Formal property verification is sometimes considered a niche methodology ideal for control path applications. However, with a solid methodology base and upfront planning, the benefits of formal property verification, such as full path confidence and requirements based property definition, can also be leveraged for protocol driven datapaths. Incorporating layered SystemVerilog constructs to provide a transaction-like protocol description simplifies property creation for both well formed packets and error scenarios. Ultimately though, the key to successful formal datapath analysis is reduction of the typically large state spaces resulting from variable and dynamic packet sizes. Proper interleaving of SystemVerilog helper constructs with protocol targeted assumptions defines a manageable state space and unlocks the promise of formal driven, full path verification for datapaths too.

METHODOLOGICAL CONSIDERATIONS FOR FORMAL VERIFICATION

With the promise of full path verification and traceable requirements, it seems formal is a clear solution to any verification need, but like any verification technique there are methodology specific considerations. First and foremost is potential non-convergence of a formal proof. As a design’s complexity, in terms of state space or controllable variables, expands formal tools require increased horsepower in order to verify the design. For datapaths, this is exacerbated since data transfers are innately temporal.

Control path functionality typically has a limited state space due to its direct map from input to outputs and limited state accumulation. Conversely, datapaths build multi-cycle transactions, each beat expanding the state space and making resolution more difficult. To complicate resolution further, transactions are usually dynamically sized due to variable payload lengths. Since formal methods are performed on synthesized models, dynamic checks aren’t available.

Variable lengths must be accounted for outside of properties in order to avoid elaboration errors. By abstracting and deconstructing the DUT interfaces, state space can be reduced and the dynamic nature of packets hidden from the proofs which verify them.

As with any testbench, careful formal testbench and methodology planning is critical for best chances of success. When dealing in formal methodologies to perform either full design or partial design verification, debug time and state space can be controlled with some early process:

• Perhaps even more important in formal verification than other methodologies is the necessity of defining a concise and complete set of requirements and assumptions from which formal proofs and constraints will be directly derived. Well thought out assumptions naturally
shrink the state space and provide known design restrictions. Since formal provides comprehensive checks, ANY way the tool can break a proof it will. In this way debug can be nondeterministic and without well placed constraints debug time is prolonged.

- State space can be controlled by recognizing components early which tend to elicit large runtimes. Counters and memories pose challenges to the tool due to their expanding state space. However, the functionality of these design components can be abstracted for formal purposes and partitioning them as submodules will ease the process. The submodules can then be verified on their own or as part of a traditional testbench.

**TRANSACTION DECONSTRUCTION**

If the proofs in the form of property defined assertions provide the muscle of the formal flow, a structured SystemVerilog RTL testbench provides the skeleton upon which everything can be built. Each piece of the structure should be built with an eye towards state reduction and efficient property design. By creating a synthesizable description of the well defined transaction header, a dynamic packet can be viewed statically and a basis for evaluation is established. As shown in the formal transaction code below, a structure is used to deconstruct and store each static component of the incoming packet. Much like a UVM driver might dictate transaction behavior at the RTL level, the formal testbench will need to define what a transaction looks like and capture the resulting behavior. By using RTL to define the packet, the formal testbench provides a structure to the transaction interface without restricting behavior. Once transaction state is captured, a basic assertion can be used to check validity at the packet boundary. Checks will be easier to implement due to the deconstructed transaction. Additionally, assumptions and assertions can now be applied at the packet boundary eliminating the need to apply across the full temporal space.

```verilog
typedef struct packed
{
  t_beat  type;
  t_beat  length;
  t_beat[1:0} addr;
}    t_pkt_hdr;

t_pkt_hdr  q_req;

always @(posedge clk) begin
  if (!rst_n) q_req <= '0;
  else if (vld) begin
    case (req_cnt)
      0 : q_req.type  <=req_data;
      1 : q_req.length <=req_data;
      2 : q_req.addr[1] <=req_data;
      3 : q_req.addr[0] <=req_data;
    endcase
  end
end

ast_pkt_chk:
  assert property always @(posedge clk)
$fell(in_pkt) | -> check_pkt(q_req);
```

**SIMPLIFICATION THROUGH CONSTRAINTS**

Once the packet has been described and captured, helper RTL can be used in combination with the previously defined formal transactions to frame and
define transaction behavior. For instance, the code below shows how helper RTL might be used to frame packet length in a synthesizable way.

```verilog
always @(posedge clk) begin
    if (!rst_n) req_cnt <= '0;
    else if (vld & req_cnt) req_cnt <= req_cnt + 1;
    else if (!vld) req_cnt <= 0;
end
assign exp_data_len = q_req.type inside {WRITE, READ_RESP} ?
    q_req.length : 0;
```

