About four years ago I gave a couple of talks on the myths surrounding formal. Although, formal has seen more adoption since then, we have a long way to go before it is recognized as a mainstream technology used throughout design and verification. I still see some of these myths clouding the judgement of end users and their managers. Last year at DVCon US, I was invited to be on a very entertaining panel on whether one should go deep or go broad with formal. We discussed whether one needs a PhD to do formal! Well one thing we could all agree was that you don’t. What you do need is the combination of a disciplined process, plan and execution of good rules.

Over the last decade, I have worked on projects with engineers at the coalface of design verification and trained nearly a hundred engineers in how to apply scalable formal verification techniques in practice at some of the leading user and EDA companies in our industry. That experience has reinforced my belief that for those of us who have moved beyond the myths and are busy applying formal on projects there are a few ground rules that help you get the most out of formal.

Here is my Top 10.

**RULE #1: THINK ‘WHAT’, NOT ‘HOW’**

By ‘what’ here, I mean requirements and specifications.

For the record, requirements are not the same as specifications. While requirements describe the high-level intent, specifications are meant to be a lot more precise and should capture detailed aspects of functionality, timing, performance, power, area, safety or security.

I have often noticed engineers get confused about what a specification should look like. A good specification describes the ‘what’ of the design, not the ‘how’. You must keep this difference in mind, otherwise what can happen is that the verification engineer very easily ends up adopting the ‘how’ of the design to architect the testbench. The problem is severe for formal as most practical formal verification relies on building glue logic using Verilog/VHDL and having been exposed to ‘how’ as opposed to ‘what’ makes the verification engineer mirror the design code into the verification test bench causing design bugs to be masked as the design and test bench end up matching each other even for buggy cases. A lot of complex end-to-end verification needs knowledge of the detailed micro-architecture of the design, but this does not mean we need to know how the design has been instrumented; all we need to know is the relationship between the inputs and the outputs - the IO boundary could be moved between the primary inputs/outputs to primary inputs/internal interfaces; or internal interfaces/primary outputs. In either case, what is the expected behavior is important, not how it has been implemented in the design.

**RULE #2: TALK TO DESIGNERS, DON’T JUST READ THE RTL**

This is the proposition that if the specification is not available, one can easily read the design code to figure out what the design does. Wrong!
Whereas, there is value in a design code review and it must be carried out at some point in the life cycle of the project, reading the design to understand the specification cannot be the first port of call in case the specification is not there or is not clear.

I have seen many bad things happen when people just read the design code as substitution for a good specification. The pressures of the project often mean that once you read the design code you are going to model your testbench much the same way as the design is. This creates common mode problems masking design bugs.

Another bigger problem is, that by reading the design code with the intent of understanding what the design does, as opposed to reading the design to review it (when you are in challenge mode), you are very likely to agree with even the broken side of the design, masking bugs in intent, and masking validation bugs.

Also, as a verification engineer one must acquire independent knowledge about the domain and challenge the designer to break the design. The goal of a good test bench is to be able to break the design, and if you cannot break it then you must strive to prove that it works correctly which is possible with only formal verification.

**RULE #3: PRIORITIZE, PICK YOUR BATTLES**

I am not a big fan of cutting corners during verification. People who know me also know that I can be a bit of a pain when it comes to signing off a design too soon.

When I say get your priorities right, what I mean is that you must think of what you would like to achieve with formal in the time you have, based on the skills your team has, and against the backdrop of known risks. Verification is risk assessment, and formal is no different.

Yes, state-of-the-art formal tools are mature enough and when combined with solid methodologies can be used for end-to-end verification on almost all kinds of designs. But the bitter truth is that we have to pick our battles.

One should use formal right from the very beginning of the design bring up down to sign-off but you must have a plan to use formal to hunt down bugs that are extremely difficult if not impossible to catch with simulation. I have seen good examples of projects that have deployed interface checks using assertions across the IPs to exercise them with both simulation and formal. However, this also comes at a cost and might not catch everything.

