Pragmatic Verification Reuse in a Vertical World

Mark Litterick
Verilab GmbH.
Munich, Germany
mark.litterick@verilab.com

Abstract— **Successful application of block-level verification reuse improves the effectiveness of the top-level environment by providing additional checks, coverage and messages (and in some cases stimulus) which, as well as detecting more bugs, helps speed up debug for other system-level defects by providing improved internal visibility and enhanced bug isolation. Despite these benefits consistent efficient reuse is simply not being achieved in many companies. This paper revisits the vertical reuse problem from a fresh standpoint and addresses the fundamental issues involved, provides a comprehensive set of pragmatic reuse guidelines and also suggests how to go about retrofitting reuse to existing block-level environments.**

Keywords—Vertical Reuse; Verification Reuse;

I. Introduction

Successful application of block-level verification reuse improves the effectiveness of the top-level environment by providing additional checks, coverage and messages (and in some cases stimulus) which, as well as detecting more bugs, helps speed up debug for other system-level defects by providing improved internal visibility and enhanced bug isolation.

The detailed understanding of block-level protocol and associated expertise that has been built into the automatic checks such as scoreboards, the ability to collect functional coverage based on coherent transactions and the capability to provide concise and informative messages based on signal decoding is of course also reused. This allows the top-level team to concentrate more on appropriate concerns such as overall functional correctness of the application, external interface behavior, module and sub-system interaction, access to shared resources, low-power operation, performance and interconnection verification requirements.

Reuse of block-level verification environments at the top-level is potentially well supported by modern verification methodologies such as the Universal Verification Methodology (UVM) [1]. However, the evidence from scores of projects in many different companies is that successful and effective vertical reuse is simply not being consistently achieved.

This paper revisits the vertical reuse problem from a fresh standpoint and addresses the fundamental issues involved[1], provides a comprehensive set of pragmatic reuse guidelines[2] and also suggests how to go about retrofitting reuse to existing block-level environments. The discussion is essentially language independent, but any major differences between *e* and *SystemVerilog* methodologies are highlighted where appropriate, however UVM terminology is used throughout.

II. Vertical & Horizontal Reuse

The world of verification is not flat. The terms "vertical" and "horizontal" are used to describe the context into which verification artifacts are reused. In this respect horizontal typically means using a verification component in a different system or project but at roughly the same level of abstraction and with the same functional role. Vertical reuse on the other hand refers to using a verification component in a different hierarchy level, usually with an implied change of role.

For example, a verification component used in block-level, sub-system, and top-level environments within the same project, exhibits vertical reuse. In this case we would expect the stimulus to be nested inside higher-level stimulus until the stage where the associated interfaces are no longer external signals, but embedded connections internal to the design, at which point the role of stimulus generation would no longer be applicable.

This paper focuses on vertical reuse attributes and especially those related to the change of role when going from block-level to top-level verification hierarchies. Comprehensive block-level verification is essential to full chip success, but at the end of the day nobody tapes-out a block, it is the top-level chip that brings in the money - in that respect we live in a vertical world.

III. Top-Down or Bottom-Up

The two engineering paradigms of top-down and bottom-up design can also be applied to the concept of vertical reuse. In an ideal world without real project pressure, it would be reasonable to assume all verification environments were composed in a bottom-up manner where block-level environments are combined and layered in a well-structured manner until we arrive at a top-level environment with implicit reuse [2]. Indeed the standard verification methodologies often portray this as normal practice rather than an ideal goal.

[1] The red warning triangle symbol, △, is used throughout this document to indicate a **hazard** observed in real verification environments.

[2] The right arrow symbol, ⇨, is used throughout this document to indicate **guidelines** and recommendations.

The evidence from many of projects at a variety of clients is that the real world does not conform to this ideal △. The main reasons for this are threefold; firstly that the top-level verification environments have different requirements; they must support highly constrained real-life scenarios and directed tests, perhaps also running real software and firmware, in many cases with accurate behavioral models for analog blocks integrated into the digital chip. Secondly, these top-level environments are almost always on the critical path and are therefore developed in parallel with the corresponding block-level environments. Thirdly, the sheer variety of quality and capability in what is often an ad-hoc collection of block-level sub-systems means that forming a coherent homogeneous amalgamation that meets all the top-level requirements is just not feasible.

This leads to the situation where the top-level environment may well be developed in a top-down manner in order to achieve its goals independently from the block-level testbench developments. At a later stage in the project the block-level verification components are imported to enhance the effectiveness of the top-level environment by supplementing the external Device Under Test (DUT) operation validation with comprehensive internal checks, coverage and messages.

IV. Verification Requirements

The top-level testbench has quite specific verification requirements that differ from the comprehensive but isolated constrained random concerns dealt with by the block-level environments. Top-level requirements include:

- Functional correctness of overall application
- Interaction of all module and sub-systems
- Access to shared resources such as memory and bus structures
- Operation with realistic clock and power domain behavior
- Overall performance of the system
- Parallel external interface behavior
- Connectivity of all blocks and sub-systems

We cannot achieve closure on all of these verification requirements by looking only at the external pins in a complex top-level chip. These requirements mean we need to measure coverage and check operation of critical functional data paths inside the DUT including validating relationships between blocks while running top-level test scenarios. If the top-level tests fail to address these concerns then it represents significant risk to the overall quality of the application.

