verification HORIZONS

A PUBLICATION OF MENTOR GRAPHICS — VOLUME 7, ISSUE 2 — JUNE 2011

WHAT'S ON THE HORIZON?

How Do You Know You Have the Right Tool for the Right Job?

By Tom Fitzpatrick, Editor and Verification Technologist

As I write this, spring appears to have finally arrived here in New England – about a month and a half later than the calendar says it should have. As much as I love warm spring weather, though, it means that I now have to deal with my lawn again. I know that many people actually enjoy working on the lawn, but as far as I'm concerned, the greatest advance in lawn-care technology happened last year when my son became old enough to drive the lawn mower. If you've ever seen a 13-year-old boy driving a lawn tractor, you'll understand my characterizing him as "constrained-random" when it comes to getting the lawn cut. I handle the "directed testing" by taking care of the edging and hard-to-reach spots, and together we manage to get the lawn done in considerably less time than it used to take me alone.

Of course, cutting the lawn isn't the only problem. We also have a rather healthy crop of dandelions this year that have to be pulled. Just like bugs in a design, they'll spread if you don't get them. Believe it or not, I actually found a tool at the hardware store specifically for pulling dandelions, so the other evening I went on a "search and destroy" mission to pull up all the dandelions on my lawn. Different classes of bugs require different tools, you see.

Our first article this month came out of a discussion I had with a colleague at DVCon. He was looking for some ideas on how to justify an investment in "methodology" to his management team who, of course, were not as steeped in these ideas as many of us are. The resulting questions and answers will hopefully serve to remind all of us of the "First Principles" behind the technologies, techniques and tools that we've come to rely on to verify our ever more complex designs.

"As a special treat in this issue, we next introduce you to the Online UVM/OVM Methodology Cookbook."

—Tom Fitzpatrick

We next introduce you to the *Online UVM/OVM Methodology Cookbook*, a new online resource from our **Verification Academy**. The biggest problem with methodology textbooks is that they often become out of date as soon as they are published. We published online to mitigate that risk and commit to update the Cookbook as the Universal Verification Methodology (UVM) from Accellera evolves. Evolution is inevitable as users and vendors explore features in UVM, and the Cookbook will be a great way for you to keep informed. This particular article is the overview page for the new UVM register modeling facility. In it, you'll see a high-level explanation of the functionality along with links to other more in-depth discussions of specific pieces of the package, a format used throughout the Cookbook. Registered users will also be able to provide feedback and updates to the articles, which we'll review and pass along as necessary.

Our next article is the conclusion of Hans van der Schoot's "*A Methodology for Hardware-Assisted Acceleration of OVM and UVM Testbenches*," which we started in the previous issue. Part two takes us through the mechanics of implementing the transaction-level interface between simulation and emulation. You'll be impressed by the results that our users have seen in adopting this powerful combination of technologies.

With the recent announcement of our Questa Ultra platform, we continue to enhance our Intelligent Testbench Automation (Questa InFact) capability. In "*Combining Algebraic Constraints with Graph-Based Intelligent Testbench Automation*", you'll see how the addition of algebraic constraints enhances the Questa InFact stimulus generation by simplifying the stimulus definition. The new import feature also allows Questa InFact to react to the current state of the design and/or testbench when producing a new stimulus item. Again, you'll see some rather impressive results from actual users of this exciting new technology.

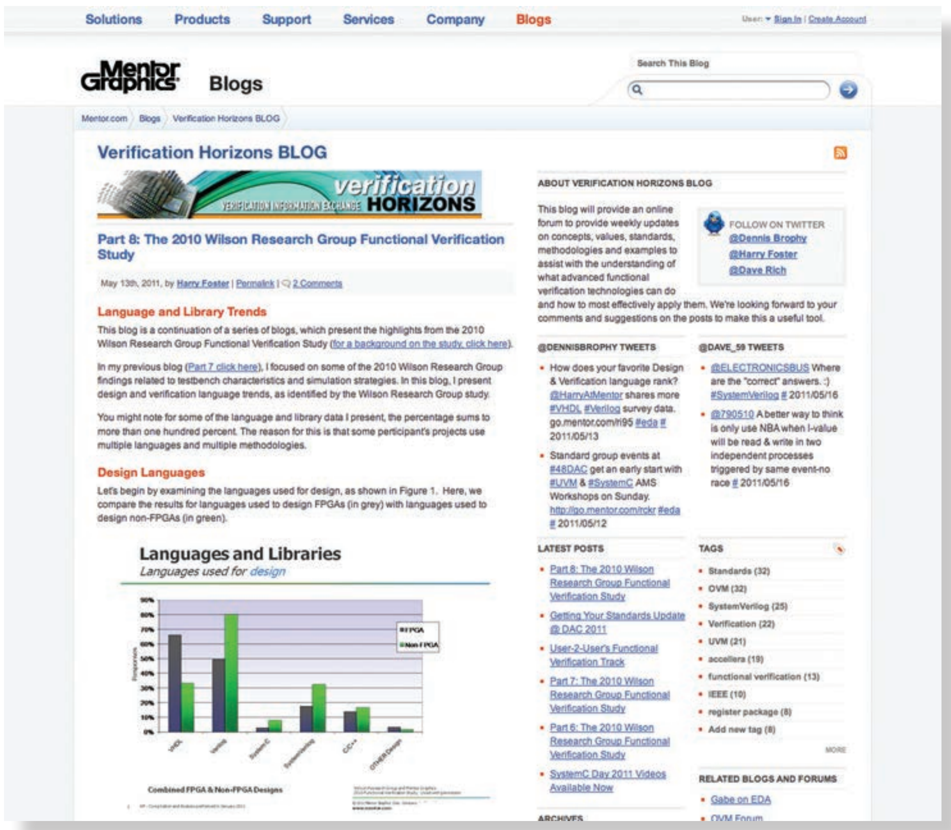
In "*Data Management: Is There Such Thing as an Optimized Unified Coverage Database?*" my colleagues Darron May and Gabriel Chidolue show yet another example of Mentor's leadership in both technology and standardization. The article provides an overview of the Unified Coverage Database (UCDB), which provides a platform for the collection and analysis of coverage data from multiple tools and verification engines. I think you'll see why the UCDB was chosen by Accellera as the basis for the upcoming Unified Coverage Interoperability Standard (UCIS).

We have four articles in our "Partners' Corner" this issue. The first, "*A Unified Verification Flow Using Assertion Synthesis Technology*", written in conjunction with our friends at NextOp, shows how their Bugscope assertion synthesis tool can be integrated into a unified verification flow with Questa and Veloce. In "*Benchmarking Functional Verification*", our friends at Test and Verification Solutions expand on the June 2009 article by Harry Foster and Mike Warner to introduce their new Functional Verification Capability Maturity Model, which helps you measure the maturity of your verification process and provides a framework for planning improvements. Putting standards into practice, our colleagues at HDL Design House next share with us their experience in creating "*UVM-Based SystemVerilog Testbenches for VITAL Models*". Find out the "four Cs" of reusability and how they've used UVM to create a family of testbenches for VITAL models while minimizing the amount of code they needed to write. We round out the Partners' Corner with "*Efficient Failure Triage with Automated Debug: a Case Study*" from our partners at Vennsa Technologies. The article shows you how Vennsa's OnPoint tools can be used with Questa to automate the identification of error sources, whether there are multiple failures from the same source or multiple sources for a given failure.

We close this issue with a special treat. We are reprinting a copy of "*Are Macros in OVM & UVM Evil?—A Cost-Benefit Analysis*", by my friend and colleague Adam Erickson. This paper won the Best Paper award at DVCon back in March, and we wanted to make sure that you saw it. Without giving away the ending, the answer is "yes." You'll find a great explanation of why, and which macros are okay to use.

I hope you'll all get a chance to stop by the Mentor booth at DAC to say "hi." I'll be happy to take any lawn care tips you might have, too!

Respectfully submitted,
Tom Fitzpatrick
Editor, *Verification Horizons*



Verification Horizons BLOG

Part 8: The 2010 Wilson Research Group Functional Verification Study

May 13th, 2011, by Harry Foster | Permalink | 2 Comments

Language and Library Trends

This blog is a continuation of a series of blogs, which present the highlights from the 2010 Wilson Research Group Functional Verification Study (for a background on the study, click here).

In my previous blog (Part 7 click here), I focused on some of the 2010 Wilson Research Group findings related to testbench characteristics and simulation strategies. In this blog, I present design and verification language trends, as identified by the Wilson Research Group study.

You might note for some of the language and library data I present, the percentage sums to more than one hundred percent. The reason for this is that some participant's projects use multiple languages and multiple methodologies.

Design Languages

Let's begin by examining the languages used for design, as shown in Figure 1. Here, we compare the results for languages used to design FPGAs (in grey) with languages used to design non-FPGAs (in green).

ABOUT VERIFICATION HORIZONS BLOG

This blog will provide an online forum to provide weekly updates on concepts, values, standards, methodologies and examples to assist with the understanding of what advanced functional verification technologies can do and how to most effectively apply them. We're looking forward to your comments and suggestions on the posts to make this a useful tool.

FOLLOW ON TWITTER
@Dennis Brophy
@Harry Foster
@Dave Rich

@DENNISBROPHY TWEETS

- How does your favorite Design & Verification language rank? @HarryAtMentor shares more #VHDL #Verilog survey data. go.mentor.com/ri95 #eda # 2011/05/13
- Standard group events at #48DAC get an early start with #UVM & #SystemC AMS Workshops on Sunday. http://go.mentor.com/rckr #eda # 2011/05/12

@DAVE_59 TWEETS

- @ELECTRONICSBUS Where are the "correct" answers. :) #SystemVerilog # 2011/05/16
- @790510 A better way to think is only use NBA when l-value will be read & write in two independent processes triggered by same event-no race # 2011/05/16

LATEST POSTS

- Part 8: The 2010 Wilson Research Group Functional Verification Study
- Getting Your Standards Update @ DAC 2011
- User-2-User's Functional Verification Track
- Part 7: The 2010 Wilson Research Group Functional Verification Study
- Part 6: The 2010 Wilson Research Group Functional Verification Study
- SystemC Day 2011 Videos Available Now

ARCHIVES

TAGS

- Standards (32)
- OVM (32)
- SystemVerilog (25)
- Verification (22)
- UVM (21)
- accellera (19)
- functional verification (13)
- IEEE (10)
- register package (8)
- Add new tag (8)

RELATED BLOGS AND FORUMS

- Gabe on EDA
- OVM Forum

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

verification HORIZONS

Table of Contents June 2011 Issue

Partners' Corner

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

First Principles: Why Bother With This Methodology Stuff, Anyway?

by Joshua Rensch, Verification Lead, Lockheed Martin and Tom Fitzpatrick, Verification Methodologist, Mentor Graphics Corporation

Many of us are so used to the idea of "verification methodology," including constrained random and functional coverage, that we sometimes lose sight of the fact that there is still a large section of the industry to whom these are new concepts. Every once in a while, it's a good idea to go back to "first principles" and understand how we got where we are and why things like the OVM and UVM are so popular. Both authors have found ourselves in this situation of trying to explain these ideas to colleagues and we thought it might be helpful to document some of the discussions we've had. If you're new to the idea of object-oriented testbenches in SystemVerilog and maybe are wondering what all the fuss about UVM at shows like DAC and DVCon is all about, or if you're getting ready to take that plunge, we think these ideas might help you "begin with the end in mind." If you're an "expert" at this stuff, we hope that this dialog will help you take a step back and appreciate how far we've come as an industry and remember not to get too hung up on the whiz-bang features of a methodology but to keep in mind the ultimate goal, which is to make sure that our chips are going to work properly.

For discussion purposes, we will refer to "Design Verification" (DV) as the process by which an ASIC or FPGA is checked to make sure the design is accurate and correct. It involves several techniques that depend on the complexity of the design and its intended application. For example, if you were interested in verifying a graphics processor for video game systems its DV would be different than a design for a safety system of an aircraft. These techniques include but are not limited to functional verification, formal verification, formal equivalence and emulation. This paper will focus on functional verification but there is a lot of crossover of skills for each of these techniques.

WHAT IS FUNCTIONAL VERIFICATION?

Functional verification is testing a design in a virtual environment crafted by a DV engineer utilizing a verification methodology such as Universal Verification Methodology (UVM). This is done by driving the various inputs to a design through simulation and checking to make sure the outputs are correct. These inputs can be various standard or proprietary buses and/or discrete signals to the design. The virtual verification environment is made up of various components, which will be defined later in the paper. [note: The DV engineer is trying to create a model of the "real world" in which the chip will operate.]

These designs can have 1000s of inputs, which would be impossible to test adequately without the use of various levels of abstraction. This allows a DV engineer to write tests at a high level of abstraction but still be able to drive the design. To give a real world example, say you need to move a house. At the top level of abstraction, you need to move the contents of the house; this is similar to moving a large piece of data from one memory to another. As you move down the levels of abstraction, you need to move a pantry; which for a design is like a transaction on the memory. To the final layer of abstraction, the jar of peanuts you need to move, which are the individual signals on whichever memory bus you are using. All you really would like to worry about is that the house was moved. But the reality is that part of move is the jar of peanuts.

HOW DO WE TEST EVERYTHING?

These designs can be massive, encompassing many different functions and paths. There is no practical way to test every permutation of all states of a large design. That is technically true, but we can test relevancy and sufficiently representative large parts of it. A DV engineer uses a technique called Constrained Random Verification (CRV) which stresses the design in the most comprehensive and efficient manner. This consists of constrained random stimulus, self-checking testbenches, and functional coverage.

Constrained random stimulus ensures that the stimulus is meaningful. The constraints are the key, otherwise we just have random stimulus that may or may not be representative of real-world stimulus. The design can be stressed more quickly and the randomness can create corner cases that a human verifier might miss. Since it is random, the test can be run multiple times and create different stimulus each time. This is controlled by a seed fed into the tool which, along with the constraints, provides

a way to produce near infinite sets of valid stimulus. Say you want to check and make sure that every part of the house's roof is waterproof. You could methodically use an eyedropper on every shingle, which would take a prohibitively long time. Wouldn't it be better to use a sprinkler on it and randomly hit most of the locations and then use the eyedropper on parts that were missed?

Self-checking is exactly what it sounds like; it automatically determines if the design performed correctly. In a traditional testbench methodology, the outputs are typically manually checked after the inputs are created. With random stimulus, that method can be tedious and prone to errors, self-checking at the proper abstraction level to the rescue. Utilizing a higher level of abstraction to check the correctness of the design allows for quicker environment creation and there are additional checkers for the lower levels of abstraction. Using the house analogy, check that each room of the house was properly moved and have lower level checkers verify that each room was moved properly.

HOW DO WE KNOW WHEN WE ARE DONE?

This is the typical question from various managers in the field. Luckily there are three metrics that help determine when the verification is completed. They are functional coverage, code coverage, and bug curves.

Functional coverage is defined by a concept of assertions and cover statements. An assertion is defining the proper behavior for something. An example assertion would be that a house should keep a person dry. Therefore, if a person inside a house ever gets wet when it rains, the assertion fails. The problem is that if it never rains, that assertion will always be true, but is never tested (we call that "vacuously true"). That is where a cover comes into play. A cover states, a person should be inside the house when it is raining outside and testing isn't complete unless that happens. One of the main responsibilities of a DV engineer is to make sure that all the coverage points are exercised.

Code coverage is built into most simulators. It gives the percentage of the code that has been exercised. This is a raw metric, which by itself has limited value. But when tied with the other metrics, it gives a good measure of confidence. The reason code coverage is not the only metric that should be used is that without an understanding of previous executed code nor the interrelationship with other statements of code and the results are not verifying the manner in which the code was executed. This is a problem because a number of latent defects are found when code interacts with other code. To take it back to the house, code coverage would give us an understanding that we have a house, that we were inside it and that it rained; it would not give us the idea that we were inside the house during the rain. It is possible to have 100% code coverage on a buggy design.

Bug metrics are the last part of this tripod. Tracking bugs found is important because the verification job isn't complete if the bug curve doesn't flatten out, meaning that if the DV engineer is pulling out four to five bugs a week, the design is not completed. But just because the bug curve is flat doesn't mean that testing is completed, the functional and code coverage need to be checked simply because the DV engineer might be testing the same small piece of logic and not getting to other logic that needs to be tested.

THIS SOUNDS COMPLICATED, IS THERE ANYTHING THAT CAN SPEED DEVELOPMENT UP?

Glad you asked, now for a brief history lesson. For a long time, the industry was fragmented, forcing vendors to have their own tools and techniques to solve this problem. That was until SystemVerilog became an IEEE standard in 2004. Vendors started to support the verification tools within System Verilog and developing their own methodology using System Verilog. It wasn't until this year that they standardized on a methodology UVM. This had all the tools to create an environment. To use the house analogy, System Verilog provided the raw materials to build a house, wood, stone, metal and glass. UVM provided frames, doors, windows and sinks to build your environment from. No longer did you need to develop your own way of doing things.

A UVM environment is built using various objects; the basic building blocks of which are called UVM sequences and UVM components.

UVM creates stimulus by using sequences and sequence items. These define the test that will create the operational parameters for the test. Think of these as a list of instructions on how to build the house. There can be several sequences, which don't have to have any knowledge of each other. In the house analogy, they describe how to build the various rooms that will make up the house. The order in which these rooms are built does not typically matter.

One pre-defined component is called an agent. These are the objects that control the various buses into the DUT. It contains both a driver that presents transactions to the bus and a monitor, which captures transactions on the bus and reports them back to the environment. It arbitrates for usage of the resource it is connected to. Arbitration is essential to make sure that the environment works in an organized manner since many sequences could be trying to access the bus at the same time.

Another pre-defined component is a scoreboard used to verify that transactions coming out of the DUT are correct. This is done by using an agent to capture the transaction and checking against a transaction that was created by some object in the environment that is predicting what the DUT will do. These predicted transactions could be created by some component in the environment getting the same as the DUT and determining what the DUT should create.

ARE THERE ADDITIONAL BENEFITS TO USING THE VERIFICATION ENVIRONMENT?

There are a couple of advantages to using a sophisticated verification environment. One is that once you have it built, adding another path or feature isn't typically difficult. If you are using a traditional testbench methodology, you may have a long task to modify the environment to test it. With this approach it could be as simple as changing the path the data takes. Also, if a bug is found later on in the development cycle, it can be replicated and triaged in the DV environment. This will lead to faster turnaround times since with simulation there is better visibility into the design.

WHAT KIND OF PERSON DO WE NEED TO DO THIS?

The typical DV engineer has to be a hybrid of a hardware and software engineer. They need to be able to understand the hardware world, comprehend the specification of a hardware product, and translate that into the environment that needs to be created. They need the skills and understanding of the world of software but with the grasp of design to create more complete and correct testing. Good DV engineers have an understanding of concepts like Object Oriented Programming and Transaction-Level Modeling (TLM) to better utilize industry standard verification techniques.

The DV engineer or team needs to be independent of the designer or design team. This provides for two sets of eyes on a design, as each will interpret a specification or requirements with their own personal bias. This independent checking will lead to a better design and a more complete set of documentation, for if the engineers don't agree; the customer will likely misinterpret it.

CONCLUSION

How do you calculate the cost of missing a bug? The typical profile is that there is some initial investment in putting the infrastructure together followed by a substantial gain as results start to come in. As a rough example, suppose a design needs to be verified and it is determined that it would take 5 days for a DV engineer to create an environment. In the same span, a non-DV engineer creates, debugs and executes a test a day so after a week, you have five working tests. After four days, the directed approach has 4 tests and the DV approach has none. On the 5th day, the directed approach has 5 tests, but the DV approach has literally 1000's of tests. Why?

A properly architected UVM environment allows you to create many variations on the theme automatically. Remember, that in addition to randomizing stimulus, you're also randomizing the structure of your testbench.

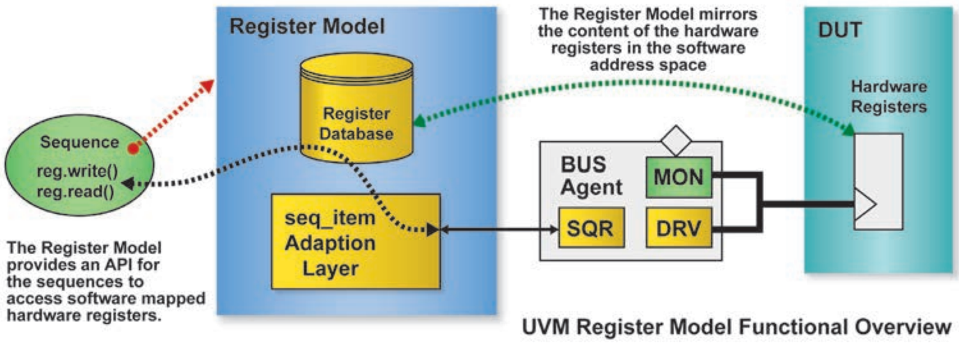
DV is standard for all ASIC companies around the world and until recently its use on FPGAs has been limited. That was until the size and complexity of FPGAs made it no longer a trivial task to debug it in the lab. The earlier in the development cycle the bug is found, the quicker and cleaner it can fixed. By doing DV in parallel with the design, it reduces time in the lab and slips in program schedule.

Online UVM/OVM Methodology Cookbook: Registers/Overview

by Mark Peryer, Verification Methodologist, Mentor Graphics Corporation

INTRODUCTION

The UVM register model provides a way of tracking the register content of a DUT and a convenience layer for accessing register and memory locations within the DUT.


Register Model
Register Database
seq_item Adaption Layer
Sequence
reg.write()
reg.read()
The Register Model provides an API for the sequences to access software mapped hardware registers.
The Register Model mirrors the content of the hardware registers in the software address space
DUT
Hardware Registers
BUS Agent
MON
SQR
DRV

UVM Register Model Functional Overview

The register model abstraction reflects the structure of a hardware-software register specification, since that is the common reference specification for hardware design and verification engineers, and it is also used by software engineers developing firmware layer software. It is very important that all three groups reference a common specification and it is crucial that the design is verified against an accurate model.

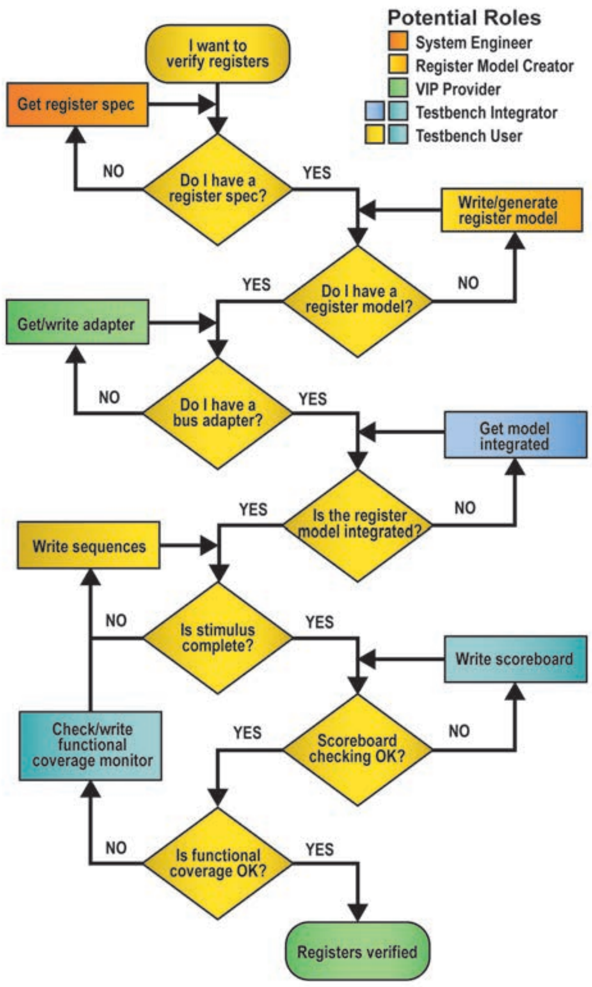
The UVM register model is designed to faciliate productive verification of programmable hardware. When used effectively, it raises the level of stimulus abstraction and makes the resultant stimulus code straight-forward to reuse, either when there is a change in the DUT register address map, or when the DUT block is reused as a sub-component.

HOW THE UVM REGISTER MATERIAL IS ORGANIZED

The UVM register model can be considered from several different viewpoints and this page is separated into different sections so that you can quickly navigate to the material that concerns you most. The diagram to the right summarizes the various steps in the flow for using the register model and outlines the different categories of users.

Therefore, the different register viewpoints are:

- The VIP developer
- The Register Model writer
- The Testbench Integrator
- The Testbench User



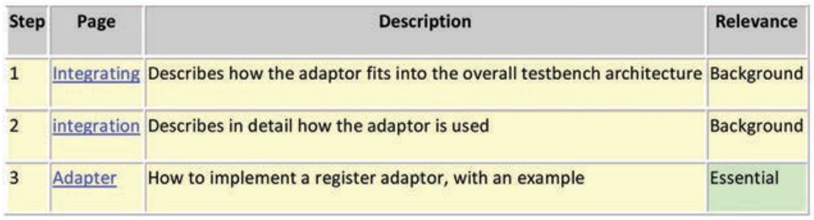
Register Model—Use Model Flow & Potential Roles

VIP DEVELOPER VIEWPOINT

In order to support the use of the UVM register package, the developer of an On Chip Bus verification component needs to develop an adapter class. This adapter class is responsible for translating between the UVM register packages generic register sequence_items and the VIP specific sequence_items. Developing the adapter requires knowledge of the target bus protocol and how the different fields in the VIP sequence_item relate to that protocol.

Once the adapter is in place it can be used by the testbench developer to integrate the register model into the UVM testbench.

