The Quest for Bugs: "Dilemmas of Hardware Verification"

(Navigating the Challenges)

12 minute read

Bryan Dickman, Valytic Consulting Ltd.,
Joe Convey, Acuerdo Ltd.

*Functional Verification for complex ASICs or IP-Core products is a resource limited '**quest**' to find as many bugs as possible before tape-out or release. It can be a long, difficult and costly search that is constrained by cost, time and quality. The search space is practically infinite, and 100% exhaustive verification is an unrealistic and non-tractable problem. The goal is to deliver the highest possible quality (measured by both product performance and absence of bugs), achieving that in the shortest possible time (in order to maximise revenue), with the lowest possible costs.*

Complexity continuously increases, and the functional verification challenge gets progressively harder. The search gets longer, and the bugs become increasingly more difficult to find. In practice, some bugs will be missed because verification is inherently imperfect and non-exhaustive. How do you find most of the bugs? How do you find all of the critical bugs? In The Origin of Bugs we asserted that:

Verification is a resource limited 'quest' to find as many bugs as possible before shipping.

Terms of Reference

So, what makes verification more challenging than other aspects of engineering when developing complex semiconductor products? Take other design workflows of semiconductor development such as RTL synthesis, timing analysis, place and route, power analysis and sign-off checks. As a rule, these workflows tend to be reasonably deterministic. They may consume significant engineering resources (skilled people and compute), but there is usually a well-defined process to converge or iterate towards a result. When you get there, there is a measurable degree of completeness in the results. We're generalizing of course. There are challenges and uncertainties with these workflows, but with verification the $64,000 question is always "Am I done yet?". We don't believe we can answer the verification sign-off question with the same degree of certainty as the design implementation sign-off. We also know that the

Valytic Consulting

ACUERDO

consequences of missed critical bugs, can be potentially catastrophic. See The Cost of Bugs for a more detailed discussion on the cost of "not finding bugs".

We characterize this verification uncertainty as the following set of dilemmas that need to be carefully navigated; ***completeness***, ***complexity***, ***constrained-random***, ***resources*** and ***delivery***.

The Completeness Dilemma

Verification is not an exact science. There is no way to achieve completeness or perfection. You have to decide when you have done enough work that you are satisfied that the risk of critical bugs remaining in the design is acceptably and manageably low.

Verification is an exercise in risk mitigation.

The problem comes down to the impossibly large state space in almost all modern ASICs. It's a non-tractable problem. You might also think of this as the

"impossibly large state-space dilemma!"

This is also related to the complexity-dilemma which we will discuss in the next section.

Some sense of completeness can be achieved when verification targets have been met. Verification becomes a data-driven quest; it's all about metrics and analysis, with verification sign-off driven by the assessment of metrics such as coverage, test-pass rates, cycles since last bug, cycles since last RTL change, formal properties proven, bug rates, etc. All of these sign-off criteria should be captured in the "test plan" and reviewed, revised and approved. You're looking for a convincing absence of bugs despite measurable efforts to find further bugs in an RTL code base that has been stable for an acceptable period of time. You can't be too complacent. The bugs are still in there. Your efforts so far have failed to find them!

All of your verification efforts will suffer from the completeness dilemma.

Test Planning

Take ***test planning***, which is the normal starting point. The process to develop the test plan is one of design and specification analysis, brainstorming, reviewing, and iterative refinement. It's probably the most important step; sometimes undertaken at the beginning of the project, sometimes evolving over time in a more agile fashion. Either way, it is impossible to develop complete

test plans. Late found bugs can usually be traced back to a shortfall in the test plan, maybe a missing corner case, or even a gap where a whole class of behaviors have been omitted from the test plan.