In our example, the packet protocol is framed using the valid signal. Deassertion of valid defines a packet boundary as shown by the counter being reset in the “Packet Length” code. When valid is held asserted, the packet size count (req_cnt) increments. By using the structure defined packet fields, an expected length for well-formed packets can be created (exp_data_len). When the packet count and expected length are constrained using SVA assumptions (asm_req_size), the packet is formally framed; asm_req_size evaluates and restricts the helper RTL at the packet boundary ($fell(in_pkt)$). The length field and packet size are thus linked by a synthesizable construct. Additionally, the overall state space has been reduced and proofs will be easier to explore as violations are synthesized away. The figure at the upper right provides a graphical representation of the reduced state space due to the packet size constraint. Truncated and overrun packets are now removed from analysis.

In order to further reduce state space, protocol and design assumptions captured within the project specifications should be implemented as formal assumptions. Each assumption is a documented verification limitation and can be evaluated for its validity and tied directly to a requirements document.

The assumptions below map into the state space chart below demonstrating state space removal for each protocol violation.

```verilog
asm_req_type: assume property
    ($fell(in_pkt) |-> q_req.type inside {READ, WRITE, RESP, ERR});
asm_req_length: assume property
    ($fell(in_pkt) |-> q_req.type inside {1:128});
asm_req_addr: assume property
    ($fell(in_pkt) |-> q_req.addr inside {ADDR_HI_ADDR_LO});
```

**Packet Protocol Restrictions**
Not only do these constraints limit state space and increase likelihood of property convergence, they also act as a natural gate to debug complexity. By including conditions which a design is unable to handle, formal will undoubtedly flag an assertion error. The first failure formal finds will halt analysis until resolved. Because of this, an underconstrained testbench will result in a protracted lesson in “peeling the onion”. A frustrating and time consuming effort as each design constraint is discovered in debug rather than up front.

**FORMAL CONTROL POINTS**

Some of the assumption statements used to limit state space are overly restrictive. For instance, when a DUT receives invalid field values it may be required to report the error and take no action. In that case, it is desirable to isolate checks for good packet behavior from bad. Different sets of assertions will need to be evaluated for each and different sets of assumptions made. Our formal testbench can take advantage of the tool’s intrinsic manipulation of undriven signals by creating new undriven wires called “formal control points”. The formal testbench uses formal control points to gate broad transaction behavior. Using them, we can take advantage of reduced state space for individual proofs while preserving important breadth within the formal environment.

In our example, we want to define “good” and “not good” behavior by adding an undriven wire called “good_pkt”. When formal drives good_pkt high, the captured header and subsequent packet conditions are well-behaved and the DUT should respond accordingly. However, when good_pkt is low, something (or everything, remember it’s formal!) violates the expected packet field definition. Perhaps the type is bad, or address is outside the expected range. In those cases we want to disable good packet checks and make sure the proper error protocol is followed.

The diagram above shows the results of our formal testbench development. An initially broad state space has been narrowed by restricting packet truncation and overrun behaviors which aren’t expected in our system, perhaps eliminated by similar testing upstream. Protocol errors are isolated in order to reduce the local state space of good packet testing which relies on good packet formation as defined by our struct and helper RTL. Using formal control points to isolate good packet behavior will also help us with initial testing. We can focus on reliable good behavior of our DUT before delving into the often treacherous realm of error detection and handling. Additionally, formal control points allowed us to recapture verification space by creating errored packet checks using the same control point.

**CONCLUSION**

Ultimately, whether it is a full formal verification environment, or a complementary piece to an existing testbench targeting critical or difficult to reach functionality, the secret to enabling
datapath verification is managing state space and constraining dynamically sized transactions. An RTL based formal testbench responsible for capturing and simplifying transactions holds and deconstructs the static pieces of the transaction. A synthesizable relationship can then be formed between the static and dynamic packet components and state space is reduced via a collection of assumptions built around the packet defining RTL.
The Most Comprehensive Resource for Verification Training

30 Video Courses Available Covering

- UVM Debug
- Sequential Logic Equivalence Checking
- Portable Stimulus Basics
- SystemVerilog OOP
- Formal Verification
- Intelligent Testbench Automation
- Metrics in SoC Verification
- Verification Planning
- Introductory, Basic, and Advanced UVM
- Assertion-Based Verification
- FPGA Verification
- Testbench Acceleration
- PowerAware Verification
- Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 8100 topics
Verification Patterns Library

www.verificationacademy.com