Detoxify Your Schedule With A Low-Fat UVM Environment

Nihar Shah
Hardware Advanced Development
Oracle Labs
Austin, Texas. U. S. of A.
nihar.shah@oracle.com

***Abstract*-This paper proposes the use of self-checking stimulus using UVM sequences as an alternative checking methodology to traditional UVM scoreboarding. The methodology proposed can significantly reduce the development time spent on building and maintaining verification infrastructure. The paper discusses the benefits and risks of self-checking stimulus and how to address the risks, especially non-portability of the results checkers from unit to full-chip.**

I. INTRODUCTION

It has been widely accepted that a test environment must be portable from unit-level to higher levels of integration [1]. Reasons cited for this range from higher verification quality [2] to increased verification efficiency [3]. However, the development cost in making things portable is often taken for granted, and the decision to port is often done without an honest cost/benefit analysis. Porting a unit environment to a higher level is not always the most practical way to find new bugs, increase coverage, or otherwise improve the overall quality of verification [4]. Yet we continue to require unit verification components to be portable as if this came for free.

This paper will examine the constraints that portability imposes on a typical UVM scoreboard-based environment [1], and how freeing ourselves from those constraints can allow for a light-weight self-checking environment. The paper will discuss:

1. The hidden costs of portability in a typical UVM scoreboard based environment (the “pay me now or pay me later” problem).
2. How to create a light weight environment using UVM sequences and UVM agents (the “self-checking stimulus”).
3. Why the proposed self-checking stimulus can save significant time in the project schedule, by eliminating specific code complexity.
4. The inherent disadvantages and risks of self-checking stimulus, especially non-portability.
5. Assessing the real benefits of portability, and why non-portability may be acceptable.
6. How to mitigate the risks introduced by self-checking stimulus.
7. What types of DUTs are a good fit for this methodology.
8. Practical comparison of results based on real world experience

The goal of this paper is to convince the reader to carefully consider both the true costs and real benefits of portability when deciding on a verification strategy. If the costs are high and the benefits are not real, then a lighter UVM environment is proposed.

Finally, it is important to note that in many cases having a portable environment and/or scoreboard is in fact desirable regardless of the cost. The author does not advocate the proposed self-checking stimulus in such cases. However, the author does advocate that we must not blindly accept that a portable environment is required for every design since many designs will not benefit from one.

II. PAY ME NOW OR PAY ME LATER: THE TRUE COST OF PORTABILITY

The biggest challenge in making an environment portable is making the scoreboard portable. For a scoreboard to be portable, it must:

1. Work independently of stimulus generators such as UVM sequences [2].
2. Only use information given by other portable components such as UVM agents [2].

At the root of the problem is that a portable scoreboard must deal with low-level transactions without directly knowing the high-level operation. This can require significant intelligence in the scoreboard, which not only leads to more development time to construct the scoreboard, but significantly more maintenance time to keep the scoreboard operating for all possible corner cases. Consider the typical UVM environment [1] example shown in Fig. 1.

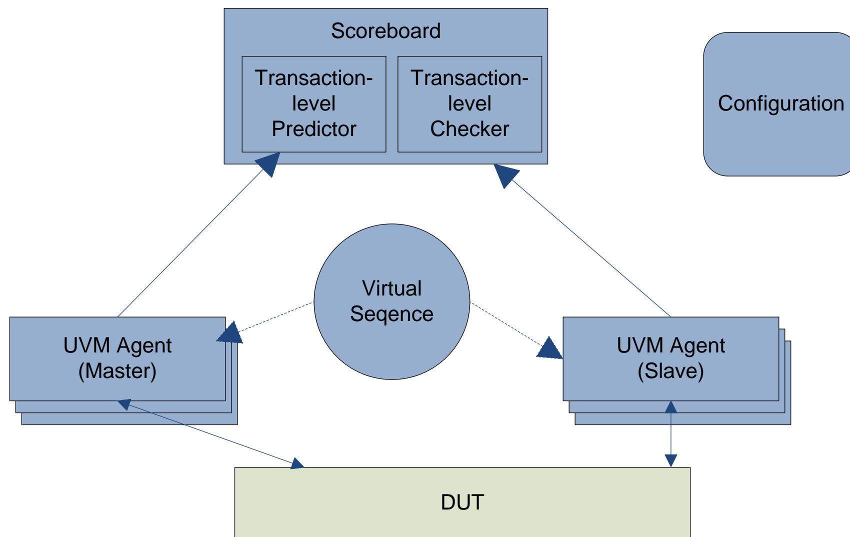

Figure 1. Typical UVM scoreboard environment

In this case, there are multiple channels of data traffic going to and coming from the DUT, which may include register configuration to the DUT. The virtual sequence orchestrates the stimulus generation given allowable configurations by:

1. Breaking up the desired high-level operations into low-level transactions that can be consumed by the UVM agents, and
2. Coordinating the UVM Agents to drive those transactions in the order desired.

