We’re finally doing it. My wife and I have been in our house for 13½ years and we’re finally redoing our kitchen, which my wife has wanted to do for about 11 years now. This is kind of a big step for us, as those of you who have gone through the exercise can attest, I’m sure. When we had the house built, we chose to limit our budget to keep “expensive” from turning into “exorbitant,” and we’ve been living with formica countertops and a linoleum floor ever since. It’s rather exciting seeing the first steps of converting these into granite and hardwood. As you might imagine, I find myself thinking of how this relates to upgrading a verification methodology because I’m sure that, by now, you know that that’s how my mind works.

When a project like this comes up, my wife and I work well together. I tend to go overboard and want to buy new appliances, add new cabinets and maybe even throw in a new “island” with a sink. Dee, on the other hand, is the more practical one so we’ll be keeping the appliances and cabinets. We are also removing the wallpaper and having the walls painted. So, we’ve got some “legacy IP” we need to work with, but we’ll be able to integrate it all with the new stuff and have a much more useful – and modern – kitchen than we had before. Keep this analogy in mind as you read the following articles.

Our feature article this time comes from my friend Andy Meyer, a fellow Verification Technologist here at Mentor. In “Graph-Based IP Verification in an ARM SoC Environment,” Andy shows how the use of graph-based verification methods not only serve to upgrade the quality of your block-level verification efforts, but also how combining graphs with your UVM environment makes it easier to extend block-level components into an SoC test environment. As you’ll see, this approach lets you reuse your block-level verification IP both to ensure that your design blocks are functioning properly in the SoC and also easily construct a complex SoC verification environment from the block-level components.

“We find myself thinking of how this (kitchen upgrade) relates to upgrading a verification methodology because I’m sure that, by now, you know that that’s how my mind works.”

—Tom Fitzpatrick
Our next article comes from Infineon who share their thoughts on how to “Use Scripting Language in Combination with the Verification Testbench to Define Powerful Test Cases.” This article shows how to use Tcl (or any scripting language) to simplify the task for designers of customizing testbenches. The article discusses both VHDL and OVM testbenches, but the technique is applicable to UVM as well.

When I was in college, I recall my dad, who was an analog designer, telling me that “it’s an analog world” and that my digital exploits would eventually run into this reality. The truth of my dad’s wisdom is evident in our next article, in which “STMicroelectronics Engineers Find Mixed Analog-Digital Verification Success with OVM and Mentor Graphics Questa ADMS Simulator.” In it, you’ll see how STMicro created analog-friendly OVM verification IP that allowed them to take advantage of OVM in generating realistic analog traffic via Questa ADMS.

We next share a couple of “how to” articles on some of the finer points of OVM/UVM testbench design. The first, “Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces,” by my colleague Shashi Bhutada, builds on prior work that explores the use of abstract classes to replace virtual interfaces for connecting your OVM/UVM testbench to the DUT. This particular application shows how this technique allows you to create a single driver class that can use different specializations of an interface without having to rewrite anything. This is followed by our monthly article from the UVM/OVM Online Verification Methodology Cookbook at the Verification Academy. This article shows how to set up a UVM environment for verifying layered protocols.

In the Partners’ Corner for this issue, our friends from Test and Verification Solutions share their experiences in helping Dialog Semiconductor to adopt OVM for their verification solution. “Successful Adoption of OVM and What We Learned Along the Way” will give you some insight into what you may expect if you’re thinking of doing the same.

Last, but not least, my colleague Darron May asks the important question, “Process Management: Are You Driving in the Dark with Faulty Headlights?” This article builds on Darron’s previous article about Questa’s Unified Coverage Database to show how we’ve developed additional technologies in our Verification Management tool to turn the data in the UCDB into valuable information for you to measure and manage your verification process.

I’m concluding this welcome message in my kitchen. The new countertops will be installed in a few days, but the walls already have a coat of primer on them. The process is underway and we’re really looking forward to it. One of the big additions for me is the new flat-screen high-definition television I set up this afternoon. Whether you’re upgrading your kitchen or your verification process, if you have a vision of where you want to be, the key is to do your planning up front, take it one step at a time, and don’t forget to enjoy the experience. The most important thing is to take that first step.

Respectfully submitted,
Tom Fitzpatrick
Editor, Verification Horizons
Now you don’t have to wait for the next printed issue to get the latest.

Hear from the Verification Horizons Team every week at VerificationHorizonsBlog.com
Table of Contents November 2011 Issue

Page 6
Graph-Based IP Verification in an ARM SoC Environment
by Andreas Meyer, Verification Technologist, Mentor Graphics Corporation

Page 10
Use Scripting Language in Combination with the Verification Testbench to Define Powerful Test Cases
by Franz Pammer, Infineon

Page 16
STMicroelectronics Engineers Find Mixed Analog-Digital Verification Success with OVM and Mentor Graphics Questa ADMS Simulator
by Alberto Allara, Matteo Barbati, and Fabio Brognara, STMicroelectronics

Page 19
Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces
by Shashi Bhutada, Mentor Graphics
Layering in UVM

(Extracted from the UVM/OVM Online Methodology Cookbook found on verificationacademy.com)

Successful Adoption of OVM and What We Learned Along the Way

by Mike Bartley, Founder, TVS and Steve Holloway, Senior Verification Engineer, Dialog Semiconductor

Process Management: Are You Driving in the Dark with Faulty Headlights?

by Darron May, Manager of Verification Analysis Solutions, Mentor Graphics Corporation

Verification Horizons is a publication of Mentor Graphics Corporation, all rights reserved.

Editor: Tom Fitzpatrick
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777
Phone: 503-685-7000

To subscribe visit: www.mentor.com/horizons
To view our blog visit: VERIFICATIONHORIZONSBLOG.COM
The use of graph-based verification methods for block designs has been shown to provide a significant time and cost reduction when compared to more traditional constrained random techniques. By filling the random constraint space in a controlled random fashion, the full set of constraints can be filled significantly faster than a random constraint solver will. Possibly more importantly, the ability to describe a graph in canonical form is likely to be easier to modify and maintain over the life of an IP block than constraint statements. Combining graphs with a UVM environment, it is possible to extend block-level verification components into a SoC-level test environment with little additional verification development.

**GRAPHS**

There are a variety of choices for stimulus generation in a verification environment. The possibilities include directed tests, constrained random testing, graphs, or a combination of approaches. Graphs have several advantages that can be important in a reconfigurable verification environment.

A graph defines a single tree of stimulus, where each branch represents one or more legal stimulus sets. The graph itself is a self-contained entity, that may provide an API for higher-level graphs to call, and provides a legal stimulus set for the underlying environment. Because a graph is a self-contained entity, it is possible to define graphs for standard protocols, or specific functions. Once a graph is defined, it can be passed between projects or layered.

A stand-alone IP block tends to have specific functionality that is accessed through one or more standard bus protocols. This can fit nicely with layered stimulus, where a lower-level graph follows the bus protocol, and a higher-level graph provides stimulus for the IP itself. By isolating the protocol knowledge from the IP, the graphs are simpler to develop and maintain, and easier to reuse or acquire from third parties. Because a graph can be expressed as a diagram, it can be significantly easier to understand the stimulus, particularly for team-members who are not familiar with verification.

A graph representation allows the input stimulus space to be expressed as a single tree, with each branch providing one or more legal stimulus sets. Because of this, the full input state space can be enumerated. With a count of all possible legal inputs, input coverage can be detected and reported automatically; in the same way that line coverage is generally provided automatically.

**UVM**

The ability to migrate from an IP-level to SoC-level verification requires an environment designed for reuse. This is the purpose of the UVM. It provides a standard library to develop modular, encapsulated components with a configurable interconnect to tie them together.

The use of UVM agents allows the details of each protocol to be separated from the stimulus source, checkers, and coverage monitors. Agents may be constructed for proprietary busses, or acquired for standard protocols. This allows the verification environment to be constructed quickly from building-block components that are connected through TLM based on the needs of a particular environment.

**STAND-ALONE VERIFICATION ENVIRONMENT**

The UVM and graph-based environment shown in figure 2 provides a stand-alone IP verification environment. Through modularity and standard interfaces, a flexible testbench structure can be built with easy access for visibility, debug, and process monitoring.
However, this environment relies on the processor bus agent (the AXI master agent in figure 2) to directly drive the ARM bus. As a result, this testbench cannot be used unmodified for an SoC integration-level test, since that would result in conflicts on the processor bus.

**SOC VERIFICATION**

Functional verification requirements at the SoC level are changing. Where there was little interaction or resource contention between IP blocks, an interconnect test was generally sufficient for SoC functional verification. As more complex IP blocks are being integrated into SoCs, system level verification is required to measure interactions, resource sharing, utilization, or power concerns. The ability to reuse existing, tested block-level IP verification components can significantly reduce the SoC verification effort.

Part of the goal of a UVM-based approach is to ensure that IP-level verification components have the potential to be instantiated unmodified into an SoC environment. Components such as scoreboards and checkers are good candidates. The IP-level graph stimulus block may also be able to be used unmodified, but the sequence items that were created by the graph can no longer be driven directly onto the processor bus, assuming that the bus only supports one master, and a processor has been instantiated in the SoC environment.