To understand how to create an adapter the suggested route through the register material is:

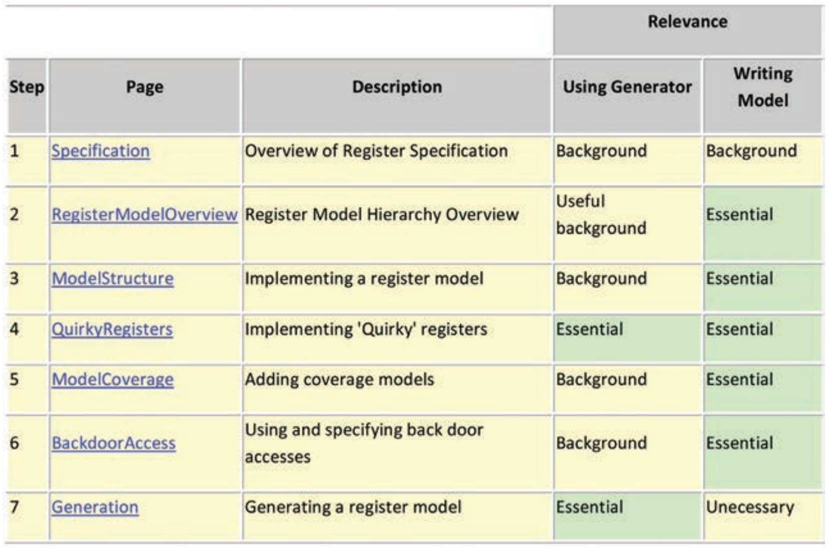


Step	Page	Description	Relevance
1	Integrating	Describes how the adaptor fits into the overall testbench architecture	Background
2	integration	Describes in detail how the adaptor is used	Background
3	Adapter	How to implement a register adaptor, with an example	Essential

CREATING A REGISTER MODEL

A register model can be created using a register generator application or it can be written by hand. In both cases, the starting point is the hardware-software register specification and this is transformed into the model.

If you are using a generator or writing a register model based on a register specification then these topics should be followed in this order:



			Relevance	
Step	Page	Description	Using Generator	Writing Model
1	Specification	Overview of Register Specification	Background	Background
2	RegisterModelOverview	Register Model Hierarchy Overview	Useful background	Essential
3	ModelStructure	Implementing a register model	Background	Essential
4	QuirkyRegisters	Implementing 'Quirky' registers	Essential	Essential
5	ModelCoverage	Adding coverage models	Background	Essential
6	BackdoorAccess	Using and specifying back door accesses	Background	Essential
7	Generation	Generating a register model	Essential	Unecessary

INTEGRATING A REGISTER MODEL

Integration Pre-requisites

If you are integrating a register model into a testbench, then the pre-requisites are that a register model has been written and that there is an adaptor class available for the bus agent that is going to be used to interact with the DUT bus interface.

Integration Process

In the testbench, the register model object needs to be constructed and a handle needs to be passed around the testbench environment using either the configuration and/or the resource mechanism.

In order to drive an agent from the register model an association needs to be made between it and the target sequencer so that when a sequence calls one of the register model methods a bus level sequence_item is sent to the target bus driver. The register model is kept updated with the current hardware register state via the bus agent monitor, and a predictor component is used to convert bus agent analysis transactions into updates of the register model, pictured at the top of the next page.

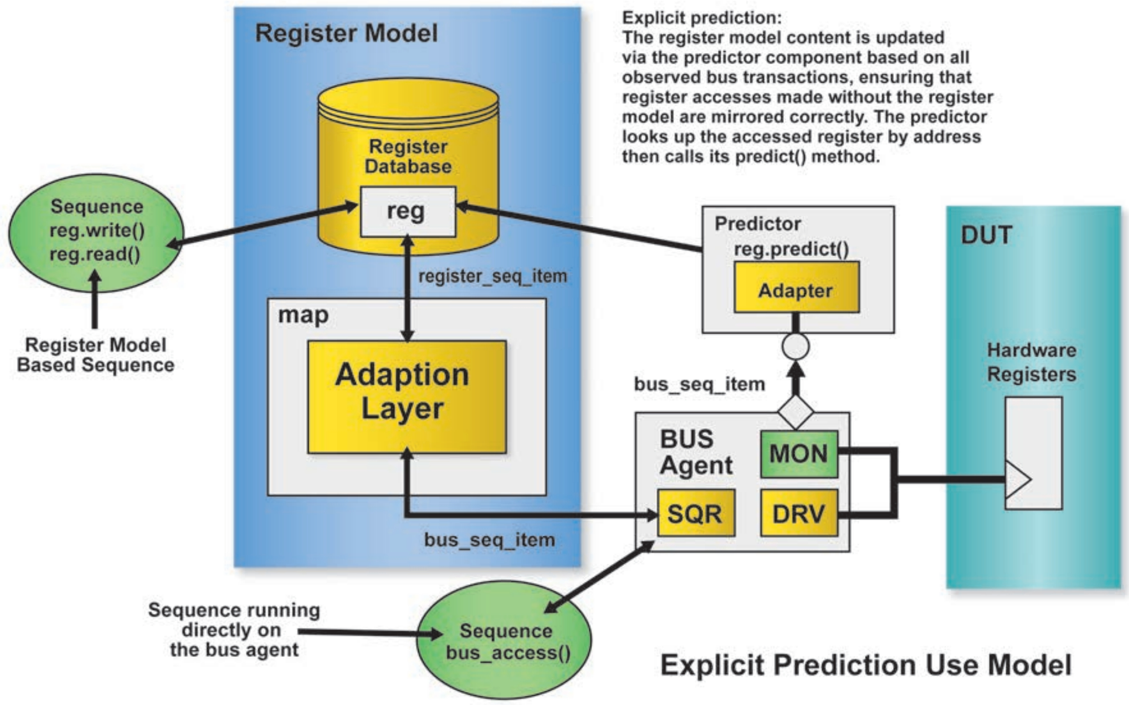
The testbench integrator might also be involved with implementing other analysis components which reference the register model, and these would include a scoreboard and a functional coverage monitor.

For the testbench integrator, the recommended route through the register material is outlined in the table to the right:

USING A REGISTER MODEL

Once it has been integrated, the register model is used by the testbench user to create stimulus using sequences or through analysis components such as scoreboards and functional coverage monitors.

The register model is intended to make it easier to write reuseable sequences that access hardware



Register Model
Register Database
reg
Sequence
reg.write()
reg.read()
Register Model Based Sequence
register_seq_item
map
Adaption Layer
bus_seq_item
Explicit prediction:
The register model content is updated via the predictor component based on all observed bus transactions, ensuring that register accesses made without the register model are mirrored correctly. The predictor looks up the accessed register by address then calls its predict() method.
Predictor
reg.predict()
Adapter
bus_seq_item
BUS Agent
MON
SQR
DRV
DUT
Hardware Registers
Sequence running directly on the bus agent
Sequence
bus_access()

Explicit Prediction Use Model

registers and areas of memory. The model data structure is organized to reflect the DUT hierarchy and this makes it easier to write abstract and reuseable stimulus in terms of hardware blocks, memories, registers and fields rather than working at a lower bit pattern level of abstraction. The model contains a number of access methods which sequences use to read and write registers. These methods cause generic register transactions to be converted into transactions on the target bus.

The UVM package contains a library of built-in test sequences which can be used to do most of the basic register and memory tests, such as checking register reset values and checking the register and memory data paths. These tests can be disabled for those areas of the register or memory map where they are not relevant using register attributes.

One common form of stimulus is referred to as configuration. This is when a programmable DUT has its registers set up to support a particular mode of operation. The register model can support auto-configuration, a process whereby the contents of the register model are forced into a state that represents a device configuration using constrained randomization and then transferred into the DUT.

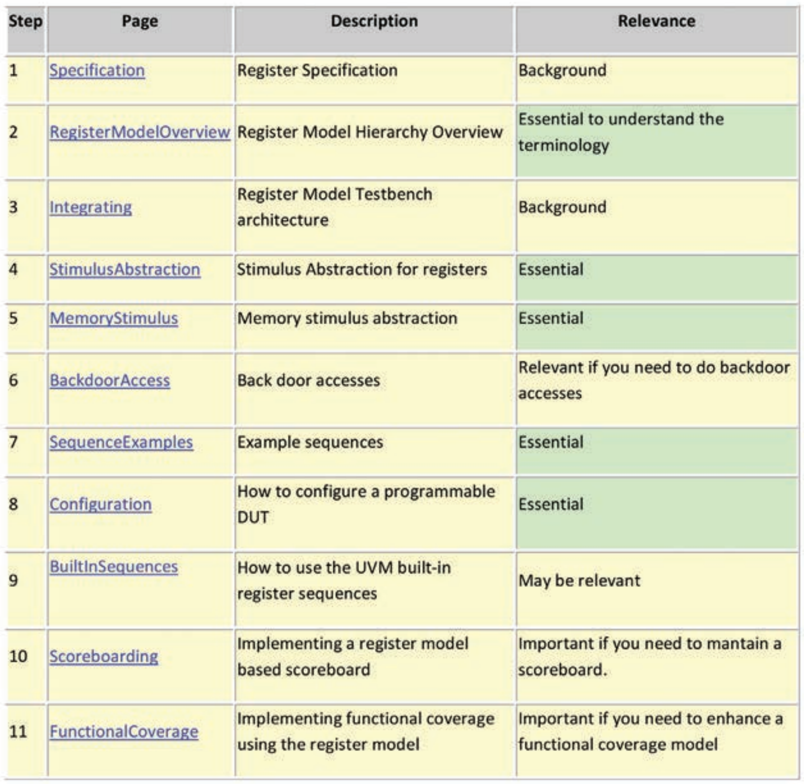
The register model supports front door and back door access to the DUT registers. Front door access uses the bus agent in the testbench and register accesses use the normal bus transfer protocol. Back door access uses simulator data base access routines to directly force or observe the register hardware bits in zero time, by-passing the normal bus interface logic.

As a verification environment evolves, users may well develop analysis components such as scoreboards and functional coverage monitors which refer to the contents of the register model in order to check DUT behaviour

Step	Page	Description	Relevance
1	RegisterModelOverview	Overview of the register model hierarchy	Useful to understand terminology
2	Integrating	Overview of the register model stimulus and prediction architecture	Essential
3	Adapter	Adapter Implementation detail	Useful background
4	integration	Register model integration detail	Essential
5	Scoreboarding	Scoreboard implementation	Useful background
6	FunctionalCoverage	Coverage implementation	Useful background

or to ensure that it has been tested in all required configurations.

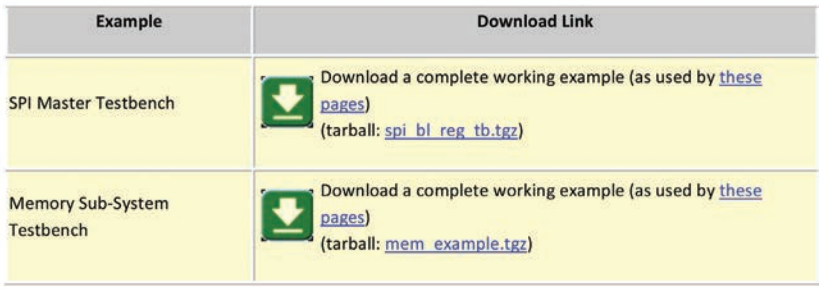
If you are a testbench consumer using the register model, then you should read the following topics in the recommended order:



Step	Page	Description	Relevance
1	Specification	Register Specification	Background
2	RegisterModelOverview	Register Model Hierarchy Overview	Essential to understand the terminology
3	Integrating	Register Model Testbench architecture	Background
4	StimulusAbstraction	Stimulus Abstraction for registers	Essential
5	MemoryStimulus	Memory stimulus abstraction	Essential
6	BackdoorAccess	Back door accesses	Relevant if you need to do backdoor accesses
7	SequenceExamples	Example sequences	Essential
8	Configuration	How to configure a programmable DUT	Essential
9	BuiltInSequences	How to use the UVM built-in register sequences	May be relevant
10	Scoreboarding	Implementing a register model based scoreboard	Important if you need to mantain a scoreboard.
11	FunctionalCoverage	Implementing functional coverage using the register model	Important if you need to enhance a functional coverage model

REGISTER MODEL EXAMPLES

The UVM register use model is illustrated by code excerpts which are taken from two example testbenches. The main example is a complete verification environment for a SPI master DUT, in addition to register model this includes a scoreboard and a functional coverage monitor, along with a number of test cases based on the use of register based sequences. The other example is designed to illustrate the use of memories and some of the built-in register sequences from the UVM library. Download links for these examples are provided in the table below:



Example	Download Link
SPI Master Testbench	Download a complete working example (as used by these pages) (tarball: spi_bl_reg_tb.tgz)
Memory Sub-System Testbench	Download a complete working example (as used by these pages) (tarball: mem_example.tgz)

Editor's Note: This article is an excerpt from the Online UVM/OVM Methodology Cookbook, available via Mentor Graphics' Verification Academy (http://verificationacademy.com)

A Methodology for Hardware-Assisted Acceleration of OVM and UVM Testbenches

by Hans van der Schoot, Anoop Saha, Ankit Garg, Krishnamurthy Suresh, Emulation Division, Mentor Graphics Corporation

[Editor's Note: This is part 2 of a two-part article on this topic. Part 1 appeared in the DVCon (February, 2011) edition of Verification Horizons. This article should serve as a great companion piece to the new Verification Academy module, Acceleration of SystemVerilog Testbenches with Co-Emulation.]

A methodology is presented for writing modern SystemVerilog testbenches that can be used not only for software simulation, but especially for hardware-assisted acceleration. The methodology is founded on a transaction-based co-emulation approach and enables truly single source, fully IEEE 1800 SystemVerilog compliant, transaction-level testbenches that work for both simulation and acceleration. Substantial run-time improvements are possible in acceleration mode and without sacrificing simulator verification capabilities and integrations including SystemVerilog coverage-driven, constrained-random and assertion-based techniques as well as prevalent verification methodologies like OVM or UVM.

IMPLEMENTING A TRANSACTION-LEVEL HVL–HDL INTERFACE

With the timed and untimed portions of a testbench fully partitioned, what remains is establishing a transaction-based communication mechanism for co-emulation. As suggested above, the use of virtual interface handles on the HVL side bound to concrete interface instances on the HDL side enables a flexible transaction transport mode for HVL-HDL communication provided thus that BFMs are implemented as SystemVerilog interfaces in the HDL hierarchy, not as modules. The flexibility stems from the fact that user-defined tasks and functions in these interfaces form the API.

Following the remote proxy design pattern discussed earlier, components on the HVL side acting as proxies to BFM interfaces can call relevant tasks and functions declared inside the BFMs via virtual interface handles to drive and sample DUT signals, initiate BFM threads, configure BFM parameters or retrieve BFM status. By retaining specifically the original transactor layer components like driver and monitor classes as the BFM proxies (see Figure 2) – minus the extracted BFMs themselves – impact on the original SystemVerilog object-oriented testbench is minimized. The proxies form a thin layer in place of the original transactor layer, which allows all other testbench layer components to remain intact. This offers maximum leverage of existing verification capabilities and methodologies.

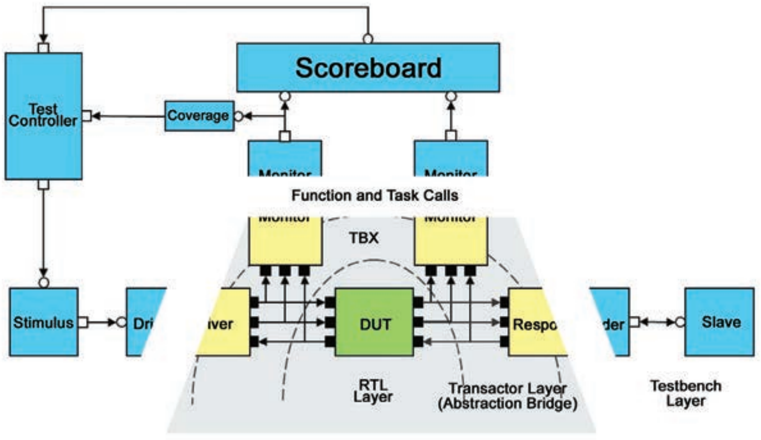
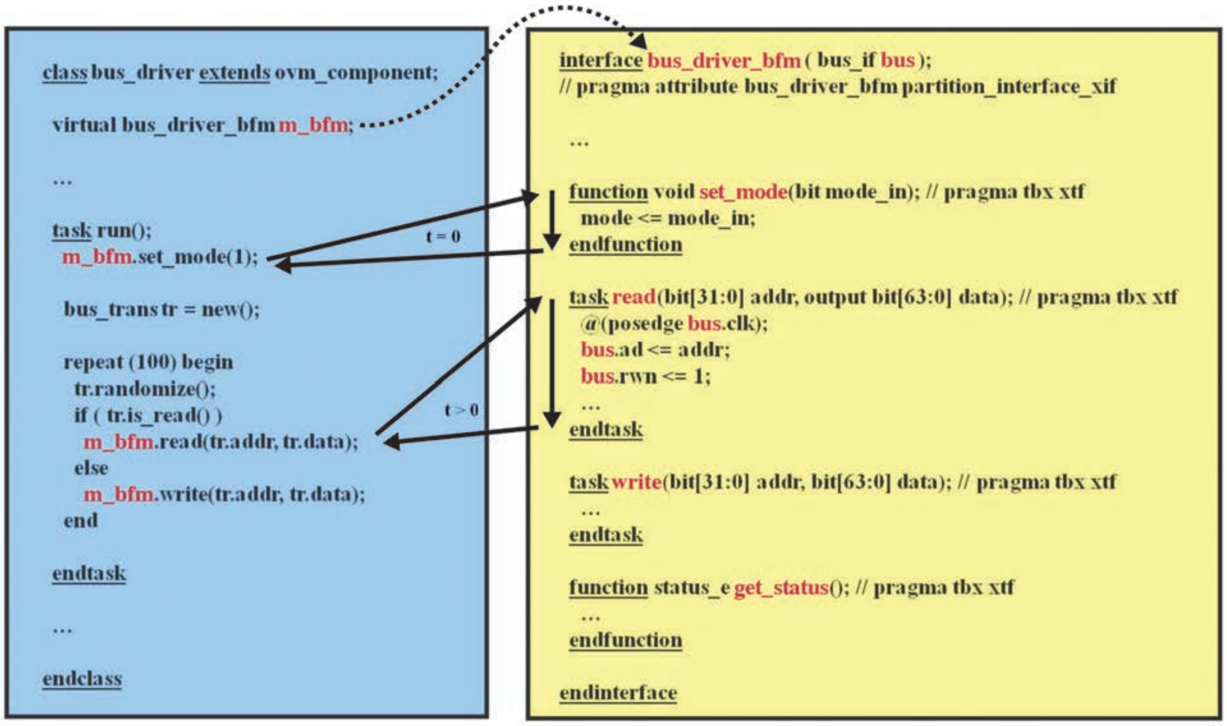


Figure 2. Transaction-based testbench with transactor/BFM proxies

The remote task/function call mechanism is based for the most part on the known Accellera SCE-MI 2 function model, and so it has the same kind of performance benefits as SCE-MI 2. In the traditional SCE-MI 2 function-based model it is the SystemVerilog DPI interface that is the natural boundary for partitioning workstation and emulator models [1], whereas the proposed methodology here uses the class object to interface instance boundary as the natural boundary for the same partitioning. Extensions specifically designed for SystemVerilog testbench modeling are added, most notably task calls in the workstation to emulator direction in which use of time-consuming/multi-cycle processing elements is allowed. This is essential to be able to model BFMs on the HDL side that are callable from the HVL side.



```
class bus_driver extends ovm_component;

  virtual bus_driver_bfm m_bfm;

  ...

  task run();
    m_bfm.set_mode(1);

    bus_trans tr = new();

    repeat (100) begin
      tr.randomize();
      if ( tr.is_read() )
        m_bfm.read(tr.addr, tr.data);
      else
        m_bfm.write(tr.addr, tr.data);
    end

  endtask

  ...

endclass
```

```
interface bus_driver_bfm ( bus_if bus );
// pragma attribute bus_driver_bfm partition_interface_xif

  ...

  function void set_mode(bit mode_in); // pragma tbx xtf
    mode <= mode_in;
  endfunction

  task read(bit[31:0] addr, output bit[63:0] data); // pragma tbx xtf
    @(posedge bus.clk);
    bus.ad <= addr;
    bus.rwn <= 1;
    ...
  endtask

  task write(bit[31:0] addr, bit[63:0] data); // pragma tbx xtf
    ...
  endtask

  function status_e get_status(); // pragma tbx xtf
    ...
  endfunction

endinterface
```

Figure 5. HDL BFM interface with HVL proxy class

The HVL-HDL co-modeling interface mechanism is depicted in Figure 5 above. A proxy class bus_driver has a virtual interface handle m_bfm to a corresponding BFM model bus_driver_bfm implemented as a synthesizable interface. Time-consuming tasks and non-blocking functions in the interface can be called by the driver proxy via the virtual interface to execute bus cycles, set parameters or get status information. Notice the 'bfm' suffix in the BFM interface name, which is recommended as a naming convention. Also notice the use of the bus pin interface confined to the BFM by inclusion through its port list.

TRANSACTION OBJECT CONVERSION

Classes and other dynamic or unpacked data types in SystemVerilog are generally not synthesizable and can therefore not be used as BFM function/task arguments. For SystemVerilog object-oriented testbenches that extensively use class-based transactions (e.g. those derived from the ovm_transaction base class in OVM) it means that these transactions cannot simply be passed as is between the BFM interfaces and their proxies. However, since BFM functions and tasks are user-defined, it may be pertinent to pass transaction class members as individual packed arguments, just as shown in the code example of Figure 5 for the address and data attributes of bus transactions.

Or one may choose to utilize special conversion routines to convert explicitly between class-based transactions and suitable packed type representations that are synthesizable such as a bit vector or packed struct. When utilized, it is recommended to standardize on from_class(...) and to_class(...) methods defined in an external converter class for each transaction type that must cross the HVL-HDL boundary. A code example is given in Figure 6.

```
class fpu_request extends ovm_transaction;

    shortreal a;
    shortreal b;
    rand op_t op;
    rand round_t round;

    ...

endclass


package fpu_trans_util_pkg;
    typedef struct packed {
      bit [31:0] a;
      bit [31:0] b;
      op_t op;
      round_t round;
    } fpu_request_s;

    typedef bit [$bits(fpu_request_s)-1:0]
      fpu_request_vector_t;

    ...

endpackage
class fpu_request_converter;

    function void to_class(
        output fpu_request req,
        input fpu_request_vector_t v);
      fpu_request_s s = v;
      req = new();
      req.a = $bitstoshortreal(s.a)
      req.b = $bitstoshortreal(s.b);
      req.op = s.op;
      req.round = s.round;
    endfunction

    function void from_class(
        input fpu_request req,
        output fpu_request_vector_t v);
      fpu_request_s s;
      s.a = $shortrealtobits(req.a);
      s.b = $shortrealtobits(req.b);
      s.op = req.op;
      s.round = req.round;
      v = s;
    endfunction

endclass
```

Figure 6. Converting transaction objects for co-emulation

Figure 7 on the following page provides an example transformation of a purely class-based FPU monitor from the OVM cookbook example kit [2] into a functionally equivalent BFM / proxy pair suited for both simulation and co-emulation. The FPU monitor proxy reimplements tasks monitor_request() and monitor_response() (i.e. lines 21-30 and 32-46 in Figure 7.b) to call corresponding tasks in the BFM (i.e. lines 58-68 and 70-73 in Figure 7.b) to perform the pin-level sampling of FPU request and response transactions and output these to the BFM proxy. External converter classes with from_class(...) and to_class(...) methods are used to convert between FPU transaction objects and convenient synthesizable packed struct representations of these transactions (i.e. lines 27 and 39 in Figure 7.b), as shown in Figure 6 for FPU requests.

For the example above it is assumed that the BFM interface is instantiated somewhere under the HDL top level hierarchy and that its corresponding proxy object on the HVL side has a virtual interface reference to the BFM. The actual binding of the virtual interface to the hierarchical HDL path of the BFM is not shown for brevity. Any such binding mechanism can be made to work also in the context of co-emulation. For OVM testbenches a recommended method described in [3] utilizes a general purpose OVM container class for wrapping any SystemVerilog type so that it can be used with the OVM configuration mechanism. It works just fine for binding BFM / proxy pairs.

HDL-TO-HVL BACK-POINTERS

For modeling flexibility and completeness a transaction-level HVL-HDL co-modeling interface can be defined in both directions. Similar to an HVL proxy class calling tasks and functions declared in an HDL interface, as discussed thus far, one can define how an HDL interface can call functions[1] declared in an HVL class. This would enable transaction-based HVL-HDL communication initiated from the HDL side. Specifically, a BFM interface may call relevant class member functions of its proxy object on the HVL side for instance to provide sampled transactions for analysis or indicate other status information.Figure 8 on the following page illustrates this. As shown, the handle of a BFM interface to the BFM proxy can be assigned simply inside the proxy itself via its virtual interface handle to the BFM. Access to any data members in the BFM proxy would not be permitted, just as cross signal references into the BFM are not allowed. Due to language restrictions on matching types, the proxy class definition together with any types it depends on must be imported inside the BFM interface via one or more packages.

