# Let's be Formal While Talking About Verification Quality: A Novel Approach to Qualify Assertion Based VIPs

Sachin Scaria
sachin.scaria@intel.com
Intel Technology India Pvt. Ltd, Bangalore
India

Surinder Sood
surinder.sood@intel.com
Intel Technology India Pvt. Ltd, Bangalore,
India

Erik Seligman
erik.seligman@intel.com
Intel Corporation, Portland
United States

***Abstract*-When developing Verification IP for a new protocol, we need to make sure it is well-qualified before release, since it is typically targeted to be used on many future products. We must pay particular attention to *consistency*, making sure all properties are mutually compatible and follow the protocol, and *completeness*, checking that protocol errors in connected models will be caught. We check consistency by building models using our Assertion based VIP with a formal verification tool in standalone mode, which will flag many types of issues. For completeness, we combine fault injection with the exhaustiveness of formal verification, currently an underexplored area in verification.**

## I. INTRODUCTION

*"Working hard and working smart sometimes can be two different things - attributed to Byron Dorgan"*

*"Did I actually finish my verification?"* has always puzzled verification engineers in VLSI industry, and continues to do so even in the current generation. Formal tools have evolved as an interesting and promising solution to intelligently address these concerns.

In this paper, we discuss the issues in qualifying Assertion-Based Verification IPs (AB VIPs), sets of pre-defined assertions that describe expected behavior for a protocol. Ideally, we would like to have high confidence that these will perform as needed for future validation teams, correctly checking compliance and flagging potential errors. If written carefully using synthesizable constructs, such IPs can be useful in both simulation and formal environments.

However, a common mistake is to 'informally' qualify such VIPs through plugging them into a project environment with lots of random simulations, and concluding that they are good if the simulations pass and flag only real errors. But if we are targeting our VIP to support formal verification as well, we have an opportunity to be much more rigorous. Even when we have seen all the properties proven with the formal tools on a targeted RTL model, we still have three big questions.

1) **Consistency:** Did I over constrain my design?
2) **Completeness:** Can my set of properties really be considered exhaustive, or are there cases where it may miss RTL bugs?
3) **Correctness:** Do our properties actually describe the intended protocol?

Even when we state that formal verification is exhaustive, these questions are still valid. They create uncertainty even for a "100% proven" result, since such a result always depends on exactly what has been proven. An approach to automatic some portion of the Correctness check is described in [Sel11], though this is inherently difficult to

automate. However, formal verification tools can play a critical role in effectively addressing and automating solutions for Consistency and Completeness.

## II. Checking Consistency of VIPs

The assertion based IP consists of a set of properties and some modelling logic to model various scenarios required for the properties. For the project we focus on in this paper, the properties are based on INITIATOR and TARGET. These properties can be categorized as:

a) Assumptions : This set define the constraints for the IP
b) Assertions : These properties check the design behavior.
c) Constraints : These properties are the stimulus filters for legitimacy as the tool generates all the possible scenarios.
d) Cover : These properties are scenario qualifiers. These also aid in checking the legitimacy of the assertions and assumptions.

The assertion based IP models all these type of properties. Once the assertion based IP is ready, two instances are connected back to back, where in one instance serves as INITIATOR while the other serves as the TARGET. The role of properties is defined as follows:

a) The INITIATOR instance properties are categorized as follows:
   i) The INITIATOR side properties serves as constraints to the formal tool
   ii) The TARGET side properties will serve as assertions to the tool. The TARGET properties will validate the slave response.

b) The TARGET instance properties are categorized as follows:
   i) The TARGET side properties serves as constraints to the tool.
   ii) The INITIATOR side properties will serve as assertions to the tool. The INITIATOR properties will validate the INITIATOR response.

Depending on the targeting of the VIP, it may be set up so that the same properties are assertions or assumptions, based on a parameter describing which 'side' is being checked, as described in [Sel15].

Once the model containing Initiator and Target instances is constructed, we use this as a basis to build a formal verification environment. To validate the legality of the properties, the cover properties are written. These properties also confirm that we cover expected behaviors of the design and in that sense they also serve a tool for measuring the functional coverage in the formal verification paradigm. Cover checks included in a particular model may include explicitly written covers for various modes and conditions; cover points that reflect example waveforms in protocol documentation; or automatic cover checks generated based on trigger properties.

