Demystifying Formal Testbenches: Tips, Tricks, and Recommendations

Shahid Ikram
Marvell, 8 Technology Drive, Westborough, MA 01581 – sikram@marvell.com

Mark Eslinger
Siemens, 46871 Bayside Parkway, Fremont, CA 94538 – mark.eslinger@siemens.com

***Abstract*- Most design and verification engineers have a clear concept of a simulation testbench. While formal verification is an integral part of most project team's verification flows, in many cases it has been limited to formal apps. As their formal experience grows, at some point a design or verification engineer will pursue model checking (AKA property checking). They may hear the term "formal testbench" which conjures up various images and may cause confusion. This paper will discuss the commonalities and differences between formal and simulation testbenches. It will also give a clear description of what components are commonly used in a formal testbench and provide a variety of techniques along with examples that can be used by the novice and expert alike as they develop their formal testbench.**

I. Introduction

As a project verification team expands their usage of formal verification they will typically go through various stages of maturity in their formal expertise. Typically, this begins with using the formal apps, which are targeted to verify specify verification challenges. The benefit of the formal apps is that they are very push button, thus the user doesn't need to know much about formal or writing assertions. Once a verification team decides to add formal model checking to their verification flow they will want to add formal to their test planning [1]. As part of this process, blocks of the design are chosen to be verified with formal and which formal strategies will be used [2]. There are 2 common formal strategies when it comes to model checking: bug hunting and assurance. Assurance is defined as fully verifying the design block with formal which typically entails finding proofs for all the assertions or in some cases finding bounded proofs for some of the assertions. This typically is a harder challenge than just using formal for bug hunting and, of course, bugs will be found when following an assurance methodology. No matter what strategy is used with formal model checking, there will need to be a formal testbench to execute the plan. When the formal testbench is used with a formal model checker, results will be reviewed and potentially many iterations of the formal testbench will be needed to achieve the final goals based on those results. Some detailed topics such as handling inconclusive properties, formal tool setup, and formal coverage are beyond the scope of this paper. The first step in using formal model checking is the development of the formal testbench and its components. This applies to the design or verification engineer who wants to do some simple bug hunting and to the formal expert doing the highest form of verification. The complexity of the formal testbench will vary from simple to complex, usually tied to the experience and expertise of the user. Having said that, simpler is better if you can manage it! The goal of this paper is to remove some of the mystery of developing a formal testbench for the user new to model checking as well as provide some ideas for more advanced users they can leverage as they refine their formal testbenches. For convenience, examples will use Verilog and SVA in this paper. The same principles apply to other property languages, though the implementation may differ slightly.

II. It Starts with a Test plan

One of Benjamin Franklin's famous quotes is "By failing to prepare, you are preparing to fail". A testbench without a test plan is risky business, whether it is related to simulation or formal. A test plan is a contract among all the stake holders of the design under test (DUT) and in its absence, there is no direction and no ending to the verification effort. Most importantly, its absence may create a false sense of security on the simulation side on what formal verification can do and has done. Formal and simulation test planning should be done at the same time. The user can then choose which requirements are to be verified with formal and which using vector-based methods. While the development of a test plan will vary for each verification team, there are some simple guidelines that can be used. When developing a formal test plan there are 7-steps [1] you will go through as outlined below:

1) Identify good formal candidates – control logic with concurrent signals, avoiding data transformation.

2) Create an overview description – Do this for each module to be verified with formal.
3) Define the interface – The simpler the interfaces the better. Identify candidates for formal VIP [9].
4) Create the verification requirements checklist – This is where each requirement is listed out.
5) Convert the requirements checklist into properties.
6) Define the formal verification strategy. The strategy may be constrained by time or design size. A block with high complexity or a verification effort limited by time may require a bug hunting strategy as compared to full proofs. A bug-hunting strategy opens the door for aggressive over and under constraining. A full proof strategy requires a lot of fine tuning of constraints, proof methods etc.
7) Define coverage goals – This may include code as well as functional coverage. Interesting and important scenarios that need to be hit to help ensure you don't have false proofs or have over constrained the design.

Below is a simple example of what a test plan might look like and some of the components to include:

Item	Type	Description	Assumptions	Check	Status
req_eventually	Assumption Cover	Env will always eventually req for each port		s_eventually pend[i] && req[i]	Covered
mode_csr_vals_*	Assumptions	CSR values range		csr_abc <= 2 && csr_xyz <=10	
requests_granted	Cover	All reqs granted		req[i] |=> s_eventually gnt[i]	Covered
req_holds_prog	Check Cover	Basic mode, all req eventually get grant or removed	enable_static	req[i] |=> s_eventually (gnt[i] || !req[i])	Passed Covered
gnts_unique	Check Cover	Only one grant can happen at a time		onehot0(gnt) cover gnt[i]	Passed Covered

The test plan typically starts in a word or spreadsheet document and is ported to a coverage database, such as the UCDB file. Each verification target is uniquely named with a description and what type it is and an expected result. Any required assumptions are listed, and a pseudo check described. Your model checking results will be mapped back into this test plan in the UCDB file as part of overall verification management.