While this changes the connection between the stimulus graph and the IP block, the stimulus itself may not need to change. To connect the graph, the sequence items need to be accessed by the processor, and then driven on the processor bus. Two new components are needed: a software routine running on the processor that reads sequence items and performs the specified operations, and a graph item buffer. Since sequence items tend to contain basic bus operations: read or write to a specific address with specific data, the software routine performs the operation, and then fetches the next sequence item. The item buffer is a verification component that is addressable by the processor that stores sequence items from the graph and delivers them to the processor when accessed, as shown in figure 3.

**Figure 3: Graph Stimulus Driven Through Processor**
The simplest software routine is a loop that waits for an item to be available from the buffer, pulls the item from the graph item buffer, executes it, and then waits for a new item to be available. For more complex operations, this routine may access multiple items. If coordination is required between existing software and stimulus, then the processor may also write commands that provide status or control to a top-level stimulus graph.

The graph item buffer can be a UVM sequencer that accepts items from any number of stimulus graphs, and has a driver to handshake with the software routine. For bi-directional operation, two buffers can be implemented to allow software to control the verification environment.

With this method, all items generated by the stimulus graph will be driven onto the processor bus, with the support of the processor. This approach requires that there is an available address space for the graph item buffer to be accessed by the processor. Because the software will execute several operations to fetch items, the timing between items is likely to change. Back-to-back operations may be difficult to reproduce, and some low-level control that was available through a master agent may be lost.

This approach will allow the IP to be driven from within the SoC integration. The main verification IP: stimulus, check, and coverage may be completely unmodified, while block-level environment is reconfigured. A top layer graph may be used to provide some control and communication between the existing software and the lower-level API stimulus graph.

SYSTEM EXERCISER

When multiple IP blocks have been independently tested and then integrated into the SoC, this method can be used as a system exerciser. For IP blocks that do not share resources, multiple stimulus graphs can be connected directly to the graph item buffer, and each IP-level test can run in parallel as shown in figure 4. Block-level scoreboards and functional coverage are used to measure IP functionality. Additional system-level scoreboards can be added to check overall system operation if desired.

Note that this environment will most likely need to include a top-level graph to coordinate and configure each of the lower-level graphs. This may include waiting until the software is running, coordinating operation for shared resources, and determining overall completion of lower-level graphs.

Using just the IP-level stimulus graphs, this simple system exerciser allows the IP blocks to run in parallel, possibly in parallel with system software. Resource sharing, bus utilization and contention, and simple system performance can be observed.
Where the IP tests are run unmodified, only tests that do not cause interference with other blocks can be run. This will likely result in a subset of the IP tests being used. If system-level tests are needed that stress IP interactions, this environment provides all the building blocks to add a new SoC-level graph that drives multiple IP blocks in a coordinated fashion. The graph item buffer can be used to allow coordination between the system-level graph and software if needed.

CONCLUSION
This approach provides a method for functional verification of an IP in a reusable environment. This allows for a low-cost method for a verification engineer to re-verify an IP after it has been integrated into an SoC. Existing, tested, verification components are used to check that the IP is operating correctly within the target SoC environment.

In an SoC environment that contains multiple independent IP blocks, this approach can be extended to run multiple IP verification suites in parallel, providing a simple system-level exerciser. By adding a top-level stimulus graph for coordination, a more complex SoC level verification environment can be constructed from the block-level components.
This article focuses on providing a methodology to make the process of writing test cases easier for verifying mixed analog digital designs. It explains the advantage of using a scripting language together with a Hardware Description Language (HDL) environment.

It shows a way to give the HDL engineers the possibility to use constrained randomized test cases without having too much knowledge about the implemented testbench.

Two possible scenarios for defining your test cases will be demonstrated. One can be used for the traditional VHDL testbench and another for the SystemVerilog (SV) Open Verification Methodology (OVM) environment. For both cases a scripting language will be used to write test cases.

At the end you will understand that with the mentioned methodology also design engineers will fully benefit of verifying their design.

Please note that within this article Tool command language (TCL) is used as scripting language. The reason for using TCL is, that Questasim is controlled by TCL and also has some more built in TCL commands included. Nevertheless this approach can be used with any scripts which are supported by your simulation environment.

WAYS OF MAKING VERIFICATION EASIER FOR HDL ENGINEERS

Nowadays designers are experienced in optimizing the circuits for power, area and implementing the algorithm effectively. Moreover, they can write very good VHDL testbenches with advanced stimulus generation and file I/O parsers, but they do not have the deep knowledge to develop a constrained randomized configurable testbench environment to verify all corner cases of the design.

Besides the designers you will find the verification engineers who have deep knowledge of modern software design and applying the newest techniques on today's higher verification languages like SV together with the OVM libraries. They are able to generate constrained randomized stimuli's with automatic checkers, scoreboards, analyzing all kinds of coverage models.

In most of the cases it is nearly impossible to distinguish between design and verification engineers. In many projects design engineers will do a basic verification of their module on their own. Reasons can be lack of resources or that the design engineer just wants to make a short rough verification of his designed module by himself. In order to avoid any misunderstanding during specification or implementation of the design, the four eyes principle is recommended. Meaning a different engineer should verify the block generated by another design engineer.

It is important to emphasize that it is really strongly recommended to make the verification process itself easy, reusable and powerful as possible. In this way only a limited amount of verification experts are needed who will build up the testbench and provide a simple interface for defining test cases in all variations. Such interfaces have to be as simple as possible but also as strong as possible. Furthermore, it should still be feasible to vary as much as possible the parameters and use the randomizing feature to catch all the corner cases. One common approach is to use SV OVM language. This methodology already provides easy test case definition by using OVM test files.

“The ovm_test class defines the test scenario for the testbench specified in the test. The test class enables configuration of the testbench and environment classes as well as utilities for command-line test selection. [...] , if you require configuration customization, use the configuration override mechanism provided by the SystemVerilog OVM Class Library. You can provide user-defined sequences in a file or package, which is included or imported by the test class.” (OVM SystemVerilog User Guide, 2009) P. 67/68

If the OVM environment is well structured you can generate all possible test cases within your test file by simply defining some of your parameters (e.g. Use the appropriate sequence, correct register settings). In addition the
modification of such test files can easily be done by the
design engineer. Depending on the setup you can define
pure randomized test cases in your test file or directed
test cases to verify dedicated use cases.

Several limitations should be pointed out with this approach.
First, for every new test case a compilation has to be done
before. Second, reusing a test file for other topics like test
engineering is nearly impossible. In addition, it may take
some time for design engineers to understand the process
of writing test cases by these test files.

With respect to the above mentioned barriers another
approach was developed. The idea is to use a scripting
language in combination with a testbench like SV OVM to
verify the AISC.

TCL VERIFICATION METHODOLOGY

TCL is used by many simulator tools (e.g. Questa, NC-
Sim) for controlling the simulators behavior. Normally TCL
is used in the simulators only to start/stop simulation and
generate some logging messages. Since TCL is a very
powerful language, you can generate whole test cases by
using TCL scripts. To this end the testbench only needs to
provide an interface to the TCL script to allow taking control
of the whole simulation flow. The interface usually will be
built up by the verification expert who has deep knowledge
of the testbench itself. Later, only the TCL script will be
implemented by the design engineers to develop the test
cases.

This methodology has many advantages which were
described here:

- Same TCL file can be used for both VHDL/Verilog
  and SV OVM environment.
- If file is kept simply it can also be used for CV
  (Component Verification) and TE (Test Engineering)
  group as input source.
- Recompile is not needed.
  If your simulator supports TCL you can simply try out
  your test case line by line within your simulator. You do
  not need to restart your simulation and change your test
case file and try it out again.
- Other tasks or scripts can be called during your test
  execution
- Make debugging easier.
  Simply think of analyzing some internal signals
  immediately or do some calculations to check for
  correctness. Temporary log signals to files during
debug session.

DESCRIPTION OF A TCL VHDL TESTBENCH

In traditional black box testing strategies the VHDL
Testbench is surrounding the Design Under Test (DUT).
The testbench controls all activities at the ports of the DUT.
It generates the stimulus and checks all the responds from
the DUT. The stimuli generation can be hard coded in some
VHDL behavioral files, or defined in some ASCII text files.
For reading in text files an extra file parser is needed which
interprets the commands in the text file. The file parser
will usually read in and execute the text file line-by-line. All
the used commands need to be defined in the parser. For
simple test cases this approach may fulfill all your needs. It
can become complicated if you want to use some variables
within your text file. You only have to let the files parsers
know how to interpret variables. It will get more and more
complicated if you want to do some math or use loops,
branches and other conditions within your text file. Thus
your VHDL parser will get more and more complex and
future adoptions will be nearly impossible.

To overcome these disadvantages of using ASCII text file
you can use simple TCL scripts. In scripting language you
already have a huge functionality (e.g. handling of variables,
loading another file, using if/while/loops). There is no need
to take care of these items.

Figure 1 on the following page gives you an overview how
such TCL scripts can be used within your environment.
The TCL Command Interpreter (TCL-CIP) is the interface between the TCL command scripts and the Register Transfer Level (RTL) environment. It contains the libraries & packages for procedure/variable definitions, file I/O functionality etc. Basically it communicates via a well defined VHDL TB peripheral or it may also directly take control of some ports of the DUT or even some internal signals would be possible. The commands and variables which were defined in the CIP can be used in the TCL scripts.

