What Do Meteorologists and Verification Technologists Have in Common?

By Tom Fitzpatrick, Editor and Verification Technologist

As I write this, New England is bracing for a snowstorm that could bring as much as two feet of snow in the next day or two. By the time you read this, we'll know a) whether the forecasters were correct and b) how well we hardy New Englanders were able to cope. I often joke that I'm going to encourage my children to be meteorologists because that's the one job where, apparently, you can be consistently wrong and not suffer any consequences (except for Bill Murray in the movie "Groundhog Day"). As verification engineers, we have to be able to forecast the accurate completion of our projects and also be able to cope with problems that may occur. Unfortunately, there are severe consequences when we get it wrong.

And the stakes keep getting higher. We've spoken for years about designs getting more complex, but it's not just the number of gates anymore. The last few years have shown a continuous trend towards more embedded processors in designs, which brings software increasingly into the verification process. These systems-on-chip (SoCs) also tend to have large numbers of clock domains, which require additional verification. On top of this, we add multiple power domains and we need not only to verify the basic functionality of an SoC, but also to verify that the functionality is still correct when the power circuitry and control logic (much of which is software, by the way) is layered on top of the problem. Who wouldn't feel "snowed under"?

In this issue, we're going to try to help you dig yourself out. As always, we've brought together authors from all over to give you their perspective on key issues you're likely to face in assembling and verifying your SoC. Our featured article, "Using Formal Analysis to 'Block and Tackle'" comes from our friend Paul Egan at Rockwell Automation. One of the critical stages of the SoC process is putting the blocks together and verifying that they are connected correctly. This is a great application for formal analysis and Paul shows how they were able to use formal to verify connectivity at both the block and chip levels.

"As verification engineers, we have to be able to forecast the accurate completion of our projects and also be able to cope with problems that may occur. Unfortunately, there are severe consequences when we get it wrong."

—Tom Fitzpatrick
chip level. As you'll see, the process is the same regardless of the size of the block, and makes it really easy to verify late-stage changes, too.

I mentioned how important software is in the SoC process. Our next article, from my Mentor colleagues Hans van der Schoot and Hemant Sharma, shows how “Bringing Verification and Validation Under One Umbrella” can both speed up functional verification and also give you a platform on which your software team can validate the software pre-silicon. This early detection of hardware/software integration issues greatly simplifies the post-silicon validation process, since problems are much easier to debug pre-silicon.

Often when developing an SoC, we want to start with a system-level model of the design, so we can analyze its performance and other characteristics to make sure it’s right before we build it. The advantage of applying “System Level Code Coverage using Vista Architect and SystemC,” is that you can gain important insight into the completeness of your system-level testbench as well as whether certain blocks you’ve included in your design are actually contributing. You’ve probably used code coverage on RTL, but now you can do it at the system level.

The Unified Power Format (UPF) is the standard for specifying low-power design intent in a way orthogonal to the RTL. The standard is being widely adopted and also being updated by the IEEE. In “The Evolution of UPF: What’s Next?” my friend Erich Marschner describes some of the interesting enhancements of UPF 2.1 that help UPF more easily and accurately model power management effects.

In our Partners’ Corner section, we first have our friends from CVC, who explain the “Top 5 Reasons Why Every DV Engineer Would Love the Latest SystemVerilog 2012 Features.” Having been involved in SystemVerilog from its earliest days, I must say that even I’m impressed with how far it’s come. The article highlights some key new features for both design and verification engineers. Keeping with the SystemVerilog theme, we next have Ben Cohen showing us how to use “SVA in a UVM class-based environment.” One of my favorite things about SystemVerilog has always been the way it combines cool features like assertions and classes into a single language. This article provides an in-depth view of how to apply SystemVerilog assertions (SVA) to simplify some of the tasks usually done by scoreboards in UVM and also to improve coverage sampling in your testbench. Our final partner article comes from our friends at FishTail Design Automation, who take us through “The Formal Verification of Design Constraints.” Timing-related design constraints can be very difficult to get right and often require the verification team to spend time manually analyzing results to determine their validity. This article shows how formal verification can simplify this process and give much more credible results.

Last but not least, our Consultants’ Corner features Mark Litterick of Verilab, who shares “OVM to UVM Migration, or There and Back Again, a Consultant’s Tale.” Mark and his colleagues have extensive experience working with users who are trying to develop new projects using UVM while simultaneously maintaining ongoing development for existing projects in OVM, including components that will eventually be migrated to UVM. This article will show you how they managed this process, and they didn’t have to kill any dragons to do it.

I hope you enjoy this edition of Verification Horizons. If you’re at DVCon, please stop by and say hello and let me know what you think. For those of you not at the show, please try to make it next year. I’ll be here.

Respectfully submitted,
Tom Fitzpatrick
Editor, Verification Horizons
Table of Contents  February 2013 Issue

**Page 4**
Using Formal Analysis to “Block and Tackle”  
*by Paul B. Egan, Rockwell Automation*

**Page 9**
Bringing Verification and Validation under One Umbrella  
*by Hemant Sharma and Hans van der Schoot; Mentor Graphics*

**Page 12**
System Level Code Coverage using Vista Architect and SystemC  
*by Ken P. McCarty, Mentor Graphics*

**Page 14**
The Evolution of UPF: What’s Next?  
*by Erich Marschner, Product Manager, Questa Power Aware Simulation, Mentor Graphics*

**Partners’ Corner**

**Page 17**
Top Five Reasons Why Every DV Engineer Will Love the Latest SystemVerilog 2012 Features  
*by Ajeetha Kumari, Srinivasan Venkataramanan, CVC Pvt. Ltd.*

**Page 24**
SVA in a UVM Class-based Environment  
*by Ben Cohen, author, consultant, and trainer*

**Page 29**
The Formal Verification of Design Constraints  
*by Ajay Daga, CEO, FishTail Design Automation Inc.*

**Consultant’s Corner**

**Page 34**
OVM to UVM Migration, or... “There and Back Again: A Consultant’s Tale”  
*by Mark Litterick, Verification Consultant, Verilab GmbH*
INTRODUCTION
Using formal analysis to validate chip level connectivity is one of “The Top Five Formal Verification Applications” (Verification Horizons Oct 2012), and has been proven as the most efficient way to verify interconnects in complex Systems on Chips (SoCs) [3, 4, 5]. So, could formal analysis also be applied at the module level, even small modules? To test our theory, formal connectivity verification was first run on a relatively small block that contained eight instances each of seven different modules (56 instances total). We discovered that, not only does it work well on small blocks, formal analysis found two bugs that would have been very difficult to find in simulation.

Traditionally, connectivity verification at the block level has been completed using dynamic simulation, and typically involves writing directed tests to toggle the top level signals, then debugging why signal values did not transition as expected. For modules with a high number of wires and many modes of operation, the number of required tests quickly becomes unmanageable. We were searching for a better approach.

This article will explain how we applied formal analysis at the block level, extended this to full chip and describe how we significantly reduced verification time at both the block and chip level. Just like a block and tackle provides a mechanical advantage, the formal connectivity flow provides a verification advantage.

APPROACH
Formal connectivity verification was first used on a relatively small block (retriggerable timers) that contained eight instances of seven different modules (56 instances total). The sub-modules are connected to common busses at the top level of the block; several instances have a port that forms a part-select (slice) for a vector net. The RTL code for this module is very compact due to the use of a generate construct (pseudo code shown here):

```vhdl
genvar gen_modules;

generate for (gen_modules = 0; gen_modules < NUM_OF_MODULES; gen_modules++)
begin : gen_blk_module_1
    module_1 u_module_1 (<port_expression>)
end
endgenerate

...:

generate for (gen_modules = 0; gen_modules < NUM_OF_MODULES; gen_modules++)
begin : gen_blk_module_N
    module_N u_module_N (<port_expression>)
end
endgenerate
```

The connectivity verification flow is a very simple three step process as shown in Figure 1 (opposite).

1. Create a spreadsheet that defines point to point connections
2. Run script to generate properties
3. Run static Formal Verification

STEP 1: CREATE THE SPREADSHEET
The connectivity is defined in a Tab Separated Value (TSV) file (shown opposite, below figure 1) that can be created using any standard text editor, but using a spreadsheet tool like MS-Excel or OpenOffice Calc makes the data entry easier. For the retriggerable timers block, the connections are all point to point without any delay, so the “connect” keyword is used for the connectivity check:

```
<table>
<thead>
<tr>
<th>gen_modules</th>
<th>module_1</th>
<th>module_N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>u_module_1</td>
<td>u_module_N</td>
</tr>
</tbody>
</table>
```

STEP 2: RUN SCRIPT TO GENERATE PROPERTIES
Once the TSV file has been created, run the utility to generate the properties used for formal verification.

```
GenConn timers.tsv -m -tsv -sva -o checkers.sv
```
STEP 3: RUN STATIC FORMAL VERIFICATION

The next step in the process is to run Formal analysis using the generated checkers. Two additional files are required to run Formal, an initialization sequence file that describes the reset sequence for the design and a directives file that defines the clock(s). The design and checkers are compiled, and Formal is run:

```
vlib work
vmap work work
vlog -f filelist -sv
vlog -sv checkers.sv
qformal -c -od conn_results -do "do directives.tcl; \
formal compile -d retriggerable_timers_wrapper \
-cuname bindings; \
formal verify -init init.seq -effort high; exit"
```

The compilation and Formal analysis commands are best done using a script in the language of your choice (i.e. TCL, Make, Perl, Python, etc). After confirming all syntax and hierarchical paths were correct on a single connection, we added the remaining paths to the TSV file. This block has 112 connections, but only 14 unique connections per instance so populating the TSV file was easily done using copy/paste/edit.

Once we were comfortable with the connectivity verification flow on a small block, the next step was to use this process on a larger, more complex block - the I/O multiplexer (pin_multiplex) block in our design. Our design has 8 groups of General Purpose Input Output (GPIO) pins, each with 16 pins that can be connected to 16 different peripherals (SPI, Ethernet, UART, I2C, etc) inside the chip, or configured as a plain old input or output. The pin_multiplex block is a large
programmable multiplexor that connects the peripherals in the SoC core to/from the I/O pads through the GPIO block. The GPIO block contains additional multiplexing logic that we verified with chip level connectivity testing.