Figure 7. Transforming an FPU monitor for co-emulation

```
class fpu_monitor extends ovm_component;

  ovm_analysis_port #(fpu_pair) pair_ap;

  // VIF handle to pin interface
  local virtual fpu_pin_if #(32) m_fpu_pins;

  ...

  function void connect();
    ... // Retrieve m_fpu_pins vif handle
  endfunction

  task run();
    fork
      monitor_request();
      monitor_response();
    join
  endtask

  task monitor_request();
    forever begin
      fpu_request req = new();

      do
        @(posedge m_fpu_pins.clk);
      while (m_fpu_pins.start != 1);

      req.a = $bitstoshortreal(m_fpu_pins.op_a);
      req.b = $bitstoshortreal(m_fpu_pins.op_b);
      req.op = op_t'(m_fpu_pins.fpu_op);
      req.round = round_t'(m_fpu_pins.rmode);

      $cast(m_req_in_process, req.clone());
    end
  endtask: monitor_request

  task monitor_response();
    forever begin
      fpu_response rsp = new();
      fpu_pair pair;

      ... // Timed code to sample response

      pair = new(m_req_in_process, rsp);
      pair_ap.write(pair);
    end
  endtask: monitor_response

endclass
```

(a) Original monitor

```
class fpu_monitor extends ovm_component;

  ovm_analysis_port #(fpu_pair) pair_ap;

  // VIF handle to XRTL BFM
  local virtual fpu_monitor_bfm m_bfm;

  ...

  function void connect();
    ... // Retrieve m_bfm vif handle
  endfunction

  task run();
    fork
      monitor_request();
      monitor_response();
    join
  endtask

  task monitor_request();
    forever begin
      fpu_request req;
      fpu_request_s req_s;

      m_bfm.monitor_request(req_s);
      req_converter.to_class(req, req_s);
      $cast(m_req_in_process, req.clone());
    end
  endtask: monitor_request

  task monitor_response();
    forever begin
      fpu_response rsp;
      fpu_response_s rsp_s;
      fpu_pair pair;

      m_bfm.monitor_response(rsp_s);
      rsp_converter.to_class(rsp, rsp_s);
      ...
      pair = new(m_req_in_process, rsp);
      pair_ap.write(pair);
    end
  endtask: monitor_response

endclass


interface fpu_monitor_bfm(fpu_pin_if fpu_pins);
// pragma attribute fpu_monitor_bfm partition_interface_xif

  ...

  wire clk = fpu_pins.clk;
```

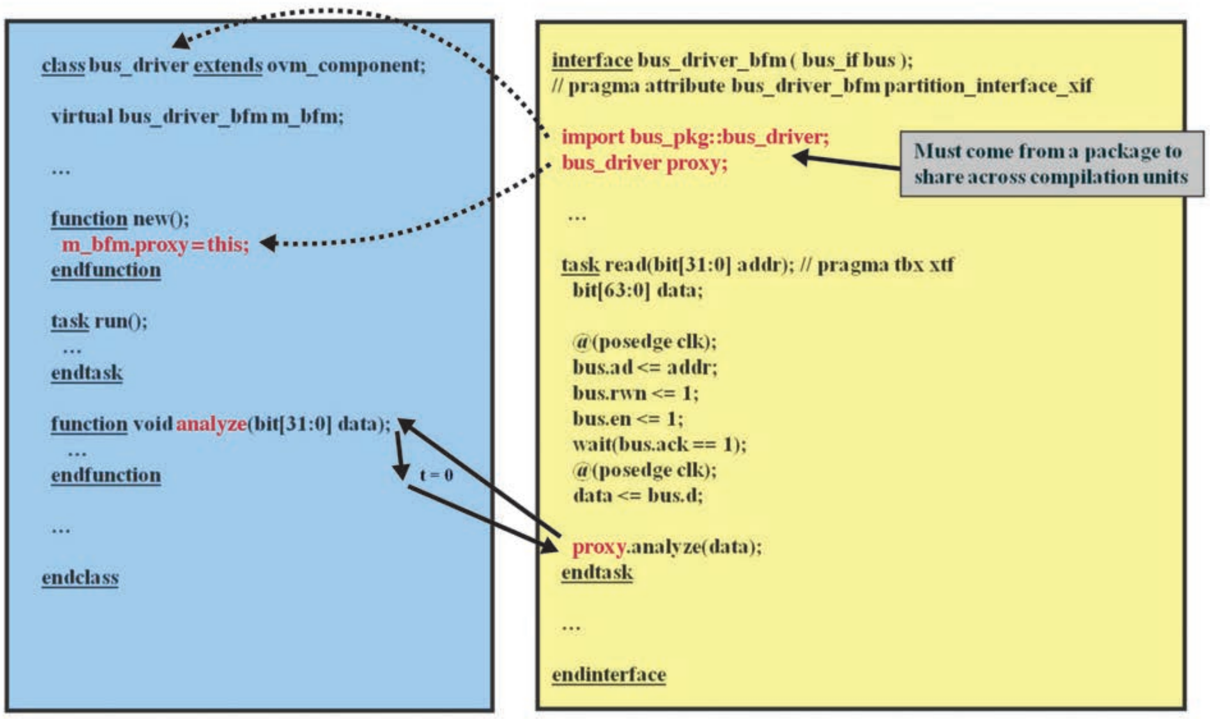
```

task monitor_request(output
  fpu_request_s req); // pragma tbx xtf
 @(posedge clk);
 while (fpu_pins.start != 1)
   @(posedge clk);
 req.a = fpu_pins.op_a;
 req.b = fpu_pins.op_b;
 req.op = op_t'(fpu_pins.fpu_op);
 req.round = round_t'(fpu_pins.rmode);
endtask

task monitor_response(output
 fpu_response_s rsp); // pragma tbx xtf
 ... // Timed code to sample response
endtask

endinterface
```

(b) XRTL monitor BFM with proxy

The use of such object handles in BFM interfaces back to their proxy classes, or 'back-pointers', is not firmly required for modeling reactive HVL-HDL communication and one can just stick to using HVL initiated 'xtf' tasks and functions[2]. Yet this is particularly useful for components like monitors. A typical monitor continuously listens to an interface to extract transactions and pass them out to other testbench components for analysis, just like the FPU monitor in Figure 7. It initiates communication of observed transactions to 'subscribers' like scoreboards, coverage collectors or interrupt monitors. It is in effect more natural to have a monitor BFM 'push' instead of the BFM proxy 'pull' these transactions out. More importantly, doing so presents opportunities for significant performance optimization. Observed transactions are commonly distributed for analysis using void functions (e.g. the TLM write(...) function in OVM – i.e. line 46 in Figure 7.a). Such one-way non-blocking calls can be dispatched and executed concurrently without even stopping the emulator clocks.

Figure 9 on the following page provides a second take on remodeling the OVM-based FPU monitor for co-emulation. This time the monitor BFM calls a void function write of its proxy via a back-pointer to push sampled FPU request-response pairs out to the HVL side (i.e. lines 62 and 22-26 in Figure 9.b). The reader is invited to inspect the example in more detail with respect to the one in Figure 7.

Figure 8. HDL Driver BFM interface with HVL proxy



```
class bus_driver extends ovm_component;

 virtual bus_driver_bfm m_bfm;

 ...

 function new();
  m_bfm.proxy=this;
 endfunction

 task run();
  ...
 endtask

 function void analyze(bit[31:0] data);
  ...
 endfunction

 ...

endclass
```

```
interface bus_driver_bfm ( bus_if bus );
// pragma attribute bus_driver_bfm partition_interface_xif

 import bus_pkg::bus_driver;
 bus_driver proxy;

  ...

 task read(bit[31:0] addr); // pragma tbx xtf
  bit[63:0] data;

  @(posedge clk);
  bus.ad <= addr;
  bus.rwn <= 1;
  bus.en <= 1;
  wait(bus.ack == 1);
  @(posedge clk);
  data <= bus.d;

  proxy.analyze(data);
 endtask

 ...

endinterface
```

ADDITIONAL METHODOLOGY CONSIDERATIONS

Prying apart transactor layer components into synthesizable BFMs on the HDL side and untimed transaction-level proxy objects on the HVL side, as described in the previous section, has the consequence that the BFMs must be elaborated statically before run-time. At first sight some of the capabilities of a truly dynamic testbench may seem lost. Recall though that it is only the timed interface protocol that is to be implemented on the HDL side. Since the DUT interface and protocol are largely static there is no real loss of functionality. The idea is to retain the bits and pieces that must be dynamic inside the BFM proxy under the HVL top level module hierarchy. It should be apparent that a BFM interface is then in principle controllable completely through its dynamic proxy, via remote function or task calls. For instance, in terms of OVM it means that while BFMs cannot be created using the OVM factory or configured using the OVM configuration mechanism, the BFM proxies can be controlled in this way and hence indirectly the static BFMs themselves.

Thanks to the application of the remote proxy design pattern, prevalent testbench topology practices can also be facilitated without much alteration. Figure 10 depicts the normal view of an OVM agent for simulation and the adapted view for co-emulation. From the perspective of the OVM testbench on the HVL side there is no difference. Certainly, a matching topology of BFM interfaces under the HDL top can be configured only statically at elaboration-time, but as suggested by the code example in Figure 11 it is rather straightforward to employ SystemVerilog conditional or loop generate constructs on the HDL side in combination with a shared package of static test parameters imported and used by both HDL and HVL sides. The topology of a typical testbench is after all static in nature since it is expected to be fully elaborated before any testbench component starts running (e.g. the ‘end-of-elaboration’ phase in OVM executes before the ‘run’ phase). In case a truly dynamic alternative is desired it is possible to elaborate a fixed number of BFMs on the HDL side of which only a subset become active as maintained by the type and number of dynamically created BFM proxy objects.

Another methodology consideration is that current synthesis technology does not readily handle SystemVerilog coverage groups. Coverage groups are well suited for implementing

```
class fpu_monitor extends ovm_component;

  ovm_analysis_port #(fpu_pair) pair_ap;

  // VIF handle to pin interface
  local virtual fpu_pin_if #(32) m_fpu_pins;

  ...

  function void build();
    ... // Retrieve m_fpu_pins vif handle
  endfunction

  task run();
    @(posedge m_fpu_pins.clk);
    fork
      monitor_request();
      monitor_response();
    join
  endtask

  task monitor_request();
    forever begin
      fpu_request req = new();

      do
        @(posedge m_fpu_pins.clk);
      while (m_fpu_pins.start != 1);

      req.a = $bitstoshortreal(m_fpu_pins.op_a);
      req.b = $bitstoshortreal(m_fpu_pins.op_b);
      req.op = op_t'(m_fpu_pins.fpu_op);
      req.round = round_t'(m_fpu_pins.rmode);

      $cast(m_req_in_process, req.clone());
    end
  endtask: monitor_request

  task monitor_response();
    forever begin
      fpu_response rsp = new();
      fpu_pair pair;

      ... // Timed code to sample response

      pair = new(m_req_in_process, rsp);
      pair_ap.write(pair);
    end
  endtask: monitor_response

endclass
```

(a) Original monitor

```
class fpu_monitor extends ovm_component;

   ovm_analysis_port #(fpu_pair) pair_ap;

  // VIF handle to XRTL BFM
  local virtual fpu_monitor_bfm m_bfm;

  ...

  function void connect();
..  // Retrieve m_bfm vif handle
    m_bfm.proxy = this;
  endfunction

  task run();
    fork
      m_bfm.request_daemon();
      m_bfm.response_daemon();
    join
  endtask

  function void write(fpu_pair_s pair_s);
    fpu_pair pair = new();
    pair_converter.to_class(pair, pair_s);
    pair_ap.write(pair);
  endfunction

endclass


interface fpu_monitor_bfm(fpu_pin_if fpu_pins);
  // pragma attribute fpu_monitor_bfm
    partition_interface_xif

  ...

  import fpu_tlm_pkg::fpu_monitor;
  fpu_monitor proxy;
  // pragma tbx oneway proxy.write

  fpu_request_s req_in_process;

  task request_daemon(); // pragma tbx xtf
    ... // Sample requests (req_in_process);
  endtask

  task response_daemon(); // pragma tbx xtf
    fpu_pair_s pair;

    @(posedge clk);

    forever begin
      @(posedge clk);
      while (fpu_pins.ready != 1)
        @(posedge clk);

      ...

      pair.req = req_in_process;
      pair.rsp.result = fpu_pins.outp;

      ...

      proxy.write(pair);
    end
  endtask

endinterface
```

(b) XRTL monitor BFM with proxy

Figure 9. Transforming an FPU monitor for co-emulation–take 2

transaction-level coverage concerned with the higher level functional requirements of a design. This stands in contrast to assertion coverage which lends itself for measuring the occurrence of lower level physical events involving the sampling of DUT signals and state variables, potentially over multiple consecutive clock cycles [4]. Assertion coverage fits naturally for BFMs and is in fact supported for synthesis by TBXTM. Moreover, while surely coverage groups could be of use in BFMs as well, key to handling any genuine transaction-level coverage requirement for a BFM interface is once again the BFM's HVL proxy object, which may have coverage groups itself and forward transactions to other transaction-level coverage analysis components (e.g. see Figure 9.b).

EMPIRICAL RESULTS

Table 1 on the following page lists empirical results of applying the proposed transaction-based SystemVerilog testbench acceleration methodology. For several different designs the run-times for executing a test with pure simulation and with co-emulation are compared. The co-emulation engine used is Mentor Graphic's Veloce TBXTM. The results clearly indicate that co-emulation can be much faster than simulation

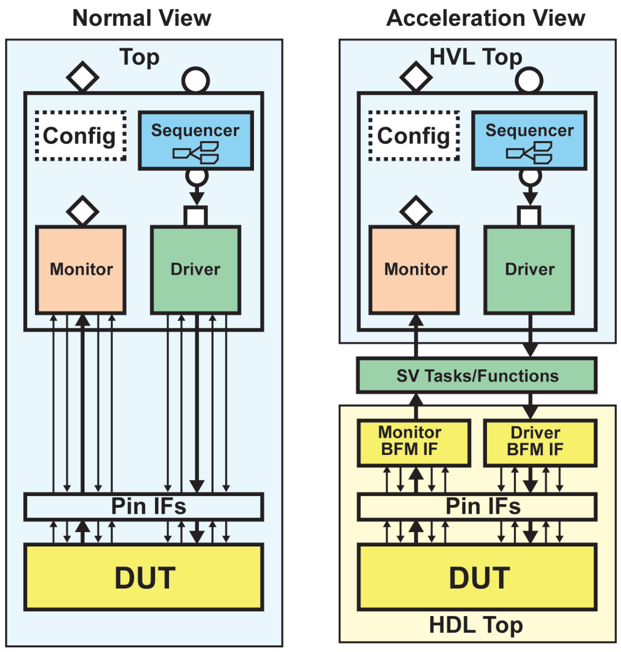


Figure 10. Normal simulation view and co-emulation view of an OVM agent

alone. Therefore, if simulation leaves you with insufficient throughput to meet your verification requirements, rather than taking calculated risks and limit the length of your simulation runs, you could greatly improve verification throughput with realistic tests using co-emulation.

SUMMARY AND CONCLUSIONS

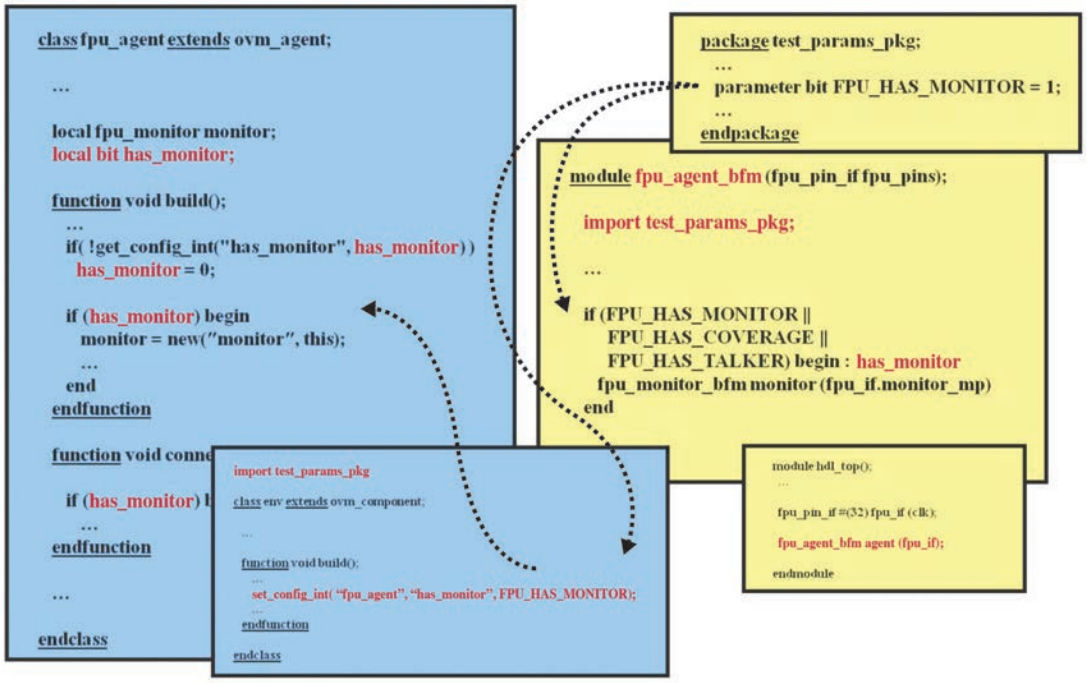
A methodology was described for writing SystemVerilog and OVM or UVM testbenches that can be used not only for software simulation, but especially for hardware-assisted acceleration. For modern transaction-level testbenches, the pragmatic approach to hardware-assisted speedup in testbench execution is to have certain testbench components – the lower pin-level components like drivers, monitors etc. – synthesized into real hardware and running inside the emulator together with the DUT, while other non-synthesizable testbench components – the higher transaction-level components like generators, scoreboards, coverage collectors etc. – remain in software running inside the simulator. Communication between simulator and emulator is then transaction-based, not cycle-based, reducing communication overhead and increasing performance because hardware-software data exchange is infrequent and information rich, and high frequency pin activity is confined to run in hardware at full emulator clock rates.

This so-called co-emulation or co-modeling approach is at the core of the methodology presented, which further maximizes reuse between pure simulation-based verification and hardware-assisted acceleration through the application of an object-oriented remote proxy design pattern. As a result, truly 'single source' and fully IEEE 1800 SystemVerilog compliant transaction-level testbenches can be created to work interchangeably for both simulation and acceleration. In acceleration mode substantial run-time improvements are made possible and without sacrificing simulator verification capabilities and integrations such as modern coverage-driven, constrained-random and assertion-based techniques and tools. Additionally, the acceleration methodology is independent of the SystemVerilog verification methodology used and applicable to all prevalent methodologies today including OVM or UVM, and VMM.

In technical summary, the proposed simulation and acceleration methodology stipulates that a testbench be partitioned into two completely separated hierarchies, a synthesizable HDL side and a strictly untimed HVL side. Cross module and signal references are not permitted between the two sides. Instead, only transaction-level data

Table 1. Empirical results

Design	Simulation Time	Veloce TBXTM	Speed-up Factor
Face Recognition Engine (1 MG)	½ hr.	6.58 secs.	128x
Wireless MM Sub-system (1 MG)	53 hrs.	658 secs	288x
Menory Controller (1.1 MG)	5 hrs.	308 secs	60x
Mobile Display Processor (1.2 MG)	5 hrs.	46 secs.	399x
Network Switch (34 MG)	16½ hrs.	240 secs.	245x
Graphics Sub-system (8 MG)	86½ hrs.	635 secs.	491x



```
class fpu_agent extends ovm_agent;

  ...

  local fpu_monitor monitor;
  local bit has_monitor;

  function void build();
    ...
    if( !get_config_int("has_monitor", has_monitor) )
      has_monitor = 0;

    if (has_monitor) begin
      monitor = new("monitor", this);
      ...
    end
  endfunction

  function void conne

    if (has_monitor) b
      ...
  endfunction

  ...

endclass
```

```
import test_params_pkg

class env extends ovm_component;

  ...

  function void build();
    ...
    set_config_int( "fpu_agent", "has_monitor", FPU_HAS_MONITOR);
    ...
  endfunction

endclass
```

```
package test_params_pkg;
  ...
  parameter bit FPU_HAS_MONITOR = 1;
  ...
endpackage
```

```
module fpu_agent_bfm (fpu_pin_if fpu_pins);

  import test_params_pkg;

  ...

  if (FPU_HAS_MONITOR ||
     FPU_HAS_COVERAGE ||
     FPU_HAS_TALKER) begin : has_monitor
    fpu_monitor_bfm monitor (fpu_if.monitor_mp)
  end
```

```
module hdl_top();
  ...

  fpu_pin_if #(32) fpu_if (clk);

  fpu_agent_bfm agent (fpu_if);

endmodule
```

Figure 11. Topology configuration

exchange is performed via 'remote procedure invocation' in SystemVerilog, and with Accellera SCE-MI 2 inspired performance benefits. Specifically, each DUT interface protocol – or BFM – on the HDL side is modeled as a synthesizable SystemVerilog interface with designated tasks and functions that can be called from the HVL side through a virtual interface by a dynamic class object that acts as HVL proxy for the BFM. Transaction objects may thereby need to be converted into synthesizable arguments. Conversely, the BFM interface may also have an object handle back to its proxy to call functions defined in the proxy. Reactive transaction-based communication is thus supported across the HVL-HDL boundary in both directions with either the HVL proxy or the BFM as call initiator. Each pair of BFM and proxy is to be viewed essentially as a joint pair representing a single transactor.

REFERENCES

[1] Accellera – Interfaces Technical Committee, "*Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual,*" Version 2.1 (Review Copy), October 21, 2010
[2] M. Glasser, "*Open Verification Methodology Cookbook,*" Springer, 2009. (Associated example kit available at www.ovmworld.org/contribution-detail/24891)
[3] A. Rose, M. Glasser, B. Osman, "*OVM Configuration and Virtual Interfaces,*" White Paper, Mentor Graphics, 2010.
[4] H. van der Schoot, J. Bergeron, "*Transaction-Level Functional Coverage in SystemVerilog,*" DVCon, 2006.
[5] A. Saha, K. Suresh, A. Jain, V. Kulshrestha, S. Gupta, "*An Acceleratable OVM Methodology Based on SCE-MI 2,*" DVCon, 2008.

NOTES

[1] Since clocks are running exclusively on the HDL side, only functions – not tasks – should be called from HDL to HVL side. Strictly speaking only time-consuming tasks are problematic, but it is recommended to avoid tasks altogether.

[2] Additionally, next to reactive communication it is possible to model non-reactive, streaming communication which can be viewed as an alternative where HVL and HDL sides are decoupled and run independent threads. Mentor Graphics' Veloce TBXTM offers this alternative via the use of so-called SCEMI pipes which are a kind of 'acceleration-friendly' buffer with unique data-shaping features for performance, described in the Accellera SCE-MI 2 standard [1]. This is not further discussed here.

Combining Algebraic Constraints with Graph-based Intelligent Testbench Automation

by Mike Andrews, Verification Technologist, Mentor Graphics

ABSTRACT:

The Questa® inFact intelligent testbench automation tool is already proven to help verification teams dramatically accelerate the time it takes to reach their coverage goals. It does this by intelligently traversing a graph-based description of the test sequences and allowing the user to prioritize the input combinations required to meet the testbench coverage metrics while still delivering those sequences in a pseudo-random order to the device under test (DUT). The rule language, an extended Backus Naur Format (BNF) that is used to describe the graph structure, has recently been enhanced to add two powerful new features. Algebraic constraints can now be included to define relationships between the fields of the stimulus description (such as the fields of an OVM/UVM sequence item). Also, external testbench values can now be imported into the graph, allowing for the definition of relationships between Questa inFact-generated field values and externally selected values. The Questa inFact algorithms can now target cross combinations of fields that are under its control with fields that are outside of Questa inFact's control. This article describes these powerful new capabilities in more detail with some simple application examples.

INTRODUCTION:

The purpose of Questa inFact is to generate meaningful stimulus automatically and efficiently from a compact description of the scenario space of interest. Currently the most widespread automated stimulus generation methodology is constrained random, which generally comes hand in hand with coverage metrics defined in the testbench in order for the verification engineer(s) to track how well the random generation performed at hitting the important cases. As every constrained random user knows, there's a tradeoff to be made in how far to constrain the stimulus generation and how comprehensive the coverage metrics should be. Increasing the scope of the coverage metrics, especially when the constraint relationships are complex, tends to push the limits on what can be efficiently achieved by purely random generation. Limiting the scope of the verification metrics makes the verification process more of a gamble, since beyond those metrics it can be very difficult to tell how effective the random generation has been.

Questa inFact has been helping verification teams to reach their desired coverage goals more predictably by defining the stimulus in a different way – specifically, using a rule based description that can be compiled into a graph. Powerful and efficient graph traversal algorithms can then more intelligently explore the stimulus space producing vectors that combine the benefits of random generation with the ability to prioritize specific coverage goals, including large cross combinations that can leave random generation stuck anywhere between 70-95% of the desired target.

Recently, the intelligence of the Questa inFact graph traversal algorithms has been significantly enhanced to allow algebraic constraints to be solved concurrently with the graph traversal process, thereby simplifying the stimulus definition process. This article describes how this powerful combination can be used to achieve more comprehensive verification goals in a more efficient and predictable way.

A DIFFERENT APPROACH TO CONSTRAINT SOLVING

When a constrained random solver generates a stimulus item its goal is to find a random valid solution to the user's stimulus description, and to continue this process repeatedly until the testbench halts and exits. The more complex the constraints are on what those valid solutions can be, the harder it is for a purely random engine to produce all the required combinations to thoroughly exercise the device under test. The primary issue is that, as complexity increases, so does the production of duplicate vectors. Many verification teams therefore can spend a lot of time analyzing the missed coverage and manually steering subsequent simulations to try to target the missing cover points – a process can that can take weeks in some cases.

As mentioned before, the stimulus description for Questa inFact can now be a mix of rules (as compiled into graphs) and algebraic constraints. The power of Questa inFact was originally in its ability to iterate through the graph description, randomly producing valid stimulus items while iterating through all combinations needed to meet the coverage goals. Where possible, the algorithms will attempt to meet more than one desired combination at a time from different cross coverage or individual stimulus field targets. As each defined coverage goal is achieved, Questa inFact will automatically revert to purely randomly generating the values for fields that are no longer involved in the remaining targets. So meeting a large total number of cover points, as defined by a number of different cover groups targeting different fields and field combinations, typically requires a number of vectors to be generated that is less than this total. This is in contrast to a pure random methodology, where the number of vectors needed to reach the desired coverage can be anything from 10x this total to effectively infinite.

Combining algebraic constraints with the traditional Questa inFact rule based description maintains this same ability to iterate efficiently through all the valid solutions. More specifically, Questa inFact is iterating through only the number of combinations that is needed to meet the goals of the various cover groups the verification engineer is targeting, since the total number of all valid solutions can be more than anyone has the time and resources to simulate.

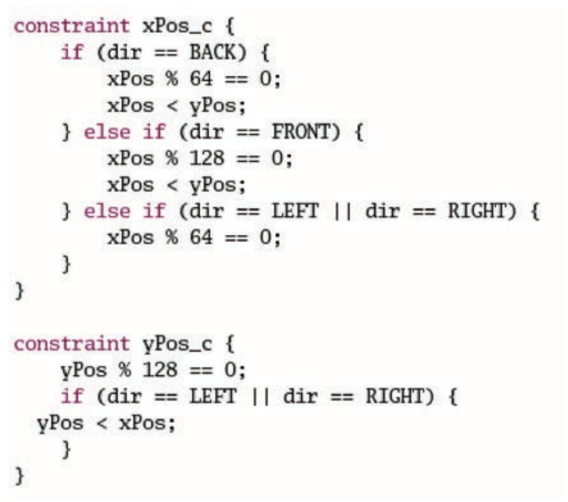
THE NEED FOR CONTEXT DEPENDENT CONSTRAINTS

Questa inFact supports two different types of constraints, one of which is a global (or static) constraint that must always be satisfied, and the other is a context dependent (or dynamic) constraint that only needs to be satisfied for specific cases. When a dynamic constraint is declared, the graph structure is used to define the applicable context.

As an example, let's consider a stimulus generation application for controlling a robot. One field of the sequence item selects a general direction for the motion between the choices LEFT, RIGHT, FRONT or BACK. If the direction choice is FRONT, for example, then the resulting motion must be be more in that direction than any other, but will allow for some sideways component to the vector. Similarly, if the choice is either LEFT or RIGHT then the robot should move more sideways than forwards. There are another two fields to determine the new relative position, called xPos and yPos.. To ensure that the general direction choice is obeyed in the FRONT and BACK case, a constraint of 'Ypos > Xpos' is placed on these fields in that case. Where the direction is either LEFT or RIGHT there should therefore be the opposite constraint of 'Xpos > Ypos'. Hence the need for context dependent constraints to be specified in the stimulus description.