Next is shown a short example how such an approach would look like. First is illustrated the TCL-CIP. For simplified reasons in this place there will not be packages and namespaces used, but also without using these techniques you can fully benefit with this methodology.

The TCL-CIP basically consists of three files.

1.) Global Signal definitions and variables

```tcl
# Physical Environment Signals
set ENV_TEMPERATUR /analog_global_pack/temperature
set ENV_PRESSURE /analog_global_pack/pressure
set VBAT /analog_global_pack/vddd_global
```

2.) Tool specific procedure definition

As an example some procedures for the Questa simulator were shown. Whenever the tool version will be changed only this file needs to be modified.

```tcl
# Clock signals
set CLK_SYS /tb/dcore/clk

# Clock period
set _CLK_SYS “100ns”
```

### Listing 1: TCL Global Signals

2. Tool specific procedure definition

As an example some procedures for the Questa simulator were shown. Whenever the tool version will be changed only this file needs to be modified.

```tcl
#Source file of Questasim specific commands
proc run_proc { runtime } {
   run $runtime
}

# Set an internal signal, by using the force command
proc set_signal_proc {netname value } {
   force –freeze $signal $value 0
}

# Disable the force of an internal signal
proc noset_signal_proc {netname} {
   noforce $signal
}
```

### Listing 2: TCL Questa Procedures
3.) Low level procedure definition

Here all the above mentioned files were loaded together with the procedure definition which can be called from the TCL test cases.

Listing 3 : TCL Basic Procedures

Below shows how such a TCL test case could look like:

```
set log [set_signal $namex [lindex $args 0]]
proc NS {name args} {
    set log "$Now : NS $name $args"
    log_test $tb_g(logfileid) $log
    set nameex [set $name]
    noset_signal $nameex
}

proc log_test {logfileid logtxt} {
    puts $logfileid $logtxt
    echo $logtxt
}

set tb_g(logfileid) [open $logfilename w 0777]
set cmd_file $1.tcl; # This is the name of the test case file
set log_file $1.log; # Logging information will be written in that log file

proc LOG {args} {
    global Now
    global tb_g tb_g
    set args [join $args]
    set log [# $args]
    log_test $tb_g(logfileid) $log
}

proc DL {time args} {
    ... run_proc $time
}

proc SS {name args} {
    global Now
    global tb_g tb_g
    set log "$Now : SS $name $args"
    log_test $tb_g(logfileid) $log
    upvar $name $name
    set nameex [set $name]
    set log [set_signal $nameex [lindex $args 0]]
}
```
WT PD_MODE 1 1ms ; # Wait until PD_MODE goes to ‘1’ for 1ms
CH PD_ADC    ; # Check High
CH PD_CPU
LOG “Test finished”

Listing 4 : TCL Test Case
A startup file is needed which will execute your test case.

# File: startup.do
tcl_cip.tcl 001_BasicTest

Listing 5 : Startup File
The startup file is loaded with the vsim command.

vsim –c top_dut_tb-cfg.vhd –do startup.do

Listing 6: Simulator start command

DESCRIPTION OF A TCL SV OVM TESTBENCH
The same approach as mentioned for the TCL-VHDL testbench can now be used also for the SV-OVM environment. Only for the TCL-CIP interface some adaptations are needed. Additionally within the OVM environment you need to write an extra TCL Interface OVC. This OVC reads via a virtual interface the commands from the TCL-CIP and takes control over other Open Verification Components (OVCs). Depending on the intention you can add a great deal of flexibility to these TCL scripts:

• Define the basic configuration
• Define which parameters to be randomized
• Define border values of randomized parameters
• Start dedicated sequencer
• Force or read back some internal signals
• Do some basic settings of the DUT
• Save Functional Coverage of your SystemVerilog Assertions (SVA)

If a SV OVM Testbench environment already exists new TCL functionality can easily be added even without having a TCL IF OVC. For the beginning it may be enough only to be able to force some internal signals or to do some interim calculation for other scripts.

Below you see an overview of how to connect your TCL script to the OVM environment. The TCL IF – OVC can be modeled in various ways. In our case it is built up like a usual OVC. The monitor is observing the TCL virtual interface for any activities. Whereas the driver will send some responds to this interface. The communication to the other OVCs can now be handled via ports or by using the sequencer as a virtual sequencer and in this way we can take control over all other sequencers within the OVM environment.

CONCLUSION / SUMMARY
Using scripting languages like TCL can give some powerful features for writing test cases. By using a generic TCL interface these scripts can be used for various testbenches. No matter which HDL (Hardware Description Language) is used. Design engineers do not need to have much verification expertise for verifying their designs. Due to easier test case definition they also can do the basic verification of their design on its own. Additionally if these TCL scripts are structured well they also can be used for other topics too. Think of using these scripts as input for component verification to control the whole equipment. Furthermore you can reuse it for test engineering to configure the test setup. Therefore only the TCL-CIP needs to be adapted.

Depending on the available resources and timeframe the TCL functionality can be extended at any time. Extending your TCL functionality will not have too much impact on your existing testbench structure.

What I want to point out is that it is never too late to implement scripting functionality to your test case by using that approach.
LIST OF ABBREVIATIONS

ASIC    Application Specific Integrated Circuit
DUT     Design Under Test
HDL     Hardware Description Language
OVCs    Open Verification Components
OVM     Open Verification Methodology
RTL     Register Transfer Level
SV      SystemVerilog
SVA     SystemVerilog Assertion
TCL     Tool Command Language
TCL-CIP TCL Command Interpreter

REFERENCES

STMicroelectronics Engineers Find Mixed Analog-Digital Verification Success with OVM and Mentor Graphics Questa ADMS Simulator

by Alberto Allara, Matteo Barbati, and Fabio Brognara, STMicroelectronics

DESIGN CHALLENGE

- Verify IP for a R/W channel to be integrated into a STMicroelectronics hard disk component (an SoC)
- Build a verification process from reusable steps

SOLUTION

- Focus on integration into the larger SoC, not just on the IP
- Make maximum use of standards, start with OVM
- Start writing verification components as soon as design work in RTL is underway
- Use Mentor Graphics Questa ADMS tool for analog accuracy and easy integration into digital verification flow

By now it’s a cliché to speak of the rise of digital technology. Follow technology coverage in the media for any length of time and it doesn’t take long to note the tacit assumption that nearly anything with an on/off switch will eventually communicate with the world at-large exclusively in strings of 0s and 1s. Of course, as long as the electronics industry is built on harnessing the laws of physics, the importance of analog signals will never go away. Nature speaks in waveforms, not regimented bitstreams. So the challenge, and one that must be repeatedly solved by those building ever more complex semiconductor devices, is how to verify what’s happening at the analog-digital interface.

One recent solution comes from a trio of engineers at STMicroelectronics in Milan, Italy working on verifying an important bit of intellectual property (IP) for an ST hard disk component. The team’s approach was to combine OVM methodology, Mentor Graphics Questa ADMS simulator and lots of collaboration among engineers at both companies.

A PRIMER ON HARD DRIVES

In general hard drives offer a case study in handling the analog-digital hand off. Consider a stream of binary data to be written to a drive. Those 1s and 0s in the bitstream must be encoded and then output as an analog waveform, a conversion handled by the ST IP. The waveform is imprinted on the magnetic regions of the drive’s spinning disk, or platter, thus storing the binary stream.

To retrieve the data from the drive, the process more or less runs backward. The head of the actuator, the writing and reading element, moves over the disk where the data are stored. The pattern of magnetization on the disk changes the current in the head, and this change can be represented as a waveform. Next comes sampling of this waveform by the ST device and finally a new stream of binary data, with all the algorithms in place to check data integrity and apply data corrections, to be fed back to that emerging digital future we mentioned earlier.

Allara and his colleagues were working to verify IP for a read/write channel to be integrated into the larger ST SoC. Perhaps the biggest challenge was how to meaningfully work across analog and digital domains. Historically, engineers specialize and develop a set of skills relevant to just one domain. Digital verification engineers eschew graphics and spend most of their time writing and compiling vast amounts of code.
By contrast, analog verification engineers look warily at code as most of their work is done via graphical interfaces. Though references to “mixed-mode simulation” abound, Allara believes the phrase generally refers to implementing an analog design digitally, not truly working across domains.

“The bottom line is that this is complex IP,” says Allara, who earned his master’s degree from Politecnico di Milano in 1994 and has been working in the industry since then. “And through the years, the complexity of the digital portion of all of our analog-digital devices has constantly increased, forcing us to look for leading edge verification technology.”

Another challenge was how to approach the verification process in such a way that at least some of the steps and tasks could be subsequently reused. A major issue, one faced by many verification engineers, is how to avoid starting from scratch with each new project. Yes, a custom approach may give a team a chance to demonstrate its technical prowess. However, one-off verification is also undeniably a huge sunk cost, particularly as verification complexity skyrockets. Recall that verification complexity increases at some multiple of the rate of increase of the number of gates, a troubling maxim given that gate counts already number in the hundreds of millions or more.