The portable scoreboard receives these snooped low-level transactions directly from the agents, and must make sense out of them by inferring the stimulus configuration and rebuilding the high-level operation. The rebuilt high-level operation is used to predict the expected low-level transactions received from the UVM slave agents. Checking is done by correlating the observed low-level results transactions with all possible correct answers derived from the transaction-level predictor. The flow is summarized in Fig. 2.

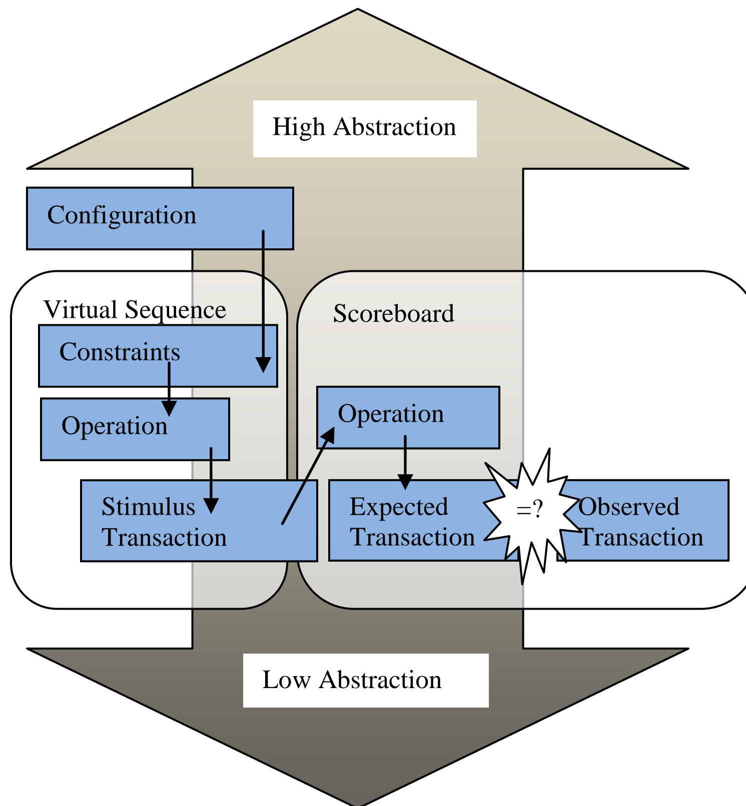

Figure 2. Stimulus lifecycle in a UVM scoreboard environment.

The downward arrows show a conversion from higher levels of abstraction to lower levels of abstraction. The upward arrows show a conversion from lower levels of abstraction to higher levels of abstraction.

A. *The first upfront cost of portability: Redundant engineering*

To identify the real cost in developing a portable scoreboard, let us consider where the verification engineer is most likely to spend valuable time. In Fig. 2, the flow traverses up and down the levels of abstraction before arriving at a checkpoint. This cost is incurred primarily in the scoreboard.

To ultimately predict expected results, the scoreboard must convert observed low-level stimulus transactions from UVM monitors to high-level operations supported by the DUT. The effort to rebuild the high-level operation by the scoreboard is redundant since the virtual sequence already knows the high-level operation by having generated and sent it to the DUT. However, the redundant effort is necessary to decouple the scoreboard from the unit-level virtual sequences and therefore keep the scoreboard portable. A portable scoreboard cannot cheat by getting information about the high-level operations from the virtual sequence.

B. *The second upfront cost of portability: Inflexibility of low-level transaction checking*

Other than this redundant effort, the portable scoreboard also introduces code complexity and maintenance requirements due to the following:

1. The portable scoreboard must account for more than one way to get the right answer. As a result, the verification engineer must take care that the portable scoreboard supports any of the following during the conversion of the high-level operation to the low-level expected transactions:
   a. Allow low-level observed transactions to be received out or order from the same agent, to the extent the design allows.
   b. Allow a variable ordering of low-level observed transactions received across all slave agents, to the extent the design allows.
   c. Allow for multiple correct possibilities for low-level transactions, to the extent the design allows.