Coverage

Take ***coverage***, which is commonly relied upon as a good measure of the completeness of verification stimulus. ***Code coverage*** does give some sense of absolute measure in that every line of RTL code and every branch and expression has been visited during testing. However, it does not inform us that functionality is correct, or about missing lines of code i.e., missing functionality. We know that code coverage is just one measure. Commonly teams will be using ***functional coverage*** to measure stimulus in terms of its ability to exercise functional behaviors observed as events or sequences or more complex combinations and crosses of coverage. These functional coverage points collectively form the ***"coverage model"***, but how do you know that the coverage model is complete? Like test planning, it is another "best-effort" exercise that relies on a process of analysis and review, refinement and iteration. You may still omit an important functional coverage point which (had it been present) would have exposed a critical gap in your stimulus that was hiding a critical bug. Best to always assume,

if it's not tested, it's broken!

Formal Verification

The same applies to most other aspects of verification. Even ***formal verification***, where you might think there is some promise of completeness (by exhaustively proving properties) is an incomplete science. Many properties cannot be exhaustively proven, only shown to not fail for a given depth of cycles (bounded proofs). You have the same issue with your formal properties as you do with functional coverage and stimulus – how do you know for certain that you have planned and implemented a complete set of properties?

System-Level

Ditto for ***system-level verification/system validation***. You will not know that your testing payloads are complete and that they will be able to find all of the remaining bugs that were missed by previous levels of verification.

Sadly, some bugs will be missed.

Those hidden bugs might never present in the field over the lifetime of the product, meaning that the product is fit for purpose and therefore meets its requirements. However, you can't be certain of what code from the software ecosystem will eventually run on your system. Software development tools may change over time and may produce different code sequences that were not present in the testing payloads at the time of verification sign-off. All of this runs the risk that eventually a previously unseen dark corner-case will be encountered with unpredictable consequences, potentially occurring for the first time in the field. If you are lucky, there may be a viable and deployable software mitigation that does not significantly degrade performance or function, in which case, you got away with it! If you are unlucky and no such software mitigation is possible, you may be looking at a hardware mitigation or a costly product update. Again, see The Cost of Bugs for a more in-depth discussion about cost impacts of hardware bugs.

So, this lack of completeness is a dilemma for the developer of complex ASICs or IP-Cores. It is something to be constantly aware of and something to be considered at length when reasoning about the age-old question,

"am I done yet?"

The Complexity Dilemma

The key causes of the completeness dilemma are the "impossibly large state-space" dilemma and the complexity dilemma. ASIC or IP-Core products only ever get more complex, they seldom deliver more capabilities by being simpler.

We're talking about complex hardware components or sub-systems such as CPUs, GPUs, ML/AI processors, integrated into multi-core SoCs and ASICs that may be multi-billion gate devices. For verification, it's a good result if tools, methodologies and platform performances and capacities can at least keep up with complexity growth, let alone getting ahead of it. That's the dilemma. Engineering teams need to understand complexity and curtail it wherever they can, but at the same time complexity is necessary to achieve performance, capabilities and ultimately, competitive advantage. Complexity that is introduced to achieve performance and functional goals, is often harder to contain, and design teams are always innovating new architectures and microarchitectures that will set their product apart from the competition in terms of performance and functional capabilities.

Furthermore, complexity is not something that you may have set out to achieve. A once clean and elegant design can degrade over time into something containing a lot of technical debt, as the code is iterated and reworked, optimized and re-factored, potentially by multiple individuals over the lifetime of the development.

When the code author admits that they really no longer have a complete understanding, it's time to panic!

Think about strategies to contain design complexity wherever possible. Reduce it by refactoring code, purging redundant code, and maintaining code readability and maintainability. Investigate suitable metrics to measure complexity if possible.

The Constrained-Random Dilemma

Over recent decades, constrained-random verification methodologies have become the norm. Given our understanding of the completeness dilemma (which means we acknowledge that it is impossible to identify all possible testing scenarios) random testing offers a way to find scenarios we had not specified thanks to "random chance". The probabilities of hitting these unknown scenarios are increased by volume of testing.

If we run enough cycles, we will eventually hit it…probably, we hope!