One additional advantage of re-using the block-level verification environments in the top-level is that we close the loop on functional expectations for the block – often a block that is comprehensively validated in a stand-alone system finds itself under different and unexpected conditions in the top-level environment (for example starved of resources) – so sometimes the block level environment and requirements are themselves improved by the application of reuse.

V. Debug Requirements

It is not enough to provide a top-level verification environment that proves functional operation of a perfect chip. The path to perfection will no doubt include failures that need to be debugged effectively and efficiently during top-level simulations. The time taken to debug failures, especially in the top-level environment, is probably the biggest drain on engineering resources at later stages of the project and can put the simulation effort firmly on the critical path. Additional internal checks and messages can go a long way to helping debug requirements become a reality. Localized checking at block and sub-system boundaries improves the observability of failures and can quickly isolate problems. Additional informative messages for operational paths and internal interfaces are also invaluable for identifying which aspects of the chip are behaving as expected even though the overall test indicates a fail. The best way to achieve this level of bug isolation within a complex top-level environment is by making extensive reuse of block-level artifacts operating in passive mode as detailed in the next section.

VI. Active & Passive Operation

All modern verification methodologies make a clear distinction between active and passive operation of verification components and environments. This can be summed up as follows:

- Active components provide and affect stimulus driven to the DUT
- Passive components do not provide or affect the stimulus in any way

In this context stimulus is more than just generating inputs to the DUT from a proactive master, but also includes providing responses from reactive slaves, clock generation and side effects like delays. All of these are affected by active components and none are affected in any way by components operating in passive mode.

Both the functional capability and the run-time architecture of the corresponding verification environment are affected by the active or passive mode setting. Consider for example the active block-level verification environment shown in Figure 1.

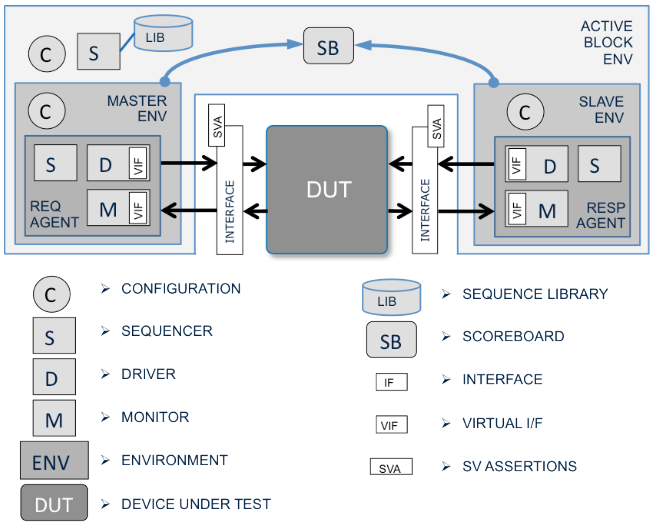

Figure 1. Active Block-Level Environment

In the active environment shown in Figure 1:

- Stimulus is provided by sequencers and drivers
    - proactive master generates request stimulus
    - reactive slave generates response stimulus
- Checks are performed by interfaces, monitors & scoreboards
- Coverage is collected by monitors
- Messages are generated by all components

The corresponding passive version of this block-level verification environment is shown in Figure 2:

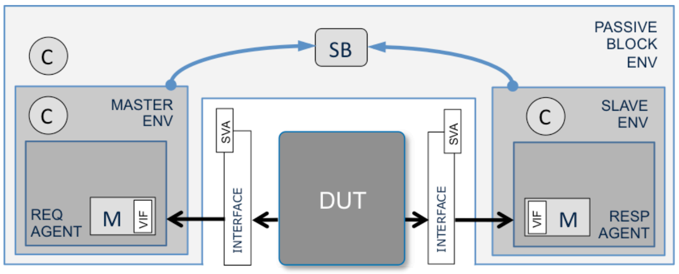

Figure 2. Passive Block-Level Environment

In the passive environment shown in Figure 2:

- No stimulus is performed by the passive components in the environment
    - sequencers and drivers are not present
- Checks are still performed by interfaces, monitors and scoreboards
- Coverage is still collected by the monitors
- Messages are generated by the remaining components

Typically passive mode is used in situations where the relevant DUT block is instantiated in a higher-level sub-system or top-level chip. In this case verification stimulus is provided by active components of the verification environment outside the scope of the re-used block-level environment. This is illustrated in Figure 3, which shows the passive block-level verification environment being used in the context of a top-level testbench.