   Furthermore, the rules that determine the specific allowable transactions can frequently be a function of design implementation rather than feature correctness. As the design evolves through the course of the project, any assumptions made to generate expected transaction sequences may change without affecting the

overall correctness of the feature being tested, and the scoreboard must keep up with these changes to avoid immediate breakage.

2. Random stimulus may continue to cause the DUT to issue transactions in new ways to arrive at the correct end result, which can break the transaction-level scoreboard checking, resulting in more false failures to debug. Depending on the design, it can be difficult to take into account all such cases. As such, it is difficult to predict the time spent on this. This problem is exacerbated when the unit environment is ported up to cluster or full-chip. Furthermore, this usually occurs late during the verification schedule, at the worst possible time.

For a scoreboard to be portable, it must get its transactions from UVM agents, which only understand low-level transactions associated with a given interface. The self-checking stimulus proposed in the next section of this paper mostly does away with understanding of low-level transactions as part of checking and hence cuts out the significant schedule impact involved in supporting the conversion back and forth between high-level operations and low-level transactions, and the unpredictable issues that go along with that.

III. Introducing The Self-Checking Stimulus

The self-checking stimulus is based on the following premises:

1. **Non-portability of the scoreboard is an acceptable trade-off**: It is not necessary to reuse the stimulus checkers outside of the unit environment. However, the UVM agents and their underlying bus protocol checks stay portable to help with debug outside the unit.
2. **Check operations, not transactions**: It is more useful to check the net result of sequences of transactions that perform a higher order function rather than check each individual low-level transaction.
3. **Stimulus centric implementation:** Inheritance is utilized to create a scalable library of UVM stimulus sequences, including a built-in checking hierarchy.

An environment using self-checking stimulus is shown below in Fig. 3. The infrastructure includes:

1. **UVM agents** - these are portable to the system level as passive monitors. The monitors will include bus protocol checkers and provide visibility to individual transactions, making them useful for debugging tests at both unit and system level.
2. **Library of operation sequences** – these are UVM virtual sequences used to generate the transactions that can be run as stimulus on the DUT. The operation sequences generate the low-level transactions needed to complete the operation, and check the results of the operation. More complex operations are built by extending simpler operations, and/or by chaining other operations together.

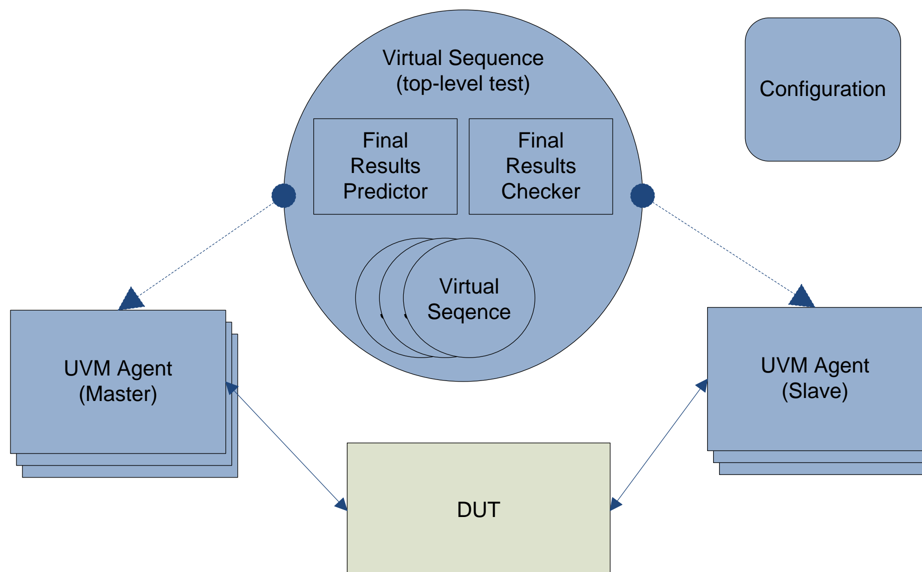

Figure 3. Self-checking stimulus environment

The following sub-sections explain the premises further by addressing the possible objections the reader may have with the proposed methodology, and will also discuss the following:

1. Assessing the real benefits of portability, and why non-portability may be acceptable. Refer to sub-section A.
2. Why self-checking stimulus can save significant time in the schedule, by eliminating specific code complexity. Refer to sub-section B.
3. The inherent disadvantages and risks of self-checking stimulus. Refer to the "objections" in sub-section B.
4. How to mitigate the disadvantages and risks introduced by self-checking stimulus. Refer to the "objections" in sub-section B.
5. How to create an environment using the self-checking stimulus. Refer to sub-section C.