But this philosophy has hugely driven up verification platform costs, and oftentimes we don't really have a good understanding of how effective all those random cycles are at finding these unknown-unknowns. Of course, there is more to it than that. This is not a fully random strategy, it is 'constrained-random'. We identify constraints for the stimulus generators to guide stimulus into specific areas of interest. We then use 'coverage' methods to measure the effect of the stimulus, and stimulus-generators and coverage models are refined over time. However, this strategy eventually leads to a saturation point, where we are no longer finding new bugs and are now running 'soak' cycles to build confidence and assurances that a "respectable" (you have to decide what that is) volume of bug-free testing has been achieved. Determining what this safe assurance level is can be difficult and you may be required to justify exceedingly large engineering costs based on your judged targets and analytics.

How do you know if you have sufficient constraints? Over-constraining means you might be missing some key areas of stimulus where bugs could be hiding. You might not realize that because your coverage model is not complete either! Some bugs may require pathological sequences that are just too improbable for the generator to produce. If you can identify these cases, you might be able to program them into your generator – but that requires you to realize that these cases exist. Shared learning from experience and historical projects can really help here.

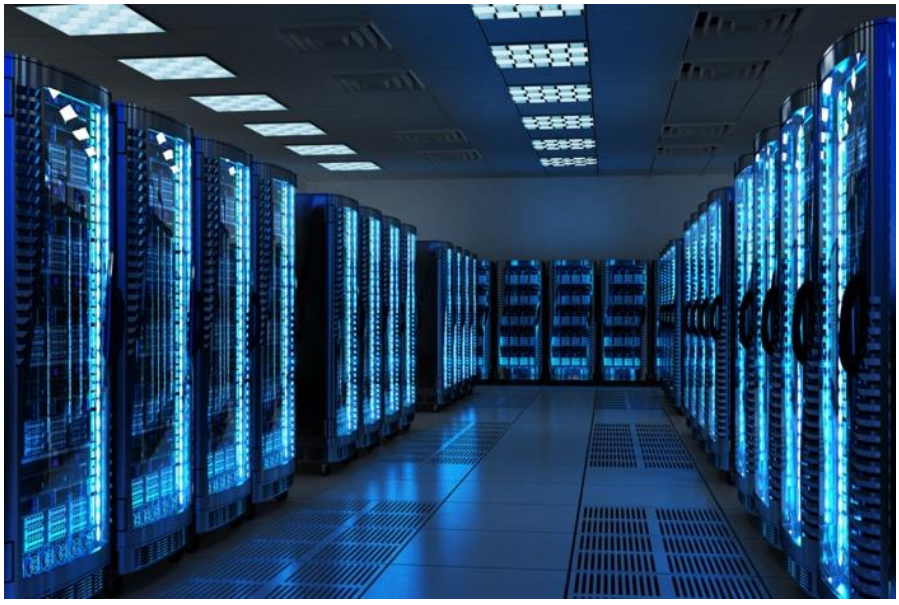
Constrained-random suffers from the completeness dilemma and the resources dilemma.

The Resources Dilemma

How do you deliver the product on-time and to the right quality level?
You had better make sure that you are using the available resources in the most efficient and effective way possible.

At the risk of repeating ourselves, but let's anyway…

Verification is a resource-limited quest to find as many bugs as possible before release.



Resources are always finite and limited, regardless of whether your resources are on-prem or in the cloud. On-prem implies investment costs to establish infrastructure and development tool capacity, and then ongoing operational costs to operate and maintain the platforms. Cloud implies that there are some cost constraints or budget parameters that you have to operate within. If your capacity demand changes, there will be additional costs of acquisition and commissioning, but there is likely to be an availability lag as it inevitably takes time to expand on-prem capacities. If you are already using cloud to provision your infrastructure, the availability lag may not be there, but the incremental usage costs will be.

Let's not forget the human resources side of this equation. Team sizes can flex, but people costs are the biggest resource cost in general, and you have to deliver your product within the constraints of the available team. Make sure that your teams have the right skills profile, the best tools, and are well-motivated and engaged, because staff turnover can be one of the most disruptive things to occur mid-project.

Engineering teams have to use the resources that are available to them, in the most effective and efficient way, to achieve the best verification results possible within these constraints.