Recommendation: Wait to write your properties until the requirements are done.

III. Formal Testbench compared to a Simulation Testbench

A formal testbench has a lot of similarities to a simulation testbench which you can leverage from a conceptual perspective as you develop your formal testbench and then start adding more advanced capabilities which are unique to formal as your experience in these techniques grows. You will find that there are many ways to develop a formal testbench, based on the style of the person(s) developing them. The various components of a simulation and formal testbench are shown below in Figure 1.

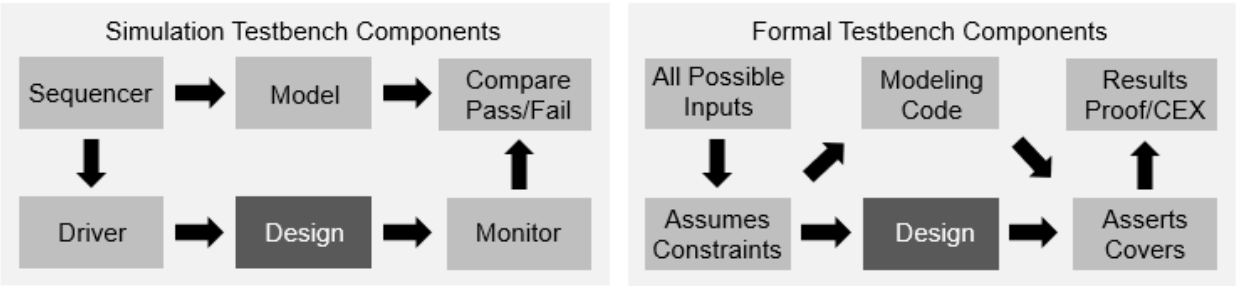

Figure 1: Simulation and Formal Testbench Components

Any type of verification needs two models to compare with each other. The testbench will contain some form of stimulus to the design and second model, and then some method of comparing the outputs of the two models. The components of the simulation testbench contain a sequencer that generates vectors which are driven into the design and model. The outputs of the design and model are compared to make sure they are the same, if not, the test fails.

Assertions, cover statements, and cover groups are often used in a simulation testbench. Property checking is predicated on the use of properties. The formal testbench has similar components with some subtle differences. The full input state space is explored cycle by cycle and is constrained with assumptions as needed. The modeling code and assertions are the second model that the design is compared against. The formal result is proven if the assertion and design function match under all conditions and inputs, otherwise there is a counter example (CEX) that is provided showing the differences. The following table shows some additional difference and similarities between simulation and formal testbenches:

Component	Simulation	Formal
Design RTL (SV/Verilog/VHDL)	Synthesizable and behavioral	Only synthesizable
Properties and coverage	Recommended	Required
Inputs – Vectors	Directed, Constrained Random, ...	All possible inputs explored by tool
Input constraints	Part of driver - transactors	Assumptions remove illegal inputs
Second Model	UVM, C, ...	Assertions and modeling code
Results	Design compared to model – pass/fail	Assertions proven or provide CEX
Implementation	Design instantiated in TB	TB bound or design instance in TB
Ports	TB drives inputs and monitors outputs	TB monitor design signals

A formal test bench can be created in two different forms:

1. *Wrapper around the DUT*: In this format, it is very similar to what is practiced in simulation. The DUT is instantiated inside the testbench. Assumptions are used to constrain the inputs and checks are written about the outputs. The wrapper can be a closed system with only clocks and resets coming in. Other inputs to the DUT can be used as inputs to the wrapper so that TCL scripts can control them directly (e.g. for mode selection). Modeling code can also be used as needed.
2. *Bound with the DUT*: In this case all the properties and modeling code are written in a module that is bound to the DUT. This is often a preferred method as it enables the use of assertions and assumptions developed for formal to be used in simulation if needed. This may be necessary when assertions are inconclusive or to verify assumptions are correct. Modeling code is of course also used here. All the inputs, outputs, and internal signals of the DUT become inputs to the formal testbench.

Next, we will look at some of the components of the formal testbench.