A. Non-portability is an acceptable trade-off

It is widely expected that a unit environment should be portable to a cluster level or full chip. However, the practicality and usefulness of this requirement are often times not fully considered.

Objection A1: I need to reuse the checkers from my unit environment to my full-chip environment to maintain a high quality of checking at full-chip.

A good unit-level verification strategy should thoroughly verify the DUT and deliver that DUT to a system-level testbench as a verified IP. System-level environments should then focus on connectivity and integration of the parts and on stressing the system as a whole, rather than stressing out any particular unit that was already verified in a unit-level environment. Therefore, the same detailed checking required at the unit-level environment is not needed at the system level. Low-level transaction checking for every unit in a system-level environment is overkill and can be replaced with self-checking tests that do not rely on unit-level infrastructure to check end results.

The key to accepting non-portability of checkers is to view the unit-level DUT as a verified black-box in the system, just like a third-party provided design IP. Rarely does one integrate into their system any scoreboard that checks every low-level transaction from a third-party verified IP. Usually the IP is verified using pre-packaged tests that at least run through all the interfaces and check the end results. Why should we expect something different from what is ultimately an 'internal' IP?

Another concern in system-level quality is related to assumptions made on interfaces of the unit-level DUT. If incorrect assumptions are made by the bus functional models at the unit level, the system integration will break. However, the entire scoreboard and transaction-level checking is not required at the system level to mitigate that risk. The self-checking stimulus environment does make use of UVM agents, which can port up to the system level to act as bus protocol checkers. Any assumptions about the bus protocol should be coded into the UVM agent as assertions or checks in the monitor. These agents used in conjunction with self-checking tests at the system level will make up the checking infrastructure required to meet the unit-integration goals of system-level verification.

Objection A2: I will spend a lot of extra time writing self-checking tests at the system level to compensate for the loss of a scoreboard.

The task of writing self-checking tests in a pre-silicon full-chip environment will overlap with the required task of writing tests for silicon validation. In fact, many of the first tests run on silicon may be existing tests from pre-silicon. If the pre-silicon tests require environment infrastructure to check results, they are not useful to validate silicon since the checking capability is lost. So why not write these as self-checking tests to start with in order to maximize their compatibility?

Objection A3: Since I already have a golden reference model (GRM), I can save time by integrating it into my unit-level environment and making the unit environment portable to leverage the GRM as a results calculator at a cluster or full-chip environment.

For a golden reference model to port easily, it must interact with the monitors in the UVM agents. The model may not map well to the individual interfaces of the DUT and requires additional work to understand the low-level transactions reported by the monitors. In the end, you could end up spending more time to retro-fit the model. The

problem does not stop there since corner cases scenarios usually trickle in at any time and break the integration. Let's consider the unit environment in Fig. 4, which utilizes a golden reference model.

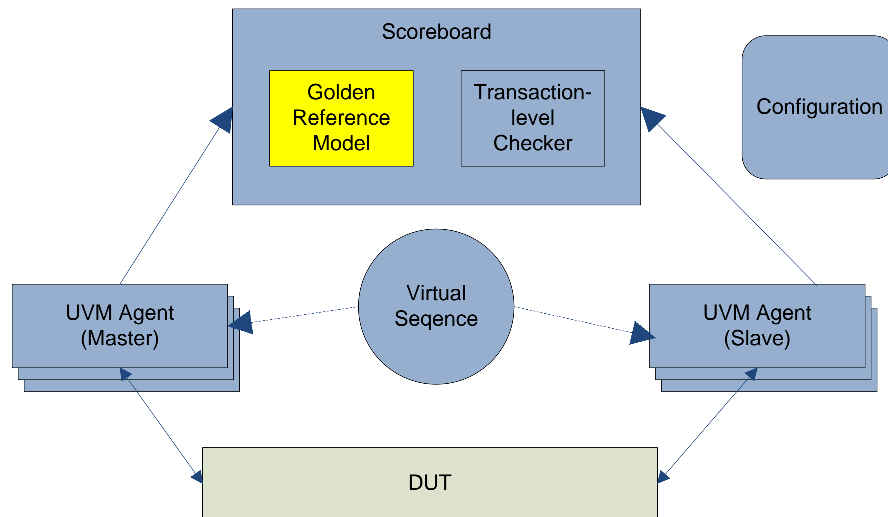

Figure 4. Simple golden reference model environment.

