starts in cycle 0, shown by the Start green line. We see three push transactions in cycles 0, 1 and 2. In cycles 1 and 2 we see the arbitrary values d_1 and d_2 appear on the input data port. In cycles 2, 3, and 4 we see a pop. After the first pop, the read pointer underflows decrementing by 1 (due to the design bug), becoming 4095 in cycle 3. The data in index 4095 is an un-initialized value that is read out in the same cycle, the tool causing it to match with d_2 thereby causing sampled_out_d_2 to go high in cycle 4, while sampled_out_d_1 is still low triggering a Fire (shown by red line) in cycle 5.

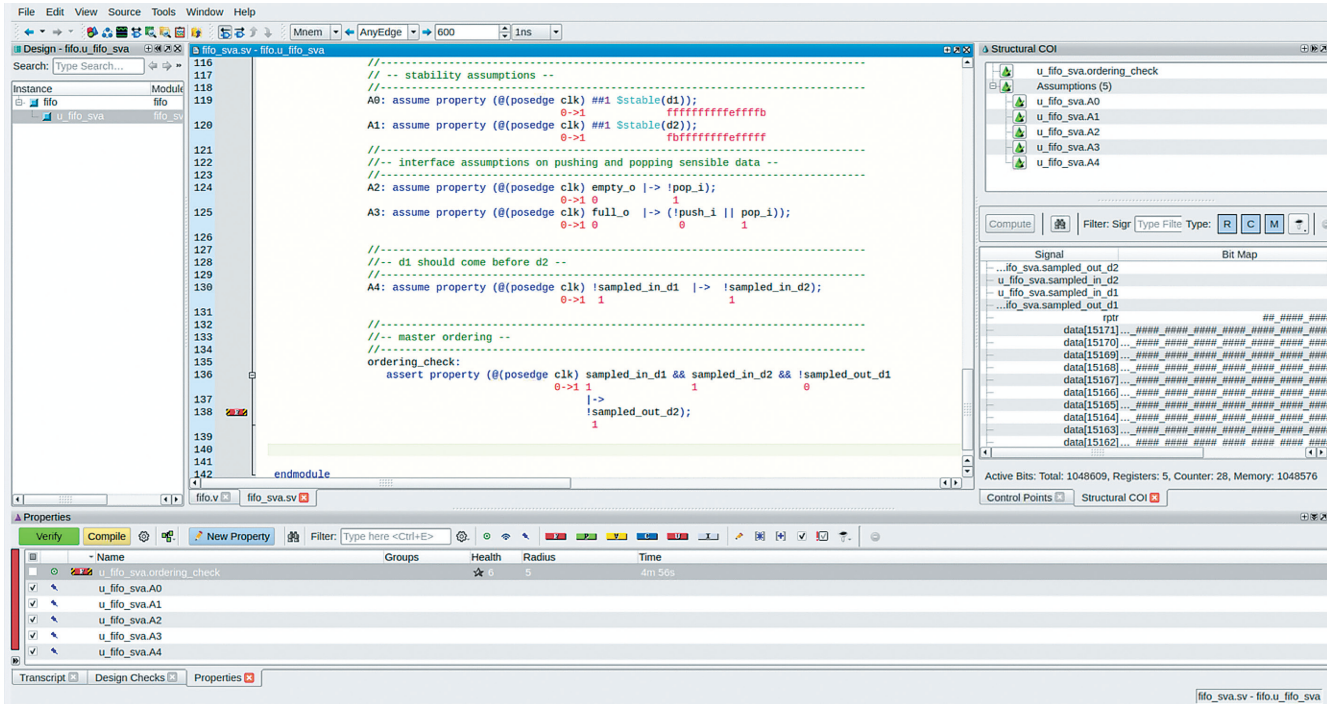


Figure 2: Questa PropCheck

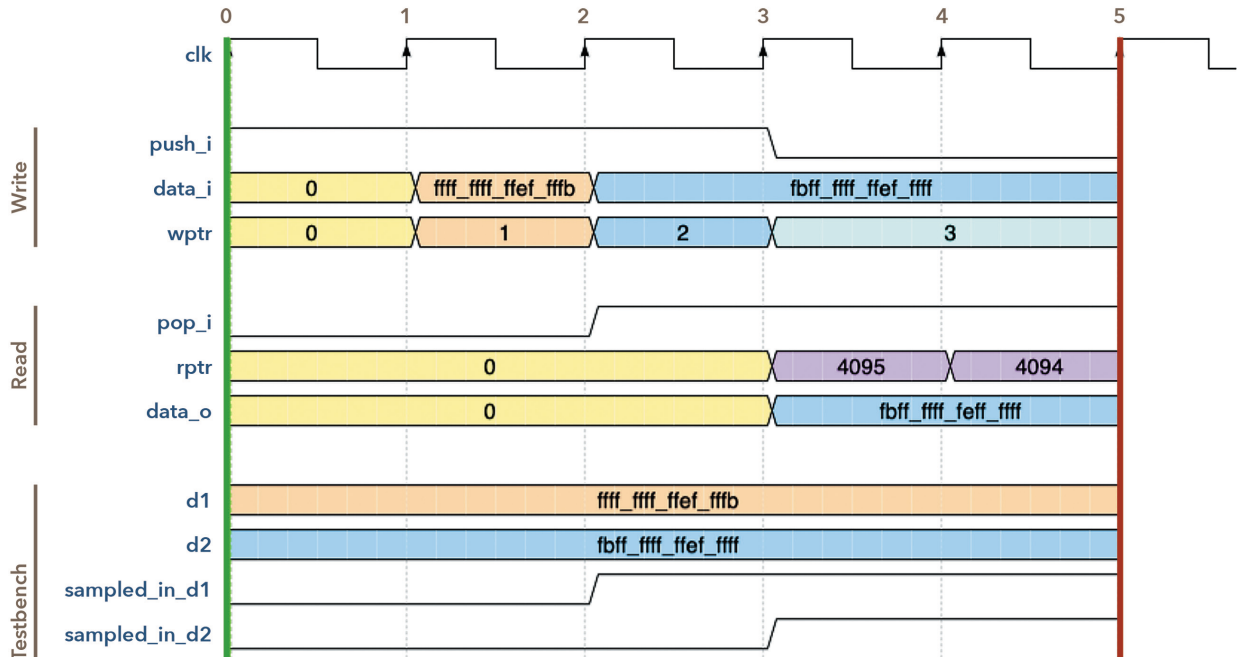


Figure 3: Waveform - Why the Assertion Failed

SUMMARY