Tip: Use the same signal names in the formal testbench as the design (ports and internal signals), can use (.*) in bind There are multiple ways to bind to the design. (Module, instance, specified port mapping.)

Tip: Formal testbench can make use of multiple bind statements, for the top module and sub modules to test various modes and features as required.

IV. Components of a Formal Testbench

A formal testbench is made up of some or all the following components:

- Clocks
- Initialization sequence
- Properties – required
    1. Assertions
    2. Assumptions
    3. Covers
- Modeling code
- Abstractions

Formal analysis is cycle based so definition of the clocks and their ratios is important. A proper initialization of the DUT is also important. The validity of your proofs is dependent on a known good starting state of the DUT. The setup of the clocks and initialization sequence is typically handled in the model checking tool.

Properties are required for doing model checking. They describe behavior to be verified against the design. Properties comprise assertions, assumptions, and cover properties. How to write a property is beyond the scope of this paper and there are many resources available on writing properties. Here are some points to keep in mind:

Recommendation: Label all properties with a meaningful name, descriptive and/or tied to a requirement or a feature of the test plan.

A. Assertions/Checkers

Assertions are what the formal tool checks against the design. When writing properties follow the 2 great laws [3].

1) Keep assertions as simple as possible.
   - The property is easier to understand and thus easier to debug when a CEX is found.
2) Keep assertions as sequentially short as possible.
   - Single cycle assertions are the most efficient for formal (can still have many cycles in the CEX).
   - Assertions also contribute to state space, the simpler and shorter, the less state.
   - Formal needs to analyze at minimum the depth of the property into the design to get any result. The longer the sequential depth of the assertion, the less likely a proof will be achieved.

Recommendation: assert label has a prefix which indicates an assertion (e.g., a_<name>, assert_<name>).

Tip: Decompose complex properties into a set of simple properties which check the same function.

Trick: When antecedent references a signal which maintains its value, use trigger (e.g., $rose) to minimize CEXs.

Tip: Run assertions in a simulation environment to double-check correctness. Debugging an incorrect assertion in simulation can be faster and easier. This can be done easily and quickly if the assertions are placed in a separate bind file.

B. Assumptions/Constraints

Assumptions limit the input state space to legal values and transactions.

Recommendation: assume label has a prefix which indicates an assumption (e.g., m_<name>, assume_<name>).

Recommendation: Start with a minimal assumption set and add them as needed (be careful not to over-constrain).

Recommendation: Make use of available verification IP for constraining bus interfaces.

Trick: Sometimes it is easier to write an assumption on an internal signal, which constrains the inputs driving it.

Trick: Sometimes moving up a level of hierarchy with a well-defined interface makes it easier to constrain inputs.

Tip: Sometimes all properties on a module's interface are specified as asserts. Then the properties on input pins are converted to assumes via formal tool commands.

Separate these "assumes" in a separate testbench/bind file that can be used in other formal runs as asserts to make sure assumptions are correct. These can be run in simulation if needed.

C. Covers

Cover properties are critical for ensuring formal analysis is exploring the design completely.

- Uncoverable cover properties can point to bad tool setup or over-constrained conditions.
- Target outputs, FSMs, counters, and other constructs which indicate the design is hitting corner cases.
- Useful when using a bounded proof methodology for determining design throughput and depth of analysis.
- Generally useful for test planning documentation.

Recommendation: cover label has a prefix which indicates a cover property (e.g. cov_<name>).

Tip: Write the cover statement using a sequence of signals to be checked.

D. Modeling code

Modeling code is just synthesizable Verilog code that is used to help model various aspects of the design to make it easier to setup your properties.

Trick: Don't define a signal as an integer (32 bits) in the formal testbench. Use exact bit size to reduce state space.

Example 1 from a DDR design [4] shows how modeling code can simplify the writing of properties.

```
parameter PRECHARGE = 7'b11_0010_0;    parameter READ     = 6'b11_0101;
parameter WRITE     = 6'b11_0100;      parameter ACTIVE   = 6'b11_0011;
```

```
reg pre_cke;        always @(posedge ddrclk) pre_cke <= ddr_cke;
wire [6:0] ddr_cmd = {pre_cke,ddr_cke,ddr_cs,ddr_ras,ddr_cas,ddr_we,ddr_addr[10]};
wire precharge    = (ddr_cmd      == PRECHARGE);
wire active       = (ddr_cmd[6:1] == ACTIVE);
wire write        = (ddr_cmd[6:1] == WRITE);
wire read         = (ddr_cmd[6:1] == READ);
```

Example 1: DDR Design Modeling Code

In the above example a "ddr_cmd" signal is created which is the concatenation of DDR control signals. The parameters define the truth table for those commands based on the signals. A flop creates the delayed version of the cke signal. Now, signals can be created which describe the various DDR commands.