The golden reference model environment is similar to our typical UVM environment from Figure 1, except that the GRM replaces the "predictor" functionality of the scoreboard. Furthermore, the GRM is usually written in a language other than SystemVerilog and likely used for a purpose other than RTL verification. The model is connected to the UVM environment using special ports or interfaces (DPI, TLM, etc) to cross language boundaries.

An initial estimate may show that using a GRM will save time in building up a portable unit environment. However, let's consider the true development costs:

1. Plumbing infrastructure is required to connect the model to the UVM environment. There can be limitations to work around when crossing language boundaries, which may not be apparent during the initial cost estimates.
2. Verification requirements for the model may be different than the goals for the golden reference model. Hence, the verification engineer may not have the flexibility to make changes to the model if the model serves a dual purpose. This will pose a challenge if changes to the model are necessary to retro-fit it to the unit environment.
3. The model may be too high level to be useful for checking transactions. A golden model usually produces end results only, and intermediate values may not match the actual transactions observed by UVM monitors. This means that the scoreboard (or GRM) must still retrofit low-level observed transactions to match high-level operations from the GRM. This may become more challenging if there are many interfaces the DUT uses before arriving at the final result.

In fact, the golden reference model may be a better fit when used with the proposed self-checking stimulus to check end results rather than the low-level transaction checking required by a portable scoreboard. This will become apparent in the next section (B) which explains the checking methodology used by the self-checking stimulus.

B. Check operations, not transactions

The self-checking stimulus will utilize a checking methodology that checks results of operations instead of individual transactions. This flow is summarized in Fig. 5.

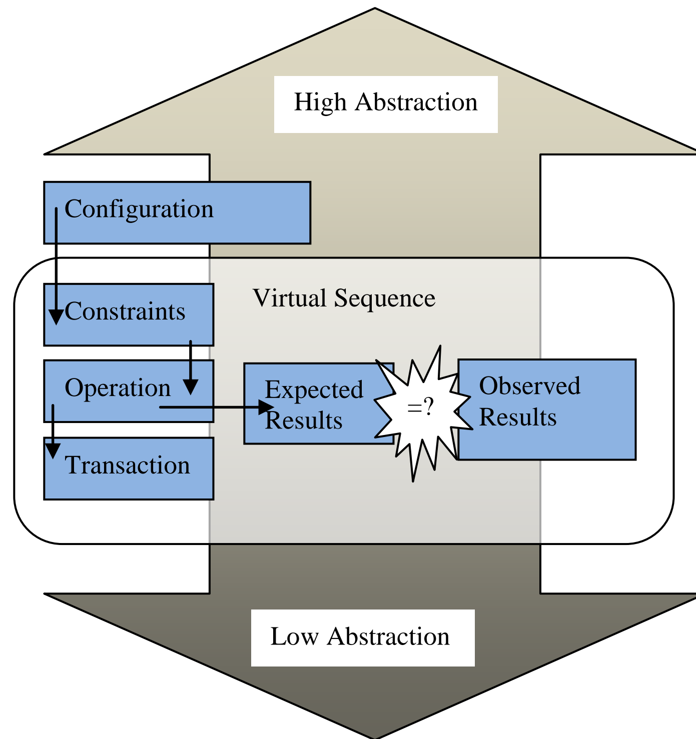

Figure 5. Stimulus lifecycle in self-checking stimulus environment.

Compare this with the scoreboard environment flow from Fig. 2. We have removed the transition from low-level transaction up to operation, and from operation down to expected transactions. This effectively eliminates all redundant paths between high and low levels of abstraction and keeps most of the environment functioning at the higher level 'operation' abstraction layer, including the checking. The result is fewer infrastructures to write and maintain, and less risk of checker breakage.

Of course, by doing away with low-level checking we can introduce some risk. Let us address some possible pitfalls with this self-checking stimulus.