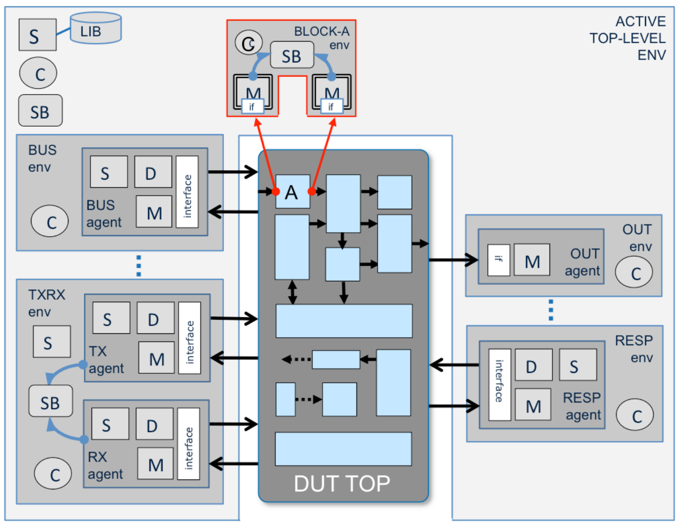

Figure 3. Passive Block-Level in Top Environment

In this case all of the checks, coverage and available debug messages are reused to ensure a more effective validation of the block in context of the top-level DUT as well as supporting more comprehensive top-level testbench operation.

A. Active/Passive Mode

In order to select between active and passive mode of operation, all relevant sub-components must include an active/passive configuration flag. Note that the complete block-level environment must take into account active and passive settings and not just the lower-level interface verification components. Often engineers make a reasonable job of the low-level verification components but fail to implement correctly structured active/passive mode settings for the enclosing environment △. The intent is to provide the top-level integrator with a complete and properly encapsulated passive environment that contains all of the necessary lower-level verification components configured in an appropriate manner.

> ⇨ Complete environment must consider active/passive configuration, not just verification components

The active/passive mode setting should be used during the creation (build or generate phases) of the environment and affects the actual constructed instances of sub-components within the agent and environment classes. Active components like sequencers and drivers are not constructed in passive mode.

⇨ Do not create active components during passive construction

Since the active components are not present in a passive agent or environment, it follows that we cannot connect to their Transaction Level Modelling (TLM) ports or assign internal pointer references to other components.

⇨ Do not attempt to connect to active components during passive mode

B. *Scoreboards*

Scoreboards are a special kind of checker that performs comparisons between transactions published by different monitors in the system. These checks may comprise transformation checks, where one transaction type is modified to generate a different transaction type, or they may be validating propagation and distribution for transactions of the same kind.

Scoreboards are critical to validating functional paths throughout the design and therefore reuse of block-level scoreboards adds extremely important checker capability to the top-level environment. Since these scoreboards live in passive environments, the transactions destined for scoreboard comparisons must come from passive components and not active components (which are not present in the passive architecture). However, even though this is a fundamental rule in each of the standard verification methodologies, it also appears to be the biggest single violation in block-level architectures △. It is also the hardest architectural defect to recover from when attempting to retrofit verification reuse capability as discussed later.

⇨ Do not connect scoreboards to active components

The main cause of bad scoreboard connectivity appears to be that some verification engineers recognize a shortcut is possible which saves effort for the monitor development, but overlook the fact that vertical reuse is totally compromised. Specifically, for an active environment the transaction field information typically exists in the form of a sequence item that is provided to the driver. It is easier to send this sequence item (which represents a request to drive a transaction) to a scoreboard than to correctly implement the monitor to fully decode the interface signals (the observed traffic), construct a transaction object from this, and then post it to the scoreboard. However, for robust functional verification, the monitor must independently decode the actual traffic on the resolved signals for the interface and publish this information via analysis ports; this ensures reliable checker operation and supports passive reuse.

C. *Functional Checks*

In addition to the transaction comparisons done in the scoreboard, all functional checks for transaction content and protocol behavior should also be done in passive components, typically monitors. Everyone knows that. Why then do people choose to implement timeout checks and error messages inside drivers? △ In principle drivers may need to check some aspects of protocol behavior in order to correctly react to DUT responses and recover correctly from anticipated failures, but actual protocol checks and error messages need to be implemented independently inside the corresponding monitor as well.

⇨ Perform functional checks in passive components

Protocol operation and timing checks are often implemented using SystemVerilog Assertions (SVA), which notionally belong to the monitor but must be implemented outside of the class-based environment, usually in the corresponding interface construct [3]. Since the interface is also instantiated in passive environments these SVA protocol checks are still operational in passive mode.

D. *Functional Coverage*

The vast majority of the functional coverage should be implemented in passive components to allow for reuse at the top-level. This approach also tends to strengthen the quality of the functional coverage since, for example, we do not cover that we requested a particular transaction to be sent to the DUT, but rather we cover what was actually sent when the monitor has decoded the observed traffic.

Occasionally some stimulus coverage is deemed appropriate and cannot be measured by a monitor, for example some error injection modes that would result in traffic that cannot be reliably decoded by a monitor [4]. In such cases separate isolated coverage in the driver is appropriate, but cannot of course be reused in the top-level environment and therefore does not contribute to the verification goals for top-level △.

⇨ Collect functional coverage in passive components

E. *Configuration Updates*

It is often the case that there are pseudo-static configuration fields related to protocol operation that must be kept up-to-date to ensure correct functional operation of the verification environment. These settings can change during the running of a test, typically in response to some traffic on one or more interfaces to the DUT. Since the value of these configuration fields is affected by stimulus in the block-level environment, it is tempting to update the configuration directly from the sequences or driver △. However, since these active components are not present in a top-level reuse scenario and it is still crucial to maintain the configuration accuracy, then these fields need to be updated by passive components, typically monitors.