Trick: Use hierarchical references to access signals in other instances of the design for your formal testbench.

```
wire [3:0] cstate = top.u1.u2.u_fsm.cstate;
a_fsm_1hot: assert property (@(posedge clk) $onehot(cstate) );
```

Example 2: Using Hierarchical Reference

Tip: Use modeling code to help make the property easier to write as shown in Example 3.

```
// Requirement: Never > 5 outstanding wr's (without a rd) and no rd before wr
reg [2:0] my_cnt;
always @(posedge clk or negedge rstn)
if (!rstn)                  my_cnt <= 3'b000;
else
    if      ( wr && !rd)    my_cnt <= my_cnt + 1;
    else if (!wr &&  rd)    my_cnt <= my_cnt - 1;
    else                    my_cnt <= my_cnt;
a_wr_outstanding_le5: assert property (@(posedge clk)    my_cnt <= 3'd5         );
a_no_rd_without_wr:   assert property (@(posedge clk) !((my_cnt == 3'd0) && rd) );
```

Example 3: Modeling Code for Easy Properties

Example 3 [3] shows how modeling code can be used to simplify the property definition. In the design spec is a requirement that there should never be 5 outstanding writes without a read and that you can't have a read before a write. Writing the SVA to check these requirements is non-trivial. It is easier to just create a simple 3-bit counter. The counter increments when there is a write (and no read), decrements when there is a read (and no write) and remains the same when both or neither write/read happens. The properties are then trivial to write. The "never more than 5 outstanding writes" property just checks that the counter is always less than or equal to 5. The "no read without a write" check just says you can't have a read when the counter is equal to 0. It is far easier and often more efficient to use modeling code like this to simplify the property and usually easier to debug as well.

E. Abstractions

Abstractions [6] are used to simplify the state space of the design or formal testbench with the goal of getting a conclusive result. There are 6 primary types of abstractions which are typically used with tool directives and typically target the design:

- Parameter reduction reduces the parameters to get a conclusive result.
- Constants configure the design for a subset of modes, can be used for over constraining. Over constraining can be useful in bug-hunting if time is limited or for initial exploration but should be fully qualified for a signoff.
- Blackbox modules or instances to remove large amounts of state space.
- Cutpoints cut out complex logic allowing formal to control the cut signal directly.
- Initial value abstractions can be used to simplify the formal run and removes the need to load values.
- Counter/Memory/Arithmetic abstract models automate the reduction of state space.

Details on various abstraction techniques is beyond the scope of this paper. Assumptions to the blackbox outputs or cutpoint can be added to further refine results. For specific cases, modeling code can be combined with simple abstractions such as a cutpoint to support certain properties. An example of this is remodeling a counter such that some critical value can be achieved in a much shorter period of time. This technique is used when a simple cutpoint won't do and is specific to the type of verification being done. In Example 4 a big counter in the design is abstracted with a cutpoint and then constrained with a new model:

```
reg [31:0] cntr;   // counter that counts first and last 16 values
always @(posedge clk or negedge rstn)
if (!rstn)                          cntr <= 32'h00000000;     else
          if (cntr == 32'h00000010)  cntr <= 32'hFFFFFFF0;
          else                       cntr <= cntr + 1;
m_abs_cntr: assume property (@(posedge clk) top.u_big.cntr == cntr );
```

Example 4: Counter Abstraction

This technique can of course be used for remodeling any type of construct in the design to make it more efficient for formal to analyze the requirement being verified.

V. Some advanced concepts

Model Checking can make use of some concepts which are foreign to simulation which include: non-determinism, data independence, symbolic variables, and phantom connections.

Non-determinism (ND) is a mathematical concept used in various algorithms including some formal algorithms. ND allows the user to take advantage of formal's capability to explore all possibilities, temporally as well as spatially. The developer of a formal testbench can make use of this concept in their modeling code and properties. Formal has the ability to drive an undriven wire which can be used to the developer's advantage in the formal testbench. One way of looking at this is the ND input allows formal to choose when and what happens with regards to the assertion being verified. When writing an assertion, the user describes what a violation or behavior is and formal figures out the how. This works great for bug hunting as well as for finding proofs. Example 5 shows the use of ND in modeling code to verify a DDR[4] requirement, leveraging signals from Example 1:

```
  wire ND_start; reg my_wr; reg [2:0] my_ba;
  wire same_pre = precharge && (ddr_ba_addr == my_ba);
  always @(posedge ddr_clk or negedge reset_n)
  if (!reset_n) begin my_wr <= 1'b0;   my_ba <= 3'b000; end else
      if (ND_start && write && !my_wr) begin
             my_wr <= 1'b1; my_ba <= ddr_ba_addr; end
      else begin my_wr <= my_wr;  my_ba <= my_ba; end
a_wr_to_pre_11: assert property (@(posedge ddr_clk)
                                 $rose(my_wr)  |-> (!same_pre)[*11] );
```