The pin_multiplex block has approximately 6144 paths to check – 2048 outputs, 2048 output enables, 2048 inputs. Creating the TSV file for this block took a bit more time, and uses a conditional connect check:

<table>
<thead>
<tr>
<th># G1-0</th>
<th>pointA</th>
<th>pointB</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>output_state[0]</td>
<td>g1_out_region_state[0]</td>
<td>(out_region_config_0_ff[3:0]==4'h2) &amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(region_enable_g1_g2_ff[0]==1'b1)</td>
</tr>
</tbody>
</table>

After we completed testing at the block level, we ran at the chip level. The pin_multiplex TSV file was used as a starting point for the top level TSV file; path names were edited using global search/replace, and the condition was updated to include the GPIO block:

<table>
<thead>
<tr>
<th># G1-0</th>
<th>pointA</th>
<th>pointB</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>neo_top_wrapper.u_.neo_top.u_.neo_core.u_.pin_multiplex.output_state[0]</td>
<td>neo_top_wrapper.u_.neo_top.G1_core_.out[0]</td>
<td>(neo_top_wrapper.u_.neo_top.u_.neo_core.u_.pin_multiplex.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>out_region_config_0_ff[3:0]==4'h2) &amp; (neo_top_wrapper.u_.neo_top.u_.neo_core.u_.pin_multiplex.region_</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>enable_g1_g2_ff[0]==1'b1) &amp; (neo_top_wrapper.u_.neo_top.u_.neo_core.u_.gpio.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>gpo_enbl_g1_g2_rd_data[0]==1'b0) &amp; (neo_top_wrapper.u_.neo_top.u_.neo_core.u_.gpio.gpo_invert_g1_g2_rd_data[0]==1'b0)</td>
</tr>
</tbody>
</table>

RESULTS

Formal analysis on the retriggeable timers block found two bugs that would have been very difficult to find in simulation: incorrect vector bit-select, and an incorrect indexed part select on a vectored net connection (see code on the opposite page).

For this particular block, tool setup was insignificant, less than an hour to create an input spreadsheet; runtime was less than 2 minutes. Closer inspection of the RTL shows there are several potential errors that were easily detected using Formal:

- Vector order on secondary_trigger_count_rd_data
- Vector bit select
- Indexed part-select base expression (vector[base_expr + : width_expr])
- Indexed part-select width expression
- Indexed part-select operator (-:, +)

Note: For detailed syntax rules see IEEE 1364-2005 section 5.2.1 Vector bit-select and part-select addressing

Analysis of the pin_multiplex block found several “showstopper” bugs:

- Incorrect output enable on a SPI port
- Incorrect output enables on the ECC pins for a memory port
- Missing pin assignments for a memory port
- Pin connections where none should exist

Since our chip is designed to boot from an external device, the first two bugs would have made the chip almost unusable. Setup, analysis and debugging failures took about 32 hours, and runtime was about 2 or 3 minutes. We estimate using formal analysis to verify connectivity was more than 10X faster than using simulation. Another critical advantage is that formal is exhaustive; not only were all “legal” pin assignments validated, we confirmed there were no “illegal” pin assignments either. For example, a SPI
clock can be connected to only 4 of the 128 available pins. Verifying the legal pin connections in simulation can be done during chip level functional testing of the peripherals, but it is much more difficult to verify if the SPI clock is connected to any of the 124 pins it shouldn’t be.

Connectivity checking at the chip level was nearly as easy as our smallest block since we had already verified the pin multiplex block. Setup and analysis was less than 16 hours; runtime was less than 5 minutes, and no bugs were found.

CONCLUSIONS

Formal analysis for SoC interconnect verification is a proven methodology and this technique should also be applied at the module level, even small blocks. This is especially true for any modules that have a large number of wide busses connected, and modules that use Generate constructs and/or Vectors with part-selects (base_expr +/-: width_expr). The setup is semi-automated, does not require a testbench, has fast runtime, and is well supported by the Electronic Design Automation (EDA) tool vendors.

Where the connectivity flow really shined was for the dreaded late stage design change. In our case a change request was made to add an output signal to two multiplexed output pins; the initial estimate of schedule impact was 1 week to implement and verify; the actual verification took less than 15 minutes – edit 4 lines in the TSV file and rerun Formal.

TIPS

- Use a SystemVerilog wrapper on the top module. Most simulation tools don’t support interfaces on top level ports. Even if there are no interfaces on the top module, having a wrapper makes it easier if an interface is added later in the design cycle.
- Always start small; verify a single path first. Once you have identified the correct design hierarchical paths and signal names, adding remaining connections is easy (copy, paste, edit).
- For chip level connectivity verification, only compile the modules you need. This will improve runtime, and
tools will automatically blackbox empty modules. There is no reason to compile modules that do not affect connectivity.

• For chip level connectivity verification, bypass internal clock generators like PLL's and connect directly to the clock you need in the top level wrapper using a hierarchical assign statement:

assign u_neo_top.u_neo_core.clk_apb1x = clk_apb1x;

REFERENCES

The standard practice of developing RTL verification and validation platforms as separate flows, forgoes large opportunities to improve productivity and quality that could be gained through the sharing of modules and methods between the two. Bringing these two flows together would save an immense amount of duplicate effort and time while reducing the introduction of errors, because less code needs to be developed and maintained.

A unified flow for RTL verification and pre-silicon validation of hardware/software integration is accomplished by combining a mainstream, transaction-level verification methodology – the Universal Verification Methodology (UVM) – with a hardware-assisted simulation acceleration platform (also known as co-emulation). Necessary testbench modifications to enable this combination are generally nonintrusive and require no third-party class libraries; thus, verification components from customer environments are readily reusable for pure simulation environments, different designs using the same block, and different verification groups.

This common-source SystemVerilog/UVM acceleration platform offers substantial speed-up for functional verification in terms of simulation cycles per second. When subsequently leveraged for software validation, hardware-assisted simulation acceleration of 2 to 3 orders of magnitude (or more) makes it practical for software engineers to co-validate software against hardware far in advance of silicon. Pre-silicon validation boosts the efficiency of post-silicon validation through early detection of hardware/software integration issues.

**A TWO TOP TESTBENCH ARCHITECTURE**

Unlike a conventional verification environment that has both synthesizable and non-synthesizable components instantiated in a single testbench top, as shown in Figure-1, we create an acceleratable testbench that is partitioned into an HDL and an HVL top.

As shown in Figure-2 on the following page, HDL_TOP has all the synthesizable components instantiated in it. HVL_TOP contains all the untimed behavioral components. Synthesized HDL_TOP runs on the hardware accelerator and HVL_TOP runs on the simulator.

The HDL and HVL partitions of the model communicate at the transaction level. This communication is enabled by using a SystemVerilog virtual interface based method and/or a core-proxy based method.

A virtual interface is a SystemVerilog variable that can hold a reference to a concrete SystemVerilog interface instance. The Mentor Graphics Veloce® emulation platform supports a synthesizable transactor interface that provides communication between the emulator and testbench. Transactor interfaces encapsulate synthesizable SystemVerilog tasks and functions. A driver calls a function or task using a reference to a synthesized SystemVerilog interface to access DUT pins and signals. Similarly, a monitor waits on...
A core-proxy is a C/C++ core model based on communication semantics between HDL and C, as defined by SCE-MI 2. A SystemVerilog API wrapper class connects a C-based proxy to the rest of the testbench. The proxy class maintains handles to components in the synthesizable transactors and uses DPI function calls or SCEMI pipes to communicate with these transactors. In this approach, C-based proxy class functions provide APIs that can be used by a SystemVerilog or SystemC driver to communicate with the DUT.

A conventional software validation platform usually involves an FPGA prototyping platform with a JTAG connection to a software debugger. Typically, a dedicated team is responsible for partitioning the system on chip (SoC) RTL for a specific hardware platform to meet capacity and peripheral constraints. As more and more SoCs have embedded processors, there is a growing requisite for hardware/software co-validation. Earlier access to complete RTL, maintaining system bus connectivity across sub-modules and protocol peripherals, is an essential step of the verification process.

Our approach for hardware/software co-validation is to leverage the verification testbench to access the same RTL as used by design verification engineers. A user can access the SoC system bus with either a UVM testbench or a C API embedded software model to access the complete RTL.

Figure-3 shows a virtual machine communicating with the Mentor Graphics Veloce emulator to validate software driver development using a proxy-based transactor. Such integration provides the complete SoC RTL in the hardware accelerator with full visibility.

### THE UNIFIED FLOW

The partitioned domains are bridged using a SCE-MI compliant, high-performance, transaction-level communication link between the hardware and software provided by the Mentor Graphics Veloce/TBX® solution. This allows the timed portions of the testbench and the DUT to be accelerated in the Veloce emulator without affecting the untimed UVM domain.

It is imperative that the entire environment —especially including the untimed UVM components and the timed transactors — is single source for both conventional simulation and the co-emulation flow. Thus, any model written for the emulator can also run in simulation alone, using a SystemVerilog compliant simulator. This eliminates the need to maintain separate models and environments.

Transaction-level models that drive bus functional models maintain a transparent separation between the stimulus and the DUT pins. UVM layering concepts and transaction-level communication techniques provide the ability to swap out UVM-based directed and pseudo-random stimulus scenarios and apply, instead, real application software developed for the DUT, without sacrificing debug capabilities.

Clock and reset transactors should be designed to take advantage of the variable clock delay feature of the Veloce/TBX emulation platform. This enables the verification and validation teams to run their tests at various clock frequencies without recreating the RTL image for emulation.
The custom drivers and monitors are modeled using abstract classes and SystemVerilog interfaces.

Last but not least, in order to monitor system bus activity and inter-block communication for improved debug visibility, inline SystemVerilog assertions are inserted into the DUT. Assertions and assertion coverage are supported by Veloce/TBX for synthesis into the emulator with highly efficient transfer of potentially large amounts of coverage data out of the emulator into the testbench domain. The well-known benefits of assertion-based verification coupled with the high-performance of emulation considerably reduced the total turn-around time to detect and resolve bugs.

A typical SoC has a single processor managing multiple subsystems or multiple processors designated to perform specific tasks — all communicating via a system bus. A proxy-based transactor can provide a master and slave interface to the system bus. Such an implementation provides a single RTL image that can be used with a SystemC testbench or a UVM testbench.

Software teams usually have a system model available for early code development. SystemC has emerged as the language of choice for these models. Hence, it is important to have language interoperability of the transactor for portability between SystemC models on the one hand and SystemVerilog testbenches on the other hand. There are various hypervisor/virtual machines available to emulate the processor architecture. One such virtualization software package is QEMU, which can run in conjunction with the Mentor Veloce emulator. Software users can run their executable code in virtual machines that communicate with the RTL via a proxy based system bus transactor. As a result, the same RTL image is available for verification and validation with full visibility to debug.