⇨ Update configuration only from passive components

F. *Information Messages*

Something that is often overlooked is the importance of preserving relevant informative messages in a passive reuse scenario in order to help debug failures and isolate working aspects of the top-level chip △. All sub-components and methods in the verification environment are allowed to generate informative messages; however, only the messages from passive components will be available when the environment is used in passive mode.

⇨ Generate important messages in passive components

G. *Warning Messages*

Many block-level verification environments include the capability to inject errors from the active components as part of the stress test features for comprehensive block-level verification requirements. Typically transactions sent to the DUT with illegal content are not reported as an error under these circumstances (since that would cause the test to fail), but rather a warning △. The verification environment will expect specific error detection and recovery operation from the DUT as a result of the error injection and if that does not occur then a real failure message is generated. In a top-level environment we may want to reclassify illegal transactions arriving at an embedded block as an error since this could help isolate real problems in the situation where the upstream RTL block generates illegal input signals.

Under these circumstances the monitor, which is always a passive sub-component even though it is present in both active and passive environments, would use the active/passive flag setting only to determine the message severity.

⇨ Consider promoting warning messages to errors in passive mode

H. *End-of-Test Control*

Normally the components of a block-level environment make a combined decision of when would be appropriate to end the test. This can be a mixture of active stimulus decisions and passive observations such as waiting for an ongoing transaction to end. The usual mechanism for controlling test flow is called objections – basically any component can object to the test ending until it sees fit.

However, when the passive block-level components are reused in a top-level context it may no longer be appropriate for the monitor on an internal interface to object to test completion △. For example we must be able to tolerate termination of the test during a partial transaction on any one of many internal or even external interfaces, likewise a direct memory access (DMA) transfer need not necessarily complete if the top-level scenario does not require this. This is another case where the passive component (like a monitor or scoreboard) needs to be aware of the active/passive flag in order to behave appropriately.

⇨ Do not control end-of-test from components in passive mode

I. *Stimulus Reuse*

When a block-level verification component is reused in active mode as part of the stimulus hierarchy in a top-level environment additional care is required with the sequence API. Top-level stimulus is typically more constrained than the equivalent stimulus at the block-level and normally the stimulus is encapsulated inside a higher-level protocol or running in parallel with multiple verification components to create interesting scenarios. It is not appropriate to put too many constraints into sequence items △ (e.g. distribution constraints) since the user will typically not be generating items directly, but rather most of the user constraints should reside in sequences and the sequence item should only contain legality constraints.

⇨ Put user constraints in sequences not sequence items

Likewise the sequence API should not be at too low a level for the top-level scenario generator (i.e. with all sequence item fields exposed as control knobs), but rather powerful high-level functional sequences should be provided as well as more generic low level sequences. It is unlikely that the block-level supplier can predict all use-cases in the top-level environment △, but they can provide a comprehensive sequence library that supports full functionality of the verification component protocol. If the verification component has more than one active agent, then the user component supplier should encapsulate all sequences into a single sequence library registered with the highest-level virtual sequencer in the verification component environment.

⇨ Provide a comprehensive sequence library that encapsulates low-level and high-level functionality

VII. Problems of Scale

The sheer scale of integrating many block-level components into a single top-level environment introduces additional demands on the block-level suppliers related to build encapsulation, configuration, namespace isolation and interface operation that may not be at all apparent when working in a block-level only domain. Consider for example the potential headaches involved with badly encapsulated block-level reuse as shown in Figure 4.

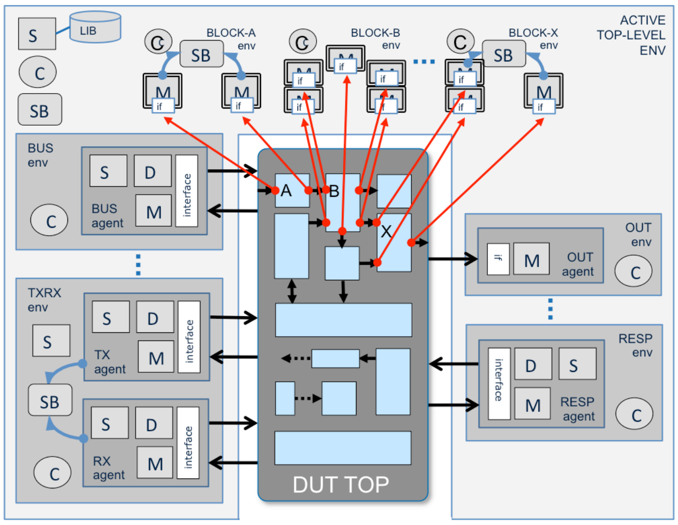

Figure 4. Un-encapsulated Reuse

Typically the top-level verification environment is already complex with many external interfaces and internal paths all operating in parallel and of course the full scope of DUT behavior needs to be catered for including modeling analog blocks, clock domains and power islands. In order to support the top-level team in validating the verification requirements necessary for tape-out, the block-level suppliers have a duty to make things as easy to integrate and reuse as possible.