While the range of xPos and yPos are specified by 12-bit values, only discrete multiples of 64 and 128 can be used for these values respectively. Also, in the special case of direction 'FRONT' xPos must also be a multiple of 128.

These constraints on xPos and yPos can be implemented in SystemVerilog as shown in Figure 1.



```
constraint xPos_c {
    if (dir == BACK) {
        xPos % 64 == 0;
        xPos < yPos;
    } else if (dir == FRONT) {
        xPos % 128 == 0;
        xPos < yPos;
    } else if (dir == LEFT || dir == RIGHT) {
        xPos % 64 == 0;
    }
}

constraint yPos_c {
    yPos % 128 == 0;
    if (dir == LEFT || dir == RIGHT) {
  yPos < xPos;
    }
}
```

Figure 1. SystemVerilog constraints

Yet another field determines the speed, which is either SLOW or FAST. All directions except for BACK can have motion at both these speeds, while the BACK direction is limited to just SLOW.

DEFINING THE STIMULUS WITH RULES AND CONSTRAINTS

One option in the Questa inFact rule description would be to simply write a rule (otherwise known as a symbol) called 'sel_vals', that simply listed the order of selection of the fields of the item. Then the SystemVerilog constraints shown above could be added to the Questa inFact rule file, with one minor syntactic difference of a ';' terminator for the constraint block. Figure 2 shows an example segment of the Questa inFact rules with the sel_val rule and the constraint on xPos.

```
symbol sel_vals;
sel_vals = dir speed xPos yPos;

constraint xPos_c {
  if (dir == BACK) {
       xPos % 64 == 0;
       xPos < yPos;
  } else if (dir == FRONT) {
       xPos % 128 == 0;
       xPos < yPos;
  } else if (dir == LEFT || dir == RIGHT) {
       xPos % 64 == 0;
  }
};
```

Figure 2. Example Questa inFact rule segment

In the Questa inFact rule description we can also define relationships using the graph structure, where a branch in the graph defines the limitations for the BACK, FRONT and LEFT & RIGHT directions. I can also apply dynamic or context-dependent constraints on those branches such that those constraints will be obeyed only in the intended context. Figure 3 shows the amended rule for 'sel_vals' and the combination of dynamic and static constraints that would be needed.

Figure 3. Example branched graph rule

```
constraint xPos_back dynamic {xPos % 64 == 0; xPos < yPos;};
constraint xPos_front dynamic {xPos % 128 == 0; xPos < yPos;};
constraint xPos_side dynamic {xPos % 64 == 0; yPos <= xPos;};
constraint yPos_c {yPos % 128 == 0;};

symbol sel_vals;
sel_vals =   (dir[BACK] speed[SLOW] xPos_back) |
             (dir[FRONT] speed xPos_front) |
             (dir[LEFT, RIGHT] speed xPos_side)
             xPos yPos;
```

In this example, the '|' choice operator is used to define the optional branches. The first branch limits the direction choice to BACK, limits the speed choice to SLOW, and then applies the dynamic constraint xPos_back. The next two branches similarly group the related field values and the algebraic constraints that must be applied to fields further down the graph path. The graph branches recombine before the selection of xPos and yPos. Figure 4 shows the graphical view of this rule graph.

This graphical view makes it much easier to visualize the stimulus description and has often helped the user to see errors that would have been much harder to discern using the text-only constraint description. The dynamic constraints appear in the graph as upside down trapeziums.

Note that in the graph view, there is an annotation on the xPos and yPos nodes stating how many bins are defined for these two fields. Given the large range of possible values, and the limits on which values are legal as defined by the constraints, it is obviously not practical to exercise all 4,096 values. A number of interesting bins are therefore defined for these two fields, and this information is used by the Questa inFact algorithms as they target the user's coverage goals. Figure 5 shows the definition of the bins for these fields.

The bins declaration follows the declaration of the domain of the field. The last bin uses a '*' to create an additional single bin which contains all remaining non-explicitly binned values. Bins can also be defined on a per coverage goal basis, so that different cross coverage goals that include the same variable can be binned differently.

COVERAGE GOALS AND THE STIMULUS SPACE

The sel_vals rule is one rule within a hierarchy of rules that define the full graph. A higher level rule call RobotCtrl includes an initialization step, a construct called a repeat, some nodes that synchronize the graph execution with the testbench, and another field of the stimulus item called 'mode.' Figure 5 shows the top-level rule.

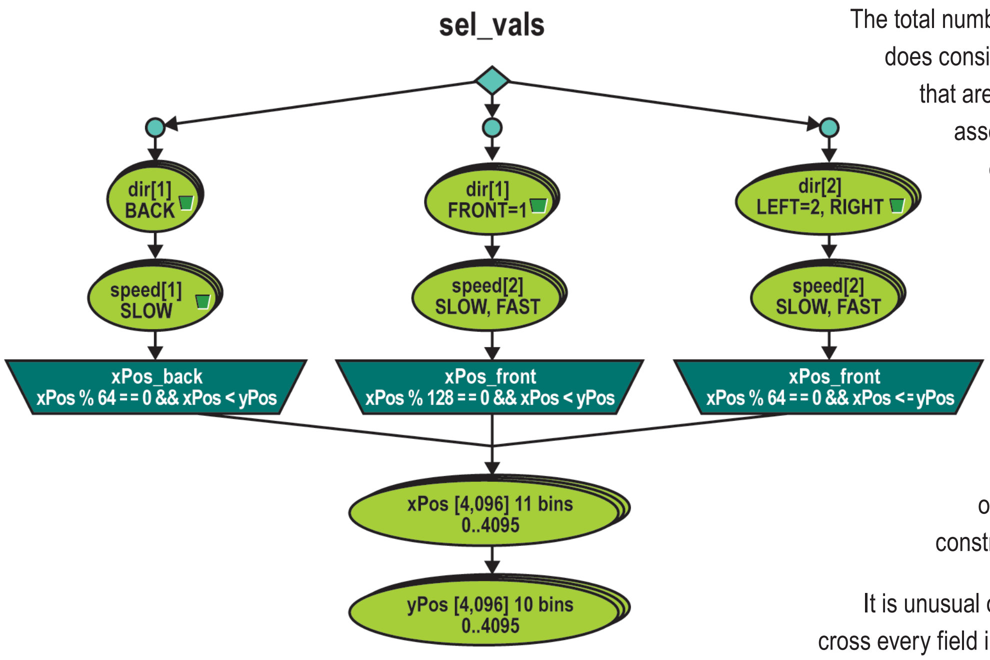
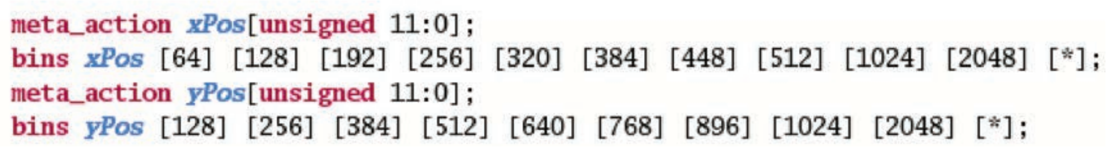

sel_vals
dir[1] BACK
dir[1] FRONT=1
dir[2] LEFT=2, RIGHT
speed[1] SLOW
speed[2] SLOW, FAST
speed[2] SLOW, FAST
xPos_back xPos % 64 == 0 && xPos < yPos
xPos_front xPos % 128 == 0 && xPos < yPos
xPos_front xPos % 64 == 0 && xPos <= yPos
xPos [4,096] 11 bins 0..4095
yPos [4,096] 10 bins 0..4095

Figure 4. Example branched graph

The total number of 3080 combinations does consider any bins defined that are global, i.e. are not only associated with a particular coverage target. It therefore reflects, in most cases, the number of cover points that would be reported for a cross cover group that included all the fields in the graph, prior to the definition of the exclusions due to constraints.

It is unusual of course to attempt to cross every field in the stimulus item, since in most cases that would be an impractical number of simulation vectors, even with sensible bins defined.

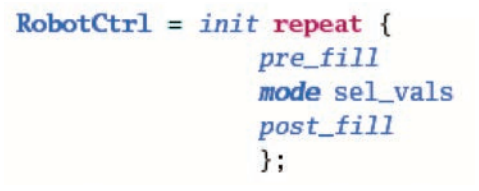


```
meta_action xPos[unsigned 11:0];
bins xPos [64] [128] [192] [256] [320] [384] [448] [512] [1024] [2048] [*];
meta_action yPos[unsigned 11:0];
bins yPos [128] [256] [384] [512] [640] [768] [896] [1024] [2048] [*];
```

Figure 5. Defining bins in the rule description

The coverage metrics tend to combine cross coverage targets with individual field targets.

Figure 6 shows the top-level graph. The numbers annotated onto the graph show the size of the stimulus space defined by the elements of the graph, without considering the effect of the constraints.



```
RobotCtrl = init repeat {
                pre_fill
                mode sel_vals
                post_fill
                };
```

Figure 6. Top-level RobotCtrl rule

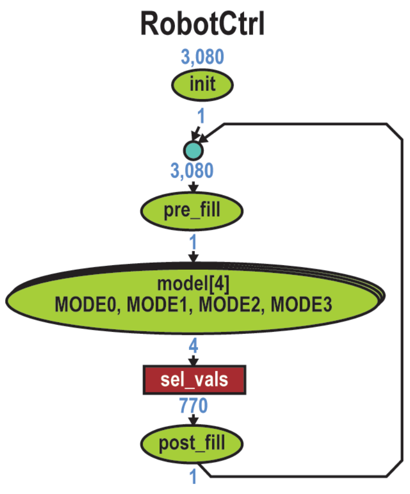


Figure 7. Top-level RobotCtrl graph with size annotation

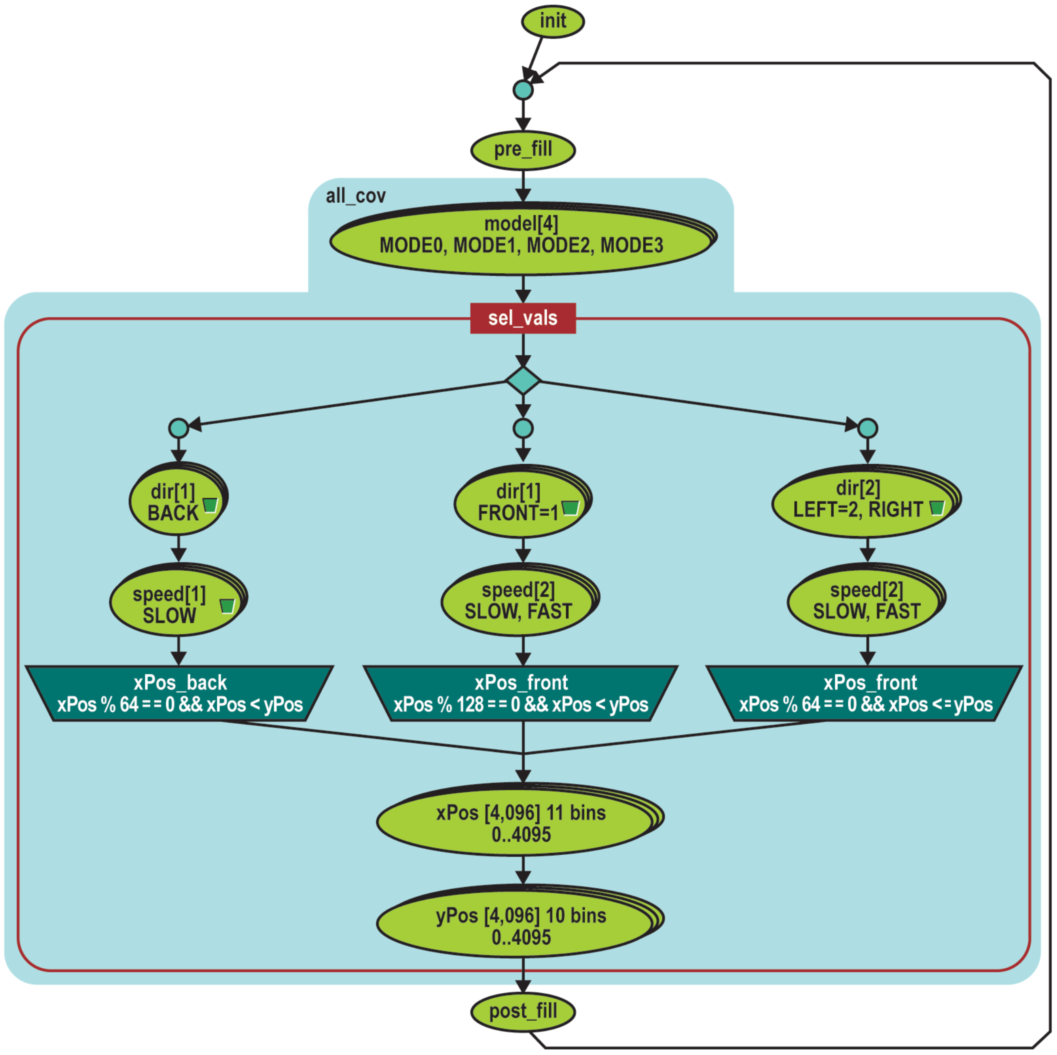


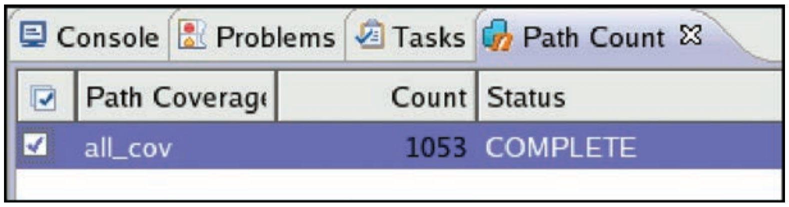
Figure 8. Definition of the coverage strategy

In Questa inFact this is achieved by overlaying a user-defined coverage strategy onto the graph. The coverage strategy mirrors the coverage goals by specifying which fields require cross coverage and which are targeted for single-value coverage.

With our robot control verification project, we will assume that the cross of all fields is the goal, so our coverage strategy contains that one goal. A graphical editor can be used to define the region of interest, which in the fully expanded graph goes from the 'mode' node down to the yPos node at the bottom. That coverage goal is called a path coverage goal in Questa inFact terminology and is given a name for reporting purposes. Figure 8 shows this goal reflected in the graphical editor.

Any of the fields can be excluded by declaring it a 'don't care' for this particular goal. An important benefit of Questa inFact's ability to comprehend the graph structure and the constraints simultaneously is that it can report the total number of valid combinations.

Our example has an additional constraint on the mode field that limits it to just the last three options, so this will also be considered in producing the total valid combinations. Figure 9 shows the result of this calculation that can be performed independently of simulation at the same time as the coverage goals are defined. This is expressed as a 'Path Count' in the Questa inFact tool.



	Path Coverage	Count	Status
✓	all_cov	1053	COMPLETE

Figure 9. Path Count Result for the Cross Coverage Goal

While we have 3520 possible combinations of these field values as binned, the graph structure reduces this number to 3080, since it reflects the relationship between dir and speed, and then the constraints further reduce the legal set to just 1053. It can be difficult to get to this final number in the presence of many complex relational constraints, and to have the cover group accurately reflect this, but Questa inFact can calculate this statically very efficiently. By adding just 3 bins to mode to reflect the constraint on that field, the cover group would contain 2640 cover points in the cross. Without specifying all the remaining illegal combinations in the cover group definition the total possible coverage that we can get is 1053/2640 = 39.886%.

COMPARING THE RESULTS

As expected, when the testbench is run with the Questa inFact graph, all 1053 legal combinations are created in exactly that number of generated items. The coverage that is reported for the cross is 39.8%. If we define this as the coverage target for a constrained random run a comparison can be made.

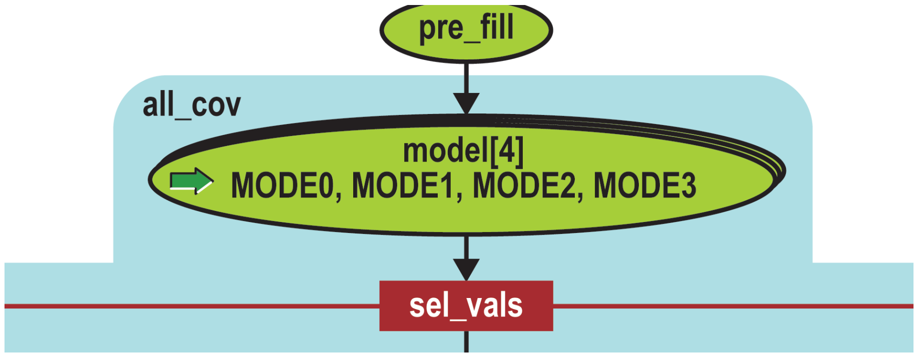
As we would expect from a constrained random generation, the progress towards the coverage goals tails off as we get closer to the goal, with significant redundancy in the vectors produced. After 120,000 items, the constrained random generation has hit 1050 of the 1053 legal combinations. It takes another 20,000 generated items to raise this to 1051. A little over another 60,000 items puts us at 1052, and the final missing combination is achieved after a total of 271,700 items. The effective improvement in the number of vectors it takes to achieve the coverage goal using Questa inFact is therefore 258x versus the constrained random equivalent. If each vector took a minute to simulate (not an unreasonable estimate) that would mean 17.5 hours of simulation with Questa inFact, and 4,528 hours (or almost 27 weeks) with constrained random. Of course, in a real verification project, other techniques would be used to get to coverage faster, such as using multiple parallel simulations, adding directed tests, or writing further constraints to steer the random generation closer to the missing coverage. A combination of these techniques would probably be used (rather than waiting for three months for the last three valid vectors).

WHAT IF I CAN'T CONTROL ALL FIELDS FROM A GRAPH?

A little more complex case is when one or more fields can't be selected by the graph, but is a product of some testbench state. In this case, we would use the new import feature in inFact to bring awareness of this field value to the inFact algorithms. With a slight modification (two lines of code affected) to the example Robot control testbench, we can move the mode field out of inFact's control and instead randomize it before we call the inFact graph to select the rest of the fields. When a variable is declared as an import in the graph, then its value is read from the testbench when the algorithms traverse through that node in the graph. Figure 10 shows the 'mode' field expressed as an import – denoted by the arrow on the left side of the ellipse.

Note that the domain of the variable is still expressed in the rules, allowing the inFact algorithms to target all the desired values and cross combinations.

Figure 10. Example of an imported variable



In this case, during simulation, a different aspect of the intelligence in Questa inFact's algorithms is exercised, that is the ability to react to the testbench state when producing a new stimulus item. If the value of mode is imported into the graph, and the coverage strategy is still to produce the cross of all fields, including mode, then Questa inFact can still far outperform constrained random, taking only 1340 items to produce all the 1053 valid items.

DO I LOSE ANYTHING WITH THIS APPROACH?

Over the years I have been working with verification teams and verification technologists I have often heard this question. After all, random generation produces more vectors than just the ones needed to meet the coverage metrics defined by the user. The generally accepted opinion is that more vectors simulated is equivalent to more verification. The promise of achieving the targeted coverage in anything from a 10th to a 1000th of the previously required vectors can therefore be a concern. The problem is there is no way to discern if those extra, mostly redundant vectors, that were generated before were actually exercising the DUT in a different or useful way. There are two ways to respond to this concern.

The first way requires virtually no extra effort on behalf of the verification team, and is to use Questa inFact to prioritize the needed vectors for coverage, and then continue to run in a purely random mode for as long as time and resources allow. A variant of this approach would be to have Questa inFact target the desired coverage more than once, taking advantage of the fact that it will produce different vectors each time, in a totally different order. This means that the combinations that the verification engineers thought were of interest get exercised in different contexts.

A second approach assumes that the original verification metrics were not as comprehensive as they could be. Expanding the scope of these metrics does of course require some effort, but this effort clearly pays off in the confidence level that can be achieved, and as proven in some cases, in the increased number of bugs that are found earlier in the verification process.

SUMMARY

As the description of the stimulus to the DUT becomes more complex, with complex constraint relationships needing to be defined, reliance on randomly generated stimulus to achieve comprehensive coverage metrics becomes a less predictable and more labor intensive process. With the addition of algebraic constraints to the Questa inFact rule based stimulus description a more intelligent approach can be taken, that can be tremendously effective in saving time and resources and is now much easier to implement.

Data Management: Is There Such a Thing as an Optimized Unified Coverage Database?

by Darron May, Manager of Verification Analysis Solutions and Gabriel Chidolue, Verification Technologist, Mentor Graphics Corporation

INTRODUCTION

With the sheer volumes of data that are produced from today's verification environments there is a real need for solutions that deliver both the highest capacities along with the performance to enable the data to be accessed and analyzed in a timely manner. There is no one single coverage metric that can be used to measure functional verification completeness and today's complex systems demand multiple verification methods. This means there is a requirement not only to unify different coverage metrics' but also to unify data from multiple tools and verification engines. Data management forms the foundation of any verification environment.

DATA STORAGE REQUIREMENTS

The reality of multiple tools, engines and metrics means the ideal verification database has to support more than just coverage. It has to have the capabilities to answer many questions posed by not only the verification engineer, but also the design engineer, the project manager and all other stakeholders in the verification process. The database infrastructure must provide the visibility into the process across many dimensions. The major requirements of such a database are as follows

Unification. No one coverage metric or a single verification engine can measure completeness. The database has to allow the storage of a large mix of coverage metrics from many data sources including simulation, emulation, FPGA prototyping, static formal analysis tools, software-driven tests and many other application-specific sources. It should be possible to combine data based on blocks, systems, instances, tests, users and time to give the most flexibility. Combining this data based on so many variables requires a flexible architecture and the need to store details about how, where and when the coverage data was generated. This allows the verification engineer to determine how and when a particular metric was or wasn't hit. The process also needs the ability to allow these metrics and measurements of certain system requirements to be associated with a verification plan and ultimately the design specification.

Capacity & Performance. Unifying the verification data storage from all tools and metrics can result in huge volumes of data. The storage capacity must be able to handle the very largest of today's designs and the designs of the future. As the stored data increases it is important to have an environment that is optimized for capacity and has the performance to manipulate and query potentially large amounts of data within workable limits. Combining results from tests that have many millions of coverage bins would require such a database. This often has a negative impact on the databases' capacity and can become a tradeoff. Ideally a solution should have the ability to solve both the capacity and performance issues within the largest of projects now and in the future.

Visibility & Analysis. Allowing queries on stored verification data requires access to the database. The results from many verification engine runs need to be combined and the verification engineer needs to be able to analyze which runs with which particular settings caused particular metrics to be hit. This type of analysis is required to figure out redundancy in tests or to isolate a particular test or set of tests of a particular feature, thus allowing the verification process to be further optimized. With the combining or merging of data it's also necessary for the Verification Engineer to be able to query the database to find out information on the history of how the data was generated. This includes not only the command line options for generating the single tool runs but also the utilities used to add and combine data to the database across the progression of the project. Reductions and optimizations are required on the data so that trends can be seen across the duration of the process. The verification process is dynamic. With the addition of new functionality in the design, as well as the process of finding and fixing bugs, there is a need to be able to look at the trends of different metrics at a higher level to determine if progress is being made towards completion.

Control. With the continual data analysis throughout the project the Verification Engineer also needs to have control over the coverage model and the ability to document decisions made during the process. The database has to have the ability to manipulate the overall coverage metrics into an overall metric showing the level of completion. It also needs the ability to trade-off one metric from another based on its importance with a user controlled weighting system. As verification progresses it's also important to document any exclusions to the coverage model and the reasons why they have been excluded. These types of exclusions could be made automatically by the verification tools. An example is a static formal tool excluding unreachable code, ahead of dynamic simulation

Extensibility/Openness. Finally, the database needs to be extensible and allow the addition of any information or metric that may be application-specific or even not currently known. An example is information or metrics from a tool yet to be developed. It also needs to be completely open and have the ability to add or remove any data with a clearly defined interface. This requirement allows any third-party tool to write data into the database or extract data, allowing the unification of data across tools from one or multiple vendors.

MENTOR GRAPHICS UNIFIED COVERAGE DATABASE

The Unified Coverage Database (UCDB) from Mentor Graphics has been architected from the ground up to meet the requirements outlined earlier. The UCDB has been the default coverage database format for storing code coverage and functional coverage metrics in both ModelSim and Questa since version 6.2. In addition, the extensibility of the UCDB has allowed test data, assertion and coverage results from Mentor Graphics Questa Formal Verification, Questa ADMS and Veloce® Emulation products to be combined. In addition to its coverage storage abilities, the UCDB also stores verification plans and test-specific data, making it a solid anchor for any verification team that intends to adopt a verification methodology that is driven from verification plans, design and/or requirements specification documents. One of the biggest verification challenges is having the ability to bring together the data and benefits from multiple verification techniques. The UCDB merge algorithms have been developed to take into consideration data from both formal (static) and dynamic verification engines. It has the ability to combine results and report on any conflicts that may occur when comparing static and dynamic techniques, as well as allowing a static formal engine to exclude coverage from the dynamic simulation engine that is flagged as unreachable.

Leveraging the unique test-associated merging capability it is possible for a verification team to maintain a single merged database that contains merged coverage data from multiple verification runs or simulations in a regression. A record of the attributes, commands and settings of any tool are associated with each test or testcase, giving it a unique label to allow test association with coverage data. The architecture allows verification plans to be imported and linked with multiple coverage metrics or tests. This single database has enough information within it to help figure out the test(s) that incremented a specific coverage bin. There is enough information to perform test ranking (aka coverage grading) on this merged database that allows the verification team to identify the most effective tests or seeds in the case of constrained random simulation. Merge performance has the biggest impact on any ranking algorithm; to gain the most optimal results the number of merges required for what if analysis is the square of the number of tests plus the number of tests, all divided by two. This greedy algorithm makes the overall performance of a ranking algorithm very sensitive to the single merge performance due to the sheer number of merges required to gain the most optimal result. With the UCDB's unique test-association merge, this analysis can be carried out from the single merge database, bringing the necessary performance for fast and accurate test analysis. Test association provides substantial disk space savings since it is no longer necessary to keep all the individual coverage databases after a test-associated merge has been performed. Another benefit is high performance of the analysis tools. Questa users benefit from two orders of magnitude reduction in their storage needs.