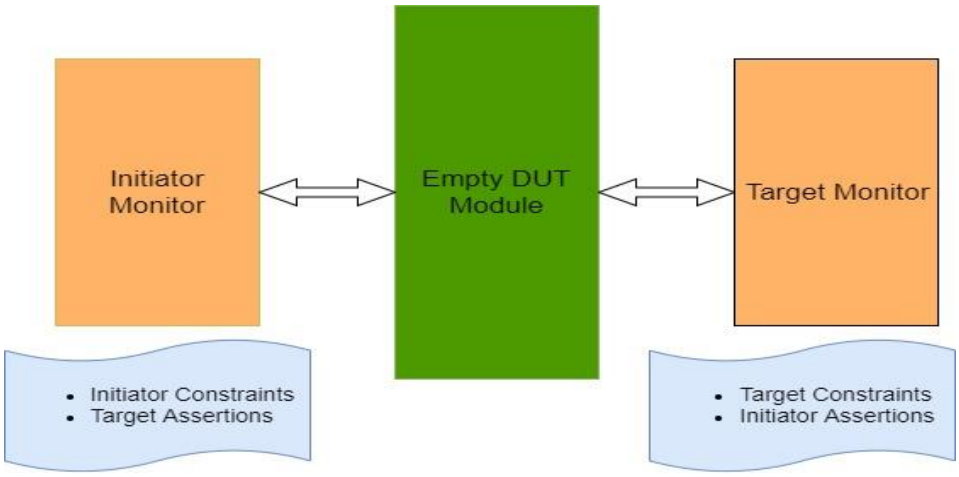


Figure 1. The Dummy DUT ABVIP

An alternative method (described in [Sel15] is to just instantiate a single model, labelling all Initiator and Target properties as assumptions. This gives similar results, though it omits the checking of the connections between VIP signals and actual RTL interface signals.

Overall, this is an excellent approach for checking consistency. If a cover point is missed, that usually indicates that our properties overconstrain the design, conflicting with each other and disallowing legal behaviors. In some cases these conflicts indicate inherent flaws in the protocol or its documentation. However, this does not really address the issue of completeness. In addition, this type of technique is harder to implement for VIPs not based on protocols or bridges.

## III. Checking Completeness through Fault Injection

When formal verification methodologies are maturing in organizations, it is very important to ensure completeness: does our property set cover, in some sense, all violations of the protocol? We know the classic quote on functional verification "who debugs a passing test case?" This is no different in formal as well, because a fully proven suite doesn't mean no bugs. While code review and the legacy quality checks are already in place, any automated way to take this quality checking efforts to an exhaustive environment outshines other approaches-- not as a replacement, but to amplify the efforts.

Today most verification sign-off checks are mandating that standard interfaces like AHB, AXI (list is unending with proprietary ones) etc. to be protocol compliant by running formal tools with ABVIPs. Here the users are expected to believe the completeness of the ABVIPs to be 100% and golden. Thus, they are silently ignoring the fact that ABVIPs are also bound to mistakes all the way from human errors to functional bugs in properties.

This method proposes combining fault injection with formal verification. Combining the power of formal with fault injection opens up a vide horizon for verification engineers, with many new possibilities. How to capitalize them for the best result lies in bringing out a structured analysis method. The unique combination of fault injection augmented with formal verification methodology lets us evaluate the quality of the properties on fault injected designs. In other words, if there was actually a fault in the design, will any property catch it? That gives a lot more confidence on the verification quality than merely running a formal tool and observing that the assertions pass..

This proposed approach needs to be applied to designs and ABVIPs which are stable and nearing production quality. Even though this method could unveil RTL bugs (it's a bonus), identifying them would be costly on fault injected RTL, when compared with non-fault injected formal approach. Also, the focus here is not to catch RTL bugs, but to qualify ABVIPs.