CONCLUSION

The co-emulation flow and resulting unified verification and validation platform has made it possible for SoC design verification and software validation teams to use the same RTL image for their respective test scenarios. Moreover, software validation teams can now have much earlier access to the RTL code than in previous projects of similar scope.

Recently deployed at multiple customer sites, this unified verification and validation flow has demonstrated a near seamless transition between RTL design verification and software/firmware validation. It significantly reduces the turn-around time for time consuming yet essential tasks, including debug and regressions. It takes advantage of very fast emulator performance to handle longer and more tests to cover more design requirements and uncover more design bugs. In essence this unified flow has eliminated the productivity and quality penalties associated with creating and maintaining different verification and validation platforms.

The single-source verification environment for both simulation and emulation provides test speed ups of at least an order of magnitude. This translates to tests that took nearly a week to run in simulation now completing in just over an hour on the co-emulation platform. With the speed of an emulator and a combination of directed tests as well as real-world scenarios, verification engineers will be able to validate the SoC more holistically.

Figure-3: SW driver development with HW-assisted acceleration.
INTRODUCTION
SoC are constantly becoming more and more complex forcing design teams to eke out as much performance as possible just to stay competitive. Design teams need to get it right from the start and can’t wait until it’s built to find out how it truly performs. This is where System Level Modeling and SystemC/TLM shine. System Level Modeling enables design teams to get it right from the start and by producing the optimal design the first time through. Vista Architect adds to this giving the designer the ability to quickly assemble a design and analyze the SoC performance before it is ever built. Code coverage will raise the level of confidence in a design.

Code coverage has been around for years. Its roots started in the software development community and can be traced back to the 1960’s. In later years it was adapted for use in modern HDL design environment. Today it has become a standard piece of all verification flows and methodology.

Now it’s time to move code coverage to the next level … the system level. One of the great advantages of SystemC is, at its core, is just C/C++ (with an additional library call for SystemC). What this means is just about any C/C++ analysis tools will work with SystemC. The GNU C/C++ compiler’s built-in code coverage mechanisms work perfectly with SystemC and Vista Architect.

THE BENEFITS
It is a huge benefit to have comprehensive Testbenches from the start at the system level. This allows the same testbench, created at the system level, to be reused throughout the design flow. This in itself drastically reduces testbench development and verification time, and it confirms that the functionality implemented at the system is consistent throughout the entire design process and flow.

Code Coverage has two main benefits, at the system level, it helps to determine if the testbenches:

• have covered all of the intended functional components in the design,
• and, in addition, it can expose components in the design that may not be used in the end product.

Typically, code coverage is used as a performance indicator on a verification strategy. At the system level in can be used to do more. It can be used to expose functionality or components of a design that are redundant or not used. These simple “extra” function or components can take up valuable Silicon real estate, become a hidden power drain, impact the performance, and increase the development time. They drive up product costs and directly impact time to market. These “extra” functions creep into a design process when components are designed individually, but when integrated into a system, become redundant or un-utilized.

System Level Code Coverage gives the architects and designers the opportunity to validate that their verification strategy is comprehensive and the design is optimal. It is important to understand that the System Level Code Coverage is a forward looking strategy, from an architectural perspective. It is preformed even before the first line of the RTL is ever written.

WHAT’S INVOLVED
Vista Architect enhances the use of Code Coverage for SystemC and the TLM based SoC designs. It makes it significantly easier to enable code coverage. Vista Architect separates the SystemC communication and timing from the functional components, thus allowing the coverage criteria to be applied only to the core functionality and not on the SystemC simulation infrastructure. Vista Architect
instrumentations are easily bypassed by the coverage tools, enabling the Verification experts to spend productive time on design optimizations.

Setting up a Vista Architect SystemC project for code coverage is very simple. It’s just a matter of setting a few options in the Vista Architect GUI, Re-compiling, and running the simulation. The final results are processed and displayed using a simple set of utilities from the command line.

The open source GNU tools “lcov” and “gcov” are frequently used to provide the necessary coverage and metrics for SystemC/C++ language based environment. These tools are easily available on the open source domains, and generate relevant outputs that can be assimilated with the UCDB (Unified Coverage Data Base).

Following is an overview of the steps required to run code coverage with Vista Architect and SystemC:

1) Start Vista architect normally.
2) Add the code coverage options to the project for the compiler and linker.
3) Recompile the project.
4) Run the simulation to completion.
5) Consolidate coverage data and Generate coverage report.
6) Review the coverage report in a standard HTML page viewer.

The Code Coverage option only needs to be set up once for a project. Once set up, it’s just a matter of recompiling, running the simulation, and analyzing results.

CONCLUSION
The process for running and generating coverage data from Vista and SystemC is straight forward and very easy to implement within any project.

For further information on the coverage tools used here please refer to associated GNU pages on “gcov” and “lcov” internet.

For the complete details of the process please look at the Vista application notes on the use of SystemC/TLM based code coverage.

You can find more information on Code Coverage by searching the Verification Academy website. It contains extensive information on the benefits and usage of code coverage.
INTRODUCTION
Usage of the Unified Power Format (UPF) is growing rapidly as low power design and verification becomes increasingly necessary. In parallel, the UPF standard has continued to evolve. A previous article described and compared the initial UPF standard, defined by Accellera, and the more recent IEEE 1801-2009 UPF standard, also known as UPF 2.0. The IEEE definition of UPF is the current version of the standard, at least for now, but that is about to change. The next version, UPF 2.1, is scheduled for review by the IEEE Standards Review Committee in early March. When it is approved, UPF 2.1 will become the current version of the UPF standard.

UPF 2.1 is an incremental update of UPF 2.0, not a major revision. That said, UPF 2.1 contains a large number of small changes, ranging from subtle refinements of existing commands to improve usability, to new concepts that help ensure accurate modeling of power management effects. This article describes some of the more interesting enhancements and refinements coming soon in UPF 2.1.

FORMAT UNIFICATION
One of the most significant aspects of UPF 2.1 is the fact that it represents a major step forward towards a unified low power format that is supported by all EDA vendors. For some time, UPF and CPF, the Common Power Format from Si2 and Cadence, have co-existed as alternative low power formats. While similar in many ways, the two formats have been different enough to make automatic translation between them difficult, which in turn has created problems for users who have a multi-vendor flow that requires use of both formats. Because of this, many low power designers and verification engineers have been pushing for unification of the two formats into a single format that all tools would support.

Si2 took some steps in that direction several years ago, by adding certain features of UPF into the most recent version of CPF, but this did not really address the interoperability issue well enough. More recently, the IEEE P1801 UPF working group has been working to include some aspects of CPF in the next version of UPF. To enable these additions, Si2 and Cadence both joined the IEEE P1801 UPF working group in early 2011, Si2 donated the definition of CPF 2.0 to the working group, and Cadence has actively contributed to the development of UPF 2.1. As a result, UPF 2.1 includes some new features derived from CPF concepts. These features appear to be sufficient to enable CPF users to move to a UPF-based low power flow. IEEE P1801 working group members who use both formats have strongly supported the integration of CPF features into the next version of IEEE standard 1801 UPF.

POWER MANAGEMENT CELL DEFINITION
The primary CPF-inspired new feature in UPF 2.1 is a set of commands for defining power management cells such as isolation, level-shifting, and retention cells. These new commands enable users to provide a complete power intent specification in UPF, without relying on Liberty libraries for the definition of power management cells. Some users require such commands because their Liberty libraries may be out of date or incorrect, and the cost of modifying and re-verifying the modified libraries is often prohibitive. With these new power management cell definition commands, power intent for a given design can be expressed without updating the Liberty libraries.

MACRO CELL MODELING
The other CPF-inspired new feature in UPF 2.1 is a set of commands for defining power intent for macro models. While not strictly necessary – macro models can be represented with existing UPF 2.0 features – these new commands provide additional convenience by packaging up the power intent for a given macro so that it can be easily applied to each instance of the macro. These commands -begin_power_model, end_power_model, and apply_power_model – effectively provide a standardized UPF subroutine definition and invocation capability.

Other enhancements in UPF 2.1 also address macro cells. In particular, refinements to the definition of the boundary of a power domain now enable isolation and level shifting strategies to apply to ports of a macro cell instance. New predefined attributes UPF_feedthrough and UPF_unconnected have been added also, to provide
information about the internals of a hard macro instance that are required during network analysis to determine driving and receiving supplies of a net.

Additional changes clarify how anonymous supply sets are constructed for ports of a hard macro based upon the related supplies associated with the port and the pg_types of those related supplies. These anonymous supply sets are considered by isolation and level shifting strategies written using -source or -sink filters, so they need to be well-defined to ensure that isolation and level shifting insertion works correctly.

SUPPLY EQUIVALENCE

UPF 1.0 defined supply nets; UPF 2.0 added the concept of supply sets, which are collections of related supply nets. Both are used to model the supply networks providing power to various portions of a design.

The supplies powering logic driving and receiving a signal affect power management requirements:

• If the driving supply can be off when the receiving supply is on, isolation is required.
• If the two supplies can operate at significantly different voltage levels, level shifting is required.

Because of this, isolation and level-shifting strategies often use the driving/receiving supply as a filter to determine where isolation or level shifting cells should be inserted.

The -source filter for supply S selects any port that is driven by logic powered by a supply that matches S. The -sink filter for supply S2 selects any port that is received by (i.e., fans out to) logic powered by a supply that matches S2. The sink filter in particular makes it easy to selectively isolate certain paths from a port while leaving other paths un-isolated, or to isolate different paths with different controls or different isolation supplies. But what does it mean for one supply to “match” another? This is where the UPF 2.1 concept of supply equivalence comes in.

At the lowest level, two supply nets are equivalent if they are connected directly or indirectly, either by HDL statements or by UPF commands or by a combination of both. This is called “electrical equivalence” in UPF 2.1.

If two supply nets are electrically equivalent, then they are also functionally equivalent: they will both always have the same state and voltage. Two supply nets can be declared to be equivalent even if they are not connected in the design, in which case the equivalence declaration acts as a constraint on the eventual implementation or on use of an IP block in a larger system.

The equivalence concept also applies to supply sets. Two supply sets are considered equivalent if their corresponding required functions (the supply nets, or placeholders for supply nets, that make up each supply set) are all equivalent. Two supply sets are also considered equivalent if they are declared to be equivalent.