Objection B1: I am compromising on quality of checking by looking at high-level operation results instead of checking every transaction.

It is important to carefully consider your design and quantify what is being compromised. Once any checking holes are identified, it is important to explore solutions that address these holes. Many times these checking holes can be addressed fairly easily. Furthermore, they can be addressed by leveraging work that is being done regardless of the choice of environment architecture.

Let's consider a standard DMA engine [5] as an example. A checking methodology using the proposed self-checking stimulus will utilize a 'results checker' to simply read the data from the destination memory block at the completion of a DMA transfer operation and compare with what was stored at the source memory block. An error is generated only if there is a miscompare at the range of addresses corresponding to the destination block. A checking methodology using a portable scoreboard will check off each read transaction as it is issued to the source addresses and each write transaction as it is issued to the destination addresses. It will also generate an error if any rogue read or write transactions occur to either the source or destination memories. What if there was a bug such that an extra write transaction was issued outside the destination address block? The portable scoreboard as described would detect that error but the results checker would not since it does not look at addresses outside the destination block.

This problem is easily addressed. Regardless of the architecture of your environment, it is likely you are using some sort of memory manager object to manage allocation and freeing of memory blocks as your test generates and completes DMA operations. It is also likely that UVM monitors are already in place at both the source and destination memory interfaces. The UVM monitor can be extended to have knowledge of the memory manager object, which will allow for a simple check in the UVM monitors to flag an error whenever a read or write access occurs to a freed address. The premise of the fix is that RTL cannot read or write to an address that the test did not expect (otherwise the test would have allocated that address). With this check, the type of bug described can be caught without need for the scoreboard.

Another bug can occur such that redundant read or write transactions occur to the same address. For example, a request ready bit may be stuck high one extra cycle causing back-to-back writes of the same data to the same address. A scoreboard would flag this immediately but a results checker would not. We can reduce the risk of this hole by capturing and checking performance metrics. Performance analysis is likely something that is being done anyways across the chip, so this should not be much extra effort. Additionally, a UVM agent can track the number of transactions captured and the test can use this to verify this number is within an acceptable range.

Objection B2: I will spend more time to root cause failures if I don't see errors until the final result is checked. I would like to see an error as soon as a transaction mismatch occurs.

The self-checking stimulus plan includes UVM agents with monitors that track transactions on every interface. These transactions can be dumped (e.g. to a file) and used as a valuable debug aid in root causing failures. Furthermore, the agents are portable to the system level since they can become passive UVM agents. Losing the ability to check these individual transactions is a trade-off to be considered when opting to use self-checking stimulus with results checking instead of a portable scoreboard.

So far we have identified the disadvantages and risks associated with self-checking stimulus in general, and discussed how to deal with or work around those disadvantages. The next section will provide more detail on how to implement the self-checking stimulus using UVM.

C. Stimulus-centric Implementation

The self-checking stimulus is implemented using UVM virtual sequences. A stimulus sequence can support anything from the simplest operations allowed on the DUT to the most complex test case that will run on the DUT. A stimulus sequence will use other stimulus sequences to build up to more complex operations, which can be used by test case sequences. An example of how the stimulus sequences can be organized is shown in Fig. 6.