The flow diagram below explains how this approach can be put to life in a systematic manner to achieve desired results. Once the initial formal proofs are done as a sanity check with ABVIP and original design code, the RTL will be fed into the tool for fault injection. Once this stage is done, the fault injected RTL is being used by the formal tool to prove the properties by iterating over the faults (single or a combination of faults, but not all at ones). Here, at least one assertion in the suite should be failing to ensure all required properties are in place for that particular fault. Otherwise, it's time to add a new property to the suite. Also there are classifications available for the type of faults injected, such as the ones on reset and clock paths, outputs, deep inside stuck at faults etc. This helps in focusing and prioritizing on the type of assertions to be enhanced in a systematic manner, than looking all of them at once (e.g. reset checks).

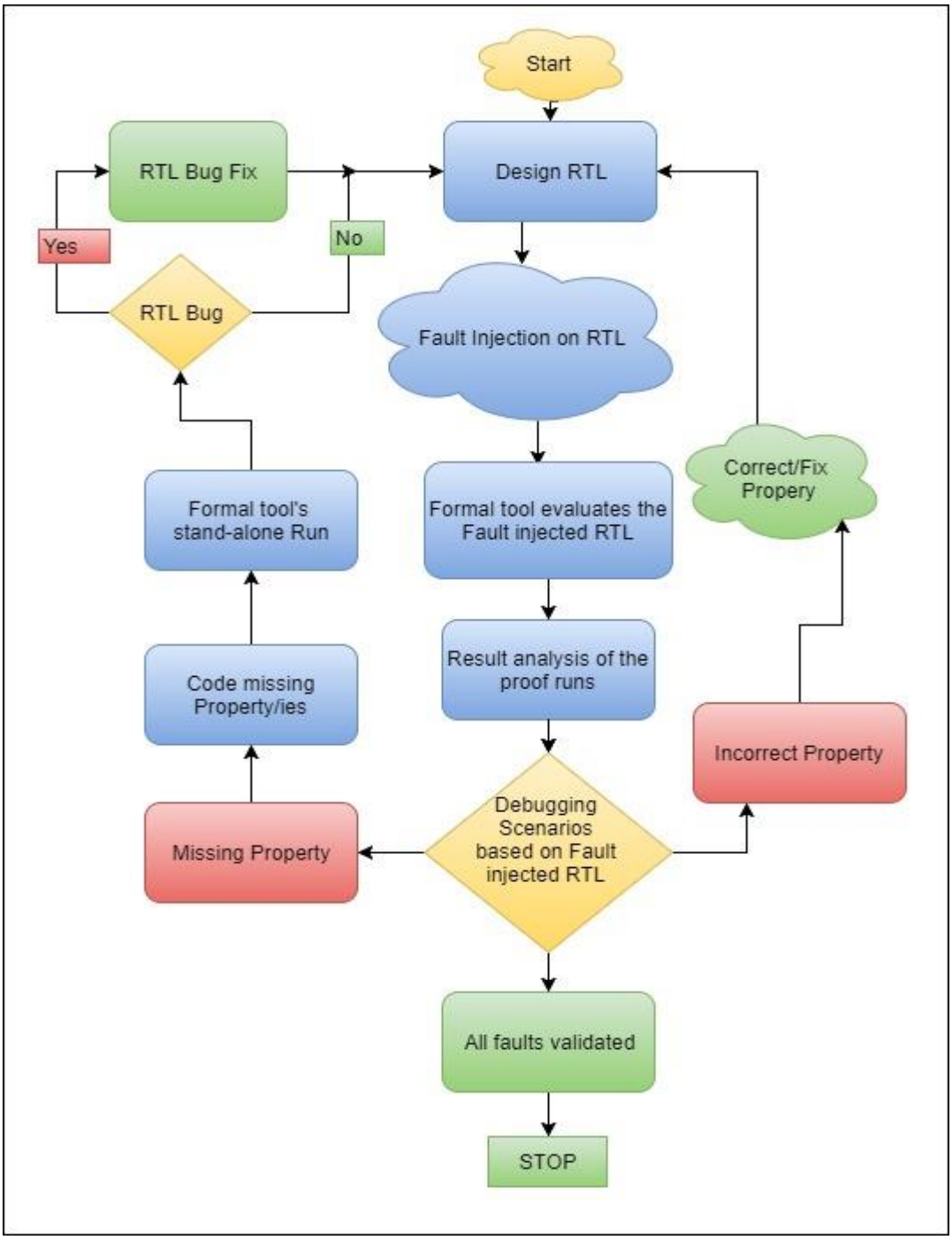


Figure 2. The Functional Qualification Framework

## III. RESULTS AND SUMMARY

Our results have shown that attention to Consistency and Completeness for ABVIPs is very important, and that these techniques can reduce risks of subtle bugs escaping to late project stages.