Once a verification project gets underway, it becomes necessary to track project momentum by looking at the trends of metrics over time. The UCDB has been architected to support shallow or "trend" merge of multiple merged UCDB databases in order to capture trend information for coverage within the merged UCDBs. The resulting trend UCDB can be visualized via graphs, HTML or CSV data that can be extracted from it for processing and analysis in other tools. Giving users an automated way of reducing and keeping the relevant data has the benefit of further reducing the volume of data required to analyze the project's progress.

The UCDB is extensible, scalable and open. Many different verification plan formats including Word, Frame, Excel, and Docbook can be imported into the UCDB. With a little XML and Style-sheet knowledge, a user can expand the list even further. The user can add any test-specific information in the form of an attribute value pair to the UCDB in addition to the attributes automatically captured by the tools. For the ultimate in flexibility, there is the UCDB C API. TCL-based CLI commands in tools such as ModelSim and Questa are implemented using the UCDB C API. Since the UCDB C API is open, it can be leveraged in many different interesting ways. For example, coverage data in third party tools can be extracted into the UCDB and then combined with coverage from Mentor Graphics tools and analyzed using the Verification Management tool suite in Questa. The UCDB C API could be used to read existing UCDB files in order to perform tasks such as generation of custom reports, performing custom queries or analysis.

Finally, the database requires different access methods to allow both performance and capacity. This has been solved in the fact that the database has both in-memory and streaming access to the stored data. This is extremely important because different analysis tools have different requirements. Some need the random access of an in-memory mode which allows multiple queries, while others need the performance to quickly access particular data for reporting purposes. The UCDB has been uniquely architected to achieve both modes of access and operation giving the best of both worlds.

THE UCDB HISTORY

Mentor Graphics began architecting the unified database nearly seven years ago. In early 2006 it released the first implementation, used natively by its simulation products to save coverage and assertion data. The database was developed from the start to store all the information needed to manage the verification process. The open API (Application Programming Interface) has been used to develop all the verification management and coverage tools within Questa and ModelSim, and is key to its extendibility to other verification tools such as Questa ADMS and Questa Formal tools. This proven implementation has been used and stressed with many users' designs and environments allowing the database itself to be refined, optimized and tuned to provide the capacity and performance required by the largest of designs.

Soon after the first implementation of Mentor Graphics' UCDB was being stressed in its use on real projects, Accellera formed the UCIS (Unified Coverage Interoperability Standard) working group. This group was formed with the goal of developing a standard for the interchange of coverage between vendors. It is made up of both EDA vendors and user representatives from the largest companies in the industry. After a period of time, Mentor Graphics representatives decided to donate its technology as a starting point for the standard due to the fact that it clearly met the requirements laid down by the group. This triggered other donations from other sources. However, the UCDB API was chosen as the basis of the standard after a lengthy period of analysis of the donated technologies, giving further credence to its capabilities. With the standardization process well under way, users will start to benefit from Mentor Graphics pioneering work and database optimizations, particularly as other vendors introduce solutions based on the UCIS.

CONCLUSION

Mentor Graphics had the vision that the cornerstone of verification solutions was an optimized and unified database. The UCDB was architected and implemented to have the capabilities to allow the unification of all verification data and allow the development of very powerful verification management capabilities, which can be found within Questa today. The UCDB implementation supports a growing number of both Mentor Graphics and third party tools, and also many custom user-created analysis tools using the Open UCDB C API. An optimized unified database is definitely a reality today, particularly when using Mentor Graphics products.

A Unified Verification Flow Using Assertion Synthesis Technology

by Yuan Lu, Nextop Software Inc., and Ping Yeung, Mentor Graphics Corporation

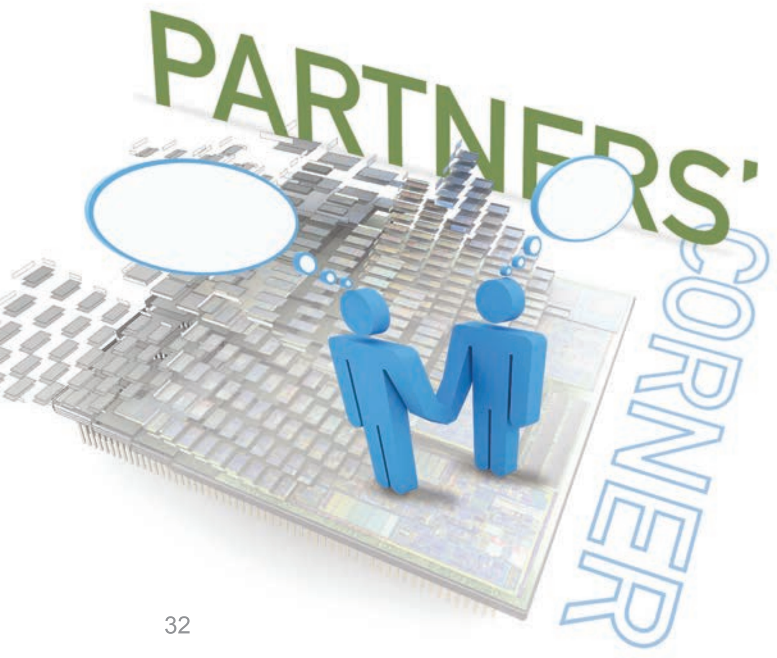
INTRODUCTION

As SOC integration complexity grows tremendously in the last decade, traditional blackbox checker based verification methodology fails to keep up to provide enough observability needed. Assertion-based verification (ABV) [1] methodology is widely recognized as a solution to this problem. ABV is a methodology in which designers use assertions to capture specific internal design intent or interface specification and, either through simulation, formal verification, or emulation of these assertions, verify that the design correctly implements that intent. Assertions actively monitor a design (or testbench) to ensure correct functional behavior. They detect design errors at their source, greatly increasing observability and decreasing debugging time.

Bugscope Assertion Synthesis

The ABV methodology is easy to adopt in most existing verification flows since it can be adopted incrementally. Besides capturing design intent manually with an assertion language such as system verilog assertion (SVA), an Assertion Synthesis tool, such as NextOp's Bugscope, can be used to create high quality assertions based on simulation activities [2].

Given a set of regression tests and the corresponding RTL, Bugscope generates properties which satisfy three requirements:



- true at every cycle of the simulation
- not easily implied by RTL only
- orthogonal to each other

These properties are further classified by designers as either assertions or cover properties. Assertions are checked in to increase verification observability while cover properties directly point to the missing functional coverage by simulation (see Figure 1). Intuitively, Bugscope captures the design/verification snapshot in the format of properties. Such information guides further verification. Therefore, Assertion Synthesis enables a progressive, targeted verification process, allowing design and verification teams to more easily uncover corner case bugs, expose functional coverage holes, and increase verification observability. The two advantages of Assertion Synthesis are

- Reduce manual assertion writing effort for designers which is believed to be the main hurdle for design team to adopt ABV;
- Synthesized cover properties provide a unique functional coverage report to the user;

In this article, we describe a unified verification flow by incorporating Bugscope into Mentor Graphics Questa/ Veloce verification flow. We will demonstrate how to integrate Bugscope with Questa simulation, Questa formal verification and Veloce simulation acceleration environment to achieve better observability.

Questa Simulation

Questa Simulation supports multiple verification methodologies including Assertion Based Verification (ABV), the Open Verification Methodology (OVM) and the Universal Verification Methodology (UVM) to increase testbench productivity, automation and reusability. It enables the automatic creation of complex, input-stimulus using scenarios described in terms of constraints and randomization using SystemVerilog or SystemC Verification (SCV) library constructs. Questa Simulation combines all of these forms of stimulus generation with functional coverage to identify the functionality exercised by the automatically generated stimulus. Using assertions as feedback for test

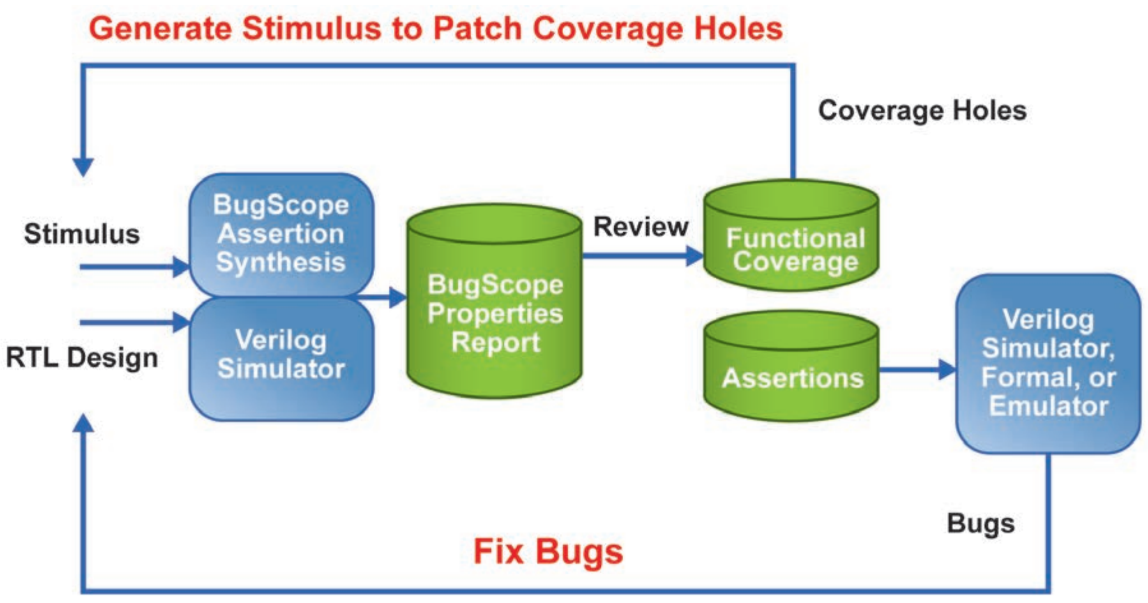

Generate Stimulus to Patch Coverage Holes
Coverage Holes
Stimulus
BugScope Assertion Synthesis
RTL Design
Verilog Simulator
BugScope Properties Report
Review
Functional Coverage
Assertions
Verilog Simulator, Formal, or Emulator
Bugs
Fix Bugs

Figure 1 Assertion Synthesis Technology

creation, engineers can adjust constraints to focus random testing on coverage holes.

Questa Formal Verification

Questa Formal Verification supports general assertion-based formal verification to ensure that the design meets its specific functional requirements. With support for PSL, SVA, and OVL, including multi-clocked assertions, Questa Formal Verification easily verifies very large designs with many assertions. Its multiple high-capacity formal engines cooperate to complete verification faster. Questa Formal Verification is integrated with the Questa Simulation for easy debug of assertion failures.

Veloce Simulation Acceleration

Veloce Simulation Acceleration speeds up block-level and full SoC regression test runs by 100s to 1000s of times. It includes a simulation-like debugging environment, has 100% internal DUT visibility, and supports traditional break pointing and ABV. In a transaction-based acceleration environment, Veloce uses TestBench XPress (TBX) Software [3], and host-based transaction-level test benches to drive transactors in the Veloce system to drive the DUT. For test benches written in C/C++ or System C, TBX interfaces directly with the Veloce and executes the test bench program. For SystemVerilog testbenches, TBX runs Questa on the host PC to drive the test bench through the Veloce-based transactors and DUT.

Questa Verification Management

Questa collects all coverage data — code coverage, assertions, formal, and functional coverage — into a single highly efficient Unified Coverage DataBase (UCDB) and makes them available in real-time within the testbench or for post-processing with Questa Verification Management. It can also capture information about the broader verification context and process, including which verification tools were used and even which parameters constrained these tools. The result is a rich verification history, one that tracks user information about individual test runs and also shows how tests contribute to the overall coverage objects.

A UNIFIED VERIFICATION FLOW WITH ASSERTION SYNTHESIS TECHNOLOGY

Given a testbench, no matter whether it is at block level or chip level, Bugscope generates assertions and cover properties based on the given tests. Expressed in SVA with binding statements ready, these property files are directly given to Questa simulator, Questa formal verifier or Veloce hardware accelerator to consume to reach intended coverage goals, check design intent, therefore reach verification signoff criteria (see Figure 2 on the following page).

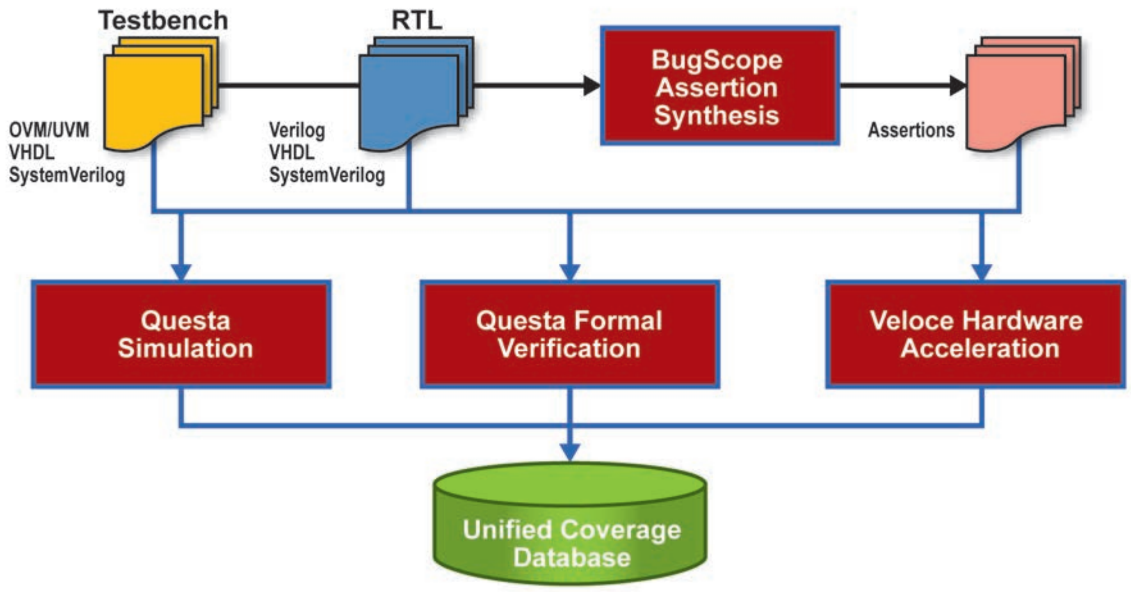

Testbench
RTL
BugScope Assertion Synthesis
Assertions
OVM/UVM
VHDL
SystemVerilog
Verilog
VHDL
SystemVerilog
Questa Simulation
Questa Formal Verification
Veloce Hardware Acceleration
Unified Coverage Database

Figure 2 Unified Verification Flow with Assertion Synthesis Technology

Unified Simulation Flow

Given a block level testbench, Bugscope generates assertions and cover properties in SVA format. The cover properties point to functional corner case holes which are missed by simulation. These SVA properties can be directly included in Unified Coverage DB which is consumed by Questa simulator to guide the user to reach 100% functional coverage.

The synthesized SVA assertions are used in block level testbench as more random seeds are executed. They are also shipped with the block RTL together to chip level simulation using Questa simulator. Note that the chip level observability is significantly improved by these whitebox assertions.

Connect Simulation and Formal Using Bugscope

Usage of formal verification suffers from two fundamental problems:

- Constraints are difficult to write and error prone. Proofs are as good as the constraints are correct. To our knowledge, there is no good way to guarantee correctness of the written constraints [4].
- Developing assertions to prove is difficult for designers.

Bugscope provides a natural solution to these two problems. Based on a Questa simulation environment, Bugscope extracts both assertions and cover properties. Note that the cover properties point to the area where simulation fails to reach. Then we give both assertions and cover properties to Questa formal verifier to prove. This methodology naturally solves the above two problems:

- Synthesized cover properties provide a good corner case target to the formal engine to reach. If they are unreachable, the constraint environment is very likely to be buggy and should be corrected;
- Bugscope supplies a high density sets of assertions for formal engine to prove. They are easier than typical manual end-to-end assertions due to its whitebox nature.

Because Bugscope can provide synthesizable set of SVAs, the integration between Bugscope and Questa Formal Verifier is painless.

New Assertion Synthesis Driven Hardware Accelerator Flow

In recent years, hardware acceleration technology has matured and been adopted by various leading chip companies. However, several fundamental issues still persist

- Testbench checking can only be applied on interface signals. Some features such as performance, whitebox behaviors are very difficult to capture at interface level.

Those types of checking are often omitted. Therefore, the verification observability is low.

- There is no coverage measurement on the quality of verification. This can be a major hurdle for the verification team to adopt hardware acceleration.

Assertions provide a natural solution to both of the above problems. As a matter of fact, Veloce hardware accelerators can accept SVA assertions as well as SVA cover properties. The problem is which assertions or cover properties should be added to the Veloce hardware acceleration. Therefore, we propose a new Assertion Synthesis driven hardware acceleration flow (see Figure 3).

First, we use Bugscope to generate assertions and cover properties in a Questa simulation environment. At classification phase, the designers will notify which assertions and which cover properties they are willing to put into the hardware acceleration environment. Second, these assertions and cover properties are filtered through Questa Formal Verifier. If an assertion is proven to be true, it will be removed from the list because it will never catch bugs as long as RTL doesn't change. Similarly, if a cover property is proven unreachable, it will be removed from the list because it will never be reachable as long as RTL doesn't change. Note that this step is important because the resource on Veloce hardware is limited. Table 1 shows the effectiveness of the Questa Formal Verifier to help reduce the number of redundant properties. Finally, the left assertions and cover properties are integrated into Veloce Hardware Accelerator to improve the observability.

CONCLUSION

This article introduces a new Quest/Veloce verification methodology based on Assertion Synthesis technology. This methodology automates the assertion based verification and solves the observability problem in the SOC verification. A number of customers have used this methodology successfully and have found bugs in their designs [5].

REFERENCE

[1] Harry D. Foster, Adam C. Krolnik, David J. Lacey, "Assertion-Based Design", Kluwer Academic Publishers, 2nd edition, 2004.

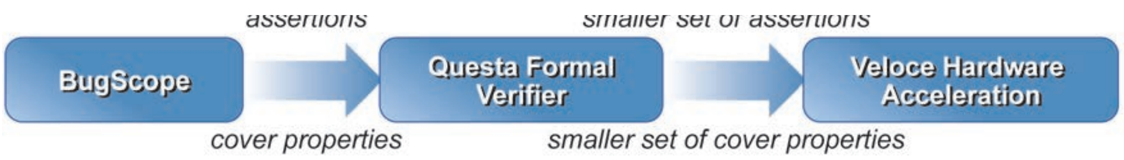
[2] Yunshan Zhu, Yuan Lu, "Assertion Synthesis: Enabling Assertion-Based Verification For Simulation, Formal and Emulation Flows", Whitepaper, http://www.nextopsoftware.com.

[3] "Transaction-based Simulation Acceleration Software - TestBench Xpress", http://www.mentor.com/products/fv/emulation-systems/veloce/testbench-xpress

[4] Alan J. Hu, Masahiro Fujita, and Chris Wilson, ``Formal Verification of the HAL S1 System Cache Coherence Protocol,'' IEEE International Conference on Computer Design (ICCD), pp.438--444, 1997.

[5] Jing Li, Nantian Qian, Yuan Lu, "Linking Multiple Verification Flows Using Automatically Generated Assertions", DVCon, 2011.

Figure 3 Assertion Synthesis Driven Hardware Acceleration Flow



assertions
smaller set of assertions
BugScope
Questa Formal Verifier
Veloce Hardware Acceleration
cover properties
smaller set of cover properties

Table 1 use Questa Formal Verified to Reduce # of Assertions on Veloce

Block Name	#FFs	#Assertions	#Proven Assertions	#Checkin Assertions
BLOCK1	10330	128	29	99
BLOCK2	20783	101	12	89
BLOCK3	169518	68	37	31
BLOCK4	69566	65	7	58
BLOCK5	270197	362	85	277

Benchmarking Functional Verification

by Mike Bartley and Mike Benjamin, Test and Verification Solutions

INTRODUCTION

This article describes "asureMark™ " - the Functional verification Capability Maturity Model (FV-CMM™) benchmarking process developed by TVS to help the user measure the maturity of their verification processes and to provide a framework for planning improvements.

When describing any activity it is important to clearly define its purpose. In this case we needed to understand how our customers benefit from applying benchmarking:

1. The constant increase in complexity of electronics means that functional verification faces an ever growing challenge. Hence it is essential not only to consider today's challenges but anticipate the future. Companies that are often in crisis because their management has been effectively ambushed by this constant march of verification complexity. Companies therefore need a process that can give them a clear warning before things go wrong!

2. Functional verification requires a vast amount of resources of all kinds: people, machines and EDA licenses. Even more importantly it has a major impact on project timescales. Yet often engineers and management in companies have very different perceptions of current capabilities and fail to identify or address key areas of weakness.

3. A process of continuous improvement needs a shared 'language' and framework that can be used to identify issues, then define, prioritize and monitor tasks. This is a key requirement for companies to ensure they will continue to be able to meet future verification challenges.

Over the years there have been numerous attempts to develop benchmarking methodologies. One of the most widely used is the Capability Maturity Model (CMMI) developed by the Software Engineering Institute at Carnegie Mellon University. Although aimed at software engineering it provides a framework that is widely applicable to most business activities. However, whilst we have drawn considerable inspiration from CMMI, it has a number of serious limitations when trying to use it to benchmark a highly specific activity such as functional verification:

The FV-CMM™ is a framework for benchmarking functional verification capability which provides:
- An integrated view of the organization from the viewpoint of functional verification
- An objective benchmark for measuring the maturity of functional verification activities
- A framework for process improvement that can help management define goals and priorities

1. The CMMI is relatively abstract and does not address domain specific 'capabilities', yet these are at the heart of effective functional verification[1]
2. Deploying CMMI is actually quite an involved process that takes considerable time and expertise. Even reading the specification is quite a lengthy commitment. Our experience suggested that this would be a major barrier to adoption.
3. Function actually follows form. The capabilities of teams are largely shaped by their organization and practices. Imposing a rigid benchmarking process can over time distort an organization and prevent necessary change. Hence any benchmarking process needed to be flexible in order to meet the current and future needs of different companies.

Much the same observations have been made independently by other industry experts (Foster & Warner, 6/2009). For the above reasons we aimed to develop a more specific, but flexible and light-weight process dedicated to benchmarking functional verification. The FV-CMM™ is a framework that provides a light weight solution for benchmarking functional verification capability which can provide:

- An integrated view of the organization from the viewpoint of functional verification
- An objective benchmark for measuring the maturity of functional verification activities

- A framework for process improvement that can help management define goals and priorities

Whilst it has some similarities to the 'Evolving Capabilities Model' Foster and Warner proposed it has a unique approach to decomposing capability in a 'top down' fashion and then evaluating maturity 'bottom up'. The rest of this article describes the three key elements of this benchmarking process: capability, maturity and the actual benchmarking process that TVS adopts.

CAPABILITY

The FV-CMM™ benchmark has a hierarchical structure that starts by breaking capability down into key process areas such as 'functional verification planning and scenario creation'. These can be customized for each client as a company developing interface IP will face different challenges to one developing CPUs or doing SoC integration. The process areas may also change over time as companies evolve and technology continues to develop. The only requirement is that each should have a clearly defined purpose and a clear impact on functional verification. We have so far defined 13 possible process areas ranging from 'metrics, coverage and closure' through 'specification and design' to 'organizational capability'.

Each process area consists of a set of specific goals (e.g. 'ensure the integrity of the code base') and practices (e.g. 'all tasks should have an agreed completion date') that capture key requirements. For example in the case of 'specification and design' the specific goals and practices for functional verification are:

- Give the verification team visibility of the architecture and micro-architecture corner cases
- Make the design 'verification friendly'
- Make the design stable to ensure verification isn't trying to hit a moving target

These in turn are broken down into example actions and activities that address that issue. These are not intended to be exhaustive but do serve to connect the abstract framework to concrete actions. For example design stability includes 'checking whether the project enforces a process of successively freezing the RTL'. This structure can easily be customized to the specific needs of different application domains, different design styles or different companies.

MATURITY

When evaluating maturity we consider three aspects:

Ownership: this can vary from tasks, tools and expertise being specific to named individuals to ownership being shared across the project or the entire company wide community. This corresponds to the level at which: adoption has occurred, decisions are made, or support can sensibly be requested. This also reflects the process for continuous improvement that can vary from best practice being owned by individuals who implement improvements in an ad hoc fashion to institutionalized fact based learning.

Visibility: this can vary from undocumented, with no external input, to living documentation with quantitative metrics and full involvement of the stakeholders. It involves the following three key aspects: the availability of documentation, the use of metrics for measuring progress and quality, and the use of reviews.

Execution: this can vary from ad hoc and incomplete to a repeatable process supporting fact based continuous improvement. Typical characteristics of a repeatable process are documentation and automation.

The maturity of each aspect is defined as being at one of five possible levels. Each of these levels corresponds to a clear step in maturity. These are:

Initial: Processes are typically ad hoc and applied incompletely or on a best effort basis, especially in times of crisis. Goals are often not satisfied. Processes are typically not documented or otherwise made repeatable and best practice remains in the ownership of individuals rather than being captured by the organization. Verification planning is either not performed or is performed and not documented, or plans are incomplete and not maintained once written. Stakeholders are not normally involved in the planning.

Managed: The processes are performed consistently and the goals are satisfied. Processes are owned and aligned at project level. They are automated, or otherwise repeatable, and will serve to locally capture best practice. However there are few specific checks on the capabilities of tools and processes. Initial verification planning is performed and documented but the plans are not maintained. Metrics are used to demonstrate progress (scenario completion, code coverage, bug rate) but not to check that the plan has been implemented. The status of the work is only visible to management at defined points and the predictability of verification completion is weak.

Defined (also known as ´Planned‘): The processes are planned in conjunction with the relevant stakeholders. Implementation is adequately resourced. The verification plan is either maintained over the life of the project or is a living plan. In either case there are checks or coverage metrics allowing the results to be monitored and reviewed. The capability of specific processes and tools is reviewed qualitatively to ensure good alignment with tasks. The predictability of verification completion is strong. Best practice is consistently shared across projects.

Quantitatively Managed: Using metrics and profiling. Living documentation ensures full visibility at all times and ensures the widest possible involvement of stakeholders in the verification process.

Optimizing: The organization practices fact based learning and continuous improvement at an institutional level using data collected across the organization and projects. Quantitative metrics are used for both coverage closure and continuous improvement of product, tools, process and organization.