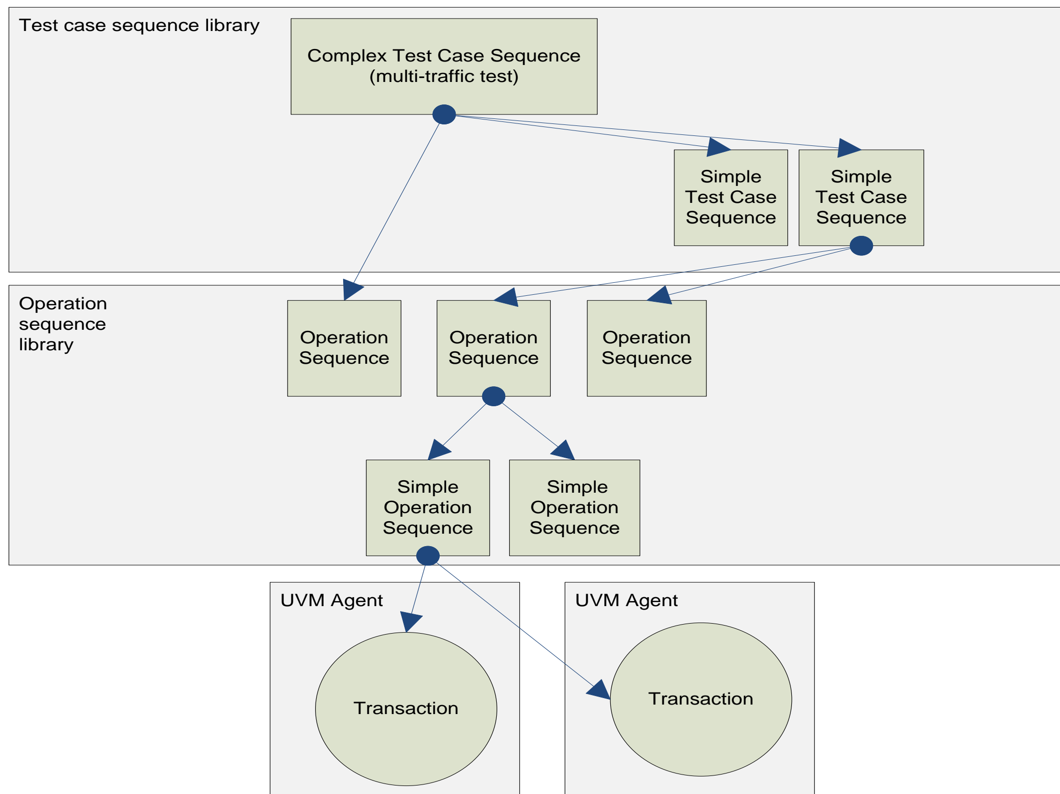

Figure 6. Stimulus sequence organization.

Each stimulus sequence object will:

1. Generate the stimulus – Populate fields based on the configuration constraints, configure any sub-sequences needed as building blocks for this stimulus.
2. Allocate resources – Any shared environment resources needed by the operation are reserved (such as memory space).
3. Calculate expected results – Predict and save off the expected end results in a local data struct. This is easier to do here than in a portable scoreboard since all post-randomized configuration and actual stimulus is known locally.
4. Configure the DUT – Program the DUT according to the desired operation by issuing transactions to the appropriate UVM agents.
5. Send the stimulus – The high-level stimulus is broken down into low-level transactions that one or more UVM agents can send to the DUT.
6. Wait for 'done' – The sequence waits for a 'done' indication from the DUT. This indication can be a bit in a register, a side-band signal that is part of a bus interface, or some logical concatenation of other signals if no such flag exists natively.
7. Collect the results – Fetch the results generated by the DUT. Ideally this is done in 0-time and non-intrusively to the state of the DUT, to avoid affecting existing stimulus. In the DMA example [5], this would amount to a backdoor memory read of the destination memory block.
8. Check the results – compare calculated results versus what was generated by the DUT.
9. Free resources – any resources allocated from step 2 are freed.

A base stimulus sequence can be defined to set the framework described. Next, the simplest operations are implemented by extending from the base stimulus. These are building blocks for more complex operations. Test cases can mix and match any of the simple or complex operations from the stimulus library to create a specialized or random test.

IV. Is Self-Checking Stimulus Right For My DUT?

To understand whether or not the proposed self-checking stimulus is the best solution for a given DUT, it is necessary to consider the following:

1. Precision of checking: To what extent is the correctness of individual transactions required in verifying the correctness of the overall design?
2. Practicality of transaction-level predictability: What sort of effort is required to accurately predict the low-level transactions required by the portable scoreboard?
3. Delineation of Results: Does the DUT allow the testbench to easily access results after all transactions for that operation have completed?

A. Precision of Checking

The required precision of checking is perhaps the most important factor to consider. In many designs like those that write results of operations to a memory, it may not be important to check every transaction that wrote the results to memory, as long as the end result is correct. In other designs such as controllers for bus protocols, it is very important that each transaction is generated by the DUT with precision.

Recommendation: If transaction-level precision is not required, then self-checking stimulus should be considered.

B. Practicality of Transaction-Level Predictability

Any design can be made predictable down to low-level transactions given enough effort to model it. However, one must consider the effort actually required to do so. If there are many ways to arrive at the correct answer, there will be many different correct combinations of low-level transactions that the scoreboard's transaction checker must be designed to accept. A portable scoreboard may require a high degree of predictability from the design to maintain accuracy in checking of low-level transactions. This level of predictability required may ultimately lead to a cycle accurate model, or may cause the scoreboard to be sensitive to simple design changes, bug fixes, or timing fixes late in the project. The cost of maintaining the scoreboard's accuracy in checking low-level transactions must be considered before choosing a portable scoreboard.