The beauty of formal verification is that problem definitions become executable solutions capable of finding bugs within minutes. Using abstractions, we can verify pretty big IP blocks with ease, finding bugs as well as building proofs. In the recent webinar, we outlined the method of finding bugs in a daisy-chain controller with 50 million flip-flops. We have been using formal for over two decades having signed-off numerous big projects. In our training programs, we focus extensively on abstraction, problem reduction techniques, and coverage. Come talk to us and start your exciting career in formal verification.

SIEMENS

Editor:
Tom Fitzpatrick

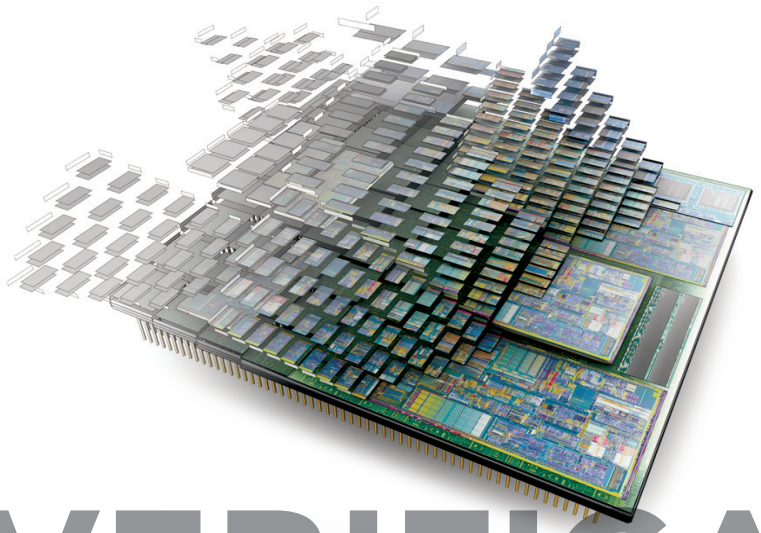
Program Manager:
John Carroll

Siemens EDA
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777

Phone: 800-547-3000

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Verification Horizons is
a publication of Siemens EDA



VERIFICATION HORIZONS

VH