A. Environment Encapsulation

Typically it is the entire cluster of components from the verification environment that is required for reuse, excluding the lower-level tests. It is not appropriate to expect the top-level to reconfigure and connect the various lower-level verification components into an appropriate topology △, but rather this must be provided by the block-level environment encapsulation itself.

The chances are that this encapsulation is already done to some degree in the block-level environment, but the real question is if it can be reused as-is without having to copy additional code into the top-level environment. For example, if the block-level environment topology is pulled-together in a base-test component, then it is by definition not reusable since the tests are not ported to the top-level △. Likewise if the uppermost environment in the block-level does not consider active/passive settings then the environment needs additional work before it can be reused. The best solution here is to encapsulate all components and settings into a single environment component and use it in both the block-level base-test and the top-level environment.

⇨ Encapsulate all sub-components in a single reusable environment

B. Configuration Encapsulation

It is assumed that most block-level verification environments make use of multiple interface and module verification components, each of which must be configured to match the requirements of the block. It is not appropriate to pass the responsibility for configuring all these sub-components up to the top-level △. Rather, the block-level environment should configure all the lower-level settings that are invariant for the block in this project setting and encapsulate what few flexible configuration settings remain into an object made visible to the top-level. Hence the top-level environment only has to care about one configuration object for each complete block-level environment and the configuration object only provides relevant fields that can be changed in the environment.

⇨ Encapsulate configuration objects correctly and manage content

C. Interface Encapsulation

Normally each agent in the verification components within an environment has its own dedicated interface construct handling a single functional group of signals. With several verification components grouped together in the environment we end up with several individual interfaces to instantiate, connect and associate with virtual interfaces inside the class world. This is not usually a problem until we consider the scale of the top-level environment encompassing many passive block-level components.

Under these circumstances it is not appropriate to present the top-level integrator with a set of fragmented signal interfaces (or signal maps for e) in order to reuse a single block-level environment △. Specifically, there is a real danger of missing a vital connection, making incorrect connections or failing to associate the virtual interface references appropriately. The risk is compounded because the reused block-level environment is connecting to internal signals in the DUT, using white-box probing, and not external ports. So we need to communicate more precisely exactly what needs to be connected for the block-level environment to continue to operate, and the best way to do this is through encapsulation.

The block-level supplier should provide a single hierarchical interface which instantiates all the required lower level interfaces. This interface encapsulation enables more effective white-box probing of the embedded block in the DUT since only one HDL path to the embedded block is required and the top-level integrator only has one virtual interface assignment to make.

⇨ Combine multiple signal interfaces into a hierarchical interface

D. SVA Encapsulation

Functional protocol checks are often implemented using SVA and therefore cannot be in located in the SystemVerilog classes for the verification component to which they belong △. These assertions should therefore be located inside the corresponding signal interface construct for the verification component so that they are automatically included when the interface is instantiated in the testbench module [3]. There is no need to independently bind a collection of block-level assertions that belong to the verification components to the internal signals in the DUT, since this connectivity is required for the interface signal connections anyway. Design assertions that are not embedded in RTL will have to be handled separately, but this can also be achieved by encapsulation inside an interface construct.

⇨ Encapsulate SVA protocol checks inside the interface

E. Namespace Collisions

Importing multiple packages into the same scope can cause namespace collisions if the artifacts in each package are not correctly named. This is not just restricted to global constructs like macro definitions, but also class, constant and type definitions within each package, including enumeration literals △. The increased scale of top-level is much more likely to uncover namespace collisions than individual block-level environments since many packages are imported into the same scope.

In general, things like constant definitions and class names do not suffer from defects here, but the smaller items get overlooked. For instance it is not appropriate to have a BUS_WIDTH macro definition since at the top-level there will be many busses each with different widths. Likewise it is tempting to use short names for the enumeration literals inside an enumerated type definition, (e.g. IDLE), but when multiple packages are imported into the same scope using the wildcard operator (*), any enumeration literals with the same name collide even though the enumeration type name itself may be unique. The recommended approach is to prefix all named items visible in the package scope (e.g. class, constant, macro and types, but not class members) with a unique string derived from the verification component's name.

⇨ Avoid namespace collisions by using unique prefix throughout package scope

F. *Benefits of Encapsulation*

Following the guidelines for correct encapsulation and naming of the block-level verification components and environments results in much easier and less error-prone integration into the top-level testbench. The resulting environment is more modular and easier to maintain as well. Figure 5 illustrates an improved encapsulation compared with that shown previously in Figure 4.