**BEFORE CODING, THINK FIRST**

The first step for the ST team had nothing to do with coding or even formal planning. Rather, they worked to begin thinking differently about the verification process. For example, rather than focus exclusively on the read/write IP, they instead chose to consider how this IP could be integrated into the SoC. The team realized that given the nuanced relationships among the various building blocks that make up the SoC, there is more to verification than just looking at the various signals expected to pass through the IP they were assigned to verify.

Allara and his colleagues also were determined to make maximum use of standards by leaning heavily on Open Verification Methodology, or OVM. (As an aside, it’s worth noting that since this ST project was done, Accellera ratified version 1.0 of Universal Verification Methodology, or UVM, a direct derivative of OVM; see http://bit.ly/eK4oUI (PDF). Allara, who with his nearly 20 years of experience counts as a seasoned pro in IP and SoC verification, says the rise of standards is the biggest change he’s experienced in how he approaches his work. It wasn’t long ago that engineers wrestled with HDL and other description languages to build and verify RTL designs. Companies cobbled together their own techniques — in fact this state of affairs still exists today — using everything from C code to SPICE simulation.

Standards came into play immediately when the actual verification work began. A second ST team working on the processor’s analog front end used VHDL AMS to build a model of the analog domain which was provided to Allara and his colleagues, Fabio Brognara and Matteo Barbati. The model allowed Allara to close the loop with the RTL describing the digital channel front end and was relatively easy to integrate with other languages, particularly VHDL AMS.

Allara says his approach was to start writing verification components as soon as his ST designer colleagues started developing the digital design in RTL. Accordingly, the first phase in Allara’s verification effort was to verify the digital design in greater and greater detail as more of the RTL took shape. Later, when the RTL was finished and Allara had received the model of the analog front end, his team moved to true mixed mode simulations, while reusing much of their verification infrastructure.

The OVM methodology requires specifying verification components for each interface of the device. These components are coordinated via a multi-channel sequencer operating at a higher, more abstracted layer of the OVM environment. The components developed by Allara’s team allowed them to program the registers of the read/write IP. For the “read” portion of the IP, they created a component that generated a model stream of bits similar to that which might be read from a hard drive’s spinning disc. Another verification component extracted the information after the stream was processed to compare the output to the expected result. The team developed a similar set of components for the “write” portion of the IP.
The simulated bitstream was produced by a pattern generator written in C code and embedded in a verification component. To reuse their environment for mixed mode simulation, the team created a layer that could convert the bitstream into a phased series of various input signals that roughly approximated the expected input to the IP's analog front end. This simulated conversion from analog to digital signal was done via VHDL AMS.

Using the methodology they built, the team was able to find bugs in the analog domain that the ST analog designers themselves missed. The reason, Allara says, is that his team’s approach was able to feed the analog front end a pattern very similar to one that might actually be read from a disc. By contrast, in analog-only verification, the simulated patterns are often simple and fairly symmetrical, two characteristics that don’t describe the complex, sprawling and often asymmetrical waveforms produced by the magnetic fields.

**FINDING BUGS IS THE FINISH LINE OF VERIFICATION**

In the end, the goal of verification at the analog-digital interface is to make a discovery, or more precisely, to find problems with the design before it’s built and those problems are found by customers. Any methodology that seems to ratify designs as mostly okay is more apt to be evidence of faulty verification than proof of a good design, which is why Allara’s success in finding bugs that others had overlooked is perhaps the best evidence of his team’s success.
INTRODUCTION
The SystemVerilog language is increasing in adoption for advanced verification techniques. This enables the creation of dynamic test environments using SystemVerilog classes among other things. The SystemVerilog virtual interface has been the primary method for connecting class-based SystemVerilog testbenches with a VHDL or Verilog DUT, but this construct has certain limitations, especially when the interface is parameterized. In this article we will discuss some of these limitations and demonstrate an alternative approach called as the Polymorphic Interface. The recommended approach can also work generically wherever one uses virtual interfaces and not just parameterized interfaces.

For the SystemVerilog testbench, we will use OVM infrastructure, but this same discussion applies to UVM testbenches. This article assumes SystemVerilog OVM/UVM class and interface syntactical understanding.

SYSTEMVERILOG OVM/UVM TESTBENCH
Design demands have grown exponentially over time as our fabrication capabilities and geometry sizes have dramatically improved. Verification methods can be advanced with the SystemVerilog language, which provides a whole gamut of methods. SystemVerilog amalgamates advanced methods including Object-Oriented Programming (OOP) by extending a well-known design language: Verilog. Since it is an IEEE standard and packaged with RTL simulators, advanced verification is now easily accessible.

The SystemVerilog language can be divided into five parts. One is an enhancement of Verilog RTL design constructs that captures design intent more accurately. The other four parts are constructs meant to deliver the advanced verification features: assertions; object oriented programming (OOP) or the ‘class’ syntax; constrained-random stimulus or automatic stimulus generation; and functional coverage.

The next big development in this area was the UVM (Universal Verification Methodology) Library, which is an Accellera standard SystemVerilog code-base jointly developed by multiple EDA Tool vendors and industry leaders. This library simplifies adoption of the object-oriented methods that are part of SystemVerilog. Simply put, UVM tames the SystemVerilog behemoth. UVM is largely based on its predecessor OVM (Open Verification Methodology) Library.

In a traditional testbench, the user is required to use a hardware description language (HDL) to also define the testbench. Unfortunately, the HDL only allowed creating static design elements. This has many disadvantages which include: testbench interfering with design interactions causing race conditions, static testbench prevents plug-n-play, verification reuse meant cut-n-paste of the test code, limited coverage of design state space and no metrics of which states were covered. In an advanced OVM/UVM testbench a user is required to use ‘class’ syntax and create dynamic testbench components which do not rely on pin-level interactions on the DUT but are at an abstraction above the pin-level interaction called as transaction-level (shown in Figure 1).

![Figure 1: Transaction-level versus Pin-level activity](image)

A typical OVM/UVM testbench (See figure 2) contains a stimulus generator that generates transaction, a driver that converts transactions into pin-level activity, a monitor that transforms pin-level activity back into transaction objects, a coverage collector to measure the transaction activity on the pins, a predictor that predicts what the expected output transaction should be based on the stimulus applied, and a scoreboard that compares observed output transaction with the expected output.

Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces
by Shashi Bhutada, Mentor Graphics
the expected transaction. This approach has the following advantages: test does not interfere with design under test, dynamic test components are plug-n-play, verification reuse is possible for a given standard bus interface protocol, transaction-level code runs typically faster than pin-level interaction, larger state space can be covered when using constrained random transaction generation, and real metrics can be obtained by observing transactions around the testbench.

- Directly setting of the virtual interface handle where needed via helper function in the class.
- Package-based global storage for the virtual interface handle.
- The preferred approach is by saving virtual interface handle into the OVM/UVVM configuration database. In OVM, this requires an extra wrapper class to hold the virtual interface.

**SYSTEMVERILOG INTERFACES AND VIRTUAL INTERFACES:**

In the OVM/UVVM dynamic transaction-level testbench one needs to ultimately connect with the DUT at different points. The driver needs access to design pins to generate the right pin-activity based on the transaction. Similarly, the monitor needs access to design pins to observe pin-activity. Dynamic constructs cannot access static design items directly.

SystemVerilog provides the ‘interface’ construct to encapsulate a set of pins on a DUT. The interfaces typically can be design dependent but often they encapsulate industry standard bus protocols for maximum reuse. A SystemVerilog interface is a static construct and resides on the static side of the testbench.

The SystemVerilog language provides the virtual interface construct that allows one to pass the ‘interface’ handle from the static side to the dynamic side. This connects the class-based dynamic testbench to the static HDL design. Following are the different approaches to pass the interface from static side to the dynamic side:

- Directly setting of the virtual interface handle where needed via helper function in the class.
- Package-based global storage for the virtual interface handle.
- The preferred approach is by saving virtual interface handle into the OVM/UVVM configuration database. In OVM, this requires an extra wrapper class to hold the virtual interface.

**LIMITATIONS WITH VIRTUAL INTERFACES:**

The SystemVerilog virtual interface based approach to connect the OVM testbench and the DUT involves passing an interface handle from the top-level module to the OVM testbench. In most of the use cases virtual interfaces work just fine. As we will see in the example below, virtual interfaces become difficult to use when it comes to parameterized interfaces.