Example 5: ND Usage

For this DDR design there was a requirement that there could not be a precharge after a write to the same address within 11 cycles. This example used formal to find a post-silicon bug. This technique can be applied pre-silicon. It is near impossible for a user to come up with the bug scenario, so let formal figure it out since formal is exploring all the state space. To setup the scenario, a ND_start signal is defined, which is the ND input. Two registers are defined, my_wr and my_ba. Next, any DDR transaction can happen, then at some point of formal's choosing, the ND_start signal is set for a specific write. That sets my_wr to be true and saves the bank address, my_ba. A signal, same_pre, is defined, which is a precharge to the same bank address as the saved bank address from the chosen write. The property is simple, saying for a chosen "write", you can't have a precharge to that address within 11 cycles. There is some sequence of DDR commands that puts the design into such a state that when a write is applied, it violates the assertion. The power of ND allows the user to choose the scenario, then formal figures out all the possibilities and corner cases to find the CEX or prove the assertion holds in all scenarios.

Recommendation: DO NOT use ND inputs in assumptions (formal can set ND to ignore illegal stimulus).
Note: Simulation doesn't handle ND inputs either and won't drive them, so Formal VIP using ND is not useable in simulations.

Data Independence (DI) is a concept that can be used to simplify the state space by using fewer data bits in analysis, typically 1 bit. The requirement here is that the system/design being tested doesn't depend on the values of the datapath. This technique is typically used in data integrity/scoreboard applications where the control logic can cause issues in the datapath and it isn't dependent functionally on the datapath. In these scenarios only 1 bit of the datapath (or more in some scenarios) is used in the analysis. A technique for data integrity called the Wolper Method [5] is used in scoreboards and can be used in similar customized data integrity applications. Figure 2 shows an example of how this method can be setup and used. These techniques also typically make use of ND.

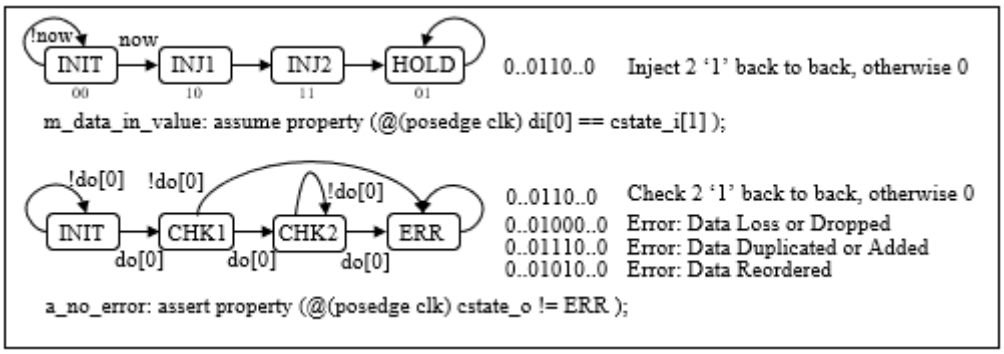

Figure 2: Data Integrity Showing ND and DI

The example in Figure 2 makes use of ND with the "now" input and DI by only working bit 0 of the data in and out busses. Two FSMs are created in the modeling code. For the input FSM at the top an ND input "now" is used to let formal choose when to inject the 2 1's back-to-back. Otherwise 0s are driven. In the assumption above the FSM encoding makes it easier to drive values on the input bit, di[0]. The 2nd FSM checks the outputs. When the 1st 1 happens, the FSM enters the CHK1 state, if another 1 is seen the CHK2 state is entered. If a 0 is seen in the CHK1 state, the ERR state is entered. From the CHK2 state if 0's are seen, then the FSM remains in the CHK2 state. If a 3rd 1 is seen from the CHK2 state, the ERR state is entered. The property then becomes trivial, simply stating the check FSM should never enter the ERR state. This data integrity check will catch data loss, duplication, and reordering. The "now" ND input allows formal to find that specific condition when there is one of the data integrity bugs. If proven correct, then there is data integrity in all scenarios. This is one of the powers of using formal in that a user doesn't need to figure out the condition of the bug scenario, let formal figure it out! In the above example specific write and read conditions aren't shown and would need to be instrumented for correct design operation.