The Consistency checks tend to only find a handful of issues, but the ones it does find are important. On a recent project, we found two cases where the specification documents contained waveform diagrams (which we had converted to cover points) that were actually not possible under the rules of the protocol. These could have caused confusion and delays during later design stages if not fixed.

The fault-injection Completeness approach gives a very powerful framework for qualifying ABVIPs. Quickly we can find out what properties are faulty or missing in our verification framework. This allows us to mature our verification environment for future projects with very high confidence. Of course, this depends on a high-quality fault injection tool to enable this qualification. In the design where we pioneered this method, we had the following results:

Total Faults injected: 1515

Total Faults Non-Activated: 136

Total Faults Non-detected: 4

The above results show that we had a large number of non-activated faults because we had not written corresponding properties which were to be mapped to those faults. Typically these properties were related to many reset, interrupt and interconnect conditions in our design which we missed to write.

Finally there are 4 faults which were not detected by any of our properties and these were reset properties, which we later on re-modelled.

We have validated this concept at IP level successfully. With little effort, we observed 4 missing reset condition checks in the ABVIP that we used in a bridge design. As time progressed, we could find the ABVIP was lacking couple of functional properties as well. Definitely, we need to do some iterations in maturing our verification environment, and this fault-based functional qualification has helped us a lot in this regard.

### *A. Challenges faced:*

The most significant challenges with both our Consistency and Completeness approaches are in ensuring that our checks are truly sufficient. We have a type of infinite regression here: we introduced additional flows and tools to gain confidence in our ABVIP, but we have no formal proof that our Consistency covers are sufficient, or that our Completeness process injected sufficient faults. While the coverage functionality introduced in recent formal verification tools can add to our confidence, an overall "proof" that we have tuned these methods sufficiently still requires further innovation.

Our fault injection approach also suffers from the limitation that at least one mature RTL model, which implements the checked protocol, is needed. While our Consistency checks can be run on the ABVIP standalone, the fault injection does not make any sense unless there is an actual design under test. Thus, this method cannot be used to pre-qualify an ABVIP, but must be used to enhance confidence in the late stages of the first project that uses it.

Aside from this, there are some practical challenges for current tools. With fault injection, the biggest challenge that came up during the experiment was the combinational loop existence for a large number of injected faults. Time had to be spent in analyzing and rectifying them.

We also need constant vigilance to ensure that our ABVIPs are implemented in synthesizable verilog, and kept compatible with both simulation and formal verification. We have had numerous cases where simulation-focused teams did not understand these issues, and tried to extend our checking with constructs that are not compilable by formal tools.

Finally, it's also the case that despite our degree of automation, microarchitecture-level design understanding is required to effectively carry out this activity. As is common with formal verification, the scenarios generated tend to poke at very obscure corner-cases behaviors, which might not be easy to understand.

## REFERENCES

[Tho07] Manoj Thottasseri, Gaurav Gupta, and Mandar Munishwar, "Developing Assertion IP for Formal Verification," Design and Reuse, 2007, http://www.design-reuse.com/articles/20327/assertion-ip-formal-verification.html.

[Sel11] E. Seligman, R. Adler, S. Krstic, J. Yang, "CompMon: Ensuring Rigorous Protocol Specification and IP Compliance," Design and Verification Conference (DVCon) 2011, February 2011.

[Sel15] E. Seligman, T. Schubert, M.V.Achutha Kirankumar, "Formal Verification: An Essential Toolkit for Modern VLSI Design,", Chapter 7, Morgan Kaufmann, 2015