To demonstrate the problem, let’s see the example code below which tries to use virtual interfaces:

```verilog
class vif_wrapper #(type T=int) extends ovm_object;
    T m;
    function new(string name="");
        super.new(name);
        endfunction
    endclass

interface bus_intf #( int aw, int dw, int tp = 8,
                    string proc="ppc") (
    //port declarations …);

module top
    parameter T1=10;
    parameter T2=T2*2;
    parameter T3=T3*5;

    bus_intf #( .aw (8), dw (12), .tp (T2), .proc("arm") )
        intf_arm(…);
    bus_intf #( .aw (16), dw (32), .tp (T3) .proc("nios") )
        intf_nios(…);
```

---

**Figure 2: Advanced Verification Testbench**

![Advanced Verification Testbench Diagram]
One must define unique and correct types

typedef virtual bus_intf
#( .aw (8), dw (12), .tp (T2), .proc(“arm”) )
    vif_arm_t;
typedef virtual bus_intf
#( .aw (16), dw (32), .tp (T3), .proc(“nios”) )
    vif_nios_t;

// If using a config object method then one can
do the following
vif_wrapper #(vif_arm_t) vif_arm;
vif_wrapper #(vif_nios_t) vif_nios;

initial
begin
    vif_arm=new();
    vif_nios=new();
    vif_arm.m = intf_arm;  //Pass the actual
                        //handle interface
    vif_nios.m = intf_nios;  //Pass the actual
                        //handle interface
    set_config_object(“*”,“wrapper”,wrapper,0);
    run_test(“test”);
end
endmodule

The issue is compounded if we want to define a generic agent based on the parameterized interface. Each agent will be instantiated to communicate with two similar but differently parameterized interfaces. As you see above the interface specialization has to be inside the top due to the dependencies on the top-level parameters. In such an agent, what is the virtual interface handle inside the driver going to look like? It has to be specific to the specialized-type of interface it will be communicating through.

Here’s the pseudo code:

class driver #(type T = int) extends ovm_driver#(T);
    `ovm_component_param_utils(driver #(REQ))
    transaction item;
    virtual intf (...) vif;
endclass

POLYMORPHIC INTERFACE:
The Polymorphic Interface approach provides a very elegant solution for sharing parameterized interfaces. This approach is also known as “Abstract/Concrete Class” approach or “Two Kingdoms” approach. This approach involves defining a pure virtual (i.e. “abstract”) class and then making it concrete inside the scope of the SystemVerilog interface (or module). The abstract class in essence provides the “interface” like construct in Java, and each concrete class inside the SystemVerilog interface (or module) provides functionality for the abstract functions in the context of the SystemVerilog interface (or module).

One could have multiple levels of inheritance hierarchy each attached to a different type of SystemVerilog Interface (or module). The OVM testbench will use the abstract class handle and, via polymorphism, the function calls get forwarded to the concrete class functions inside the right interface - hence the name “polymorphic interface.” As we have seen, the virtual interface is not polymorphic. In the example below, the polymorphic interface approach elegantly solves the issue with parameterized interfaces.

CREATING POLYMORPHIC INTERFACE:
Following are the steps involved in creating a Polymorphic Interface:

Step 1: Create an abstract or virtual class:

virtual class abstract_c extends ovm_object;
    function new(string name);
        super.new(name);
    endfunction
    pure virtual task wr_cycle(int addr, int data);
    pure virtual task rd_cycle(int addr, ref int data);
endclass

package abs_pkg;
`include “ovm_macros.svh”
import ovm_pkg::*;
`include “abstract_c.svh”
endpackage
Step 2: Define interface (or module) that will contain the following things apart from regular port/parameter/modport/task/function declarations:

- a **concrete class** that derives from the abstract class above
- a concrete class handle
- a utility function to create the concrete class instance

```java
interface bus_intf #( int aw=8, int dw=8, int tp=8,
  string proc="ppc")
import abs_pkg::*;
class concrete_c extends abstract_c;
function new(string name="");
  super.new(name);
endfunction

task wr_cycle(int addr, int data);
  $display("wr_cycle: accessing %s interface pins", proc);
endtask

task rd_cycle(int addr, ref int data);
  $display("rd_cycle: accessing %s interface pins", proc);
endtask

class driver #(type T = int, string proc="ppc") extends ovm_driver#(T);
  `ovm_component_param_utils(driver #(req,proc))
  ovm_sequence_item item;

  abstract_c api;
  function new(string name, ovm_component p);
    super.new(name, p);
  endfunction
endinterface
```

Step 3: Use the interface in the top level testbench module to connect to the DUT just like you would normally use. We then add one extra step to **create the concrete class** (and add it as a config object) using the utility function we created earlier inside the interface as shown below:

```java
import ovm_pkg::*;
import test_pkg::*;
module top;
parameter T1=10;
parameter T2=T1*2;
parameter T3=T1*5;
business #( .aw (8), .dw(12), .tp(T2), .proc("arm") )
intf_arm();
business #( .aw (16), .dw(32), .tp(T3), .proc("nios") )
intf_nios();

initial
begin
  set_config_object("*","intf_arm",
    intf_arm.get_concrete_c_inst(),0);
  set_config_object("*","intf_nios",
    intf_nios.get_concrete_c_inst(),0);
  run_test("test");
end
endmodule
```

Step 4: In the OVM testbench the **driver will call utility functions** (like `wr_cycle`) **provided in concrete_c via abstract_c** (polymorphism) to drive the DUT pins:

- declare a handle to the abstract class
- get the configuration object to assign the concrete object to the abstract handle
- inside the run function call the utility functions (such as `wr_cycle`)

```java
class driver #(type T = int, string proc="ppc") extends ovm_driver#(T);
  `ovm_component_param_utils(driver #(req,proc))
  ovm_sequence_item item;

  abstract_c api;
  function new(string name, ovm_component p);
    super.new(name, p);
  endfunction
endclass
```
Step 5: There are no changes in how one creates the OVM Environment and Test. One can create the agent with sequencer, driver and monitor. For the driver, specifically, use the one created earlier in step 4 that leverages the polymorphic interface. The agent’s driver will now be customized with a different parameter which then customizes the underlying interface. Here’s the example:

```verilog
package comp_pkg;
    `include "ovm_macros.svh"
import ovm_pkg::*;
import abs_pkg::*;
    `include "driver.svh"

class agent #(string proc="ppc") extends ovm_agent;
    `ovm_component_param_utils(agent#(proc))
    driver #(ovm_sequence_item, proc) drv;
    ovm_sequencer #(ovm_sequence_item) sqr;

function new(string name, ovm_component p);
    super.new(name, p);
endfunction

function void build();
    super.build();
    drv = driver#(ovm_sequence_item, proc)
        ::type_id::create("drv", this);
    sqr = new("sqr", this);
endfunction

function void connect();
    drv.seq_item_port.connect(sqr.seq_item_export);
    drv.rsp_port.connect(sqr.rsp_export);
endfunction
endclass

class env extends ovm_env;
    `ovm_component_utils(env)
    agent #("arm") arm_agent;
    agent #("nios") nios_agent;

function new(string name, ovm_component p);
    super.new(name, p);
endfunction

function void build();
    super.build();
    arm_agent = agent#("arm"):type_id::create
        ("arm_agent", this);
    nios_agent = agent#("nios"):type_id::create
        ("nios_agent", this);
endfunction
endclass
endpackage

package test_pkg;
import ovm_pkg::*;
import comp_pkg::*;
    `include "ovm_macros.svh"

class test extends ovm_test;
    `ovm_component_utils(test)
    env e;
endclass
```
CONCLUSION:
Virtual Interfaces themselves do not have a notion of polymorphism. The virtual interface is the standard documented mechanism for communication between dynamic (OVM/UVM) testbench and the static design under test. In OVM, the preferred interface-based approach is to wrap the virtual interface in an object and save the wrapper into the OVM/UVM configuration database. In UVM, the virtual interface can be passed into the uvm_config_db directly without a wrapper class.

As shown in this paper, there are situations, such as parameterized interfaces, when virtual interface approach doesn’t work due to limitations in the interface syntax itself. The Polymorphic Interface (Abstract/concrete class) approach has the following advantages over virtual interface:

- It provides a cleaner mechanism to deal with parameterized interfaces, as shown here.
- It also provides polymorphism with interfaces, hence the name Polymorphic Interface.
- It gets rid of the extra virtual interface wrapper completely, when using the preferred the configuration database, since the approach uses SystemVerilog class syntax. The concrete class can directly be made available via OVM/UVM configuration database to the driver/monitor or any other transactors as needed.

REFERENCE:
Layering in UVM

*Extracted from the UVM/OVM Online Methodology Cookbook found on verificationacademy.com*

Many protocols have a hierarchical definition. For example, PCI express, USB 3.0, and MIPI LLI all have a Transaction Layer, a Transport Layer, and a Physical Layer. Sometimes we want to create a protocol independent layer on top of a standard protocol so that we can create protocol independent verification components (for example TLM 2.0 GP over AMBA AHB). All these cases require that we deconstruct sequence items of the higher level protocol into sequence items of the lower level protocol in order to stimulate the bus and that we reconstruct higher level sequence items from the lower level sequence items in the analysis datapath.

**THE ARCHITECTURE OF A LAYERING**

In order to do this we construct a layering component. A layering component:

- Must include a **child sequencer** for each non-leaf level in the layering.
- Must create, connect and start **translator sequence** for each non leaf level in the layering.
- Must have a handle to the leaf **level protocol agent**. This protocol agent may be a **child** of the layering or **external** to it.
- May include a **reconstruction monitor** for each non leaf level in the layering.
- Should create and connect external **analysis ports** for each monitor contained within the layering.
- Will usually have a configuration object associated with it that contains the configuration objects of all the components contained within it.

**CHILD SEQUENCERS**

A child sequencer in a layering is simply the usual sequencer for that protocol. Very often an appropriately parameterized uvm_sequencer is quite sufficient. If the higher level protocol has been modeled as a **protocol UVC**, then the layering should instantiate an instance of the sequencer used by the agent for that protocol so that sequences can be targeted either at the bus agent or the layering.