Table 1 below details how the five maturity levels map onto three aspects of ownership, visibility and execution

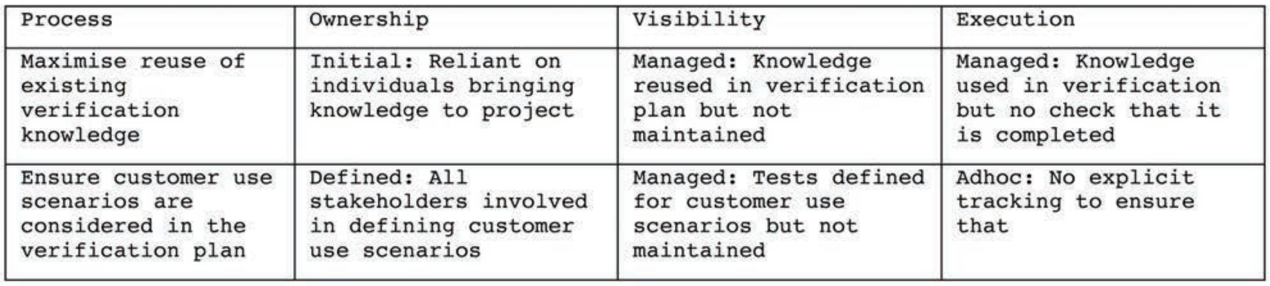
Process maturity is not a substitute for skilled and dedicated Engineers but it will make the work of those individuals more predictable and repeatable, and make it easier for the organization to learn from best practice.

PROCESS

Evaluation against the FV-CMM™ benchmark proceeds ‘bottom up’ using the example actions and activities to

Table 1: Maturity levels for the three key aspects

	Initial	Managed	Defined	Quantitative	Optimising
Ownership	Individual	A Project Team. Normally belonging to one group and located on a single site.	Project Stakeholders, possibly spread across different groups and sites, or ad hoc groups of projects	A community, normally spread across multiple sites, independent of any specific project. (Participation is normally voluntary.)	Company wide or institutionalised. Part of the formal processes and organisation defined by the company. (Participation is normally compulsory.)
Visibility	Undocumented. No reviews. No metrics.	Documents incomplete or unmaintained. Progress metrics defined. Point reviews (typically at alpha, beta and first release).	Maintained docs. Regular tracking against progress metrics and at least basic quality metrics.	Living docs. Continuous tracking against progress metrics and quantified quality metrics.	A consistent view of all projects. Data, including progress metrics and quantified quality metrics, integrated across the organisation.
Execution	Ad hoc	Tasks performed but completion not explicitly checked	Tasks planned and implemented in a systematic fashion. Check completion of planned tasks.	Quantifiable metrics used for coverage closure and release determinism	Quantifiable metrics used to drive continuous improvement.



Process	Ownership	Visibility	Execution
Maximise reuse of existing verification knowledge	Initial: Reliant on individuals bringing knowledge to project	Managed: Knowledge reused in verification plan but not maintained	Managed: Knowledge used in verification but no check that it is completed
Ensure customer use scenarios are considered in the verification plan	Defined: All stakeholders involved in defining customer use scenarios	Managed: Tests defined for customer use scenarios but not maintained	Adhoc: No explicit tracking to ensure that

Table 2: Example application of FV-CMM™ to UVM adoption

structure evidence gathering. This typically takes the form of in depth interviews with key project or department staff including verification managers, design managers and project managers as well as key verification experts. This may be backed up by reviewing project documents and data but it differs in subtle ways from an audit. Here the intention is to facilitate discovery and draw out the collective knowledge of the team rather than enforce practices. The observations are recorded and validated by being fed back for comment to the interviewees and other relevant staff. The reviewers then use their expertise and this evidence to 'score' the maturity of each of the three key aspects of ownership, visibility and execution for the associated goal or practice. Overall maturity is then evaluated based on the maturity of the three component aspects. Rather than impose an arbitrary algorithm we make this a subjective process, the only restriction being that the overall rating can't exceed the range set by the individual aspects, hence three wrongs can't make a right! The results for the individual goals or practices are in turn are used to guide the overall evaluation of each process area. All these results are captured in a single easily accessible spread sheet and can be made even more visible through the use of spider graphs to present the key results. Sometimes there is a mismatch in perception between various team members, or between engineers and management. This can be identified by following a 360 feedback process where staff, as well as the reviewers, score the maturity of the different process areas.

Whilst this evaluation is partially subjective the evidence based 'bottom up' flow aims to ensure the conclusions are fact based. By defining target maturity levels appropriate to the business and its future product roadmap a gap analysis can be conducted. The results can then be used to identify key issues and plan improvements in either specific processes or in overall functional verification maturity. Regular reviews against this model can ensure the organization maintains an appropriate level or help drive a process of continuous improvement, though subsequent audits should aim to apply common standards for evaluating maturity.

ASUREMARK™ IN ACTION: APPLYING FV-CMM™ TO UVM ADOPTION

TVS is not able to talk in detail about the application of FV-CMM™ with customers. Instead, this paper will discuss how it is applied to a company considering adoption of UVM [2] (the Universal Verification Methodology). UVM is being developed within Accellera's Verification Intellectual Property Technical Subcommittee[3] and is supported by Mentor Grtableaphics, Cadence and Synopsys. It is gaining rapid widespread adoption within the verification community but, in the experience of TVS, mere adoption of a constrained random methodology such as UVM will not necessarily lead to verification improvements. The FV-CMM™ benchmarking process will enable a company to understand better it's readiness for the adoption of UVM.

For example, 'functional verification planning and scenario creation' is an important process area within constrained random verification. This process has a number of goals such as 'Ensure the widest possible input into verification planning' and 'Make verification plan and its scenarios visible' which break down into practices. The table above considers two of the practises that contribute the first of these two goals.

We have found that the output of the benchmarking process is best presented as a spider diagram such as the one shown in Figure 1 on the following page. The figure shows three assessments: internal, external and a target assessment. Having an internal and external assessment captures the differences in perceptions between the internal

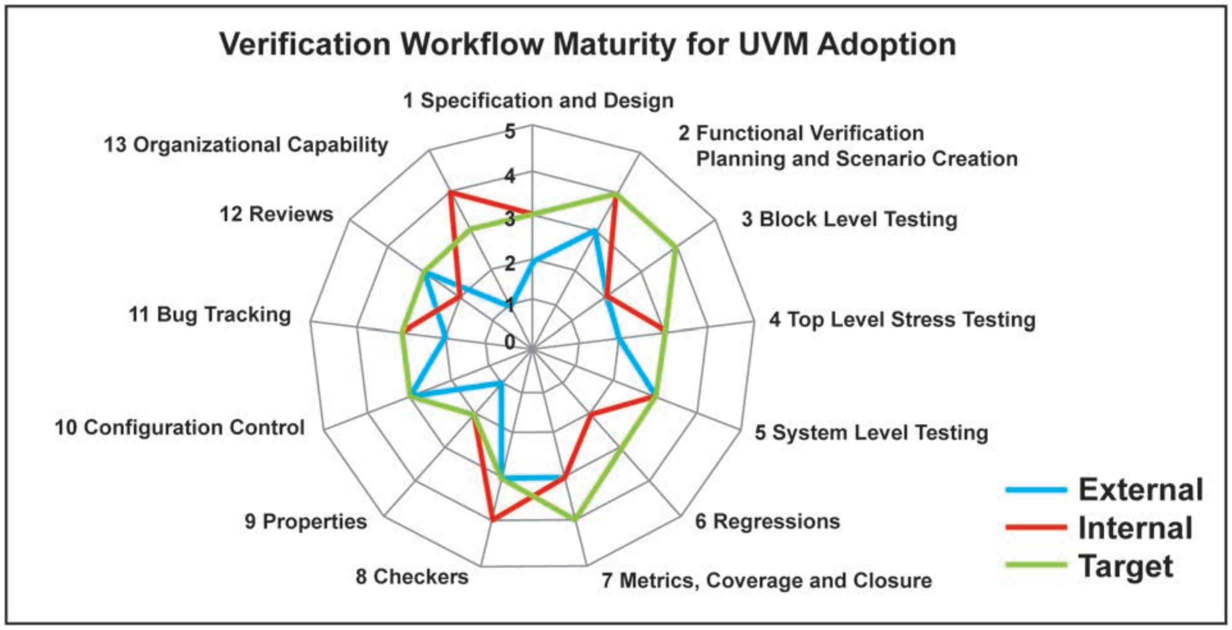


Figure 1: Spider diagram for verification maturity

team and the external TVS consultants which can lead to very valuable discussions. Finally, the target assessment allows for improvement plans to be defined in terms of the goals and practises required in order to maximise the benefits of UVM adoption.

SUMMARY

The FV-CMM™ is a flexible, light-weight benchmarking process specifically targeting functional verification. This avoids some key limitations of a more general framework such as CMMI.

Capability is captured top down by identifying process areas which are then decomposed into goals and practices that can then be linked to example actions and activities that connect the benchmarking process to concrete actions.

Evaluation then proceeds bottom up by considering ownership, visibility and execution. Maturity is rated using five clearly distinct levels from 'ad hoc' to 'Optimizing'.

Doing a gap analysis against the business requirements helps TVS' customers identify weak areas of their verification process in a timely fashion and the FV-CMM™ also provides a framework for planning improvements.

WORKS CITED

Foster, H., & Warner, M. (6/2009). Evolving Verification Capabilities. Verification Horizons.

NOTES

[1] For this reason software testing has developed the domain specific ´Test Maturity Model Integration' (TMMi)

[2] See http://www.uvmworld.org/

[3] See http://www.accellera.org/activities/vip

ABOUT THE AUTHORS

Mike Bartley and Mike Benjamin have over 35 years of hardware verification between them gained at STMicroelectronics, Infineon and start-up fabless companies. They have both worked as verification engineers, team leaders, led central functional teams and worked as consultants advising on company-wide verification strategy. They have experience in verifying blocks, CPUs, systems and SoC's and have applied various techniques and tools including formal, constrained random, assertion-based, OVM, testbench qualification, etc.

Both authors work with TVS, an independent hardware verification consultancy that provides both high-level consultancy on verification strategy and verification execution resources in several locations around the world (www.tandvsolns.co.uk).

Universal Verification Methodology (UVM)-based SystemVerilog Testbench for VITAL Models

by Tanja Cotra, Program Manager, HDL Design House

With the increasing number of different VITAL model families, there is a need to develop a base Verification Environment (VE) which can be reused with each new VITAL model family.

UVM methodology applied to the SystemVerilog Testbench for the VITAL models should provide a unique VE. The reusability of such UVM VE is the key benefit compared to the standard approach (direct testing) of VITAL models verification. Also, it incorporates the rule of "4 Cs" (Configuration, Constraints, Checkers and Coverage). Thus, instead of writing specific tests for each DUT feature, a single test can be randomized and run as part of regression which speeds up the collection of functional coverage.

The results show that UVM VE testbench, with respect to a standard direct test bench, requires nearly equal time to develop. In return it provides re-usability and much faster verification of each new VITAL model. The changes one needs to do are mainly related to the test where the appropriate configuration must be applied.

The prevailing method in verification of VITAL models was based on the use of a direct testbench where we can point out two basic problems:

1) For each model, a new testbench needs to be developed, which is time consuming, and may only be used with that specific VITAL model.

2) The direct testing does not provide functional coverage information as the main parameter for the overall verification progress.

In order to overcome the issues above, the answer was to migrate to UVM which is based on the well proven OVM Verification Methodology. The UVM methodology applied to the SystemVerilog Testbench for VITAL models should provide a unique VE that can be reused later with minimal changes.

The initial version of the SystemVerilog VITAL testbench, which is based on UVM, is intended for verification of serial flash family of VITAL models. One serial flash VITAL model contains a set of specific instructions which is common for all models belonging to the serial model family.

Having reusable verification components, we can significantly reduce the time needed to set the environment for verification of each new serial flash model. By incorporation of "Three Cs" rule (Constraints, Checkers and Coverage), instead of writing specific tests for each DUT feature, a single test can be randomized and run as part of a regression which speeds up the collection of functional coverage.

The main parts of our UVM environment are:

- Top module
- Test
- Configuration class
- Testbench class
- Environment class

The top module instantiates device-under-test (DUT) and DUT interface, used for connecting the VE with the DUT. Also, the top module generates the necessary clock signal and calls the predefined UVM task run_test(), which is used for running the specific test. The name of the specific test must be provided on the command line.

Since all verification components are defined as classes, they cannot be directly connected to the actual DUT interface but rather through the construct of virtual interface. The virtual interface is a SystemVerilog type and it is instantiated inside a specific VE component which has the need to access some signals on the actual interface (such components are the driver and monitor, for example). The actual interface is made visible to all components through the use of a predefined configuration table.

The main purpose of the configuration table is to parametrize the VE components so they can be easily customized from the specific test. Since UVM does not allow the interface to be directly added to the configuration table, a wrapper is defined around each interface.

This wrapper is stored in the configuration table by using the set_config_object() method. This enables every component, which needs access to the interface, to be able to retrieve it from the configuration table by using the get_config_object() method.

The wrapper is a user defined class which extends uvm_object. It contains the instance of the virtual interface and its constructor takes the virtual interface as an argument. At the top level, where the actual interface is instantiated, this wrapper is also instantiated where, its constructor is called and the interface is passed in as an argument. This wrapper is added to the configuration table. Code example would be:

```
class dut_if_wrapper extends uvm_object;
   virtual dut_interface dut_if_vi;
   function new (string name, virtual dut_interface arg)
       super.new();
       dut_if_i = arg;
   endfunction: new
endclass

module top;
    ...
    dut_interface dut_if ();

    initial begin
       dut_if_wrapper if_wrapper =
           new("if_wrapper",dut_if);
       set_config_object("*","dut_if_wrapper",
            if_wrapper,0);
      ...
     run_test();
   end
endmodule
```

The set_config_object() sets the object into the configuration table. The first argument of the set_config_object() method is *, which is the hierarchical path of the VE component for which we are setting information. In this case, the wildcard * makes the DUT interface available to the entire VE. The second argument is the name of the configuration parameter that we are setting. The third argument is the value of that parameter. The last argument determines if we are adding only the reference to the object (0) or to the actual object (1). The configuration parameter stored can then be retrieved inside the build() method of any component by using the get_config_object(). In this way, every component gets access to the actual DUT interface. This verification methodology simplifies accessing DUT's signals and significantly improves the verification environment's reusability.

Each test contains an instance of the configuration class and testbench class. Inside the test, the object of the configuration class is constructed, and if necessary randomized. Then it is added to the configuration table.

The configuration, as "4th C", is implemented through a configuration class. The fields of this class represent the values which will change when a new serial flash VITAL model appears. In this case, the fields are timing parameters (setup and hold times, for example) which will change with each new model. Similar to the interface wrapper, set_config_object() and get_config_object() are also used for this class, so that any component in the environment can retrieve it, if needed. It is set during the build phase of the test, so the new DUT will only be required to set different configuration parameters for the test without changing the rest of the environment.

The testbench class, consists of the following blocks:

1) Scoreboard
2) Coverage collector
3) Environment

Inside the build_phase() method, these three components are created. Also, the testbench extends two predefined methods:

```
connect_phase() -
   to subscribe the scoreboard and coverage collector
   to monitor analysis port
end_of_elaboration_phase() -
   to set report verbosity level and to print
   testbench topology.
```

To provide checkers and coverage, two components are subscribed to the monitor: scoreboard (which performs data checks) and coverage_collector (which contains SystemVerilog coverage groups).

The scoreboard class is defined by extending the base class uvm_scoreboard. The scoreboard represents a type of data checker as it checks for the written data integrity, erasure of particular locations, erasure of whole memory, read operation results, register access, etc.

The scoreboard collects information on the DUT's inputs. Depending on the data driven into the DUT, the DUT's functional specification and the current configuration, the expected output data is calculated and placed in the scoreboard list. When the output data from the DUT is collected, the data checker part of the scoreboard checks whether the DUT's output data matches the expected scoreboard's data. The scoreboard is a crucial block since it models the DUT behavior.

The scoreboard contains the memory model, resembling its organization (banks, sectors, subsectors). This memory model is preloaded with the same data as the DUT.

The scoreboard receives the monitor's transactions. When data is being written into a memory location, the monitor sends this data to the scoreboard, which in turn provides that this data is written to the memory model.

In the case of reading data, the data checker compares the DUT's data with the memo model's scoreboard data.

During the erase operation, the erase checker checks if both the memory model and the DUT's memory were erased.

This testbench environment allows for functional coverage to be collected by the use of a coverage collector component. The coverage collector uses the collected bus transactions from the monitor and checks if all of the DUT's features are covered. Coverage groups include different coverage items, checking if all of the cover items relevant combinations were covered by the transactions.

For example, coverage group read_cg checks if all locations have been read; coverage group write_cg checks if all locations have been programmed.

All of the necessary coverage groups are defined within the verification environment in order to check the device's functionality.

The environment class instantiates only the agent component. Since data needs to be driven to the DUT, this environment requires an active agent which instantiates the sequencer, driver and monitor.

The DUT being verified with this VE is a memory with Serial Peripheral Interface (SPI). The main transaction class, by which all sequences are parametrized, is defined so that it contains all the necessary information for memory access (instruction_type, address, data, etc.). It has constraints to keep the values inside allowed ranges as specified by the DUT protocol. For example, the number of address bytes sent to the DUT depends on the instruction. If READ is issued, constraints keep the number of address bytes according to protocol to either 3 or 4.

The constraints are an important part of the environment. They are specified inside the main transaction, and also inside sequences and tests. The idea is to set constraints in the test on a specific sequence, and then all lower constraints (inside sequences and the main transaction) are set automatically.

All sequences are written and stored inside a separate file called seq_lib.sv. Inside the specific test one or more sequences from the seq_lib file are created, randomized if necessary, and started on the sequencer. For example, the read sequence contains the fields: address, number_of_address_bytes and number_of_read_bytes. These fields are prefixed with rand, but also they are kept in reasonable range by using constraints. This approach enables the creation of a smaller number of sequences with random fields instead of manually writing a large number of specific sequences in a standard testbench.

When a new serial flash VITAL model appears, the changes can be made from the test by setting the appropriate configuration. For example, a new serial flash model will have different timing parameters. From the test, the inline constraint is used to set appropriate parameters:

```
        function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            spi_configuration =
spi_configuration_c::type_id::create("spi_configuration");
                    ...
            assert(spi_configuration.randomize() with { ... });
            set_config_object("*", "spi_configuration",
spi_configuration, 0);
            spi_configuration.print();
        endfunction: build_phase
```

Basically, all parameters that change with th new serial flash model, can be added to the configuration class and set appropriately from the test. Although this VE cannot be completely reusable with other families of VITAL models, it still offers a certain amount of reusability through UVM features like instance and type overrides.

The results show that this kind of testbench initially requires more time to develop, but in return provides re-usability and much faster verification of each new serial flash VITAL model.

Efficient Failure Triage with Automated Debug: a Case Study

by Sean Safarpour, Evean Qin, and Mustafa Abbas, Vennsa Technologies Inc.

Functional debug is a dreadful yet necessary part of today's verification effort. At the 2010 Microprocessor Test and Verification Workshop experts agreed that debug consumes approximately one third of the design development time. Typically, debugging occurs in two steps, triage and root cause analysis. The triage step is applied when bugs are first discovered through debugging regression failures. This occurs at the chip, system or sub-system level, where a verification engineer will group different failures together and perform an initial investigation to determine which engineers should deal with the bug. We refer to this as the triage step due to its similarity to how hospital emergency rooms assess incoming patients and determine the next step for treatment. Once the bug has been passed on, the root cause analysis step begins. Here, the engineer responsible for the bug will determine its full cause and how to fix it. Both triage and root cause analysis must be performed accurately and efficiently to result in an effective debug process. This article focuses on the often neglected pain of triage. It presents a case study where a UVM verification environment is enhanced with powerful automation tools to improve the overall debug effort. More specifically, the use of Vennsa's OnPoint and Mentor Graphics' Questa suite of tools can distinguish between multiple error sources in the design and the testbench as well as reduce the debugging time by combining different failures of the same error source.

INTRODUCTION

Consider the following scenario: a verification engineer comes to work and his first task is to go through the previous night's simulation regression failures. He must sort through the failures based on the simulation log files and error messages to determine the appropriate engineers to assign the failures to. Identifying the rightful owner without wasting other engineers' time and without submitting duplicate failures greatly affects the debug efficiency of the entire team. More specifically, the verification engineer must answer the following questions as accurately as possible.

- Which of the failures are due to the same error sources?
- Which of the failures are due to distinct error sources?
- Which of the failures are "new" bugs and have not been filed?
- Which of the failures have already been filed as bugs but have not been fixed yet?
- Who is the rightful owner of the block to assign the failure to?

Answering these questions is difficult because there is limited visibility to the design's inner workings; which paths are taken, what conditions are activated, and where the bug source originates. For example, determining whether two failures are due to the same error source cannot be known with full confidence until detailed root cause analysis is performed and the bug has been removed. In other words, this is a "catch 22" problem: triage cannot be performed perfectly until the bug is removed, and debug cannot be done efficiently until triage is performed.

To understand the inefficiencies stemming from triage it is worth looking at three typical cases. In case 1, after triage, the failure is forwarded to an engineer to perform root cause analysis and remove the bug. After hours of in-depth analysis, this engineer realizes that the problem does not originate in his/her block but from another block. This case is typical and can consume many engineering resources until the rightful owner is identified and the problem is corrected.

In case 2, shown in Figure 1 on the following page, there are two distinct bugs in the design. Both bugs are caught by a single checker in multiple simulation failures (i.e. multiple failing log files with a single type of error message). A checker is any mechanism that can distinguish between the correct and buggy behaviour of the design such as a monitor, comparator, scoreboard, or assertion. In this case, without looking into the design, tracing the signals and doing root cause analysis, one cannot determine that there are two distinct bugs in the design. As a result, assuming

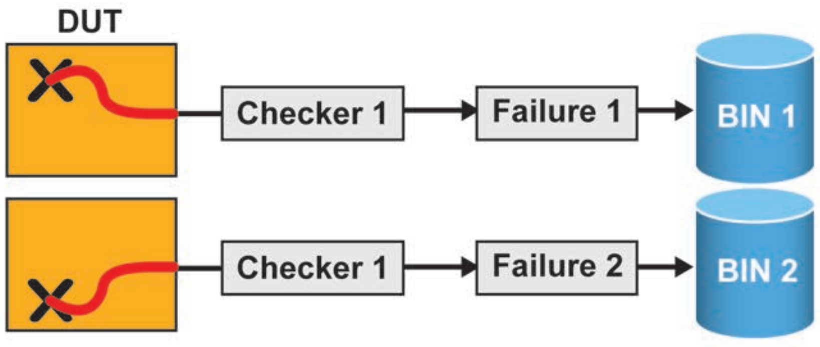


Figure 1: How 2 distinct bugs can fire the same checker

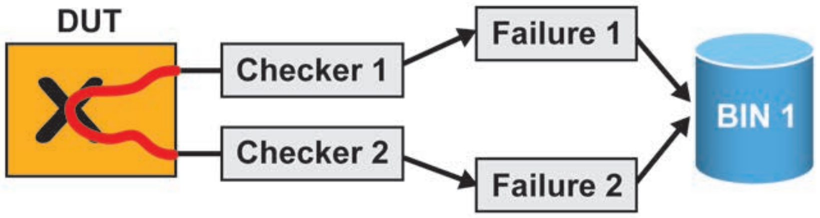


Figure 2: How the 1 bug can fire two different checkers

that there is only a single bug source, one of the failures is diagnosed and fixed. The other bug may not be caught and fixed until a few days later when the first fix has been applied.

Case 3, shown in Figure 2 above, contains a single bug source which is caught by two different checkers in different tests. For example, in one test an assertion may catch the bug while in another a scoreboard checker may catch it. In this case, the problem is that during triage one cannot tell that the failures are caused by the same error source. As a result, both failures will be forwarded to one or more engineers to do the root cause analysis only to find out that they are looking for the same bug.

Fortunately, there is an automated solution that can address the triage problem. This article presents a case study where Vennsa's OnPoint and Mentor Graphics's Questa are used together to automate the triage task in a simulation based verification environment. The solution proposed answers the major questions posed by triage engineers and addresses the difficulties of the three cases described previously. The next section provides an overview of the design and testbench used in the case study, followed by a description of the failing test cases. The remainder of the article details the proposed triage solution followed by a discussion of the superior results achieved.

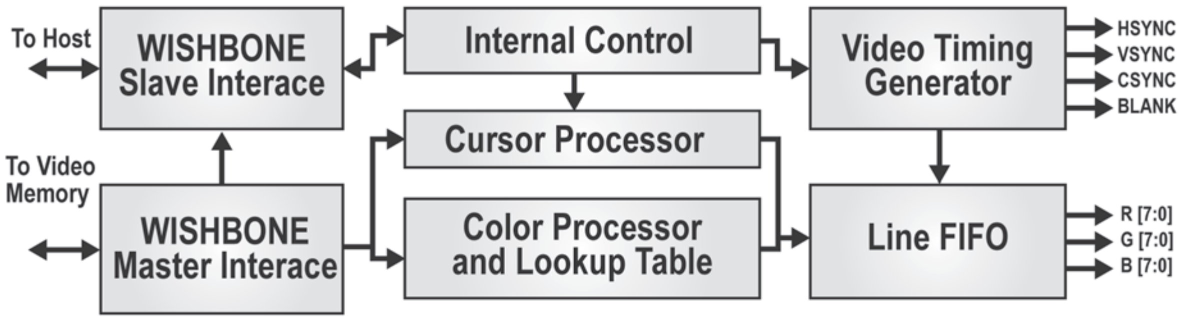
DESIGN AND TESTBENCH OVERVIEW

The following two sections provide an overview of the design used and its verification environment. The design is a VGA core and the verification environment is constructed using Unified Verification Methodology (UVM).