Symbolic variables, also known as Rigid variables [7], are signals that are stable and hold a value for each trace that formal can choose. When setup correctly for a formal run, formal analyzes all possible values of the symbolic variable and will choose a specific value if formal finds a bug in the design. When setup, the symbolic variable has the effect of doing a bunch of formal runs in parallel. It is a state space reduction technique in that the symbolic variable doesn't change value during a formal run, yet all values are analyzed. These can be setup on primary inputs of the design, internal signals of the design, and/or signals in the formal testbench. Example 6 shows an internal configuration register setup as a symbolic variable.

```
m_config_1_lo_val:  assume property (@(posedge clk) config_1[3:0] <= 4'h4 );
m_config_1_hi_1hot: assume property (@(posedge clk) $onehot(config_1[7:4]) );
m_config_1_stable:  assume property (@(posedge clk) $stable(config_1) );
```

Example 6: Symbolic Variable Usage

In Example 6, the "config_1" signal is an 8-bit configuration register. The lower nibble can take on any value 4 or less. The upper nibble values are onehot. The user would use a cutpoint on this register so formal can control it directly. The first two assumptions define the legal values for the lower and upper nibbles respectively. The final

assumption makes the “config_1” signal stable for the duration of the formal run. The benefit of this is that state space and depth of analysis is saved by adding the cutpoint (no CSR transactions are needed to configure specific values for “config_1”) and the values of “config_1” remain stable during the formal analysis (which is typically true of a configuration register). Since it is a symbolic variable, formal (while keeping the value constant) will explore all possible combinations of these bits given the assumptions that were specified. If there is a bug, the configuration that caused it is known immediately. If it is proven, it is proven for all combinations of the register.

Tip: Use symbolic variables instead of setting inputs to constants allowing formal to explore all legal values.

Phantom connections are signals that are connected by how the property is defined and not with any real wires. This is an example of using the antecedent of a property to further constrain the design. Your imagination is the only limitation. An example is doing ECC verification as shown in Figure 3.

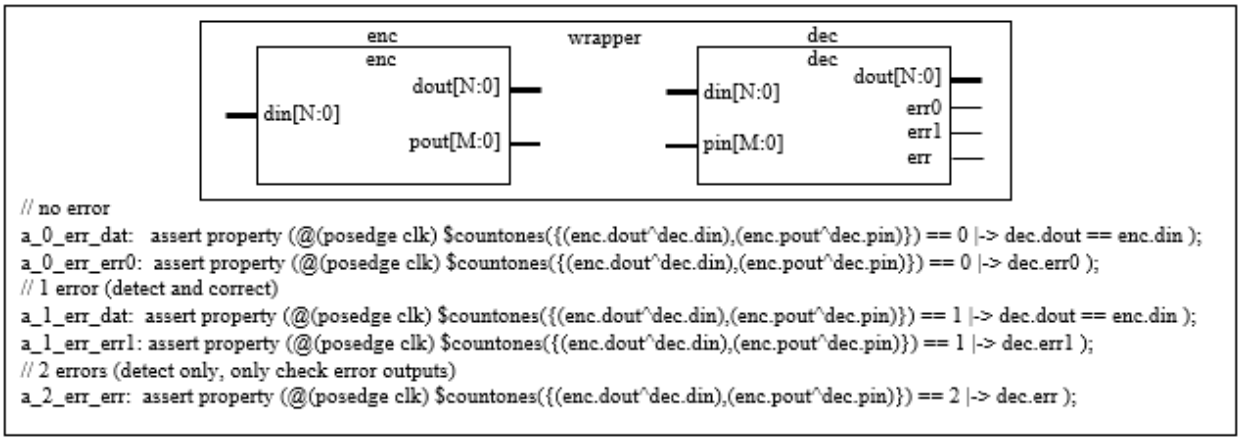

Figure 3: Phantom Connections in ECC Verification

ECC is commonly used with memories to ensure a bit flip in the memory isn’t passed down stream when being read. The ECC encoder and decoder logic can be completely verified without knowing what the algorithm is. In these cases, the algorithm is calculated in a single cycle and is typically combinatorial in nature. The encoder will calculate some parity bits, normally on the output, and these parity bits from the encoder go into the memory. The same bits come out of the memory into the decoder which calculates the syndrome and outputs the original data. These ECC algorithms typically correct single bit errors and detect double bit errors. In the above example, both the encoder and decoder are instantiated in a wrapper. The assertions check the data and the error signal. The antecedent connects the “phantom” wire between the encoder and decoder. The XOR of the encoder outputs with the decoder inputs, which formal is controlling, makes sure the data is passed across as is, with 1 error, and with 2 errors as specified with the $countones function. It is possible to perform this check in the context of the real design if cutpoints are used on the inputs of both instances. Including the memory in this verification model would increase the state space too much. Other verification would be done on the data going into and out of the memory. The output error signals would also be checked that they are onehot, that check isn’t shown above.