For example, the ABC layering below has an A_sequencer and a B_sequencer.

class ABC_layering extends uvm_subscriber #(C_item);
'`uvm_component_utils(ABC_layering)

... 
A_sequencer a_sequencer;
B_sequencer b_sequencer;
...
C_agent c_agent;

function new(string name, uvm_component parent=null);
super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
a_sequencer = A_sequencer::type_id::create
("a_sequencer",this);
b_sequencer = B_sequencer::type_id::create
("b_sequencer",this);
endfunction
...
endclass
TRANSLATOR SEQUENCES
A sequence which translates from upstream items to downstream items runs on the downstream sequencer but has a reference to the upstream sequencer. It directly references the upstream sequencer to call get_next_item and item_done to get upstream items and tell the upstream sequencer that it has finished processing the upstream sequence item. Between get_next_item and item_done it sends data to and gets data from the lower level sequencer by calling start_item and finish_item. A simple BtoC translator sequence is shown below:

```
class BtoC_seq extends uvm_sequence #(C_item);
  `uvm_object_utils(BtoC_seq);

  function new(string name="");
    super.new(name);
  endfunction

  uvm_sequencer #(B_item) up_sequencer;

  virtual task body();
    B_item b;
    C_item c;
    int i;
    forever begin
      up_sequencer.get_next_item(b);
      foreach (b.fb[i]) begin
        c = C_item::type_id::create();
        start_item(c);
        c.fc = b.fb[i];
        finish_item(c);
      end
      up_sequencer.item_done();
    end
  endtask

endclass
```

The run phase is responsible for creating the translator sequences, connecting them to their upstream sequencers, and starting them on their downstream sequencers:

```
virtual task run_phase(uvm_phase phase);
  AtoB_seq a2b_seq;
  BtoC_seq b2c_seq;

  a2b_seq = AtoB_seq::type_id::create("a2b_seq");
  b2c_seq = BtoC_seq::type_id::create("b2c_seq");

  // connect translation sequences to their respective upstream sequencers
  a2b_seq.up_sequencer = a_sequencer;
  b2c_seq.up_sequencer = b_sequencer;

  // start the translation sequences
  fork
    a2b_seq.start(b_sequencer);
    b2c_seq.start(c_agent.c_sequencer);
  join_none
endtask
```

THE PROTOCOL AGENT
Every layering must have a handle to the leaf level protocol agent. If we are delivering verification IP for a layered protocol, it usually makes sense to deliver the layering with an internal protocol agent. On the other hand, we may be adding a layering for use with a shrink wrapped protocol agent instantiated elsewhere in the testbench. Under these circumstances we will want the leaf level protocol agent to be outside the layering.

**Internal Protocol Agent**
In the case of an internal protocol agent, the layering component inherits from uvm_component and creates a child layering agent:
class ABC_layering extends uvm_component;
    `uvm_component_utils( ABC_layering )

    ... A_sequencer a_sequencer;
    B_sequencer b_sequencer;
    ... C_agent c_agent;

    function new(string name, uvm_component parent=null);
        super.new(name, parent);
    endfunction

    function void build_phase(uvm_phase phase);
        a_sequencer = A_sequencer::type_id::create
                        ("a_sequencer",this);
        b_sequencer = B_sequencer::type_id::create
                        ("b_sequencer",this);
        c_agent = C_agent::type_id::create("c
                        sequencer",this);
    endfunction

endclass

Really, there is nothing special in the analysis path of a layering. For each layer in the monitoring we provide a reconstruction monitor which assembles high level items from low level items. These reconstruction monitors have an analysis export which is connected to the analysis ports of the lower level monitor and an analysis port. This analysis port is connected to an external analysis port and the analysis export of the upstream monitor if there is one.

An outline of a reconstruction monitor is shown below:

class C2B_monitor extends uvm_subscriber #(C_item);
    // provides an analysis export of type C_item
    `uvm_component_utils(C2B_monitor)

    uvm_analysis_port #(B_item) ap;
    // declarations omitted ...

    function new(string name, uvm_component parent);
        super.new(name, parent);
        ap = new("ap",this);
    endfunction

    function void write(C_item t);
        // reconstruction code omitted ... 
        ap.write( b_out );
    endfunction

endclass

External Protocol Agent

In the case of an external protocol agent, the layering is a subscriber parameterized on the leaf level sequence item and the agent is not constructed inside the layering. The code introduced above shows the code for an external agent.

THE ANALYSIS PATH

---

www.mentor.com
The reconstruction monitors are connected up in the normal way:

```verilog
class ABC_layering extends uvm_subscriber #
    ( C_item );
`uvm_component_utils( ABC_layering )

uvm_analysis_port #( A_item ) ap;
A_sequencer a_sequencer;
B_sequencer b_sequencer;
C2B_monitor c2b_mon;
B2A_monitor b2a_mon;
C_agent c_agent;

function new(string name, uvm_component parent=null);
    super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
    a_sequencer = A_sequencer::type_id::create
                  ("a_sequencer",this);
    b_sequencer = B_sequencer::type_id::create
                  ("b_sequencer",this);
    c2b_mon = C2B_monitor::type_id::create
                  ("c2b_mon",this);
    b2a_mon = B2A_monitor::type_id::create
                  ("b2a_mon",this);
    ap = new("ap", this);
endfunction

function void connect_phase(uvm_phase phase);
    c2b_mon.ap.connect(b2a_mon.analysis_export);
    b2a_mon.ap.connect( ap );
endfunction
...
endclass
```

For more details or to download the examples, go to http://verificationacademy.com/uvm-ovm/Sequences/Layering
**ABSTRACT**

OVM has gained a lot of momentum in the market to become the dominant verification “methodology” and the indications are that UVM will gain even greater adoption. OVM is built around SystemVerilog and provide libraries that allow the user to build advanced verification test benches more quickly. There is extensive documentation, training and support for how to best develop such test benches and to encourage test bench re-use. However, there is less advice on how to adapt your verification processes on your project to best use OVM and even less advice on how to do this for company wide adoption.

In this article we discuss the experiences of the authors of a company-wide adoption of OVM. We consider the original motivations for that adoption and the expected benefits, and the actual results achieved and problems that have been overcome. The aim of the article is to give advice to other companies considering adopting OVM.

**WHY ADOPT A METHODOLOGY? AND WHY OVM?**

Figure 1 shows a brief of industry test bench methodologies and goes beyond OVM to UVM.

Dialog had already decided that they wanted to improve their verification process by moving it from a directed testing approach to use pseudo-random verification. The main question was the most effective way of making such a transition and it was clear that adopting an existing methodology that supported a pseudo-random verification was the best way forward.

OVM is currently the leading industry pseudo-random verification methodology and has a well planned and supported roadmap to UVM (the Universal Verification Methodology) which is supported by Cadence, Mentor Graphics and Synopsys. Thus the decision by Dialog to adopt OVM was reasonably obvious.

Figure 2 (on the following page) “OVM Verification Hierarchy” shows how OVM is made up of a class library and methodology. The OVM class library is built on SystemVerilog which has been standardised (IEEE 1800). The class library is freely distributed and can be run across multiple industry simulators from different vendors. The OVM methodology is then supported by the OVM class libraries. The methodology is also supported by other languages: SystemC® (IEEE 1666), and e (IEEE 1647) languages.
The fact that the language, library and methodology have been standardised and are supported by multiple companies and vendors brings significant additional advantages to Dialog:

- **Availability of training & support:** As the adoption of a common methodology gains widespread industry acceptance then the market for services and products based on the standard increases. Thus vendors are able to start providing those services and products.

- **Compatibility & availability of 3rd party IP:** Similar to training and support, the availability of 3rd party IP increases with growing adoption of a methodology. In particular, vendors are able to produce and sell their verification IP (VIP) to a larger market. It should also be easier for the user to incorporate that VIP into their verification environment if that environment and the VIP are compatible through the standard – a “Write once, run everywhere?” concept. Note also that the OVM standard supports the use of plug-and-play verification IP (VIP) written in SystemVerilog (IEEE 1800), SystemC® (IEEE 1666), and e (IEEE 1647) languages.

- **Cross-tool compatibility:** The standardisation process should ensure that tools from different vendors support both the methodology and the underlying class libraries making it easier for users to switch tools and thus vendors more easily.

We can all point to differences in the ways that simulators interpret the semantics of a particular language. Indeed, in many places the language standard does not completely cover the implementation and the tool vendor is left to make their own decision – race conditions being an obvious example. However, these issues are usually minor enough not to block a switch to an alternative supplier. The ability to switch supplier is vitally important commercially when negotiating with tool vendors. It also means that the user has an alternative should their current supplier cut support for their simulator.

- **Reduced ramp up time for subcontractors & new starters:** Skilled verification resources remain in high demand and short supply. Use of an industry standard methodology both increases the probability of finding resource with the appropriate skills and knowledge, and reduces the time to ramp those people in your particular verification environment.