THE DESIGN

The design used in this case study is a VGA/LCD controller core written in Verilog that is composed of 17 modules totalling 4,076 lines of code and approximately 90,000 synthesized gates. The controller provides VGA capabilities for embedded systems. The architecture consists of a Color Processing module and a Color Lookup Table (CLUT),

Figure 3: VGA Core High Level Functional Block Diagram



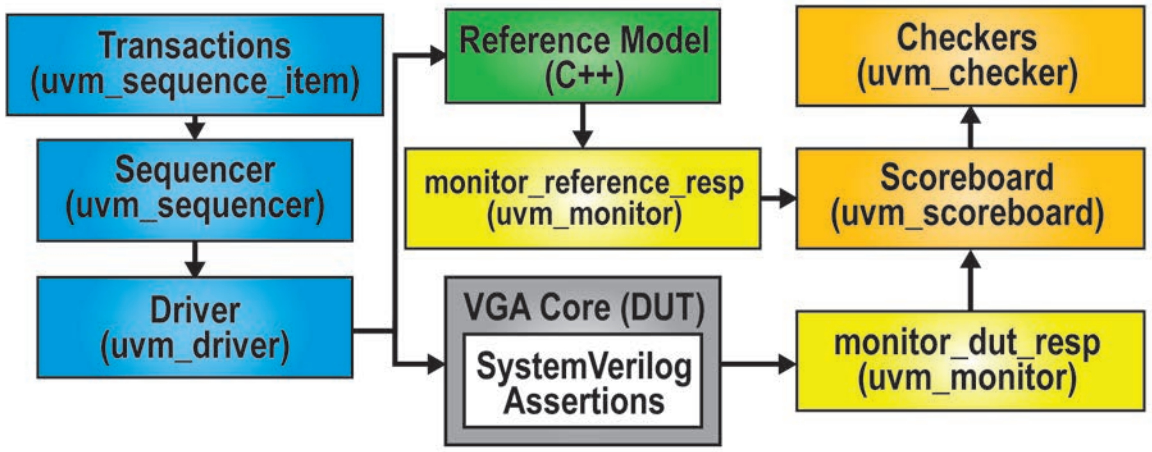

Transactions (uvm_sequence_item)
Sequencer (uvm_sequencer)
Driver (uvm_driver)
Reference Model (C++)
monitor_reference_resp (uvm_monitor)
VGA Core (DUT)
SystemVerilog Assertions
Checkers (uvm_checker)
Scoreboard (uvm_scoreboard)
monitor_dut_resp (uvm_monitor)

Figure 4: Verification Environment for the VGA Core with UVM classes

a Cursor Processing module, a Line FIFO that controls the data stream to the display, a Video Timing Generator, and WISHBONE master and slave interfaces to communicate with all external memory and the host, respectively. A diagram outlining the major system components is outlined in Figure 3 at the left.

The operation of the core is straight forward. Image data is fetched automatically via the WISHBONE Master interface from the video memory located outside the primary core. The Color Processor then decodes the image data and passes it to the Line FIFO to transmit to the display. The Cursor Processor controls the location and image of the cursor processor on the display. The Video Timing Generator module generates synchronization pulses and interrupt signals for the host.

THE TESTBENCH

The test suite for the VGA core is constructed using UVM. Figure 4 above describes the structure of the verification environment. The testbench consists of the following major components:

- uvm_sequence_item – generates the stimulus using random stimulus constraints
- uvm_sequencer – sets up the sequences of inputs for the test
- uvm_driver – sends the stimulus to the design under test (DUT)
- uvm_monitor – collects and monitors the output/response
- uvm_scoreboard – checks the correctness of the outputs

Four main tests are introduced for testing this design. These include register, timing, pixel data, and FIFO tests. The transaction (uvm_sequence_item class) has randomly generated control-data pairing packets under certain constraints. These transactions are expected to cover all the VGA operation modes in the tests (and they may be reused to test other video cores such as DVI, etc). The sequencer (uvm_sequencer class) exercises different combinations of these transactions through a given testing scheme so that most corner cases and/or mode switching are covered. The monitors (uvm_monitor class) are connected to the DUT and the reference model respectively. They check the protocols of the responses, and make sure that the data being sent to scoreboard has correct timing. The scoreboard (uvm_scoreboard class) and checkers (uvm_checker class) contain all the field checkers which compare the data from the DUT, and reports the mismatches.

The golden reference model is implemented using C++. It receives the same set of stimulus from the driver (uvm_driver class) and produces the expected value of the outputs. Along with the reference model, 50 SystemVerilog Assertions (SVA) are also used to do some instant checks. While running simulation, SVA can catch unexpected behaviours of the design and prevent corrupted data going through the flow.

```
. . .
# ******************************
  **********************
# *** Register Test                              ***
# *****************************************************
#
# Testing Reset Values ...
# Testing Pattern R/W ...
#
# ERROR @ Time:   412 ns, Location WISHBONE _ SLAVE _ DATA port wbs _ dat _ o[31:0] REFERENCE
  0x00000000 OBSERVED 0xffffffdf
#
. . .
# ERROR @ Time:  1005 ns, Location SYNC _ VSYNC port vsync _ pad _ o REFERENCE 0b0 OBSERVED 0b1
#
. . .
#
# ERROR @ Time: 10831 ns, Location WISHBONE _ SLAVE _ DATA port wbs _ dat _ o[31:0] REFERENCE
0xffffff9f OBSERVED 0xffffffaf
#
. . .
#
# ** Error: Assertion error.
#    Time: 12831 ns Started: 12825 ns Scope:
test.dut.wbm.data _ fifo.chk _ fifo _ i1. _ a _ read _ pointer File: /home/mustafa/regressions/
nightly/vga/sva/vga _ fifo.sv Line: 32
#
. . .
#
# ERROR @ Time: 15897 ns, Location WISHBONE _ SLAVE _ DATA port wbs _ dat _ o[31:0] REFERENCE
  0xaaaaaaaa OBSERVED 0xffffff9f
#
. . .
```

SIMULATION AND FAILURES - WE HAVE BUGS

The design and testbench are simulated with Mentor Graphics' Questa UVM and SVA support. After simulation is complete the log shows that a total of 5 failures occurred as shown in log snippet above.

We can see that the UVM checker has fired four times, three on the WISHBONE slave data port and once on the vsync port, and an SVA assertion has also fired once. Without doing any debug, the only triage that can be done at this stage is by inspecting which checkers failed and putting into bins the failures accordingly. For instance, based on the checkers, the following binning can be performed.

Failure	Time	Checker	Bin
WBS _ SLV-01	412ns	WISHBONE	1
WBS _ SLV-02	10831ns	WISHBONE	1
WBS _ SLV-03	15897ns	WISHBONE	1
SYNC _ OUT-01	1005ns	SYNC	2
SVA _ 01	12831ns	SVA _ FIFO	3

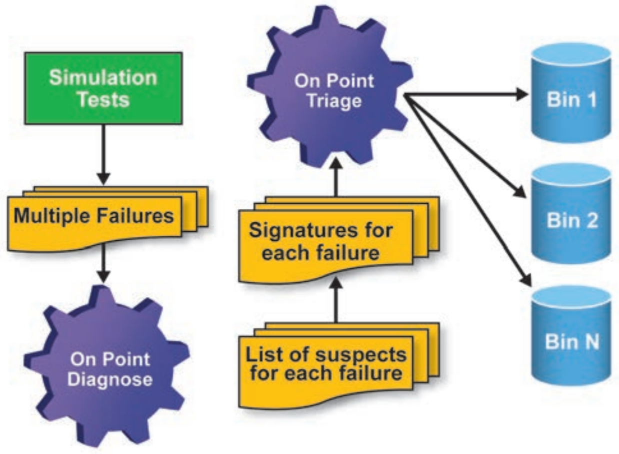
After binning is done, an engineer is assigned to each bin to further diagnose each failure and fix the problems. This binning may seem intuitive and simple to implement but, as discussed in the Introduction, there are cases where inefficiencies exist as one cannot distinguish between failure messages and the root cause of the problem.

VERIFICATION FLOW WITH VENNSA ONPOINT

Error! Reference source not found. describes the verification flow with Vennsa OnPoint's diagnose and triage engines. As usual, the design and testbench are simulated with Mentor Graphics' Questa simulator. Once a failure occurs a simulation value dump file such as wlf, vcd, or fsdb file is generated and provided to OnPoint. Vennsa's OnPoint is an automated debugger tool that analyzes a simulation failure (along with the RTL files and a functional mismatch) and determines the root cause of errors. For each failure, OnPoint's diagnose engine automatically generates a list of suspects that point to locations in the design that may be the culprit of the failure. For triage, the suspects are converted to signatures and passed to OnPoint's triage engine, which then bins similar failures together. The bins contain failures that have a high likelihood of being caused by the same bug source. The bins are then assigned to the proper engineer who will perform in depth root cause analysis and fix the error. The suspects OnPoint finds for each failure can also be used by the engineers to perform root cause analysis and fix the bug.

The flow in Figure 5 below consisting of Questa simulation, OnPoint diagnose and OnPoint triage is applied to the case study design. OnPoint's binning results are summarized below. Note that in addition to the bin number, each failure also has a hint describing whether the bug is in the RTL, the testbench stimulus side or the testbench checker side. Furthermore, the number of suspects found by the OnPoint diagnose engine is shown in the final column.

Figure 5: Verification flow with Vennsa Onpoint



```
Failure          Time          Checker          Bin      Hint          # Suspects
----------------------------------------------------------------------------------------
WBS _ SLV-01       412ns       WISHBONE          1       Stimulus       4
WBS _ SLV-02     10831ns       WISHBONE          2       RTL            35
SVA _ 01         12831ns       SVA _ FIFO        2       RTL            31
WBS _ SLV-03     15897ns       WISHBONE          3       RTL            42
SYNC _ OUT-01     1005ns       SYNC              4       Checker        10
```

Note that the binning done using Vennsa's triage engine is very different from the basic approach based on error messages alone. Next, the triage results are analyzed with a discussion detailing the reasoning behind each bin.

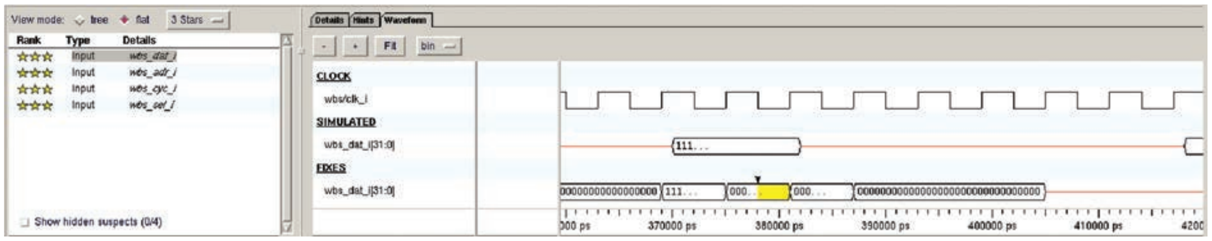
ANALYZING TRIAGE PERFORMED BY ONPOINT

In bin 1 of the OnPoint triage results, there is only one WISHBONE failure, WBS_SLV-01. This failure was binned together with the other WISHBONE failures in error message based triage. However, OnPoint's triage engine has binned it in isolation because rather than just analyzing which checker has failed, the engine analyzes the path of the suspects taken from error source to the checker boundary. In this case, it can differentiate this path against those taken in the other failures. Furthermore, OnPoint has provided a hint that this failure is likely caused by a bug originating from the testbench stimulus side due to the high number of primary input suspects found. Figure 6 shows a waveform inside the OnPoint user interface where the value and time that a fix can be applied on the primary inputs is shown. A close look at the input suspects confirms that the wrong value of 111... instead of 000... was applied to the input wbs_dat_i at time 375ns-380ns. Indeed the bug was in the testbench during the initial read write tests for the WISHBONE interface which was sent an incorrect value to the DUT.

The other two WISHBONE failures, WBS_SLV-02 and WBS_SLV-03, are also binned separately by OnPoint (bin 2 and bin 3). Again, the triage engine determines that the paths taken from the bug sources to the failure points are distinct enough to conclude that they stem from different bug sources. Figure 7 shows a side by side comparison of each failure's suspect list with OnPoint's user interface. Notice that the suspects generated for each failure are very different. Using the suspects generated, it is quickly found that the failure WBS_SLV-03 was caused by an incorrect initial register state in Color Processor, while the WBS_SLV-02 failure is caused by an incorrect assignment to a register in the FIFO module.

Bin 2 also contains the firing assertion SVA_01. Although the checkers are different, OnPoint binned the assertion

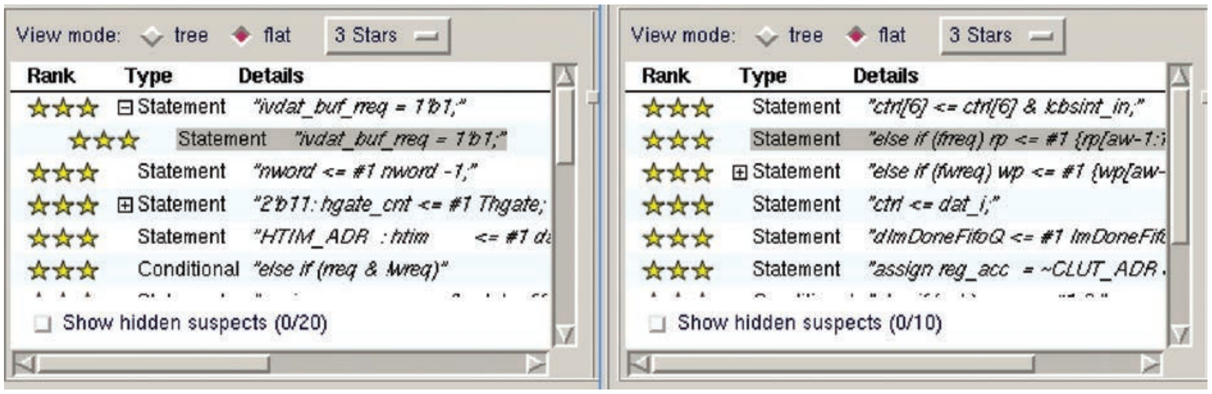
Figure 6: Input suspect view showing that wbs_dat_i is the most likely fix



Rank	Type	Details
★★★	Input	wbs_dat_i
★★★	Input	wbs_adr_i
★★★	Input	wbs_cyc_i
★★★	Input	wbs_sel_i

Show hidden suspects (0/4)

Figure 7: Suspect view for the WBS_SLV-02 (right) and WBS_SLV-03 (left) failures



Rank	Type	Details
★★★	⊟ Statement	"ivdat_buf_rreq = 1'b1;"
★★★	Statement	"ivdat_buf_rreq = 1'b1;"
★★★	Statement	"rrword <= #1 rrword -1;"
★★★	⊞ Statement	"2'b11: hgate_cnt <= #1 Thgate;
★★★	Statement	"HTIM_ADR : htim <= #1 d
★★★	Conditional	"else if (rreq & !wreq)"

Show hidden suspects (0/20)

Rank	Type	Details
★★★	Statement	"ctrl[6] <= ctrl[6] & !cbsint_in;"
★★★	Statement	"else if (fireq) rp <= #1 {rp[aw-1:'
★★★	⊞ Statement	"else if (fwreq) wp <= #1 {wp[aw-
★★★	Statement	"ctrl <= dat_i;"
★★★	Statement	"dImDoneFifoQ <= #1 ImDoneFif
★★★	Statement	"assign reg_acc = ~CLUT_ADR .

Show hidden suspects (0/10)

together with the WBS_SLV-02, after analyzing their suspects. Therefore, fixing the FIFO bug results in both the assertion and WBS_SLV-02 passing.

Finally, the failure SYNC_OUT-01 is binned on its own, similar to regular triage. However, OnPoint gives a hint that actual root cause of the failure may be in the checker itself rather than the DUT. This is because the suspects are all very close to the failure point. Indeed the bug was found in the C++ reference model that generated the wrong expected value for the checker.

CONCLUSION

Failure triage is a difficult problem. Relying on error messages to bin failures results in wasted time and lost productivity because of wrongfully associating bugs. In this article, Vennsa's OnPoint and Mentor Graphic's Questa tools are combined to develop an automated simulation and failure triage flow. A case study on a VGA controller design demonstrates that OnPoint's triage engine can provide insight into the failures thus allowing the failures to be binned appropriately without any user guidance. As a result, the efficiency of both the verification and design engineers can be significantly improved.

Are OVM & UVM Macros Evil? A Cost-Benefit Analysis

by Adam Erickson, Mentor Graphics Corporation

ABSTRACT

Are macros evil? Well, yes and no. Macros are an unavoidable and integral part of any piece of software, and the Open Verification Methodology (OVM) and Universal Verification Methodology (UVM) libraries are no exception. Macros should be employed sparingly to ease repetitive typing of small bits of code, to hide implementation differences or limitations among the vendors' simulators, or to ensure correct operation of critical features. Although the benefits of the OVM and UVM macros may be obvious and immediate, benchmarks and recurring support issues have exposed their hidden costs. Some macros expand into large blocks of complex code that end up hurting performance and productivity, while others unnecessarily obscure and limit usage of otherwise simple, flexible APIs.[1]

The 'ovm_field macros in particular have long-term costs that far exceed their short-term benefit. While they save you the one-time cost of writing implementations, their run-time performance and debug costs are incurred over and over again. Consider the extent of reuse across thousands of simulation runs, across projects, and, for VIP, across the industry. These costs increase disproportionately with increased reuse, which runs counter to the goals of reuse.

In most cases, it takes a short amount of time and far fewer lines of code to replace a macro with a "direct" implementation. Testbenches would be smaller and run faster with much less code to learn and debug. The costs are fixed and up-front, and the performance and productivity benefits increase with reuse.

This article will:

- Contrast the OVM macros' benefits (what they do for you) with their costs (e.g. inflexibility, low performance, debug difficulty, etc.) using benchmark results and code analysis.
- Identify which macros provide a good cost-benefit trade-off, and which do not.
- Show how to replace high-cost macros with simple SystemVerilog code.
- Provide insight into the work being done to reduce the costs of using macros in the UVM, the OVM-based Accellera standard verification library currently under development.

1.INTRODUCTION

The hidden costs associated with using certain macros may not be discovered until the economies of scale and reuse are expected but not realized. A VIP defined with certain macros incurs more overhead and may become more difficult to integrate in large-scale system-level environments.

The following summarizes our recommendations on each class of macros in the OVM.

Table 1. Summary Macro Usage Recommendations

'3ovm_*_utils	Always use. These register the object or component with the OVM factory. While not a lot of code, registration can be hard to debug if not done correctly.
'ovm_info | warning | error | fatal	Always use. These can significantly improve performance over their function counterparts (e.g. ovm_report_info).
'ovm_*_imp_decl	OK to use. These enable a component to implement more than one instance of a TLM interface. Non-macro solutions don't provide significant advantage.
'ovm_field_*	Do not use. These inject lots of complex code that substantially decreases performance, limits flexibility, and hinders debug. Manual implementations are significantly more efficient, flexible, transparent, and debuggable. In recognition of these faults, the field macros have been substantially improved in the UVM.
'ovm_do_*	Avoid. These unnecessarily obscure a simple API and are best replaced by a user-defined task, which affords far more flexibility and transparency.

‘ovm_ sequence-related macros	Do not use. These macros build up a list of sequences inside the sequencer class. They also enable automatic starting of sequences, which is almost always the wrong thing to do. These macros are deprecated in the UVM and thus are not part of the standard.

Application of these recommendations can have a profound effect. If the ‘ovm_field macros were avoided entirely, several thousands of lines of code in the OVM library would not be used, and many thousands more would not be generated (by the macros).

The following section describes the cost-benefit of each macro category in more detail.

2. COST-BENEFIT ANALYSES

2.1 ‘ovm_*_utils

Always use.
The ‘ovm_*_utils macros expand into code that registers the class with the OVM factory, defines the create() method, and, if the type is not a parameterized class, the get_type_name() methods. Because type registration with the factory must be performed in a precise, consistent way, and the code involved is small and relatively straightforward, these macros provide convenience without significant downside.

2.2 ‘ovm_info | warning | error | fatal

Always use.
Issuing a report involves expensive string processing. If the message would be filtered out based on the verbosity, or if it’s configured action is OVM_ACTION, all the string processing overhead would be wasted effort. These report macros improve simulation performance by checking verbosity and action settings before calling the respective ovm_report_* method and incurring the cost of processing the report.

These macros also conveniently provide a report’s location of invocation (file and line number). You can disable file and line number by overriding the ovm_report_server or by defining OVM_REPORT_DISABLE_FILELINE on the command line.

2.3 ‘ovm_*_imp_decl

OK to use.
These macros define special imp ports that allow components to implement more than one instance of a TLM interface. For example, the ovm_analysis_imp calls the host component’s write method, of which there can be only one. Multiple such ovm_analsys_imps would all call the same write method. To get around this, you can invoke the ovm_*_imp_decl macro to define an imp that calls a different method in the component. For example:

```
 ‘ovm_analysis_imp_decl(_exp)
 ‘ovm_analysis_imp_decl(_act)
class scorebd extends ovm_component;
 ovm_analysis_imp_exp #(my_tr,scorebd) expect;
 ovm_analysis_imp_act #(my_tr,scorebd) actual;
 virtual function void write_exp(my_tr tr);
     ...
 endfunction
 virtual function void write_act(my_tr tr);
     ...
 endfunction
endclass
```

Writes to the expect_ap analysis imp will call write_expect, and writes to the actual_ap analysis imp will call write_actual.

The imp_decl macros have a narrow use-model, and they expand into a small bits of code. They are OK to use, as they offer a convenience with little downside.

If you do not want to use the *_imp_decl macros, you could implement the following. Define a generic analysis_imp that takes a “policy” class as a type parameter. The imps’ write method calls the static write method in the policy class, which calls a uniquely-named method in the component. You will need to define a separate policy class for each unique instance of the analysis interface, much like what the ovm_*_ imp_decl macros do for you.

```
class aimp #(type T=int, IMP=int, POLICY=int)
    extends ovm_port_base #(tlm_if_base #(T,T));
  `OVM_IMP_COMMON(`TLM_ANALYSIS_MASK,
"ovm_analysis_imp",IMP)
  function void write (input T t);
    POLICY::write(m_imp , t);
  endfunction
endclass

class wr_to_A #(type T=int, IMP=int);
  static function void write(T tr, IMP comp);
    comp.write_A(tr);
  endfunction
endclass

class wr_to_B #(type T=int, IMP=int);
  static function void write(T tr, IMP comp);
    comp.write_B(tr);
  endfunction
endclass


class my_comp extends ovm_component;
  aimp #(my_tr, my_comp, wr_to_A) A_ap;
  aimp #(my_tr, my_comp, wr_to_B) B_ap;
  virtual function void write_A(my_tr tr);
   ...
  endfunction
  virtual function void write_B(my_tr tr);
   ...
  endfunction
endclass
```

2.4 *'ovm_do_**

Avoid.

The 'ovm_do_* macros comprise a set of 18 macros for executing sequences and sequence items, each doing it a slightly different way. Many such invocations in your sequence body() method will expand into lots of inline code. The steps performed by the macros are better relegated to a task.

The 'ovm_do macros also obscure a very simple interface for executing sequences and sequence items. Although 18 in number, they are inflexible and provide a small subset of the possible ways of executing. If none of the 'ovm_do macro flavors provide the functionality you need, you will need to learn how to execute sequences without the macros. And once you've learned that, you might as well code smartly and avoid them all together.

```
virtual task parent_seq::body();
  my_item item;
  my_subseq seq;
  'ovm_do(item)  <-- what do these do?
  'ovm_do(seq)   <-- side effects? are you sure?
endtask

-----------
task parent_seq::do_item(ovm_sequence_item item,...);
  start_item(item);
  randomize(item) [with { ... }];
  finish_item(item);
endtask

virtual task parent_seq::body();
 my_item item = my_item::type_id::create("item",,get_
full_name());
 my_seq  seq =  my_seq::type_id::create("seq",,get_full_
name());
 do_item(item);
 seq.start();
endtask
```

Most uses of the inline constraints seen by this author set the address or data member to some constant. It would be more efficient to simply turn off randomization for those members and set them directly using '='. Encapsulating this procedure in a task is also a good idea. A task for simple reads/writes is shown on the following page:

```
task parent_seq::do_rw(int addr, int data);
  item= my_item::type_id::create
             ("item",,get_full_name());
  item.addr.rand_mode(0);
  item.data.rand_mode(0);
  item.addr = addr;
  item.data = data;
  item start_item(item);
  randomize(item);
  finish_item(item);
endtask

virtual task parent_seq::body();
  repeat (num_trans)
    do_rw($urandom(),$urandom());
endtask
```

2.5 'ovm_sequence macros

Do not use.

The macros, 'ovm_sequence_utils, 'ovm_sequencer_utils, 'ovm_update_sequence_lib[_and_item] macros are used to build up a sequencer's "sequence library." Using these macros, each sequence type is associated with a particular sequencer type, whose sequence library becomes the list of the sequences that can run on it. Each sequencer also has three built-in sequences: simple, random, and exhaustive.

When a sequencer's run task starts, it automatically executes the default_sequence, which can be set by the user using set_config. If a default sequence is not specified, the sequencer will execute the built-in ovm_random_sequence, which randomly selects and executes a sequence from the sequence library.

These macros hard-code sequence types to run on a single sequencer type, do not support parameterized sequences, and cause many debug issues related to random execution of sequences. In practice, the sequencer can not start until, say, the DUT is out of reset. When it does start, it typically executes a specific sequence for DUT configuration or initialization, not some random sequence.

Users often spend lots of time trying to figure out what sequences are running and why, and they inevitably look for ways to disable sequence library behavior. (Set the sequencer's count variable to 0, use 'ovm_object_utils for sequences, and use 'ovm_component_utils for sequencers.)

The problems with the sequence library and related macros grow when considering the UVM, which introduces multiple run-time phases that can execute in parallel and in independently timed domains. A single, statically-declared sequence library tied to a single sequencer type cannot accommodate such environments. Therefore, the Accellera VIP-TSC committee decided to officially deprecate the sequence library and macros. The committee is currently developing a replacement sequence library feature that has none of the limitations of its predecessor's and adds new capabilities.

2.6 'ovm_field_*

Avoid.

The 'ovm_field macros implement the class operations: copy, compare, print, sprint, record, pack, and unpack for the indicated fields. Because fields are specified as a series of consecutive macros calls, the implementation of these operations cannot be done in their like-named do_<operation> methods. Instead, the macros expand into a single block of code contained in an internal method, m_field_automation. Class designers can hand-code field support by overriding the virtual methods— do_copy, do_compare, etc.. Users of the class always call the non-virtual methods—copy, compare, etc.— methods, regardless of whether macros or do_* methods were used to implement them. For example, consider the implementation of the ovm_object::copy non-virtual method:

```
 function void ovm_object::copy(...);
  m_field_automation(COPY,...); //'ovm_field props
  do_copy(...);          // user customizations
endfunction
```

The non-virtual copy first calls m_field_automation to take care of the 'ovm_field-declared properties, then calls the corresponding virtual do_ copy to take care of the hand-coded portion of the implementation.

Because of the way the ‘ovm_field macros are implemented and the heavy use of policy classes (comparer, printer, recorder, etc.), macro-based implementations of the class operations incur high overhead. The next few sections provide details on this and other costs..