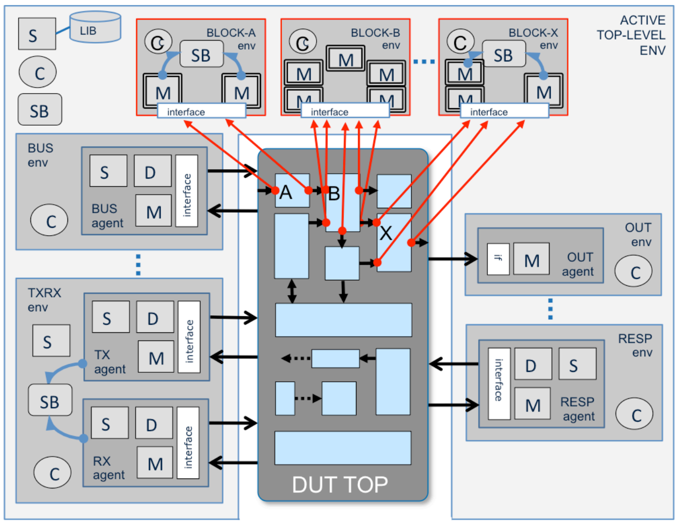

Figure 5. Encapsulated Reuse

VIII. Additional Concerns

Not all vertical reuse concerns are directly related to the change of role associated with block-level verification environments being reused in the top-level hierarchy, although that is the focus for this paper. This section outlines some additional concerns that need to be taken into account in many projects and is included in order to give the reader a more comprehensive view of the overall problem space; unfortunately a detailed analysis of each of these topics is outside the scope if this paper.

A. *Formal Verification*

Formal verification is normally applied at the block-level due to tool limitations and the need to contain the scope of the mathematical proofs. Vertical reuse of formal verification artifacts is however not only desirable, but also a requirement, in order to ensure correct integration and dynamic operation of the associated block △. In particular it is essential to validate that the assumptions used in the formal analysis hold true in the top-level environment by reusing the assume statements as assertions in the simulation environment. The functional properties may, in general, also be reused in the top-level environment to validate correct dynamic behavior of the block and provide confidence on the accuracy and completeness of the formal analysis.

⇨ Validate assume statements from formal verification

B. *Power-Aware Simulations*

Most block-level verification environments do not execute power-aware simulations and yet this might well be a critical verification requirement for the top-level environment. Specifically most signal-based transactors (e.g. drivers and monitors) in the verification components and associated SVA checks may not behave correctly, or recover in an appropriate manner, if the signals (including clock and reset) transition to undefined values in the top-level simulation due to power-domain manipulation △. All verification components in applications where power-aware simulations will be required need to take power-intent attributes into account [5]. As a minimum the drivers, monitors and SVA checks need additional power state indicators which can be used to prevent false failures, ensure correct recovery from power-off states and of course validate the actual power state sequencing. If the block-level verification requirements include power-intent simulations, then the corresponding meta-data must be correctly encapsulated for reuse in a higher-level environment.

⇨ Verification components must be power-aware for low-power applications

C. *Assertion-Based Verification*

Many RTL blocks are instrumented with embedded, or separately bound, assertions intended to validate important design requirements. These assertions can be invaluable in detecting and isolating functional failures and abuse of the RTL block in the top-level system. However, caution is required. Typically RTL assertions are focused at a very low-level and could therefore quite significantly affect the performance of the top-level verification environment if all the assertions for all blocks are active and evaluating some property on every clock cycle △. Having many thousands of relatively uninteresting design assertions can also contribute to a false sense of security △ – these assertions are only valid if they really protect the block from misuse or they contribute directly to top-level verification requirements. For some reason the quality of design assertions varies significantly, perhaps due to the different mind-sets of the originators – while this situation might be tolerable at block-level it can be a killer in the top-level situation.

Block-level RTL assertions should be well documented and encapsulated such that individual checks can be selectively enabled or disabled in the top-level environment. This can be achieved using labeled assert statements in SVA or by providing control knobs for continuous assertions embedded in supplementary checker code.

⇨ Enable only appropriate RTL assertions for top-level

D. Clock-Domain Crossing

Verification of Clock Domain Crossing (CDC) signals is almost always part of the requirements for modern top-level environments. Even if the CDC behavior is partially visible at the block level, this might not be enough to validate interaction of all the clock domains at the top-level △. This is especially true when low-power features like Dynamic Voltage and Frequency Scaling (DVFS) are controlled by application-specific software for example. If CDC assertions are available for the block-level environment, then these should be reused at the top-level as discussed in the previous section.

⇨ Reuse CDC assertions in top-level

For a bottom-up CDC flow it is also possible to attach waivers to block-level artifacts and import these into the top-level analysis in order to minimize the information overload often associated with full-chip CDC analysis. Note however that extreme caution is required here since the clock relationships and operational modes may not be fully understood at block-level and can change for new derivatives using the same legacy blocks △.

⇨ Exercise caution with bottom-up CDC waiver reuse

E. Transaction-Level Operation

An additional application of vertical reuse concerns targeting verification components to operate with a different abstraction level for the DUT, for example RTL and transaction-level SystemC models. Typically the same environment can be used to validate cycle-accurate SystemC models and RTL implementations (in fact the SystemC model does not normally have to be cycle-accurate but does require a signal interface for this to work). However, if the verification requirement needs to support transaction-level modeling without a signal interface, including stimulus and monitoring, then the drivers and monitors need to be designed with that in mind △. This would typically involve separating protocol layers from the physical signal interface layer in the agent architecture – when done correctly most of the remaining architecture is unchanged and reusable between the two abstraction levels including sequence libraries, checker operation (especially scoreboards), and functional coverage.