This definition of equivalence enables other UPF commands to function in intuitively natural ways. For example, the set_isolation command will by default use supply equivalence in determining which ports are to be isolated when set_isolation is specified with a –sink or –source filter. The command create_composite_domain will also consider supply equivalence when ensuring that all subdomains have the ‘same’ primary supply. In general, where UPF 2.0 required two supplies to be “the same”, UPF 2.1 now requires them to be equivalent.

SUPPLY SET REFINEMENTS

Several other changes in UPF 2.1 also focus on supply sets and related concepts. For example, UPF 2.0 allows specification of attributes representing the driver supply or receiver supply of a port, in case the logic driving or receiving that port is not in the design (for a primary input or output) or not visible (for a hard macro instance). However, UPF 2.0 assumes that all ports have drivers or receivers, as appropriate; the possibility that a port is left undriven or is not connected to a sink is not considered. In UPF 2.1, this is modeled explicitly with the above-mentioned UPF_unconnected attribute, and the semantics of supply sets and driverreceiver supply analysis account for this, even to the point of making different assumptions about primary ports, which are assumed to be connected externally, and ports of...
leaf-level instances, for which the same assumption may not hold.

UPF 2.1 also refines the definition of default supplies for isolation and level shifting cells inserted by strategies that do not explicitly identify the relevant supplies. For level shifters, the default supplies are now those of the logic driving the input side and the logic receiving the output side, respectively, provided that the domain crossing involved is not overly complex. For isolation, the default isolation supply now reflects the fact that the location in which an isolation cell is inserted typically determines what isolation supply will be used for that cell. UPF 2.1 also adds the concept of supply availability: which supplies are available in a given domain to power buffers inserted during implementation. This allows the user to constrain implementation tool decisions to avoid routing difficulties.

STRATEGY REFINEMENTS
In UPF 2.0, buffers can be inserted at a given domain boundary port by associating a “repeater supply” attribute with that port. However, this attribute can only be associated with outputs of a domain, not with inputs; this limits its utility. In UPF 2.1, the repeater supply attribute has been replaced with a new strategy command, set_repeater, which supports much more flexible buffer insertion and functions consistently with other strategy commands for isolation and level shifter insertion.

The retention strategy command, set_retention, has also been improved in UPF 2.1. The initial definition for set_retention in UPF 1.0 was modeled on the use of a balloon latch to which a register’s value could be saved before power down and from which the value could be restored after power up. UPF 2.0 added the --use_retention_as_primary option to reflect the fact that retention might be accomplished by just keeping the latch powered up. However, UPF 2.0 still requires both save and restore control signals even in this case. Also, the UPF 2.0 semantics for retention do not model very well some aspects of practical implementation methods, such as the “live slave” approach used with master-slave flipflops. In UPF 2.1, the semantics of set_retention have been adapted to allow for various types of retention, with various combinations of control signals, and the UPF 2.1 specification explains how each of these types is modeled in simulation.

SUMMARY
The changes in UPF 2.1 described above are only some of the many refinements that have been made in the new version of UPF. If these changes strike you as minor and incremental, you would be partly right: the changes are indeed incremental, but the aggregate effect is quite significant. Many of these changes focus on improving the precision, accuracy, and fidelity of power intent expressed in UPF, to make the standard easier to use and to ensure that it intuitively captures the concepts and functionality we need to represent. Just as a high-resolution display represents the same images as a lower resolution display but with much finer detail, UPF 2.1 will enable users to model their power intent using familiar concepts but with much greater finesse. Ultimately, the enhanced modeling precision and accuracy provided by UPF 2.1 should translate to higher quality and productivity in design and verification of low power systems.

Although UPF 2.1 is close to final approval, the 16+ user and vendor organizations making up the IEEE P1801 UPF working group will continue driving the evolution of UPF to address new and evolving requirements for low power design and verification. In fact, the next phase of UPF standard development will most likely begin soon after UPF 2.1 is approved, starting with decisions about what issues to tackle next and collection of requirements in those areas. If you are interested in influencing and/or contributing to the further development of UPF, you can join the working group and take part in its regular meetings.²

UPF is supported today by many of Mentor Graphics’ products, including Questa Power Aware Simulation (PASim), Olympus, Veloce, Tessent, and Calibre. If you are interested in how these tools support and make use of UPF power intent specifications, contact the corresponding product managers for more information.

END NOTES
2 For more information about the IEEE P1801 UPF working group, go to the P1801 web page at http://standards.ieee.org/develop/project/1801.html or contact the author at erich@p1801.org.
**INTRODUCTION**

SystemVerilog has become the most widely deployed Verification language over the last several years. Starting with the early Accellera release of 3.1a standard, the first IEEE 1800-2005 standard fueled the wide spread adoption in tools and user base. Since 2005 there is no look-back to this “all encompassing” standard that tries to satisfy and do more for RTL Designers and Verification engineers alike. The RTL designers benefit from the enhanced SV-Design constructs aimed at better modeling, lesser typing and new features. Many RTL designers also add simple assertions via SVA into their code as their RTL develops – they do themselves a big favor by doing so as this leads to less time to isolate a design (or environment at times) errors when a verification test fails later in the design flow. Adding SVA inline also assists engineers to formally document their assumptions about interface signals, corner cases etc. in an executable form. The verification engineers, arguably the more empowered ones with SystemVerilog’s vast new features, simply cherish the ability to build high quality verification environments, stimulus, reference models etc. all by staying within a single language. SystemVerilog being built on Verilog’s syntax significantly reduces the barrier between designers and Verification engineers thereby leading to more productive debug cycles as the designers are willing to look at “seemingly similar syntax”.

Long term DV enthusiasts and pioneers have been used to certain advanced features in the adjacent languages such as PSL (IEEE 1850) and E (IEEE 1647) and once in a while miss those “handy ones” in SystemVerilog 2005 standard. The good thing about having a standard in IEEE is that it has a well defined process that allows looking for contributions, attracting donations from adjacent technologies and growing the language further. In this sense SystemVerilog is more like the English language, which has been adopting/including words from various other languages and by doing so becomes more and more prevalent in the spoken world. In that process there has been several significant upgrades done to SystemVerilog 2005 standard, one in 2009 (IEEE 1800-2009) and the standardization committee has approved another update, the IEEE 1800-2012 standard.

We at CVC have been working very closely with customers in deploying this wonderful standard that is destined to touch every electronics engineer in one way or the other since its early Accellera 3.1a version days. We have been educating customers on the new features as and when they become available both as a standard and of course on the tools like Questa, Questa-Formal etc. In this endeavor we have experienced several user queries, issues and woes with current standard features and suggest the new features as soon as they get evolved.

In this article we list the top 5 features that would enhance current SV users to do their job much better. So whether you are a RTL designer or a die-hard verification engineer, sit-back, relax, and indulge in these evolving new standard features. The added benefit for Questa users is that most of this works today in Questa simulator, so read, enjoy and start using these features in your projects!

**SOFT CONSTRAINTS**

A well known aspect of constraint solving is the ability to classify hard vs. soft constraints. Let us take a lifestyle example for understanding the new “soft constraints”; consider the case of an inter-city travel. Let’s say that we want to travel from City-1 to City-2 (Say Bangalore to
Chennai); the source & destination are fixed and are non-negotiable. But how we travel can be based on preference/availability/cost etc. For instance one may have options of:

- Flight
- Bus
- Train
- Car

Now modeling the above scenario using a constraint specification language like the one in SystemVerilog, there are “constraints” as below;

```verilog
class my_travel_c;
  rand places_t src, dst;
  rand int cost;
  rand travel_mode_t travel_mode;

  // Non-negotiable src and dest locations
  constraint src_dst_h { src == BLR; dst == CHENNAI;}

  // Nice to have low cost, but a “soft” constraint perhaps
  constraint minimize_cost { cost < 2000;}

  constraint travel_by {travel_mode inside {AIR, BUS, TRAIN};}
endclass : my_travel_c
```

Voila! With the addition of this “soft” keyword, the constraint solver (at right) inside Questa respects your constraint as long as possible. However when confronted by other, simultaneous hard constraints, the soft constraint “relaxes” itself automatically without user intervention leading to a practical solution. It is as-if the user has added an explicit constraint_mode(0) “on need basis”.

For long time CRV (Constrained Random Verification) users, this feature has proven to be of great use with E language in the past to setup default values, payload lengths etc. A specific user test may add additional hard constraints that may provide a seemingly conflicting value to the variable being solved, but being “soft” the new style constraints adapt themselves well!

**UNIQUE CONSTRAINTS**

How do you generate a set of unique numbers in SV today? Say an array of unique, random valued integers? One usually needs to write a bunch of interacting constraints to get there. To give a background, consider a classical crossbar switch.

Now depending on various factors, the cost & travel-mode constraints may not be solvable. For instance if only AIR travel is desired, the cost constraint is likely to be violated. As an avid traveler you may not mind that violation and say YES it is fine! But how do you tell that to SystemVerilog? In the past one may go and turn-off the relevant/violating constraint via constraint_mode(0);

However that becomes tedious as before every randomize call you would need to do it (perhaps across testcases).

Welcome the all new SystemVerilog 2012 soft constraint.
While every CPU can talk to/access every memory, for every access uniqueness must be maintained in terms of one-to-one connection. This is usually referred to as “Uniqueness Constraint” (ref: http://www.wonko.info/ipt/iis/infosys/infosys5.htm). In SV 2012, with unique constraint feature one may code this very easily. Below is a screenshot of Questa with the output plotted.

MULTIPLE INHERITENCE

As some of our customers ask during our advanced SystemVerilog/UVM training sessions, SystemVerilog doesn’t allow multiple-inheritance. Or to be precise “DID NOT have”, now in SV-2012/2012 it does!
For those yet to get there – here is a quick recap:

Simple inheritance

```
base_class
members
methods
```

```
derived_class_1
members
methods
```

Few derived classes

```
base_class
members
methods
```

```
derived_class_1
members
methods
```

```
derived_class_2
members
methods
```

One can of-course “derive” from another “derived class” too, as-in:

```
base_class
members
methods
```

```
derived_class_1
members
methods
```

```
deep_der_class
members
methods
```

This is used widely in good System Verilog code and in general any OOP code. UVM uses this a lot. However what was not allowed in older SystemVerilog (2005/2009) is:

Multiple inheritance, as in:

```
base_class_1
members
methods
```

```
base_class_2
members
methods
```

```
derived_class_1
members
methods
```

Come to think of it, UVM supports TLM ports and that requires multiple inheritance at the core: See the UML diagram for today’s UVM TLM:

i.e. the class `uvm_port_base` extends (and implements) a standard TLM-INTERFACE class (call it `uvm_tlm_if_base`). It is also “hierarchical” and can do better with certain characteristics of a `uvm_component` such as “instance path”, “parent” etc. The current UVM implementation “works-around” the limitation of SV 2005/2009 by instantiating a local `uvm_component` instance as a member and provides “proxy” methods to mimic the effect of multiple inheritance.