**ADVANTAGES OF ADOPTING EXTERNAL VIP**

In addition to the above reasons from Dialog, TVS has noticed additional advantages that other clients have seen from the adoption of OVM. The first of these is in the use of 3rd party OVM VIP:

- The ability to more rapidly develop complex test benches
- Independent interpretation (of protocols) leading to higher quality verification of interfaces
- Ensure higher coverage of protocols through the cover points and assertions built into the VIP and mapped to the protocol specification (assuming your VIP supplier does this)
- Easier re-use of VIP between OVM-compliant projects and test benches, and across hierarchies in the same project (e.g. from block to SoC – the VIP is used more for traffic generation in the latter given the interface is already proven)
- Leverage knowledge and experience from others
- Huge potential ROI (return on investment) from the VIP

**ROLLING OUT OVM ACROSS**

Figure 3 “Dialog OVM test bench structure” shows a typical OVM-style test bench and this is the style of test bench that Dialog chose to adopt. The agents shown could be either internally developed or external VIP (such as the VIP from TVS).
In the rest of this section we look specifically at the methods that Dialog used to roll out adoption of OVM across their organisation. It should be noted that this was a multisite roll out. [1]

The following is the list of steps deployed by Dialog.

1. External training courses & workshops: As discussed above the fact that Dialog were adopting an industry standard methodology meant that there were external providers of training and workshops.

2. Internal seminars to consolidate knowledge & discuss best practices: Dialog were of course not starting from scratch. They had engineers with experiences from: previous jobs; attending industry events; reading online, press and papers, etc; external training and workshops. Dialog found it useful to consolidate that knowledge via internal seminars where they also discussed best practise.

3. Set of golden rules for implementing OVM environments: Distilled from the internal seminars and discussions, and from externally available materials.

4. Internal Wiki for knowledge sharing: This allowed Dialog to have a “living” knowledge base where engineers could add their growing knowledge and practical experiences of using OVM.

5. Using external resources on OVM: For example, the Mentor Graphics Verification Academy and TVS white papers on deploying OVM VIP.

6. Set of “template” components: These templates were a good way to encapsulate knowledge in a practical format and were distributed as an example environment
   a) Demonstrate best practise
   b) Form the basis of module environments
   c) Scripts automated generation of environments from template code

7. Code review sessions: These were used to ensure that engineers started on the right track and give them feedback on their coding style. These were used to continuously improve the code being written and the environments being developed

8. Library OVC components created: An internal library was created containing components for potential re-use.

9. Standard VIP procured: For example, Dialog procured OVM compliant VIP from TVS.
PROBLEMS IN EXECUTION

Dialog overcame the following problems in their execution of OVM adoption.

1. Problems with implementing checking at the appropriate level of abstraction
   a) Data & transformation checking in scoreboard
   b) Timing and protocol checking with SVA

2. Problems with expending effort on coverage closure metrics
   a) Appropriate priority setting on coverage goals
   b) Keeping vplan “live” – linking coverage model and reviewing coverage model
   c) If functional coverage is high, but code coverage is low then suspect the completeness of the model...
   d) Achievable coverage goals (e.g. defining appropriate coverage buckets for a 32-bit real-time counters)

3. Problems with appropriate constraints on stimulus.
   For example,
   a) Don’t randomly toggle test mode signals in functional tests
   b) Use highly-directed sequences required to arrive at system state of interest and then use CRV
   Domain specific knowledge is usually required in these circumstances.

4. RTL-centric engineers found it hard to grasp ideas from the software world
   a) Encapsulation
   b) Inheritance
   c) Polymorphism
   d) The OVM factory & overrides
   TVS have encountered this situation in numerous organisations. OVM often requires a software mindset which RTL engineers often find it hard to adopt.

5. Initially the OVM reuse paradigm was not fully leveraged. TVS have found that other companies overcome this through the use of a central VIP library and “librarian” who is able to actively market the VIP across the company and help engineers in the reuse of the VIP.

6. Reuse from module level to chip-level is non-trivial
   a) Many issues were found when vertical reuse was done (requiring rewrite of module environments)
   Dialog sees this as a learning process and is now achieving significant vertical reuse.

7. SV is a “rich” language, so there are many ways to code functionality. For example, it is not always obvious how to...

1. Reusable Verification Components encapsulate a number of:
   • Agents
   • Scoreboards
   • Checks
   • Coverage

2. Verification Components shall be scalable for module, subsystem or chip level

3. Verification Components shall be located in SystemVerilog packages

4. Agents shall follow OVM guidelines

5. DUT interaction should not be done outside the agent

6. Synchronization should be handled with ovm_event_pool

7. Assertion-based checks should be placed in interfaces

8. TLM should be used wherever possible to communicate data

9. Configuration should be stored in a config component

10. Class and Module based code should only communicate via interfaces

Figure 4: Basic Dialog OVM Guidelines

a) make appropriate choice of language constructs
b) write the interface between module-based code and class-based code

8. OVM is merely a library of parts and a user guide so a strong adherence to the methodology is needed. Dialog resisted the temptation to create Dialog-specific class libraries, macros or an OVM wrapper layer. This may ease the introduction of OVM but ultimately this approach is potentially restrictive and requires more ramp-up and longer-term maintenance. Dialog thus had a strong drive to stick with “vanilla” OVM and the approach suggested in the reference guide examples.

9. Dialog had an expectation that OVM would reduce their verification effort whereas the reality was that the effort increases (at least until reuse is leveraged and engineers become more familiar and experienced in its usage). The initial perception of the verification engineers was “why do I need to write so much code”? Dialog thus sees the main initial advantage of OVM as improved quality in verification. This will be of high
importance to Dialog going forward as the complexity of their designs increases. Dialog also expects to improve efficiency within a relatively short time period.

THE MAIN RESULTS ACHIEVED
In this section we first consider the main results achieved by Dialog in their rollout:

• The engineers were able to get running quickly with random stimulus generation and transaction coverage more quickly when they were supplied with reusable templates.
• Dialog did indeed reap all of the benefits normally associated with a Constrained Random Verification (CRV) methodology: high rates of bug discovery; easier tracking of real progress; managed closure driven by a metric-driven approach.
• Dialog was able to add coverage collected on directed tests (in passive mode) to that achieved via CRV. This was of high importance as it meant that Dialog were able to exploit the value contained in their legacy tests within their OVM environment.

Dialog now has a well-defined approach to exploiting OVM and a roadmap for its deployment on future projects.

RECOMMENDATIONS
From their experience, TVS and Dialog would recommend the adoption of the vanilla OVM with a planned roll out taking most of the points from the Dialog roll out plan. Realistic expectations need to be set: expect improved verification but that improved efficiency will follow in time after the initial ramp up.

From experience through working with other adopters of OVM, TVS also recommends a multi-layered building approach to test benches and verification environments:

• Have OVM experts develop the main infrastructure and test benches.
• Use verification engineers to develop project sequences.
• Consider using non-verification engineers to develop the specific scenarios.

This allows engineers to work at different levels of the test bench according to their need to understand OVM and their project specific knowledge. For example, at one TVS client a software engineer used the TVS SDCard VIP to set up various scenarios to allow them to test corner cases in their driver code.

BACKGROUND
Test and Verification Solutions (TVS, Bristol, UK) delivers an independent testing service that not only reduces costs and time-to-market, but also improves product quality. TVS combines skills and experience in software testing, hardware verification and outsourcing to provide customers with an efficient, well-managed, quality assurance service. The company provides both consultancy and execution services using experienced engineering resources in several locations around the world.

Dialog Semiconductor creates energy-efficient, highly integrated, mixed-signal circuits optimised for personal portable, short-range wireless, lighting, display and automotive applications. TVS have been helping Dialog with their successful adoption of OVM.

For more information about TVS, please visit www.testandverification.com

END NOTE
[1] Dialog has development sites around the world including Asia and the US, as well as numerous countries in Europe.
Process Management: Are You Driving in the Dark with Faulty Headlights?

by Darron May, Manager of Verification Analysis Solutions, Mentor Graphics Corporation

INTRODUCTION

If your car’s headlights were faulty, would you even think about leaving home in the dark bound for a distant destination to which you’ve never been? Even if you were armed with a map, a detailed set of instructions and a good flashlight, the blackness on the other side of your windshield would still make it difficult to navigate. And even with reliable headlights and maps, you’ll invariably still encounter obstacles and detours.

These hassles are nowadays mitigated somewhat by car navigation systems that tell us where we are, how far we have travelled and, so we can consider alternate routes and estimate how long it will take to get to our final destination, how bad the traffic is ahead.

Verification of SOCs and electronic systems is certainly a little more complex than road navigation. However, the process of planning what you are verifying and then constantly measuring where you are in the process is equally important whether your final destination is a swanky new hotel a few hours away or a successful tape-out by the end of the year.

Verification of SOCs and electronic systems is certainly a little more complex than road navigation. However, the process of planning what you are verifying and then constantly measuring where you are in the process is equally important whether your final destination is a swanky new hotel a few hours away or a successful tape-out by the end of the year.

A good verification methodology enables electronic closure of verification against a plan written in a human readable form, and an ability to make real measurements during verification and then link those to the original verification plan. To offer such features, the verification process requires not only the unification of different coverage metrics but also the unification of data from multiple tools and verification engines. Accordingly, data management is the foundation of any verification environment, which means that a well architected database offering both performance and capacity is critical to success.