VI. Architecting a Formal Testbench

When creating a formal testbench for a block, the verification model should consider the objectives regarding targeted bug hunting or full proofs [6]. State space is another consideration and is something a user will have some control over in the development of the formal testbench. All these considerations come into play in how you approach your model checking setup. Figure 4 below shows a multiplexed bridge design, that has 256 different configurations:

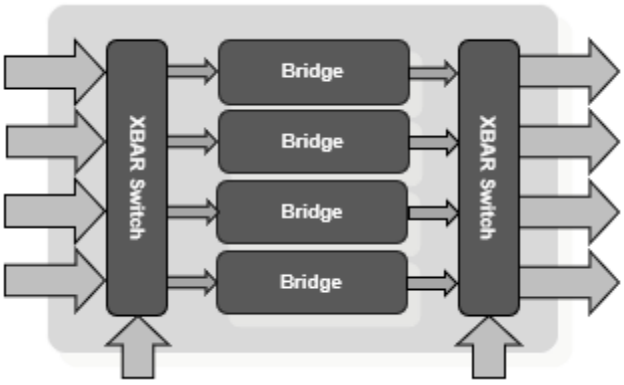

Figure 4. Multiplexed Bridge Design – 256 combinations of paths – selects stable during data transmission

There are several ways to approach verifying this design with your formal testbench including:

Divide and Conquer - if the state space is large, the problem can be broken up into smaller sections:

- Verify data integrity on the bridge with a scoreboard, if all bridges in the design are the same only verify it once.
- Verify the XBAR switch function, if the same in the design, only need to verify the XBAR once.
- Use a connectivity app to verify the connections between the blocks.

Brute Force - Verify the block with 4 scoreboards end to end with the selects hard coded. This approach will need 256 runs. This is also a method which may be used if a more general setup was producing inconclusive results.

Multiple Configurations – While this method may not apply here, in some cases the user may swap in different property sets and modeling code using ifdefs or `include statements. In many cases there will be multiple bind files at different levels of hierarchy. If there are assumptions needed on the inputs, those can be in a separate bind file so that they can be used as assertions in the block driving this block to verify their correctness, or potentially taken to simulation if needed. Earlier guidelines mentioned to name properties such that it is easy to convert them to assumes or asserts depending on the initial specification.

Elegant – This method makes use of most of the advanced techniques mentioned above and is efficient from a state perspective as you can verify the whole block using only 1 scoreboard checker as shown in Example 7.

Trick: The old concept of design for verification can often make setting up your formal testbench easier.

```
// ND: formal picks the value for these signals
wire [1:0] i, j, k;
// create symbolic variables, only pick one input, bridge, and output channel
m_i_stable: assume property (@(posedge clk) $stable(i) );
m_j_stable: assume property (@(posedge clk) $stable(j) );
m_k_stable: assume property (@(posedge clk) $stable(k) );
// REQ: Hold the select lines stable during the run (only change when no data)
m_sel_i_stable: assume property (@(posedge clk) $stable(sel_i) );
m_sel_o_stable: assume property (@(posedge clk) $stable(sel_o) );
// Formal selects 1 input channel(i) to go to one bridge channel(j)
m_ch0_si_10: assume property (@(posedge clk) j == 2'b00 |-> sel_i[1:0] == i );
m_ch1_si_32: assume property (@(posedge clk) j == 2'b01 |-> sel_i[3:2] == i );
m_ch2_si_54: assume property (@(posedge clk) j == 2'b10 |-> sel_i[5:4] == i );
m_ch3_si_76: assume property (@(posedge clk) j == 2'b11 |-> sel_i[7:6] == i );
// Formal selects 1 output channel(k) to come from one bridge channel(j)
m_ch0_so_10: assume property (@(posedge clk) k == 2'b00 |-> sel_o[1:0] == j );
m_ch1_so_32: assume property (@(posedge clk) k == 2'b01 |-> sel_o[3:2] == j );
m_ch2_so_54: assume property (@(posedge clk) k == 2'b10 |-> sel_o[5:4] == j );
m_ch3_so_76: assume property (@(posedge clk) k == 2'b11 |-> sel_o[7:6] == j );
```

```
// scoreboard - formal selects 1 of 4 inputs to 1 of 4 bridges to 1 of 4 outputs
// Analyzes all combinations using ND, DI, and symbolic variables
scoreboard u_scoreboard (.clk(clk), .rstn(rstn),
           .din(din_lsb[i]), .enq(wr[i]), .dout(dout_lsb[k]), .deq(rd_r[k]) );
```