So below is a sample code using multiple inheritance feature that is now added to 1800-2012 SystemVerilog standard. It does so by introducing few new keywords/constructs:
Now consider a simulation trace that ends prematurely – i.e. req is asserted and within next few clocks the test ends before the grant was asserted by the DUT. This scenario may or may not be acceptable depending on the intent of the test. In SV-2009, such an incomplete temporal shall be treated as “weak” and hence a tool like Questa shall provide you a warning and not an error. This is because the default strength of the temporal in a property is “weak”. However one can make it a stringent requirement that the test should wait at least till the temporal finishes by using the new strong keyword around the consequent:

```verilog
a_arb_latency : assert property
  (req => strong(#1:10) grant); // assumes default clocking
```

As shown below, this can help in quickly locating an environment/testcase issue by flagging the error. Without this added help, one would need to debug a possibly long log file to isolate the missing grant with the aid of waveforms, etc.

The concept of weak vs. strong temporal evaluations is well known in the formal world and in LTL domain. The power it brings to functional verification in terms of expressing various design behaviors is now fully available to SystemVerilog users! There are also a series of handy new LTL operators such as nexttime, always, until, until_
with, eventually added to the SVA – they all have origins in the adjacent IEEE 1850 PSL standard and as noted in the introduction, SystemVerilog is open to more such donations making it a natural choice for every DV engineer (a la English as a common language for human communication).

GLOBAL CLOCKING
With the recent developments in formal verification (model-checking), tools like Questa Formal make it more compelling for the mainstream DV engineers to leverage on this great technology. These DV engineers hitherto have been writing assertions but primarily aimed at simulation. The possibility of running formal verification on their RTL blocks with the same assertions written for simulation is quite inviting to them as they can leverage the work done already and get additional verification for free! However one of the challenges in writing SVA code that is compatible across simulation and formal verification has been the identification of primary system clock. One needs to make the same information visible to both these technologies in order to get similar verification setup. SV 2009 added the concept of **global clocking** to facilitate this. With this a user can designate a clock as global clock (per-module/interface etc. or for the entire system as the case may be). The SVA code can then refer to this global clock via newly introduced system function `$global_clock` instead of a local clock name. This makes the SVA code portable across technologies.

Occasionally DV engineers want to verify the clock generation logic itself via assertions. Typically they look for issues like:
- Clock period violations
- Glitches
- Unknown values etc.

This has proven to be very useful especially in establishing Gate-Level Simulations where-in the run times are huge and any additional help in debug saves days if not weeks. The new global clocking feature extends the power of SVA in cases such as these as well.

CONCLUSION
SystemVerilog with all its advanced features is becoming more and more capable with every upgrade. Tools like Questa have started supporting the latest additions as well. As explained in this article, the latest updates cater to RTL designers, assertions enthusiasts, testcase writers and methodology developers. So whichever task you do with your next SystemVerilog project, make sure you do better with latest SystemVerilog features! We at CVC (www.cvcblr.com) work closely with Mentor via their QVP program to train your team on all these latest developments, so contact us for all your training needs on SystemVerilog and Verification in general.
ACKNOWLEDGEMENTS
We would like to thank Josef Derner, Mentor Graphics for providing us early access to Questa 10.2 so that we could try out some of the latest features in simulation. We also would like to sincerely thank Ben Cohen (www.systemverilog.us) for his pedantic review of this article.

REFERENCES

ABOUT THE AUTHORS
Ajeetha Kumari is the founder & CEO of CVC Pvt. Ltd. (www.cvcblr.com). She has co-authored several books on the topic of functional verification including PSL, SVA & VMM. She has trained, mentored several engineers in India on these topics. She consults on key verification topics and is an active member of several online forums such as Verification Academy (www.verificationacademy.com).

Srinivasan Venkataramanan is currently the CTO at CVC Pvt. Ltd. and has more than 14 years of experience in ASIC design, synthesis and verification. He has worked at various semiconductor companies such as Philips, RealChip, Intel and also recently at Synopsys. He has presented at various conferences worldwide such as DesignCon (East), DVCon, VLSI Conference etc. He has trained several practicing engineers on SystemVerilog & UVM across the globe. He is currently the co-chair of IEEE-1647 E standard and is an active member of several committees on standards development.
SVA in a UVM Class-based Environment

by Ben Cohen, author, consultant, and trainer

INTRODUCTION
Verification can be defined as the check that the design meets the requirements. How can this be achieved? Many verification approaches have been used over the years, and those are not necessarily independent, but often complementary. For example, simulation may be performed on some partitions while emulation in other partitions. The verification process in simulation evolved throughout history from simple visual (and very painful) examination of waveforms with the DUT driven by a driver using directed tests, to transaction-based pseudo-random tests and checker modules. This has led to the development of class-based frameworks (e.g., e, VMM, OVM, UVM) that separate the tests from the environment, are capable of generating various concurrent scenarios with randomness and constraints, and are capable of easily transferring data to class-based subscribers for analysis and verification. In parallel, designers and verification engineers have improved the detection and localization of bugs in their design code using assertion-based languages (e.g., PSL, SVA).

Those recent verification technologies need not be independent. Specifically, verification approaches consist of several methodologies that work well, particularly when combined in symphony. UVM is a class-based methodology that, because of its nature, tends to stress the use of scoreboarding, which is supported by the analysis ports. Though scoreboarding solves many verification tasks, SystemVerilog Assertions (SVA) provide another complementary and supporting set of solutions that can speed up the verification and model building processes and can provide additional verification data to UVM for further actions. Specifically, SVA provides a simpler definition of temporal design properties for verification and coverage; for the support of covergroup sampling-controls that are based on a sequence of events; and for localized detection of errors and effective debug via on-demand waveform/transaction recording.

This article demonstrates how SVA complements a UVM class-based environment. It also demonstrates how the UVM severity levels can be used in all SVA action blocks instead of the SystemVerilog native severity levels.

SCOREBOARD IN UVM
Figure 1.0 demonstrates a typical verification model. The scoreboard is part of the analysis group and is most often used for the verification of the design behavior.

Scoreboards are analysis components that collect the transactions sent by the monitor and perform specific analysis activities on the collection of transactions. Scoreboard components determine whether or not the device is functioning properly. The best scoreboard architecture separates its tasks into two areas of concern: prediction and evaluation. The prediction is based on the requirements. A predictor model, sometimes referred to as a “Golden Reference Model”, receives the same stimulus stream as the DUT and produces known good response transaction streams. The scoreboard evaluates the predicted activity with actual observed activity on the DUT. Thus, in essence, a scoreboard is a piece of checking code that keeps track of the activities of the design and calculates the expected calculated responses based on scenarios, and compares those against actual.
SVA as Complementary/Substitution Approach

The concept of assertions is not new; any verification methodology or code “asserts” that the design meets the properties of the requirements. Thus, broadly speaking, one can state that the UVM scoreboarding approach provides assertions about the correctness of the design, and in its own rights is a coding style/language/methodology with its macros, data structures, and libraries. Assertion languages, such as PSL and SVA, evolved to address the need to specify the temporal requirement properties of a design in an executable language that is easy to write and read, and yet meets the needs of the design space for both simulation and formal verification. Thus, in a manner similar to frameworks, SVA is a language that expresses behavioral properties and supports directives to do something with those properties, such as assert for their correctness, assume for the input space constraints, and cover for the coverage of those properties.

The unique capabilities of SVA in a verification environment

SVA provides two types of assertions: immediate and concurrent. Immediate assertions are executed like a statement in a procedural block and do not need a clocking event to trigger the assertion, as that is controlled by the procedural block when the code reaches the assertion. An example of an immediate assertion commonly used in UVM is something as the following:

```
assert($cast(rsp_item, req_item.clone()) else 'uvm_error("driver", $sformatf("%m : error in casting rsp_item")));
```

Concurrent assertions are temporal, are based on clock semantics, and use sampled values of static variables. Assertions can also access static variables defined in classes; however, access to dynamic or rand variables is illegal. Concurrent assertions are illegal within classes, but can only be written in modules, SystemVerilog interfaces, and SystemVerilog checkers. As will be shown in the next subsections, assertions provide many benefits in a verification environment, and if values of class variables are needed for use in concurrent assertions, those variables can be copied onto class-static variables or in SystemVerilog virtual interfaces (to be connected later to actual interfaces), and then used in the assertions.

Assertions provide several unique capabilities, as described in the following subsections.

Detection of temporal violations and use of action block to modify flow

SVA handles cycle accurate checks more effectively than with an application using scoreboarding because the verification of the signals is done at every cycle rather than at the completion of the data gathering process. The action block of an assertion can modify SystemVerilog interface variables or static class variable on either a pass or a fail action of the assertion. Those variables can then be used by a class instance to drop an objection or to modify the flow of the class. Figure 2.0 demonstrates these concepts.