2.6.1 Code bloat

Consider the simple UBUS transaction definition below.[2]

```
class ubus_transfer extends ovm_sequence_item;
   rand bit [15:0             addr;
   rand ubus_op               op;
   rand int unsigned          size;
   rand bit [7:0]             data[];
   rand bit [3:0]             wait_state[];
   rand int unsigned          error_pos;
   rand int unsigned          transmit_delay = 0;
   string                     master = “”;
   string                     slave = “”;

  `ovm_object_utils_begin(ubus_transfer)
   `ovm_field_int  (addr,               UVM_ALL_ON)
   `ovm_field_enum (ubus_op, op,        UVM_ALL_ON)
   `ovm_field_int  (size,               UVM_ALL_ON)
   `ovm_field_array_int(data,           UVM_ALL_ON)
   `ovm_field_array_int(wait_state,     UVM_ALL_ON)
   `ovm_field_int  (error_pos,          UVM_ALL_ON)
   `ovm_field_int  (transmit_delay,     UVM_ALL_ON)
   `ovm_field_string(master, UVM_ALL_ON |
                             UVM_NOCOMPARE)
   `ovm_field_string(slave,  UVM_ALL_ON |
                             UVM_NOCOMPARE)
  `ovm_object_utils_end
endclass
```

After macro expansion, this 22-line transaction definition expands to 644 lines, a nearly 30-fold increase. Real-world transaction definitions far exceed 1,000 lines of code. The following table shows the number of new lines of code that each of the ‘ovm_field macros expand into, for both OVM 2.1.1 and UVM 1.0. In UVM 1.0, the macros underwent significant refactoring to improvement performance and provide easier means of manually implementing the do_* methods.

Table 1 Macro expansion – lines of code per macro

Macro	Lines of Code OVM[3]	Lines of Code UVM[2]
`ovm_field int|object|string|enum	51,72,17,41	50,75,43,45
‘ovm_field_sarray_*	75-100	117-128
‘ovm_field_array_*	127-191	131-150
‘ovm_field_queue_*	110-187	133-152
‘ovm_field_aa_*_string	76-87	75-102
‘ovm_field_aa_object_int	97	111
‘ovm_field_aa_int_*	85	85
‘ovm_field_event	16	29

In contrast, the manual implementation of the same UBUS transaction consists of 92 lines of code that is more efficient and human-readable.

2.6.2 Low performance

The lines of code produced by the expansion of the ‘ovm_field macros do not actually do much of the actual work. That is handled by nested calls to internal functions and policy classes (e.g. ovm_comparer, ovm_printer, etc.).

Table 2 shows how many function calls are made by each operation for the macro-based solution and the equivalent manual implementation of the do_ methods. As a control, the size of the data and wait_state members were fixed at 4.

Table 2 Function calls per UBUS operation

Operation	OVM Macro/Manual	UVM Macro/ Manual
copy	38 / 9	8 / 9
compare	51 / 18	17 / 18
sprint - table	1957 / 1840	187 / 160
sprint - tree	518 / 441	184 / 157
sprint – line	478 / 405	184 / 157
pack / unpack	140 / 28	80 / 28
record (begin_tr / end_tr)	328 / 46	282 / 36

Compare these results with a theoretical minimum of one or two calls, depending on whether the object has a base class. Calling copy in a macro-based implementation incurs 38 function calls, but only 9 in a do_compare implementation—a four-fold difference. Compare incurs 51 method calls with macros versus do_compare's 18 calls. Sprinting (and printing) incur thousands of calls for each operation.

Each function call involves argument allocation, copy, and destruction, which affects overall performance. The results were alarming enough that significant effort was taken to improve the macro implementations in UVM. The UVM column shows this.

Table 3 shows the run time to complete 500K operations for the macro-based and manual implementations of the do_* methods.

Table 3 Performance – 500K transactions, in seconds[4]

Operation	OVM Macro/Manual	UVM Macro/ Manual
copy	43 / 2	8 / 2
compare	60 / 6	9 / 6
sprint - table	1345 / 1335	165 / 159
sprint - tree	215 / 165	137 / 137
sprint – line	195 / 165	137 / 132
pack / unpack	100 / 19	37 / 18
record (begin_tr/end_tr)	533 / 40	413 / 37

The poor performance results in OVM prompted a significant effort to improve them in UVM. The results of this improvement effort show that performance issues for most operations have largely been mitigated.

Amdahl's Law [5] states that testbench performance improvements are limited by those portions of the testbench that cannot be improved. Although this author still cannot recommend field macro usage over manual implementation, the macro performance improvements in UVM are very welcome because they afford significant performance improvements achievable in emulation and acceleration.

Note that the sprint times are comparable between the macro-based and manual implementations. This is because there is no equivalent manual replacement for the formatting capabilities of the printer policy class, the primary source of overhead for this method. The UVM provides an improved uvm_printer policy class that makes performance less sensitive to output format.

2.6.3 Not all types supported

The 'ovm_field macros do not support all the type combinations you may need in your class definitions. The following are some of the types that do not have 'ovm_field macro support.

- Objects not derived from ovm_object
- Structs and unions
- Arrays (any kind) of events
- Assoc arrays of enums
- Assoc arrays of objects indexed by integrals > 64 bits
- Assoc arrays—no support for pack, unpack, and record
- Multi-dimensional packed bit vectors—For example, bit [1:3][4:6] a[2]. The [1:3][4:6] dimensions will be flattened, i.e. treated as a single bit vector, when printing and recording.
- Multi-dimensional unpacked bit vectors— For example, bit a[2][4]
- Multi-dimensional dynamic arrays, such as arrays of arrays, associative array of queues, etc.

2.6.4 Debugging difficulties

The 'ovm_field (and, still, the `uvm_field) macros expand into many lines of complex, uncommented code and many calls to internal and policy-class methods.

If a scoreboard reports a miscompare, or the transcript results don't look quite right, or the packed transaction appears corrupted, how is this debugged? Macros would have been expanded, and extra time would be spent stepping through machine generated code which was not meant to be human readable.

The person debugging the code may not have had anything to do with the transaction definition. A single debug session traced to the misapplication, limitation, or undesirable side effect of an `ovm_field macro invocation could negate the initial ease-of-implementation benefit it was supposed to provide. Manually implementing the field operations once will produce more efficient, straight-forward transaction definitions.

As an exercise, have your compiler write out your component and transaction definitions with all the macros expanded.[5] Then, contrast the macro-based implementations with code that uses straight-forward SystemVerilog:

```
function bit my_obj::do_compare(ovm_object rhs,
                          uvm_comparer comparer);
   do_compare =
     ($cast(rhs_,rhs) &&
      super.do_compare(rhs,comparer) &&
      cmd  == rhs_cmd &&
      addr == rhs_.addr &&
      data == rhs_.data);
endfunction
```

2.6.5 Other limitations

The ‘ovm_field macros have other limitations:

- Integrals variables cannot exceed ‘OVM_MAX_STREAMBITS bits in size (default is 4096). Changing this global max affects efficiency for all types.
- Integrals are recorded as 1K bit vectors, regardless of size. Variables larger than 1K bits are truncated.
- The ovm_comparer is not used for any types other than scalar integrals, reals, and arrays of objects. Strings, enums, and arrays of integral, enum, and string types do not use the ovm_comparer. Thus, if you were to define and apply a custom comparer policy, your customizations.
- The ovm_packer limits the aggregate size of all packed fields to not exceed OVM_MAX_PACKED_BITS. This large, internal bit vector is bit-by-bit copied and iterated over several times during the course of the pack and unpack operations. If you need to increase the max vector size to avoid truncation, you will affect efficiency for all types.

2.6.6 Dead code

The ‘ovm_field macros’ primary purpose is to implement copy, compare, print, record, pack, and unpack for transient objects. None of these operations are particularly useful to OVM components. Components cannot be copied or compared, and pack and unpack doesn’t apply. Print for components are occasionally useful for debugging component topology at start of simulation, but you could get that and more from a GUI debugger without having to modify the source. In most cases, a simple $display(“%p”,component) would suffice.

The ‘ovm_field macros also implement a little-known feature called auto-configuration, which performs an implicit get_config for every property you declare with an ‘ovm_field macro inside an ovm_component. While convenient sometimes, it presumes all macro-declared fields are intended to be user-configurable, and you sacrifice control over whether, when, and how often configuration is retrieved. For ovm_objects, auto-config code is never used. For ovm_components, this feature incurs significant time to complete and is in many cases unwanted. To avoid this overhead, users often disable auto-config by not calling super.build() and simply call get_config explicitly for the properties intended to be user-configurable.

Despite performance improvements in UVM, the field macros still incur code bloat, performance degradation, debug issues, and other limitations. The UVM also provides small convenience macros for helping users manually implement the do_* methods more easily. For these reasons, this author continues to recommend against using the field macros.

3. ALTERNATIVE TO ‘OVM_FIELD MACROS

The following sections describe how to write implementations of copy, compare, etc. without resorting to the ‘ovm_field macros. In all cases, you override the do_<method> counterpart. For example, to manually implement copy, you override the virtual do_copy method. For UVM, change the O’s to U’s.

3.1 do_copy

Implement the do_copy method as follows:

```
1  function void do_copy (ovm_object rhs);
2    my_type rhs_;
3    if (!$cast(rhs_,rhs))
4      ‘ovm_fatal(“TypeMismatch”,”...”);
5    super.do_copy(rhs);
6    addr = rhs_.addr;
7    if (obj == null && rhs_.obj != null)
8      obj = new(...);
9    if (obj!=null) obj.copy(rhs_.obj);
10 endfunction
```

Line 1—This is the signature of the do_copy method inherited from ovm_object. Your signature must be identical.

Lines 2-4— Copy only works between two objects of the same type. These lines check that the rhs argument is the same type. If not, a FATAL report is issued and simulation will exit.

Line 5—Here, we call do_copy in the super class so any inherited data members are copied. If you omit this statement, the rhs object will not be fully copied.

Line 6—Use the built-in assignment operator (=) to copy each of the built-in data types. For user-defined objects, assignment is copy-by-reference, which means only the handle value is copied. This leaves this object and the rhs object pointing to the same underlying object instance.

Lines 7-9—To deep copy the rhs object's contents into this object, call its copy method. Make sure the obj handle is non-null before attempting this.

3.2 do_compare

Implement the do_compare method as follows:

```
1 function bit do_compare (ovm_object rhs,
ovm_comparer comparer);
2   mybusopmanual rhs;
3   do_compare =
4     ($cast(rhs_,rhs) &&
5      super.do_compare(rhs,comparer) &&
6      addr == rhs_.addr &&
7      obj != null && obj.compare(rhs_.obj)
9     );
10 endfunction
```

Line 1—This is the signature of the do_compare method inherited from ovm_object. Your signature must be identical.

Line 3—This line begins a series of equality expressions logically ANDed together. Only if all terms evaluate to true will do_compare return 1. Should any term fail to compare, there is no need to evaluate subsequent terms, as it will have no effect on the result. This is referred to as short-circuiting, which provides an efficient means of comparing. We don't need to check the rhs object for null because that is already done before do_compare is called. Be sure to use triple-equal (===) when comparing 4-state (logic) properties, else x's will be treated as "don't care."

Lines 4-— Compare only works between two objects of the same type. The $cast evaluates to 'true' if the cast succeeds, thereby allowing evaluation of subsequent terms in the expression. If the cast fails, the two objects being compared are not of the same type and comparison fails early.

Line 5—Here, we call do_compare in the super class so any inherited data members are compared. If you omit this expression, the rhs object will not be fully compared.

Lines 6—The equality operator (==) can be used to compare any data type. For objects, it compares only the reference handles, i.e. it returns true if both handles point to the same underlying object. You should have one of these expressions for each member you wish to compare.

Lines 7-8—To compare different instances of a class type, call the object's compare method. Make sure the object handle is non-null before attempting this.

3.3 convert2string

The convert2string method is used to print information about an object in free-format. It is as efficient and succinct as the class designer wants, imposing no requirements on the content and format of the string that is returned. The author recommends implementing convert2string for use in `uvm_info messages, where users expect succinct output of the most relevant information.

```
1 function string convert2string();
2   return $sformatf("%s a=%0h, s=%s,
                          arr=%p obj=%s ",
       super.convert2string(), // base class
       addr,                   // integrals
       str,                    // strings
       arr,                    // unpacked types
       obj.convert2string());  // objects
3 endfunction
```

Line 1—This is the signature of the convert2string method inherited from ovm_object. Your signature must be identical.

Line 2—This line returns a string that represents the contents of the object. Note that it leverages the built-in $sformatf system function to perform the formatting for you. Use format specifiers to %h, %d, %b, %s, etc. to display output in hex, decimal, binary, or string formats. For unpacked data types, like arrays and structs, use %p for the most succinct implementation. Be sure to call super. convert2string.

3.4 do_print

To implement both print and sprint functionality, you only need to override do_print as follows:

```
1 function void do_print (ovm_printer printer);
2   super.do_print(printer);
3   printer.print_generic(“cmd”,”cmd_t”,
                          1,cmd.name());
4   printer.print_field(“addr”,addr,32);
5   printer.print_array_header(“data”,
                               data.size(),
                               “byte[$]”);
6   foreach(data[i])
7     printer.print_generic($sformatf(“[%0d]”,i),
                          ”byte”,
                          8,
                 $sformatf(“%0h”,data[i]));
8  printer.print_array_footer(data.size());
9 endfunction
```

Line 1—This is the signature of the do_print method inherited from ovm_object. Your signature must be identical.

Line 2—Call super.do_print() to print the base class fields.

Line 3-4—We call methods in the ovm_printer class that correspond to the type we want to print. Enum types use the print_generic method, which has arguments for directly providing field name, type, size, and value.

Line 5-8—Print arrays by printing its header, elements, and footer in separate statements. To print individual elements, the author recommends using print_generic, which allows you to customize what is printed for the element name, type name, and value.

3.5 do_record

Implement do_record as follows. First, define a simple macro, ‘ovm_record_field, that calls the vendor-specific system function for recording a name/value pair, e.g. $add_attribute. The macro allows you to pass the actual variable—not some arbitrarily large bit-vector—to $add_ attribute. (The UVM will provide these macro definitions for you.)

```
‘ifdef QUESTA
  `define ovm_record_att(HANDLE,NAME,VALUE) \
      $add_attribute(HANDLE,VALUE,NAME);
‘endif
‘ifdef IUS
  ‘define ovm_record_att(HANDLE,NAME,VALUE) \
       <Cadence Incisive implementation>
‘endif
‘ifdef VCS
  ‘define ovm_record_att(HANDLE,NAME,VALUE) \
       <Synopsys VCS implementation>
‘endif
`define ovm_record_field(NAME, VALUE) \
   if (recorder != null &&
      recorder.tr_handle!=0) begin \
       `ovm_record_att(recorder.tr_handle, \
                  NAME,VALUE) \
  end
```

These macros serve as a vendor-independent API for recording fields from within the do_record method implementation. Note that, for these macros to work, the ovm_recorder::tr_handle must be set via a previous call to ovm_component::begin_tr or ovm_transaction::begin_tr.

The do_record method simply invokes the `uvm_record_ field macro for each of the fields you want recorded:

```
function void do_record(ovm_recorder recorder);
  super.do_record(recorder);
  `ovm_record_field(“cmd”,cmd.name()) // enum
  `ovm_record_field(“addr”,addr) // integral
  foreach (data[index])        // arrays
    `ovm_record_field(
$sformatf(“data[%0d]”,index), data[index])
  obj.record(recorder);        // objects
endfunction
```

Line 1—This is the signature of the do_record method inherited from ovm_object. Your signature must be identical.

Line 2—Be sure to call super.do_record so any inherited data members are recorded.

Lines 3-7—Records enums, integral types, arrays, and objects using invocations of the ‘ovm_record_field macro, or calling a sub-object’s record method.

3.6 do_pack / do_unpack

These operations must be implemented such that unpacking is the exact reverse of packing. Packing an object into bits then unpacking those bits into a second object should be equivalent to copying the first object into the second.

Packing and unpacking require precise concatenation of property values into a bit vector, else the transfer would corrupt the source object’s contents.

To help reduce coding errors, the author advises using small convenience macros.[6] These types of macros are “less evil” because they expand into small bits of readable code that users might otherwise have to write themselves. In fact, the UVM will offer versions of these macros to facilitate robust manual implementations of do_pack and do_unpack.

```
`define ovm_pack_intN(VAR,SIZE) \
   packer.m_bits[packer.count +: SIZE] = VAR; \
   packer.count += SIZE;
`define ovm_pack_array(VAR,SIZE) \
    `ovm_pack_scalar(VAR.size(),32) \
    foreach (VAR `` [index]) begin \
      packer.m_bits[packer.count+:SIZE]=\
                                  VAR[index]; \
      packer.count += SIZE; \
    end
`define ovm_pack_queueN(VAR,SIZE) \
   `ovm_pack_arrayN(VAR,SIZE)
`define ovm_unpack_intN(VAR,SIZE) \
   VAR = packer.m_bits[packer.count +: SIZE]; \
   packer.count += SIZE;
`define ovm_unpack_enumN(TYPE,VAR,SIZE) \
   VAR = TYPE’(packer.m_bits[packer.count +: \
                               SIZE]); \
   packer.count += SIZE;
`define ovm_unpack_queueN(VAR,SIZE) \
    int sz; \    `ovm_unpack_scalar(sz,32) \
    while (VAR.size() > sz) \
      void’(VAR.pop_back()); \
    for (int i=0; i<sz; i++) begin \
      VAR[i]=packer.m_bits[packer.count+:SIZE];\
      packer.count += SIZE; \
    end
`define ovm_pack_int(VAR) \
    `ovm_pack_intN(VAR,$bits(VAR))
 `define ovm_unpack_enum(VAR,TYPE) \
    `ovm_unpack_enumN(VAR,$bits(VAR),TYPE)
 `define ovm_pack_queue(VAR) \
    `ovm_pack_queueN(BAR,$bits(VAR[0])
```

The ‘ovm_pack_int macro works for scalar built-in integral types. You can add your own simple macros to support other types, if you like. For example, reals would need the $realtobits and $bitstoreal system functions.

The macro implementations manipulate the m_bits and count properties of the packer object. m_bits is the bit vector that holds the packed object, and count holds the index at which the next property will be written to or extracted from m_bits.

With these simple macros defined, you can implement pack and unpack as follows:

```
function void do_pack(ovm_packer packer);
  super.do_pack(packer);
  `ovm_pack_int(cmd)
  `ovm_pack_int(addr)
  `ovm_pack_queue(data)
endfunction

function void do_unpack (ovm_packer packer);
  super.do_unpack(packer);
  `ovm_unpack_enum(cmd_t,cmd)
  `ovm_unpack_int(addr)
  `ovm_unpack_queue(data)
endfunction
```

Line 1—This is the signature of the do_pack method inherited from ovm_object. Your signature must be identical.

Line 2—Always call super.do_pack first.

Lines 3-5—For each property, invoke one of the convenience macros, which concatenates values into the packer's internal m_bits field and updates the count variable. Here, we've leveraged some convenience macros to make it simple and less error prone.

Line 8—This is the signature of the do_unpack method inherited from ovm_object. Your signature must be identical.

Line 9—Always call super.do_unpack first.

Lines 10-12—You must unpack each property in the same order as you packed them. You will need to cast the bits when unpacking into strongly typed data types like strings and enums.

4. CONCLUSION

This articlehas provided insight into the hidden costs behind the various macros provided in OVM. Some macros expand into small bits of code that the user would end up writing, or ensure the correct operation of critical features in the OVM. Other macros expand into large blocks of unreadable code that end up hurting performance and productivity in the long run, or unnecessarily obscure and limit usage of a simple, flexible API.

In summary:

We recommend always using the 'ovm_*_utils macros and the reporting macros: 'ovm_info, 'ovm_warning, 'ovm_warning, and 'ovm_fatal. These macros provide benefits that far exceed their costs.

The 'ovm_*_imp_decl macros are acceptable because they provide a reasonable trade-off between convenience and complexity.

The 'ovm_field macros have long-term costs that far exceed their short-term benefit. They save you the one-time cost of writing implementations. However, the performance and debug costs are incurred over and over again. Consider the extent of reuse across thousands of simulation runs, and across projects. For VIP, reuse extends across the industry. The more an object definition is used, the more costly 'ovm_field macros become in the long-run. While the UVM improves the performance of the field macros, it also provides "less evil" macros that help make the do_* methods easier to implement. In this author's opinion, it is still better to implement simple, manual implementations.

The 'ovm_do macros attempt to hide the start, start_item, and finish_item methods for sequence and sequence_item execution. This is unnecessary and confusing. The current 18 macro variants with long names and embedded in-line constraints cover only small fraction of the possible ways you can execute sequences and sequence items. It is easier to learn to use the simple 3-method API, encapsulating repeated operations inside a task.

The 'ovm_sequence-related macros hard-code a sequence to a particular sequencer type and facilitate the auto-execution of random sequences. Sequences should not be closely couple to a particular sequencer type, and they should not be started randomly. Stimulus generation is typically preceded by reset and other initialization procedures that preclude their automatic execution. You should declare sequences with 'ovm_object_utils and sequencers with 'ovm_component_utils, then start specific sequences explicitly using the start method. The UVM recognizes these and other shortcomings by deprecating the macros and OVM sequence library API. A new, superior sequence library implementation that is decoupled from the sequencer is currently being developed.

5.ACKNOWLEDGEMENTS

The author wishes to acknowledge the significant collaborative contributions of John Rose, Senior Product Engineer at Cadence Design Systems, Inc., toward improving the performance of the field-macro definitions in the UVM.

6.REFERENCES

[1] “IEEE Standard for SystemVerilog- Unified Hardware Design, Specification, and Verification Language,” IEEE Std 1800-2009, 2009.

[2] OVM 2.1.1 Reference, ovmworld.org

[3] OVM User Manual, ovmworld.org

[4] Accellera Verfication IP Technical SubCommittee (UVM Development Website); http://www.accellera.org/apps/org/workgroup/vip

[5] Amdahl’s Law: http://en.wikipedia.org/wiki/Amdahl’s_law

7.NOTES

[1] References to OVM macros shall also apply to UVM macros unless otherwise stated.

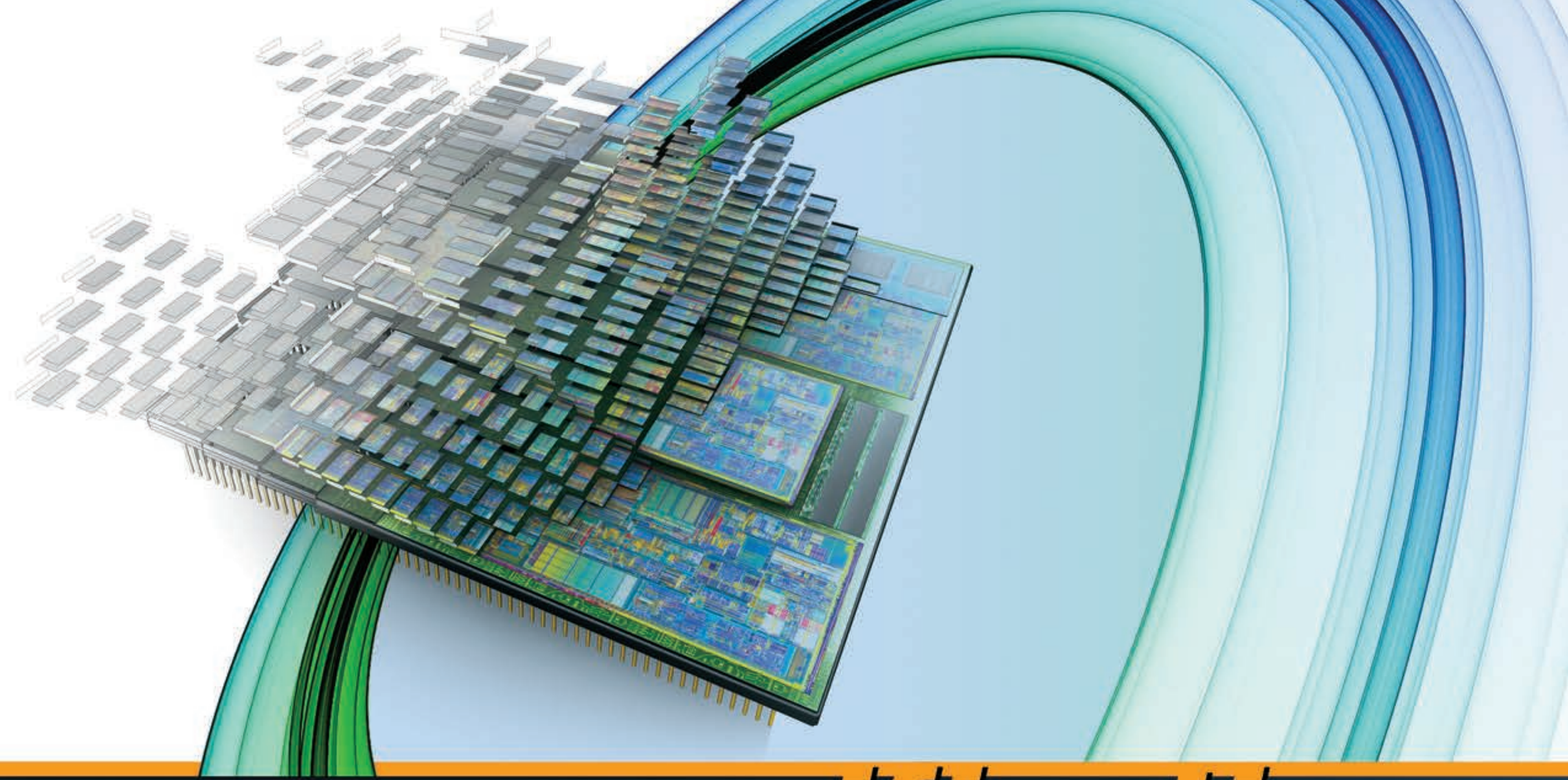
[2] The UBUS is a contrived bus protocol used in examples in the UVM 1.0 User Guide. It’s predecessor in OVM was XBUS.

[3] ‘ovm_field_aa_* macros do not implement record, pack, or unpack; line counts would be much greater if they did.

[4] Simulation results depend on many factors: simulator, CPU, memory, OS, network traffic, etc. Individual results will differ, but relative performance should be consistent.

[5] For Questa, the vlog option is -Epretty <filename>.

[6] Simulators supporting bitstream operators should make packing and unpacking easier, less error prone, and macro free: bits = {<<{ cmd, addr, data.size(), data, …};



verification HORIZONS

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