This is the second article in a verification management series that shows how Mentor Graphics Unified Coverage Database (UCDB), donated to Accellera in 2008, has allowed powerful tools to be developed around the contents of the database. One key result of the database and its increasing use: verification engineers and managers are now in the position of having their own navigation system for the verification process.

GUIDING THE VERIFICATION PROCESS

All projects start with a design specification detailing what needs to be built. These specs are used to develop a strategy about what to verify to ensure that every design requirement is ultimately implemented, tested and functioning correctly. This verification plan, or testplan, is then used to guide the process of verification. Often some requirements are more important than others, though these relative rankings can change during a project either due to adjusted specs or the addition of new features. Indeed, a process guided by the verification plan needs the ability to react dynamically to changing project circumstances to ensure that acceptable quality is reached on time and with the allocated resources.

The process starts by decomposing each of the features of the specification down into test requirements. Each of the low level requirements then has one or multiple metrics associated or linked with them which indicate whether the design is doing the right or wrong thing. This, in turn, allows a coverage model to be written that is implemented within the language of choice and gives the verification team indication of where it has been. The link information from the testplan to the coverage model should be able to be put into the document itself or into the design or testbench source code.

TESTPLAN TRACKING

A list of sections or verification requirements comprise the testplan and allow metrics to be associated directly with each tested feature. Executing electronic closure then allows this information, which can be written in nearly any document format, to be read into the database and merged with the coverage results from multiple verification runs. The UCDB has the ability to store testplans as a hierarchy and associate any combination of the stored coverage metrics or directed tests with the testplan sections. Questa
Verification Management (VM) can read and import testplans from any file format, including Excel or Calc spreadsheets, and Word, Adobe or Write documents.

The utility can be customized to read any format of data into the database, so these test requirements can even come from an external requirements database like IBM Doors or Mentor Graphics ReqTracer. The UCDB itself unifies coverage from multiple verification engines, including analog and digital simulation, formal verification and emulation. This enables the testplan to pull together the results of all verification methods. And the testplan can also carry other planning information such as a feature’s priority, the engineer responsible for verifying it and the verification method that will be used.

Attaching this kind of information to the plan allows results to be filtered in any number of ways. For example, by breaking down the verification requirements into priority categories, it’s possible to assign numbers to the must-have and nice-to-have features, and those in between. This makes it possible to focus verification on the most important features first. Likewise, by adding information on engineering resources to the testplan, it is possible to track each engineer’s progress in verifying features for which they are responsible. This allows managers to spot problems and shuffle resources around during the project to ensure that schedules are still met.

![Figure 1- Example of an Excel testplan.](image)
CONTROLLING MEASUREMENT

Because metrics from multiple sources are being combined to give an overall measurement of completeness, the testplan tracker must allow users to control calculations within the testplan document. Other attributes such as weights and goals can be attached to the testplan sections to allow further control over how the metrics are combined. By giving a higher weighting to a section of the testplan, it’s possible to increase its contribution to the overall testplan coverage calculation. If a more automated way of weighting sections against each other is required, then other coverage calculation algorithms can be used. (An example is a calculation based upon the number of linked objects.) It’s also important to bear in mind that not all coverage is implemented when a testplan is first generated. Thus, the fact that a coverage object doesn’t yet exist needs to be taken into account. By marking links within the testplan as yet to be implemented, Questa testplan tracking automatically generates coverage objects as part of the merge process. Until they are implemented, these objects then show as being uncovered in the overall testplan coverage data.

REUSE

It’s extremely common today for the latest project to use IP or VIP from a past project or external source. Verification of this IP also requires a testplan, thus the need to reuse testplans within new projects. For example, all Mentor Graphics Questa VIP is shipped with testplans that can be read into the Questa testplan tracking tools. This means that support for hierarchical testplans is required not only to allow reuse but also to make it possible to break down large testplans into more manageable pieces. Because different users and IP vendors rarely provide their testplans in the same file formats, it also must be possible to mix and match the testplans from any source document. Questa testplan tracking provides the ability to add links within a document to refer to any other document type. This means that an Excel spreadsheet can have a testplan section that refers to a Word document or vice versa.

Another important requirement that enables reuse is the ability to parameterize settings that vary based on environment. For example, some links embedded within a testplan may have path settings for a particular coverage object. If a particular setting refers to a path in a piece of IP, then this path will change when used in different
environments. Likewise, if a specific use of a given block of IP means that a coverage object will never be used, then the object needs to be excluded. Accommodating these kinds of changes require parameters that can be set based on testplan usage.

**IMPROVING TESTPLAN ENTRY**

Forcing a user to capture their testplan with a particular entry method or proprietary editor creates heartburn. The good news is that building a proprietary testplan editor is not the only way to make capturing testplans easier. Most document applications like Excel, Calc and Word have their own extension languages that make it possible to integrate custom features. Questa testplan tracking provides add-ins that enable creating testplans with the application of choice. For example, the Excel add-in has a testplan creation wizard to make generating a testplan template very straightforward. Working in Excel allows for easy addition and deletion of sections and user attributes, and also automatic re-numbering and formatting. And the Excel testplan in this example interfaces directly to the UCDB for information on the objects available for linking.

Testplan validation can be run to ensure the plan is in the right format before importing it to Questa and it is possible to save the testplan directly to a UCDB file, thus completely automating the import process. Once coverage data is available, the add-in is able to annotate coverage information back into the spreadsheet. To ease adoption it will also regenerate a spreadsheet directly from a testplan that’s already been imported.

**COVERAGE MANAGEMENT**

Another important part of the process is how coverage is captured over regressions throughout the project. The UCDB is used to capture both coverage and test information from single verification runs across all engines. Then UCDB utilities combine, merge or optimize the data via ranking. A merged database can be used to answer all kinds of queries about the effectiveness of a particular testplan section, while at the same time reducing the space needed to store all the single verification runs. The database still holds information such as how and where the test was run, and how long it took. This allows for optimization of test suites by using the ranking utility to either reduce the number of simulations (by finding redundancy) or find a particular set of tests that will run in the shortest possible time while still achieving a particular coverage goal. Data management over multiple regressions and the overall project duration are flexible due to the fact that the data is available on a test by test basis.

![Figure 3 - Excel add-in making creation easy.](image-url)
Normally data is combined from verification runs of the same source code, making it straightforward to locate the same coverage objects and calculate total coverage. When changes are made to the source code and different coverage models are combined, the merge process must decide what to do when objects do not match.

Questa provides either a union merge or a base merge of data. A union merge combines all coverage from all source databases and can lead to a disconnect between the coverage data and the current source code. Great care should be taken when combining data this way. One reason: code coverage, with its dependencies on code lines, becomes stale very quickly as source code changes. Merging data from the same source code builds is recommended, though using a second merge method can also help with the problem of stale coverage. Base merging allows a base database to hold the objects that will be merged from all other provided databases. This allows the current database represented by the source code to be used as a filter at merge time. If an object in the base database matches the database being merged, the object will be included. This method is beneficial for functional coverage merging because it excludes old coverage and also helps to merge slightly different code coverage models.

**TREND ANALYSIS**

As the project progresses problems can arise with data storage requirements. Keeping the results from multiple regressions over time requires vast amounts of storage. Consider, too, the changes in the source code over time, which can make it very hard to combine the detailed information. What’s required is a different kind of merging that is suited to looking at the progress of coverage over longer periods of time. By taking snapshots of relevant data across regressions, it’s easier to spot higher level trends even without keeping all the details of every single test. The UCDB database has been formatted to allow the user to decide how to manage the data by combining runs from regressions. Then, with a trend merge of the database, it’s possible to take snapshots from these regressions over time. The trend merge is a shallow merge of the data that extracts coverage on the basis of design unit, testplan and functional coverage, or instance by instance. This reduces the amount of data that needs to be stored for each regression run. It also makes the stored numbers less susceptible to changes in source code because only metrics for higher level objects are stored within the database.

Of course other metrics within the verification process can be useful for tracking other tasks and milestones. Being able to track metrics within other parts of our verification methodology and view those with the coverage data can give excellent evidence of progress being made towards tape-out. For example, seeing the number of open, closed and assigned bugs over time gives an indication of whether bugs are still being found. Looking at the number of lines.
of source code and the number of lines that have changed from week to week can suggest how design stability is changing. Monitoring the number of tests that are run, the type of tests, and the number that have passed and failed over time also gives us a measurement of completeness. The trend merge allows any user metrics to be added to the UCDB and trended over time with the coverage metrics.

**CONCLUSION**

Mentor Graphics anticipated that the cornerstone of verification solutions would be an optimized and unified database. Thus, UCDB was architected and implemented to allow the unification of all verification data and the development of powerful verification management capabilities, all of which can be found within Questa today. These capabilities include testplan tracking to allow electronic closure of the process with a plan, utilities that allow data to be combined and optimized when generated from multiple tests and engines, and the ability to reduce data over time while still exposing data needed to see project progress.

Today, you wouldn’t leave home without your car navigation system to guide you to your destination and give you real-time location information. Questa VM gives you the same visibility and guidance for your verification process.

*Figure 5 - Trending metrics over the duration of the project.*