Figure 2.0 Using the results of an assertion to stop the simulator

```python
import uvm_pkg::*; `include "uvm_macros.svh"
class dut_xactn extends uvm_sequence_item;
endclass : dut_xactn
class C;
    static int k=9;
    static bit h;
    int v =5;
    rand int r;
    task test();
        // some actions
        if(v==5) k=1000; else k=0;
        endtask : test
endclass : C
interface bus_if(input bit clk);
    C c;
    bit a, b, d;
    bit done_abc; // for use by class
clocking cb @ (posedge clk); endclocking
```
update static class variable and interface variable
for use by other classes
task updates(bit done);
    done_abc <= done;
c, h <= done;
endtask : updates
apEndProp : assert property (@ (cb)
$rise(a)&& c.k==1000 => b ##[1:10] d)
updates(1'b1);
endinterface
class dut_driver extends uvm_driver #(dut_xactn,
dut_xactn);
C c; // Class has the static variables read
and modified by the assertions
in the SV interface
virtual interface dut_if.bus_if vif;
// ...
task run_phase(uvm_phase phase);
    phase.raise_objection(this,"dut_driver");
    fork
    begin
        //...
        get_and_drive();
    end
    begin
        wait (this.vif.done_abc && c.h);
        phase.drop_objection(this,"dut_driver");
    end
endtask : run_phase
endclass : dut_driver

Minimizing use of scoreboarding
with assertions
In class-based model, the scenario may modify class
variables (e.g., the random variables) that are used in the
verification properties of the DUT. Those variables are
often transferred to the scoreboard for the necessary
computations and comparisons. However, assertions offer
another, easier to use alternative that in some cases may
also work in conjunction with scoreboards. For example, if a
new occurrence of signal a occurs, then DUT signal b must
remain at 0 for a number of cycles, as determined by the
scenario (the class). Signal a should then go to a logical 1
within 5 cycles after that. To achieve this with assertions, a
task in the class copies the value of that number of cycles
into a SystemVerilog interface (or a static class variable, as
discussed previously). The assertion in the interface makes
use of that copied variable and other signals of the interface
to determine compliance to the requirements. Any detected
error is reported by the assertion; that error can also be
reported back to the class via an interface variable
(or a static class variable). That variable is set or reset by the
action block. Below is a snapshot of the key elements of
this concept and assertion in figure 4.0.

Sequence match to modify interface variables
or trigger system level functions
Assertions can be used to call functions or system functions
during any cycle of a sequence upon the success of an
evaluation during that cycle. This capability also allows for
changes to the SystemVerilog interface variables or static class
variable for use by the class. For example, in Figure 3.0:

Figure 3.0 Variables and function
calls from assertion statements

interface b_if (input logic clk);
// ... signal declarations
bit abc, a, b, c;
clocking cb @ (posedge clk);
endclocking
function void f(logic v);
    abc = v;
endfunction : f

variable abc is accessible by a class instance. It is modified by
a function called from within a sequence of an assertion.

Minimizing use of scoreboarding
with assertions
In class-based model, the scenario may modify class
variables (e.g., the random variables) that are used in the
verification properties of the DUT. Those variables are
often transferred to the scoreboard for the necessary
computations and comparisons. However, assertions offer
another, easier to use alternative that in some cases may
also work in conjunction with scoreboards. For example, if a
new occurrence of signal a occurs, then DUT signal b must
remain at 0 for a number of cycles, as determined by the
scenario (the class). Signal a should then go to a logical 1
within 5 cycles after that. To achieve this with assertions, a
task in the class copies the value of that number of cycles
into a SystemVerilog interface (or a static class variable, as
discussed previously). The assertion in the interface makes
use of that copied variable and other signals of the interface
to determine compliance to the requirements. Any detected
error is reported by the assertion; that error can also be
reported back to the class via an interface variable (or a
static class variable). That variable is set or reset by the
action block. Below is a snapshot of the key elements of
this concept and assertion in figure 4.0.

Figure 4.0 Use of values of class variables and
assertions in interfaces
(Code and simulation results are in http://SystemVerilog.us/
verizons/test_if_assertions.zip)
This is then followed by $b = 1$ within 5 cycles after that.

```
// This is then followed by b==1 within 5 cycles after that.

property p_ab_min;
int count_v;
($rose(a), count_v=count) =>
(lb, count_v=1'b1)* 1:1 #1 count_v=0
##[1:5] b;
endproperty : p_ab_min

ap_ab_min: assert property (posedge clk)
            (p_ab_min)
            ab_error <= 1'b0; else ab_error <= 1'b1;
endinterface : b_if
```

For more complex problems, a scoreboard data structure can keep track of conditions (driven and expected), and the assertions in the SystemVerilog interfaces can then determine the correctness of DUT’s responses. For example, assume that a scoreboard keeps track of the expected caching of data in the variable abc is accessible by a class instance It is modified by a function called from within a sequence of an assertion. Other functions can also be called, including system level functions. DUT’s memory based on a caching algorithm. The scoreboard can keep track of an expected miss or hit (cache_hit), and transfer that information, along with the value of any expected data (expected_data) into a SystemVerilog interface. Assertions in that interface can then verify the correct response from the DUT and log in any detected errors (cache_hit_error, cache_miss_error), which can then be read by the class instance.

```
Figure 5.0 Use of scoreboard values from class variables for use in assertions
```

```
// Enhanced coverage sampling
SystemVerilog assertion statements enable the control of transaction recording from action blocks or sequence match item, i.e., from any clocking event when a variable is true. This can be used in the setting of values within a module or interface, or to call any function, including the call of the sample() function to sample group coverage. The following example demonstrates the concepts.

```
Figure 6.0 Use of assertions for covergroup sampling
```

```
typedef enum {NOP, ADD, SUB, MULT, JMP, MOV, READ, WRITE, IDLE} instr_e;
typedef enum {FETCH, DECODE, EXECUTE} mode_e;
typedef enum {PC, IR1, IR2, REGA, REGB, MAR, REG_FILE, STK} resource_e;

interface cpu_if(input logic clk);
clocking cb @ (posedge clk);
endclocking
instr_e instr;
resource_e resource;
mode_e mode;
logic [15:0] addr;

covergroup instr_cg;
InstXrsc : cross instr, resource;
endgroup : instr_cg

instr_cg instr_cg1 = new();

covergroup addr_cg @(clk);
addr_cp : coverpoint addr
{ bins instruction = {[0:255]}; bins data = {[256:32767]};
}
endgroup : addr_cg

addr_cg addr_cg1 = new();

sequence q_fetch; @ (clk)
mode==FETCH #1 (1, instr_cg1.sample(),
              addr_cg1.sample());
endsequence : q_fetch

cover property (q_fetch); // Enables the sampling in cycle following the mode==FETCH

endinterface : cpu_if
```
UVM SEVERITY LEVELS IN SVA

UVM provides several macros that resemble the SystemVerilog severity levels, but provide more options, verbosity, and consistency with the UVM verification environment. In our SystemVerilog Assertions Handbook, 3rd Edition, we recommend the use of the UVM severity levels in all SVA action blocks instead of the SystemVerilog native severity levels. For example,

```systemverilog
string tID="UART "; ap_MEDIUM: assert property(a) else
  `uvm_info(tID,$sformatf("%m : error in a %b", a), UVM_MEDIUM);
ap_handshake : assert property ($rose(req) ||=> ###[0:4] ack) else
  `uvm_error(tID, $sformatf("%m req = %0h, ack=%0h", $sampled(req), $sampled (ack)));
```

See http://systemverilog.us/uvm_hi_low3.pdf for a discussion on this topic along with simulation results.

CONCLUSIONS AND RECOMMENDATIONS

Verification consists of several methodologies that work well, particularly when combined in symphony. UVM is a class-based methodology that tends to stress the use of scoreboarding to compare expected results against actual observations. Though the scoreboarding approach solves many verification tasks, SVA provides another complementary approach that works with UVM to speed up the model building process, to support the class-based environment, and to quickly localize errors at the cycle level.

SVA provides complementary advantages for UVM. This includes the simpler definition of temporal design properties for verification and coverage; the support of covergroup sampling control based on sequence of events; the localized detection of errors and effective debug via on-demand waveform/transaction recording; the control of system level tasks based on sequence of events; and the reporting of errors back to the UVM environment for flow control.

For bus protocol types of verification properties (signal level verification), we recommend SVA at the interfaces instead of using scoreboarding with compare as a verification technique. If a tool supports SystemVerilog checkers (1800-2009 and 2012), use checkers bound to interfaces. Assertion results can be written into the interface variables or class static variables for control and access by the UVM environment. For coverage of temporal properties, use assertions in SV interfaces. For covergroups that require the completion of temporal sequences as the sampling trigger, use SVA sequences that can transfer the sampling triggers to class variables. Use UVM severity levels in all SVA action blocks instead of the SystemVerilog native severity levels. Use SVA immediate assertions to flag unsuccessful randomization and terminate the simulation run.

END NOTES

1. MENTOR GRAPHICS UVM/OVM documentation verification methodology online cookbook
http://SystemVerilog.us
The Formal Verification of Design Constraints
by Ajay Daga, CEO, FishTail Design Automation Inc.

There are two approaches to the verification of design constraints: formal verification and structural analysis. Structural analysis refers to the type of analysis performed by a static timing tool where timing paths either exist or not based on constant settings and constant propagation. Formal verification, on the other hand, establishes the condition under which a timing path exists based on the propagation requirements for the path. These path propagation requirements are then used to prove or disprove constraint properties based on a formal analysis of the design. Structural analysis is fast because it is simple. Formal verification, however, is more complete and less noisy. Formal verification allows engineers to guarantee that their design is safe from silicon issues that result from an incorrect constraint specification. Structural analysis cannot make this claim because it cannot independently establish the correctness of a variety of design constraints.

This document illustrates aspects of constraint analysis that are better handled by formal verification. The issues belong to the following categories:

1) Clock propagation
2) Generated clocks
3) Logically exclusive clock groups
4) Physically exclusive clock groups
5) Case analysis
6) False paths
7) Multi-cycle paths

CLOCK PROPAGATION VERIFICATION
For the design shown in Figure 1, assume the following constraints are in place:

create_clock -period 5.0 CLKA
create_clock -period 3.0 TCLK
create_clock -period 10.0 CLKB

Structural constraint analysis simply establishes whether a clock propagates to a pin or not. Case analysis and disable timing commands are the only reasons for the propagation of a clock along a combinational path to stop. Formal constraint verification, on the other hand, establishes not only the pins that a clock propagates to but the condition under which a clock propagates to that pin.

For example, in Figure 1, formal constraint verification establishes that for the clock TCLK to propagate to U1/Y the requirement is that TEST must be 1 (note that there is no case analysis specified on TEST). As the requirement for TCLK to propagate to U3/Y is that TEST must be 0 it is impossible for TCLK to propagate through U1/Y and then also through U3/Y. As a result, the propagation of TCLK will halt at U3/A. Formal constraint verification flags this as an issue - the impact of which is that the flop FF2 will not be clocked by TCLK. This issue could have been the result of a design bug where instead of an or-gate driving the select line of the clock mux U3, a nor-gate was intended. Structural analysis will completely miss this issue and instead indicate that flop FF2 is clocked by TCLK. This provides a false assurance to a design engineer and misses an important bug in the clock-generation logic – a bug that could cause a silicon issue.

VERIFICATION OF GENERATED CLOCKS
Generated clock definitions specify a periodic waveform on a pin. The complexities associated with the different options used to create a generated clock coupled with design complexities result in the risk that an engineer specifies an incorrect waveform on a generated clock. Such a mistake is serious because it impacts the stat...
Engineering has specified a divide-by-3 waveform relative to the master clock when, in fact, the logic of the design results in a divide-by-4 generated clock. A formal logical analysis of the design flags this issue and provides a much more complete check regarding the correctness of the waveform specified on a generated clock.

**LOGICALLY EXCLUSIVE CLOCK GROUP VERIFICATION**

For the design shown in Figure 3, assume that the following constraints have been defined:

- `create_clock –period 5.0 clk`  
- `create_generated_clock –source clk –divide_by 2 genclk`  
- `set_false_path –from clk –to genclk`

Structural analysis would flag the false path from clk to genclk as incorrect because it is specified between two clocks that have the same master clock. Formal verification, on the other hand, would provide a more interesting result. Formal verification would indicate that the path between FF3 and FF4 is correctly constrained because the propagation requirement for clk to reach the clock pins on these flops is “!sel” and the propagation requirement for genclk to reach the same clock pins is “sel”. As the propagation requirements are exclusive to each other, the two clocks are logically exclusive in the way they propagate to flops FF3 and FF4. Formal verification would indicate that the path between FF1 and FF2 is incorrectly constrained because there

**Figure 2: Verification of generated clocks.**

![Figure 2: Verification of generated clocks.](image)

**Figure 3: Logically exclusive clock group verification.**

![Figure 3: Logically exclusive clock group verification.](image)
is no propagation requirement for either clk or genclk to reach the clock pins on these flops. For this reason, clocks clk and genclk are not exclusive in the way they propagate to FF1 and FF2.

Formal verification provides a more nuanced assessment of clock crossing issues. Instead of complaining about a clock-to-clock exception simply based on the fact that two clocks share the same master clock, formal verification analyzes the design to establish if the clocks are logically exclusive to each other and only complains if they are not. As a result, a designer only needs to review the real clock crossing issues on a design.

PHYSICALLY EXCLUSIVE CLOCK GROUP VERIFICATION

For the design shown in Figure 4, assume that the following constraints have been defined:

```verbatim
create_clock –period 5.0 clk
create_generated_clock FF1/Q –source clk –divide_by 2 –name genclk1
create_generated_clock FF2/Q –source clk –divide_by 2 –name genclk2
set_clock_group –physically_exclusive –group genclk1 –group genclk2
```

Structural analysis would flag the physically exclusive clock group specification between genclk1 and genclk2 as incorrect because the two clocks are not defined on the same pin. Formal verification, on the other hand, would establish that the propagation requirement for the master clock clk to genclk2 is “!sel”. As a result, the condition under which generated clock genclk1 exists on the design is “sel” and the condition under which genclk2 exists in the design is “!sel”. As these conditions are exclusive the two generated clocks are physically exclusive – when one exists, the other does not.

Again, formal verification goes a step further than structural analysis and performs a more sophisticated analysis of the design. The benefit of this is less noise in the warnings reported by formal verification and this is key to maintaining the attention span of a seasoned designer.

CASE ANALYSIS VERIFICATION

For the design shown in Figure 5 assume that the following constraints are in place:

```verbatim
set_case_analysis 0 U1/S
set_case_analysis 0 U2/S
```

A structural analysis of the design will flag no issue with these constraints because there is no conflict with the specified constant values when they are propagated forward in the design. Formal constraint verification will, on the other hand, establish that the case analysis of 0 on U1/S requires flops FF1 and FF2 to both be high. The case analysis of 0 on U2/S requires flops FF1 and FF2 to both be low. These requirements conflict and so the specified case analysis is cumulatively illegal. On complex designs it is easy to specify case analysis values that are in conflict with each other. These values will be honored without complaint by a
static timing or implementation tool and could easily result in a silicon issue because timing signoff will be performed with the design placed in a bogus state – the actual analysis that is meant to take place will never be performed.

**FALSE PATH VERIFICATION**

For the design show in Figure 6 assume that the following constraints are in place:

- `create_clock –period 5.0 CLKA`
- `create_clock –period 10.0 CLKB`
- `set_false_path –from FF1/CP –to CLKB`
- `set_false_path –from FF2/CP –to CLKA`

Structural analysis will not help establish whether the specified false paths are correct or incorrect. At best structural analysis will tell a designer that the specified paths do apply to real paths on a design. Whether they are meant to apply or not, is not something structural analysis will help with. Formal verification on the other hand will establish that the propagation requirement for the path from FF1 to FF3 is “!sel” and the propagation requirement for FF2 to FF3 is “sel”. Also, the propagation requirement for clock CLKA to FF3 is “!sel” and the propagation requirement for clock CLKB to FF3 is “sel”. As a result, since the propagation requirement of “!sel” from FF1 to FF3 conflicts with the propagation requirement of “sel” for CLKB to propagate to FF3 the false-path definition from FF1 to CLKB is correct. By a similar formal analysis the false-path definition from FF2 to CLKA is also correct.

The value of formally verifying false paths is to only leave for designer review the fpaths that cannot formally be proven based on the design description, but require additional architectural considerations to be taken into account. To the extent this architectural information such as static registers or restrictions on the way configuration registers are programmed can be communicated using SVA assertions, formal verification is able to leverage these assertions by treating them as assumptions and prove more of the false path definitions in a constraint file. The inability to establish the correctness of any false-path definitions in a constraint file results in a gap between what structural analysis is able to do and what is required for constraint signoff.

**MULTI-CYCLE PATH VERIFICATION**

For the design shown in Figure 7, assume the following constraints are in place:

- `create_clock –period 5.0 CLK`
- `set_multicycle_path –from FF1/CP –to FF2/D`

Structural analysis is in no position to verify whether the specified multi-cycle path is correct or not. Formal verification, on the other hand establishes that for the FF1/Q pin to transition in the current clock cycle valid must be high in the previous clock cycle. Also, for the clock clk to propagate to FF2, valid must be high. The signal valid has a periodic relationship and toggles from high to low every clock cycle. For this reason, if valid is high in the current clock cycle, causing FF1/Q to transition in the next cycle, then valid is also low in the next cycle thereby disabling...
the clock clk from propagating to FF2. This ensures multicycle behavior and the specified multi-cycle path from FF1 to FF2 is correct. MCP verification requires a sequential formal analysis of the design, working back several clock cycles until a multi-cycle property is proven. Again, the use of assertions to specify architectural restrictions (for example, the way configuration registers are programmed) is key to enabling formal verification when MCP behavior is not supported by just the synthesizable design description.

SUMMARY
Formal verification is key to obtaining a thorough and independent assessment of design constraints. Formal verification results play the role of devil’s advocate, questioning the rationale for constraints and driving an engineering review discussion. These results are credible because there is less noise and more nuanced issues are presented to designers. Structural analysis is far too limited in its scope leaving a whole range of constraint issues unverified. While fast, it relies on engineers to separate the wheat from the chaff. While the examples in this document are simple, the value of formal verification comes through on large complex designs where the correctness of constraints is not as obvious. Clean formal verification results eliminate the risk of silicon failure from bad constraints.
ABSTRACT
Many companies have a goal to migrate to UVM but this must be achieved without disrupting on-going projects and environments. In our current clients we have all possible combinations of migration paths, a desire to move new projects to UVM and a definite need to support OVM for a good few years. This article presents an interesting OVM to UVM migration story where we successfully translated a whole family of verification components from OVM 2.1.2 to UVM, assessed the impact, and then reworked the original OVM code, which was still live in a series of ongoing derivative projects, to make the ongoing translations totally automatic and part of the project release mechanism.

The key was to push all functional modifications possible into the OVM world including code that is deprecated in the UVM like the old sequence library utilities, and to perform minimal UVM work in the very last layer. This approach means that the ongoing project can still endure with OVM, but new projects and near future derivatives can go straight to UVM. In fact the translation layer is such that with only a few limitations we can actually translate back from UVM to OVM in order to deliver VIP to other projects that do not yet have the ability to migrate to UVM.

INTRODUCTION
While it is generally accepted that the Universal Verification Methodology (UVM) is the way forward for modern SystemVerilog based verification environments, many companies have an extensive legacy of verification components and environments based on the predecessor, the Open Verification Methodology (OVM). Furthermore single push migration of all block-level and top-level environments between the two methodologies may be considered too disruptive for most companies taking into account families of complex derivative under development and scheduled for the near future. Several of our major clients are in the transition phase of initiating new product developments in UVM while having to maintain ongoing development in OVM. Furthermore it is imperative for the new development that we migrate the vast majority of existing verification components from OVM to UVM even though the projects providing the VIP remain with OVM for the time being.

OVM to UVM Migration, or “There and Back Again: A Consultant’s Tale.”
by Mark Litterick, Verification Consultant, Verilab GmbH
This transition between OVM and UVM is represented by Figure 1, which shows overlapping development cycles for multiple projects and derivatives that share a common VIP database. In this case some projects remain on OVM for some time moving forward, but they must supply VIP, environments and tests to UVM for use in the new project families currently evolving in UVM.

This scenario is a bit different to that described in [1], resulting in additional restrictions and goals for our migration flow, that can be summed-up as follows:

- Support ongoing development in OVM for established projects with scheduled derivatives and intense reuse
- Validate all functional changes and enhancements to VIP using regressions environments for established projects
- Provide a path for all VIP to migrate to UVM in order to provide a comprehensive library and support new project development in UVM
- Demonstrate a mechanism to move entire project derivatives to UVM when appropriate for specific projects
- Ideally allow for limited backward translation of some new UVM VIP into ongoing legacy projects and for other OVM-based groups in the company

The process presented in this article was established by performing two full translations from OVM to UVM using a top-level environment with many in-house proprietary OVCs. The aim of this first pass was to establish if there were any major gotchas in our code-base and to establish credibility for further work. The purpose of the second pass was to optimize the flow to provide an automatic and reliable translation process that would support our multi-methodology requirement without compromising current ongoing project derivative or future new developments.

**STARTING POINT**

It is important to clarify that our starting point for this process enhancement was a family of OVCs that were developed from scratch using a clean OVM-2.1.2 implementation. The OVCs are for fundamentally different application protocols but they have architectural and coding similarities since they were developed together by the same team of people. There was no AVM or URM legacy and no previous code translation or inter-operation compromises.

This has two bearings on the discussion that follows. Firstly, this is probably as easy as it gets! Specifically, the careful adherence to a common coding style and specific attempt to keep the VIP architectures similar for the different protocols means that our translation scripts are more effective and less general-purpose. Secondly, it means that the reader may have to adapt the findings slightly to suit their own circumstances, but we hope that the information provided here will still be of benefit in assisting with that process.

**FIRST PASS — EARLY UVM TRANSLATION**

For our first pass we performed full translation of all the OVCs and the environment to UVM essentially using the process outlined in the Verification Academy [2]. The main purpose here was to determine if we had any unexpected problems in our database and to establish where the major effort would appear. We wanted to know if we could fully automate the translation process to allow continued development in OVM but reliable delivery to UVM from the parent project.

This process involves executing the ovm2uvmm translation script provided with the UVM release early on in the flow as shown in Figure 2 on the following page.

During the execution of this flow we performed regression checks at each stage of the process, and compilation and sanity checks for each of the minor steps. In order to efficiently convert our whole family of OVCs we generated, or enhanced, automatic scripts for each stage of the
process. Some of these scripts were fully automatic as shown in Figure 2, but others required additional work after execution (e.g. all of the items in the OVM deprecated.txt file are identified by a simple script, but the user has to implement any code changes). Figure 3 provides a table of the script setup and usage for the first pass translation.

In fact the translation process using this flow went quite well. We could successfully run UVM-based sanity tests quite early on and validate the subsequent enhancements incrementally. The only real issue was that we had too much work to do after the translation to UVM which would have to be done repetitively since, in our case, the providers of the VIP are live OVM projects.

**PUSH BACK TO OVM**

During the execution of the first pass stage it became clear to us that some of the steps in the process could be pushed back into the OVM world so that they were only executed once and did not become part of the translation flow. In particular, because the OVCs were developed with a clean OVM-2.1.2 implementation, the following features are already in the original code:

- only objection handling was used for end-of-test control
- improved message macros used instead of methods
- using back-ported uvm_reg in OVM environment

Furthermore we realized that if we invested some effort up front in OVM we could greatly simplify the translation of two key steps, which provided the most manual repair work in our first pass flow:

- modify the virtual interface configuration setup
- deprecate the old OVM sequence utils and sequence library in OVM

Making these changes was also implemented with automatic scripts, followed by manual repair work, but the key point is that this was done only once in the actual OVM

<table>
<thead>
<tr>
<th>Step</th>
<th>Identity</th>
<th>Modify</th>
<th>Repair</th>
</tr>
</thead>
<tbody>
<tr>
<td>deprecated OVM</td>
<td>script</td>
<td>manual</td>
<td>—</td>
</tr>
<tr>
<td>non-recommended OVM</td>
<td>script</td>
<td>manual</td>
<td>—</td>
</tr>
<tr>
<td>ovm2uvm</td>
<td>script</td>
<td>script</td>
<td>—</td>
</tr>
<tr>
<td>regression &amp; test scripts</td>
<td>—</td>
<td>manual</td>
<td>—</td>
</tr>
<tr>
<td>convert phases</td>
<td>script</td>
<td>script</td>
<td>—</td>
</tr>
<tr>
<td>convert *stop</td>
<td>script</td>
<td>manual</td>
<td>—</td>
</tr>
<tr>
<td>convert objections</td>
<td>script</td>
<td>script</td>
<td>minor</td>
</tr>
<tr>
<td>convert config_db</td>
<td>script</td>
<td>script</td>
<td>minor</td>
</tr>
<tr>
<td>virtual interfaces</td>
<td>script</td>
<td>manual</td>
<td>—</td>
</tr>
<tr>
<td>deprecated UVM</td>
<td>script</td>
<td>script</td>
<td>yes</td>
</tr>
<tr>
<td>message macros</td>
<td>script</td>
<td>script</td>
<td>—</td>
</tr>
<tr>
<td>command args</td>
<td>script</td>
<td>manual</td>
<td>—</td>
</tr>
<tr>
<td>register model</td>
<td>—</td>
<td>manual</td>
<td>—</td>
</tr>
</tbody>
</table>

**Figure 3: Translation Automation Scripts**
project codebase. The modifications then became part of the normal OVC behavior for the live OVM project and all related regression activity. In other words we prepared the OVM codebase for UVM translation as part of our standardized OVM coding style guidelines, this allowed the provider OVM project to continue and simplified subsequent OVM to UVM translation.

OBJECTION HANDLING
Our original OVCs use objections to handle end-of-test scheduling so there were no issues deprecating stop() and global_stop_request() from the code-base. But one important point to note here is that although we do translate from uvm_test_done to phase-based objection handling, we currently discourage the use of multiple run phases. The main reason here is that in line with the rest of the industry, we are waiting for stability in the proposed UVM phasing mechanism and in fact we have an adequate and flexible sequence-based phasing solution currently in use where required. One advantage of the sequence-based phasing solution is that it does not put any additional demands on the OVM to UVM translation process.

MESSAGE FORMAT
The message macros, which improve the performance of the OVM and UVM messages by evaluating the verbosity settings before calculating formatted string parameters, are already available in OVM-2.1.2 and are used throughout OVCs.

REGISTER MODEL
During the lifetime of the original project development Mentor contributed a back-ported version of the UVM register model (or register layer) to the OVM community. At that stage we modified our register generation tools to switch to uvm_reg with a specific goal of forward compatibility with UVM. Almost two years later we get payback! We extended the original register model to add several new features, but the translation from OVM to UVM is currently trivial for the register code and user environments.

VIRTUAL INTERFACE CONFIGURATION
It was only really at the UVM translation stage that we realized we had an opportunity for improving the virtual interface configuration in OVM in such a way that it would allow for easier conversion to UVM syntax. Specifically we replaced our previous implementation that used an explicit interface wrapper class instance with a much more generic container solution, very similar to that proposed in [3]. By providing static methods for set and get in the container that closely match the signature of the uvm_config_db API, we can create an easy translation path for going from OVM to UVM as shown in the following code snippets:

```plaintext
// example set in OVM testbench module
my_container#(virtual my_if)::set("", "cif", mif);
// example get in OVM agent or test class
if (!my_container#(virtual my_if)::get(this, "cif", vif))
  `ovm_fatal("NOVIF","...")
```

```plaintext
// example set in UVM testbench module
uvm_config_db#(virtual my_if)::set(null, "", "cif", mif);
// example get in UVM agent or test class
if (!uvm_config_db#(virtual my_if)::get(this, "", "cif", vif))
  `uvm_fatal("NOVIF","...")
```

SEQUENCE LIBRARY CHANGES
The original sequence utility macros and sequence library are deprecated in UVM partly because they were badly conceived. Specifically the use-case of having a default sequence and count is not generally appropriate to the way we construct interesting scenarios for most verification environments, especially proactive master agents. Analysis of the solution proposed in [4] led us to realize that in fact we can live without the library setup entirely in OVM and UVM for this family of verification components. This was first tested in UVM and then the scripts were modified to perform the same changes in OVM. The main steps are to replace the sequence and sequencer util with object and component utilis respectively, and to deprecate the default_sequence and count setup. Since many of our sequences access sequencer attributes via the p_sequencer field, we added this to the base sequences for each “library” (i.e. collection of sequences targeted for a particular sequencer). The Perl script extract on the following page provides an example of the level of complexity for the sequence translation step in OVM:
After running this script we do have manual repair work to do, but because this is done in the OVM code-base, the repairs are a one-off effort. Specifically we have a few empty methods lying around that we wish to remove for maintenance reasons, but more importantly a few of our OVCs are reactive slaves which generally require a default sequence in order to provide responses to the DUT. In this case we chose to start a default sequence explicitly in the run phase for the slave agent. This run method can be overloaded using the factory mechanism by tests that wish to select a different default sequence for the slave (e.g. an error response sequence) or callbacks can be built into the driver for error injection if required.

### SECOND PASS — LATE UVM TRANSLATION

With the major functional changes successfully implemented in the OVM source we are now in a better position to handle fully automatic OVM to UVM conversions using the flow shown in Figure 4.

Notice that Figure 4 shows the full process, but in fact the initial steps (inside audit OVM code) were only performed once, for all the OVCs in the original OVM source and never repeated. Therefore these steps are part of the overall migration to UVM but not part of the translation process; in fact the audit step can now be replaced with a script that simply identifies any possible hazards to the user. In addition convert steps do not need any significant manual repair work and can therefore now be bundled into a combined script. The resulting conversion from OVM to UVM can now be performed automatically as shown in Figure 5 on the opposite page, whenever a new version is released in the OVM domain.

### UVM TO OVM

With such a slim OVM to UVM conversion process it is possible to re-engineer the scripts to provide translation in the opposite direction. This potential path is required at some clients to support delivery of UVM verification components back into an OVM-based project. In order to

---

**Figure 4: Full Process for Late UVM Translation**

- **audit OVM code**
  - remove deprecated OVM
  - delete non-recommended OVM
  - convert *stop to objections
  - convert virtual interfaces to container
  - remove deprecated UVM seq lib
  - use improved OVM message macros
  - use UVM register model (back-ported)

- **translate OVM to UVM**
  - execute ovm2uvm script
  - update regression & test scripts

- **convert to UVM style**
  - convert phases to UVM
  - convert objection raise/drop to UVM
  - convert set/get to config db
  - convert virtual interfaces to config db

- **use new UVM features**
  - use improved UVM command args
validate the feasibility of this approach we successfully modified scripts and back-ported a couple of UVCs that originally came from OVM and managed to compile and use them successfully. What we have not attempted at this stage is to take a pure UVC, which was conceived in UVM, and apply these same scripts to translate it to OVM. Obviously this path is less critical than the forward path from live OVM to UVM and is therefore a lower priority; nonetheless analysis suggests it is possible to back-port (like the UVM register model) provided the following limitations are considered.

**UVM LIMITATIONS**

If the OVM to UVM transition period is expected to endure for a long period of time (or many project and reuse lifetimes) then it is worth considering limiting the use of the following UVM constructs to ensure backward compatibility:

- avoid run-time phases (use sequence-based phasing instead)
- avoid TLM 2.0 (or use only where required and encapsulate well)

Some of these points also reflect a state of flux in the UVM user community (e.g. the run-time phase discussions) and should probably be avoided until a consensus or more flexible solution is achieved.

Other improved aspects of UVM (such as the modified end-of-test objection handling using phases, updated phase methods, command line processor, and config_db changes) will require a little more effort to translate back to OVM but should be used as standard practice for UVM development.

**REFERENCES**