Sometimes these constraints help to drive engineering innovation to improve the efficiency and effectiveness of verification workflows. How do we achieve better Quality of Results (QoR), from the same or less resources, in the same or less time, thus reducing the cost of verification and increasing product ROI? In our experience, engineers love to innovate, so direct them towards these challenging problems. After all,

scarcity of resources, drives innovation.

Optimization is a never-ending quest that requires you to measure everything. Optimize your workflows; optimize your tools and select the ones with the best performance; profile and optimize your testbenches and verification code to make them run as efficiently as possible.

Securing resources for your project is oftentimes a process of negotiation. Good resource forecasting is essential to ensure you have planned ahead for the resource demand, but these forecasts need to be reviewed and refined throughout the project. If you are competing for shared resources, human behavior can lead to negotiation tactics, e.g., figure out what you think you need and add a buffer or sandbag your estimates by an amount that you think you will be negotiated back down by! Forecasting really needs to be a transparent and data-driven process where predictions are accurate and based on best-practice analytics.

Conclusion - the Delivery Dilemma

Finally, you have to deliver your product on time and on cost. Endlessly polishing the product beyond the point where the documented goals and sign-off criteria are met will erode the product ROI and left un-checked can destroy it. Remember…

perfection is the enemy of success.

The delivery dilemma can lead to some tough calls on occasions. It's a matter of risk management. This is where you have to be very clear about what your sign-off criteria are and how you are measuring up against them. You started with good intentions and a comprehensive test plan, but now you need to assess status. Look at all the data and make a judgement call on the remaining risk. You can think of this signoff in terms of ***"must do", "should do", "could do"***, and

"things that will help you to sleep better at night!"

By the time you get here you have probably achieved all of the first three items and are making a judgment call on the fourth. Consider the following:

- Delaying the final release will block resources; people and infrastructure, that are needed to execute on the delivery of other revenue bearing projects and the overall business roadmaps.
- Delaying the final release will increase the product cost and erode ROI.
- Delaying the final release will have a downstream impact on the customer's schedules, which in turn impacts their ROI (and potentially your future opportunities).

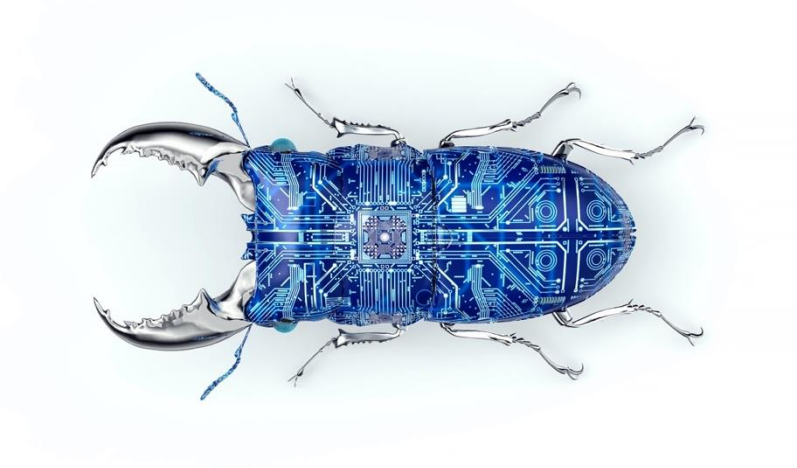
Get this wrong however, and you might incur substantial rework costs, opportunity costs, and reputational costs, as a consequence of an impactful bug!

Having made the release, there is still a window of opportunity where you could continue to make marginal improvements to the verification so that any new bugs can be intercepted and mitigated before the product is widely deployed into the field. As verification engineers, we know that some level of extended background verification can be a good investment of engineering resources, especially if we are still in a pre-silicon situation.

The real challenge is in deciding when to stop!

Although this paper does not prescribe solutions to these dilemmas, having an understanding of them can help in navigating good verification choices.

Happy sailing!



Valytic Consulting

ACUERDO