⇨ Supporting multiple abstraction levels requires architectural partitioning

F. Analog-Mixed-Signal

Analog Mixed Signal (AMS) simulation is used to validate correct interoperation of analog and digital sub-components. These simulations are usually done at a block-level due to tool performance issues involved with evaluating continuous analog behavior (as opposed to discrete event-based digital simulation). The AMS simulation environment is also used to validate behavioral models of the analog blocks, which are then used in the higher-level digital simulations. Typically the behavioral model would be instrumented with a comprehensive set of assertions (either actual Analog-SVA, or ad-hoc continuous assertions). Often these assertion-style checks are extremely inefficient and can severely affect performance in the top-level environment if the DUT has a significant analog content △. For that reason the analog assertions also need to be well documented and individually controllable in order to get the most effective usage checks and support top-level verification requirements without compromising overall top-level regression effectiveness.

⇨ Enable only appropriate AMS assertions for top-level

G. Emulation and Acceleration

Many top-level verification strategies make use of hardware-assisted emulation or acceleration to enable much faster "simulation" of scenarios closer to real application speed. These environments synthesize the DUT and part of the testbench (typically the bus-functional signal interfaces as well as both RTL and testbench assertions) into actual hardware (FPGAs in the emulator system). The remainder of the testbench runs in the simulator, which communicates with the emulator box interactively during test execution. Architecture guidelines for vertical reuse in an emulation system are outside the scope of this paper, but some limited information is available in [6].

H. Multi-Language Operation

Another aspect of verification reuse that should be mentioned for completeness is that of multi-language inter-operation. Most simulators support the ability to operate with different functional languages such as e, SystemVerilog and SystemC, however the actual verification components from each language do not communicate in a standard manner and the current situation is far from the desired plug-and-play scenario. Relevant topics include: coordination of phases, scheduling, configuration, stimulus generation, constraint solving, data communication, message maintenance and functional coverage unification. Full analysis of this topic is also outside the scope of this paper, but more information is available in [7].

IX. TUNING REUSE

Taking into account the change of role associated with vertical reuse and the different focus for higher-level verification requirements, many block-level components need to have their behavior modified in order to add value in the top-level environment.

A. Tuning Coverage

Since the top-level environment has different verification requirements, it is unlikely that the block-level functional coverage groups can be reused effectively without modification △. The top-level environment makes use of the block-level decoding capability and coverage collection mechanisms, but may require to modify the actual coverage groups and bins for each cover-point. In addition the top-level environment needs to create additional coverage to measure relationships between different blocks and sub-systems, which can be layered on top of existing block-level mechanisms.

Verification environments using the e language can tune coverage directly using the Aspect Oriented Programming

(AOP) mechanisms to redefine the original classes. SystemVerilog environments need to overload the coverage group definitions typically by using a factory pattern to do class substitution (e.g. as provided by UVM). In either case, functional coverage code is quite different to the normal functional code in the corresponding monitor component and benefits from correct encapsulation in a separate coverage-only class definition. For SystemVerilog it is essential that only the coverage is isolated in this class to enable safer factory class substitution without affecting normal monitor behaviour and minimizing maintenance problems when bugs are fixed in monitor functional code, whereas for e this just represents good coding practice.

⇨ Encapsulate functional coverage constructs in dedicated class

There are several models for doing coverage encapsulation; the most appropriate for a particular application depends on the coverage requirements. Valid options include:

- Instantiate a coverage class in the monitor
- Extend the monitor to add (only) coverage code
- Provide subscriber class coverage component

Instantiating a coverage class inside the monitor is the most flexible solution and can provide transaction coverage as well as cross coverage of other monitor utility fields. Subscriber coverage is limited to transaction content, since it relies on the transaction being published to trigger the coverage collection. Extending the monitor to add coverage can lead to complications if the factory is used to replace the original monitor with a modified class.

B. Filtering Messages

It is absolutely essential that reused block-level environments do not pollute the top-level logfiles with excessive information about block behavior except when configured to do so by the controlling environment △. This is also somehow related to scale; at block-level we probably do not care if we have too much information about this particular block, since that is the focus for the tests, and as a result inappropriate messages with incorrect verbosity settings may be present in the final environment. Examples of such defects include printing messages without verbosity controls (e.g. using the *SystemVerilog $display* function, or an unguarded *item.print*) or printing too much information at too low a verbosity setting (e.g. printing a complete transaction at LOW verbosity) △.

Typically the top-level will disable all messages from block-level components during regression runs and only selectively enable message generation for the block-level environments in order to debug errors. Errors and warnings are not affected by verbosity levels and are always printed. The block-level should stick to a simple verbosity scheme such as that shown below and the supplier should check this as part of the release procedures by actually running the block level regressions under different verbosity settings.

- LOW: messages that happen once per run or reset
- MEDIUM: single-line summary of each transaction
- HIGH: print each transaction or sequence once
- FULL: anything, including method parameter settings

⇨ Ensure message verbosity is appropriate and controllable

C. Controlling Checks

In general we expect to reuse all available checks from the block-level environment in order to validate continued correct operation of blocks embedded in the top-level DUT. However, there are some checks that might cause problems due to the different use case. One good example of this is an end of test check for a scoreboard, which generates a failure if the scoreboard is not empty △. At block-level this is appropriate behavior since the environment is in control of when the test ends, however at top-level different criteria are applied for managing end of test conditions as previously discussed.