It may be better to deploy formal to catch lockups, livelocks and other such artifacts that are difficult to catch with simulation. The right approach here is to combine formal with simulation so that you get the best of both worlds. Whereas with simulation, stimulus generation is the most challenging task, with formal, you get that for free. Develop the right strategy to deploy formal that covers your requirements.
RULE #4: PLAN, MEASURE AND TRACK
Modern formal verification on big projects is not just a matter of writing a few assertions or running automated apps from your favorite vendor. Use a proper verification planning tool and devise a clear verification strategy document. And subject that strategy to timely review by your design and verification leads to ensure everyone is on the same page.

Tracking the progress of formal verification runs is very much doable with verification planners and several EDA vendors offer such solutions. The exercise entails a sequence of planning, recording, executing the testbench and then integrating it back with coverage results to give you an end-to-end view of what has been achieved. I will not lie to you, but I have seen big organizations with great technology forget this simple rule.

Here are some examples of the things you should understand:

- What do the different error messages mean?
- How do you set up a design to get the most out of proof engines (also called solvers)?
- How do you effectively cluster properties in different sessions to maximize throughput?
- How do you determine what to do with the tool when you get inconclusive proofs?
- And, critically, do you know what your tools and apps can do - and what they cannot?

A few years ago, I was using a tool that claimed to have deadlock detection. When I inspected it closely, it appeared, as far as I was concerned, not to have that feature. It turned out that the definition of deadlock used by the vendor was not the same as mine. The moral: Don’t be afraid to ask questions of your vendor. Most will provide helpful answers. No question is a bad question.

RULE #5: KNOW YOUR TOOL, USE IT PROPERLY
It is very easy to get started with a formal tool these days but knowing how to exploit it in enough detail is another matter. For example, compiling, elaborating and running a tool does not take long, as long as you don’t run into multi-language issues with VHDL and SystemVerilog. However, it pays off if you understand a bit more on how your tool can help.

A few years ago, I was using a tool that claimed to have deadlock detection. When I inspected it closely, it appeared, as far as I was concerned, not to have that feature. It turned out that the definition of deadlock used by the vendor was not the same as mine. The moral: Don’t be afraid to ask questions of your vendor. Most will provide helpful answers. No question is a bad question.

RULE #6: START SMALL, THINK BIG
I have met many engineers, including formal experts, who are keen on such well-known formal verification reduction tactics as black-boxing or cut-pointing. However, this is not the best starting point for problem reduction. Often compiling a smaller representative design will help you achieve bigger gains if the design being verified is very close to the original design. This is what I call ‘Start small, think big.’ The idea is you get a snapshot of the original big design but you get to play with a much smaller state space.
Black-boxing and cut-pointing are indeed very useful when applied in the correct setting. But they can create more problems than you imagine. For example, when done aggressively they can cut off design logic when you actually need it, causing under-constraining and yielding spurious assertion failures.

Under-constraining can be both good and bad. When done within a context where you are exploring failures it can be a very powerful technique. But most of the time, it is considered a bad practice as it causes a lot of time to be wasted on debugging spurious failures.

Over-constraining is considered dangerous as it causes legal input stimulus to be blocked by the formal tool and hence the tool cannot search the state-space blocked by the over-constraint masking bugs in the design. However, over-constraining can also be beneficial in some circumstances. An example of this is when a verification engineer has failing traces and the designer doesn’t believe the trace represents a valid design bug. This happens because the waveform has some signals in a state that the designer believes is not valid. By selectively applying constraints and rerunning the formal runs, one is able to obtain a trace which looks realistic to the designer convincing him that it is a design bug. Once the design bug is fixed, one must remember to finally relax these over-constraints before tagging verification as complete.

There are other well-known reduction techniques such as the use of induction, abstraction, and invariants that can assist in breaking down a big state-space search into a very small state-space search providing exhaustive proofs and very fast bug hunting capabilities, but they will be a topic for another day. Recently, I reported finding bugs in a design with 1/2 billion gates (approx. 126 million+ flops) using a combination of end-to-end checks and abstraction. Some of the properties that caught the bug had the whole design in the cone-of-influence (literally 126,760,721 flops).

**RULE #7: EXPLOIT YOUR RESETS**

Most designs have one reset pin, but it is not uncommon to find designs with several resets – especially designs with low power having a warm and a cold reset.