Recommendation: If the prediction effort is not practical, then the self-checking stimulus should be considered.

C. Delineation of Results

For self-checking stimulus to work well there must be a clear boundary between results across multiple operations, and a clear indication of completeness when an operation finishes. So the questions to consider are:

1. Can a 'done' flag or signal be extrapolated from the DUT as it completes each operation?
2. Can the DUT generate results without corrupting them before the operation is complete and the testbench has had a chance to check the results?

Recommendation: If the answer is 'no' to either of the above questions, then the self-checking stimulus cannot be used. Otherwise, the self-checking stimulus should be considered.

V. RESULTS

Having coded/architected several unit environments of varying degrees of complexity and portability with scoreboards and golden models, I have seen the pitfalls and benefits of these.

On a past project, we had access to a 'golden model' primarily written for purposes other than verification, and in a language that is not SystemVerilog. At the time, it was thought to be a good idea to leverage this work in our unit-level UVM environment, and port it up to full chip. I needed a checker well before the model was ready for use, so I decided on the low-cost solution of writing self-checking stimulus, with the intent of later phasing it out in favor of the 'golden model' based scoreboard. Although the initial integration of the golden model went well, we started finding many end cases that did not work well with the transaction-level scoreboard. The scoreboard required frequent attention, and frequent debug of false fails. The effort was initially estimated at 3 man-weeks to complete, but ultimately consumed 1 full time engineer for more than 3 months without completion. Fortunately, we had been maintaining the self-checking stimulus and eventually were able to abandon the 'golden model' scoreboard altogether, and cut our losses.

Looking back on the experience, I found that I was able to spend significantly more time on actually verifying the DUT with the self-checking stimulus instead of fixing/maintaining environment infrastructure and chasing down false fails. I was able to get a lot more verification for my effort without the golden model, and was able to mitigate the loss of benefits from not having a scoreboard/model.

VI. CONCLUSIONS

The rising complexity of designs has made verification environments even more complex. If we continue to focus on portability, we continue to chase a rising cost for a benefit that may no longer have the value we think it does. It is important to reevaluate past assumptions on what is required in a verification environment, and carefully consider the cost/benefit tradeoffs when deciding how to construct the environment.

Many times strategic decisions in verification are made based on fear, rather than careful consideration of the risk/benefit tradeoffs. There is fear that not having low-level checkers port to system-level tests will result in missed bugs. While in many cases there are very good reasons for porting these checkers, we must first consider these reasons before blindly investing the effort to support the portability. Taking the time to do so could save considerable time and resources during the course of the project.

The checking capability of the self-checking stimulus will not port up to other environments. However, this desired portability must first be evaluated for its true benefit. UVM sequences make it even easier to create self-checking stimulus because we can now abstract up from a flat transaction object that relies on other infrastructure to manage it, to a mostly self-contained 'live' transaction or operation sequence which lives and dies with little management from outside infrastructure. This is important because it allows us to efficiently build up libraries of self-checking stimulus sequences, using inheritance. The time saved by constructing self-checking stimulus in lieu of a traditional scoreboard can be used towards improving stimulus quality and finding more bugs.

ACKNOWLEDGMENT

I would like to thank Tom Symons and Ram Narayan for the encouragement and support in challenging what is widely accepted as standard verification practices. It is always risky to try something new and unproven under a development schedule. However, often we find the bigger risk lies in traditional approaches. Your coaching has helped me understand the risks and push the boundaries of accepted verification practices. In addition, I would like to thank Doug Good for supporting this methodology to finish a critical project. Thanks also to Jeff Montesano for your contributions to make this a better paper.

REFERENCES

[1] Accellera, Universal Verification Methodology, www.uvmworld.org
[2] M. Litterick, “Pragmatic Verification Reuse in a Vertical World”, DVCon 2013
[3] H. Chan, et al., “Maximize Vertical Reuse, Building Module to System Verification Environments with UVM e”, DVCon 2013
[4] “Verification Challenges: Past, Present & Future”, DVClub Panel discussion, September 9 2015, Austin, Texas
[5] ARM, CoreLink DMA-330 DMA Controller Technical Reference Manual r1p2, www.arm.com