In this case the user should provide a separate control flag for the end of test scoreboard check so that the user can disable this check if required. Typically disabled checks should generate a warning instead of an error under these circumstances.

⇨ Allow checks to be disabled where appropriate

X. Retrofitting Reuse

It is not absolutely necessary to validate internal top-level verification requirements by applying a reuse strategy, since we could chose to develop additional monitors, coverage and checks independent of the available block-level components. Reuse does not come for free, but the implementation effort is not huge and is generally much less than the cost of reinventing the code especially in terms of engineering time. In reality the trade-off is generally not a choice between opting to reuse or reinvent, but rather whether or not to validate some of the internal top-level verification requirements at all △, which carries an extremely high price tag in terms of quality and risk.

It would not be far off the mark to assume that if a verification component has not yet been reused in a different context, then it is not yet reusable △. The good news is that retrofitting reuse to an existing environment is a feasible proposition and the effort can be justified by the consumer's project as a cost comparison between modifying the code for reuse or reinventing from scratch. It is very unlikely that the cost of fixing even seriously broken block-level environments will be more than coding a passive equivalent from scratch. In addition, changes to support vertical reuse are generally related to encapsulation and composition, and therefore are relatively low risk in terms of introducing bugs but just show up as compile or elaboration types of errors. The following flow can be used:

- Determine the scope of effort
- Review against reuse guidelines

- Implement all repairs in the supplier code-base
- Validate changes using block-level regression

Experience suggests that determining the scope of effort is usually hampered by out-of-date and inaccurate documentation for the verification environment, or indeed no documentation at all; however, using the *factory.print* mechanism from OVM and UVM or the *eDocs* facility from Specman is usually a good place to start to explore the actual topology of supplied verification environments. It is absolutely essential to have a working block-level regression to facilitate code improvements and prevent accidental damage.

Once the reuse enhancements have been retrofitted to the block-level environment it is time to test it out. Instantiating an extra instance of the block-level environment in the block-level base-test, and configuring it for passive operation as shown in Figure 6, is a great way for suppliers and integrators to weed out most of the reuse defects described in this paper. This extra passive environment shadows the normal active testbench behavior; its only purpose is to validate passive operation of the full block-level environment.

- Instantiate a shadow instance of the environment in block-level base-test
- Configure this shadow instance to passive mode
- Run block-level regressions

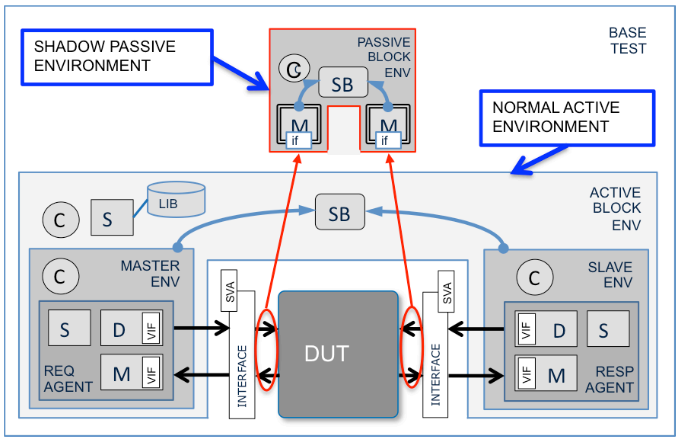

Figure 6. Shadow Passive Environment

⇨ Validate passive operation in block-level environment

Once passive mode has been validated in the block-level environment an instance of the passive environment can then be integrated into the top-level environment and attempt to tune the coverage, control the messages and validate if any checks need additional control flags to disable them.

- Instantiate passive instance in top-level environment
- Validate coverage tuning, check control and message verbosity

Any defects or improvements should be fed back to the supplier's code-base which will ensure much more effective reuse for other consumers at a later date and effectively improve the quality of the block-level verification environment.

XI. Conclusion

Many of the guidelines presented in this paper are standard engineering practice and are repeated here for completeness together with additional guidelines based on real-life experiences. The techniques discussed in this paper have been successfully used and adapted at a number of different verification groups in various clients. It is hoped that combining pragmatic observations with comprehensive reuse guidelines and practical suggestions for retrofitting reuse into real world projects will empower the reader to achieve improved vertical reuse in their own project scenarios.

References

[1] Accellera, Universal Verification Methodology, www.uvmworld.org
[2] R. Wang, "A comprehensive approach to scalable framework for both vertical and horizontal reuse in UVM verification", CDNLive Beijing 2012
[3] M. Litterick, "SVA Encapsulation in UVM – enabling phase and configuration aware assertions", DVCon 2013
[4] J. Montesano, "UVM Sequence Item Based Error Injection", SNUG Ottawa 2012
[5] Mentor, "Low Power Design and Verification Techniques", white paper
[6] Cadence, "Comprehensive UVM/OVM Acceleration", white paper
[7] Accellera, "Verification Intellectual Property (VIP) Recommended Practices", 2009