Most formal tools are smart at detecting a reset pin and applying the reset for the right duration. However, controlling the duration of the reset is something a verification engineer should be aware of as is the application of the reset sequence for the right number of clock cycles. Things get a lot more interesting with resets as most checkers are disabled on a reset to avoid spurious fails (using the `disable iff` construct in SVA for example).

However, some checks may need to be active during warm resets and that is something else to keep in mind. Also in many cases, interesting states can be reached by the tool by not applying a reset which will often lead to some bugs being found a lot quicker. By not applying the reset, the tool is free to explore states that are usually beyond the reachable state space from reset, causing the tool
to take on any start state and in some cases, this can cause the tool to pick a start state which is only a few clock cycles away from the real bug. However, this technique must be used with caution as it can cause spurious states to be reached that are otherwise unreachable. Nevertheless, don’t be too scared to play with reset settings during your verification.

**RULE #8: CONFIGURATIONS GALORE**

Most IP designs have massive reconfigurability. So, checking them exhaustively for all configurations is prohibitively expensive.

There is no commercially available tool that can check all configurations without recompiling the designs for each configuration, wasting a lot of time. The best thing to do in this case is to obtain the most commonly used configurations as well as take a small sample of random configurations to test the design. It would be nice here if the EDA tool vendors provided a slick way of verifying multiple configurations at once.

**RULE #9: DON’T LEAVE COVERAGE TO CHANCE**

This topic has been close to my heart for a good few years. Coverage with formal can be achieved today based on a range of different techniques available from different EDA vendors. It comes in many flavors.

Starting from structural coverage one explores reachability conditions for design implementations and looks for dead code and unreachable states. However, where coverage is badly needed so that you can sign-off verification with formal is when a combination of coverage techniques evaluate the quality of the verification environment using a very detailed coverage analysis exploring proof bounds, cone-of-influence analysis and mutation-based coverage. Each technique offers some insight into what is covered by functional verification checks and what is not. I have used all these techniques in practice and there is a trade-off that one has to make between what the tool can report back to you versus run time and memory footprints. Whereas proofcore and cone-of-influence analysis represent a good starting point, you cannot sign-off formal verification purely on their basis. If you’re doing this currently, you’re probably missing real design bugs. I am a big fan of mutation coverage!

**RULE #10: REVIEW, REVIEW, AND THEN REVIEW SOME MORE**

A thorough design and verification review cannot be ruled out from any best practice strategy. This is certainly true of formal as well. However, the review needs to be:

- Consistent.
- Applied regularly.
- Undertaken against a set of guidelines laid out by the DV team and based on verification’s foundational principles.
Signed-off by an independent engineer not related to the nitty-gritty planning and execution of the verification in any way, so no common mode problems can arise.

I hope you find these rules useful in your daily practice of formal verification. Do let me know whether you agree or disagree as to whether you find them useful.

**About the author**

Dr. Ashish Darbari is the founder and CEO of Axiomise® – a formal verification training, consulting, and services company founded in London in 2018. Ashish obtained his DPhil from the University of Oxford and has been a user of formal methods for over 20 years. Before starting Axiomise®, Ashish worked at Intel®, ARM®, Imagination Technologies®, and OneSpin® Solutions. At Imagination, he founded and led the AVM group where he trained nearly one hundred engineers in formal verification. He led the AVM team to deploy formal on over 50 projects in a little over three years. Ashish completed a three-year Royal Academy of Engineering Visiting Professorship at the University of Southampton, England in 2018. With 18 US, UK and EU patents and over two dozen papers in conferences, Ashish is a Senior member of IEEE and ACM; and a Fellow of the British Computing Society, and IETE.
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

33 Video Courses Available Covering
• DO-254
• UVM Framework
• UVM Debug
• Portable Stimulus Basics
• SystemVerilog OOP
• Formal Verification
• Metrics in SoC Verification
• Verification Planning
• Introductory, Basic, and Advanced UVM
• Assertion-Based Verification
• FPGA Verification
• Testbench Acceleration
• Power Aware Verification
• Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 10,000 questions asked
Verification Patterns Library

www.verificationacademy.com