Example 7: Elegent Formal Testbench Setup for Multiplexed Bridge Design

There are 3 ND variables (i,j,k) used to select the input (1 of 4), the bridge (1 of 4), and the output (1 of 4). DI is also used to only check the LSB of the data path. This can be seen in the scoreboard checker port connections. The bit of the data path could also be selected by formal if desired. The 3 ND inputs and selects are made stable (symbolic variables). The mux settings are determined by the i,j,k values. With this setup, only 1 path is verified by the scoreboard as chosen by formal. Since symbolic variables are used, formal analyzes all 256 combinations!

VII. Proof of Completeness

The last mile of the any verification effort is a proof of completeness, that is, are we done or not. Theoretically speaking, for any reasonable design, it is impossible to achieve proof of completeness whether we use simulation, emulation, or formal. The next best goal is to reduce the risk of missing something important. To that end, we can separate what features should be covered and what invariants must hold true for the DUT. The first challenge can be dealt with using functional coverage. The second challenge requires checkers or assertions.

A. Functional Completeness:

The goal of functional completeness is to ensure we are covering all the functional intentions of the design under test. This is a subjective target and hence demands inputs from all the stake holders. A formal test plan can help greatly to this end. This formal test plan must be part of overall test plan of the DUT and must clearly state what features will be (partially of fully) verified by formal. Every feature must map to a set of coverage and assertion properties with back annotation in the testbench as well. The simulator deals with functional completeness using functional coverage metrics. Formal tools are available to generate functional coverage. Most of the available tools let you merge simulation and formal coverage to generate a comprehensive report.

B. Checkers Completeness

Checkers are needed to ensure there are no violations of basic propositions of the design. They are the main bug-hunters and a missed bug almost always mean a missed checker. One way to deal with this is using classic fault injections methods to insert random faults in the implementation and watch if at least one check fires for it. The advantage of this approach is that it is indifferent to the design details. However, there are cases where a more subjective, directed approach can be used through design insight, like disabling certain feature should be a problem.

Coverage is a big topic which is beyond the scope of this paper. At some point you will need to merge coverage metrics together with other formal engines. Formal coverage is different and there are considerations when merging formal coverage [8] with coverage from other engines.

VIII. Putting It All Together

By now you know there is nothing mystical about a formal testbench. Each one is as unique as its creator. Whether you are just starting out with model checking or a seasoned veteran, you can make use of these techniques to create your own style. The main thing is to start where you are and add to the tools in your formal toolbox. Remember to consider the following when developing your formal testbench:

- Proper test planning is important before development of your formal testbench to ensure success.
- Decide on a structure for your formal testbench and make use of techniques mentioned above.
    - Start simple and expand as you gain more experience with these techniques.
- Have a coverage strategy in place to make sure your formal testbench is complete and can be tied back to the test plan and other verification management systems.

Over time you will be using most of the techniques that have been discussed in this paper and expanding on them. This paper is too short to list every tip or trick that can be used. Discuss and share with your colleagues and continue to learn. You will only learn by doing and experimenting.

REFERENCES

[1] H. Foster, L. Loh, B. Rabii, V. Singhal, “Guidelines for creating a formal verification testplan,” DVCon US 2006.
[2] Ram Narayan, "The future of formal model checking is NOW!", DVCon US 2014.
[3] Jeremy Levitt, et al., “It’s Been 24 Hours – Should I Kill My Formal Run?” Short Workshop 6, DVCon US 2019, February 2019.
[4] Blaine Hsieh, et al., "Every Cloud - Post-Silicon Bug Spurs Formal Verification Adoption" DVCon US 2015.
[5] Bernard Murphy, et al., “The Wolper Method” semiwiki.com and Oski Technology, June 2018.
[6] Vaibhav Agrawal, “Formal Verification of Macro-op cache for ARM Cortex-A77, and its successor CPU”, DVCon 2021, February 2021.
[7] Eduard Cerny, et al., “SVA: The Power of Assertions in System Verilog”, Second Edition, Springer 2015.
[8] Joe Hupcey, et al, “How to Avoid the Pitfalls of Mixing Formal and Simulation Coverage” DVCon US 2022.
[9] Eduard. Cerny, et el, “Guidelines for System Verilog Assertion IP Development”, DVCon 2006