Welcome back to Verification Horizons! And for those of you reading this at the 49th Design Automation Conference, welcome to San Francisco! We’re looking forward to a great show.

On a recent visit to the Evergreen Aviation & Space Museum in Oregon, I had an opportunity to see some great examples of what, for their time, were incredibly complex pieces of engineering. From a replica of the Wright brothers’ Flyer, to the World War I Sopwith Camel (sure to be recognized by you Snoopy fans out there) to various World War II fighter planes, including the Spruce Goose, the aviation side of the museum does a great job showing how the technology of flight evolved in just over 40 years. The space side of the museum has a marvelous collection of jet airplanes and rockets, such as replicas of the Gemini and Apollo spacecraft, including the Lunar Module, and pieces of a Saturn V. When you think about the effort and risk people took to advance from Kitty Hawk, North Carolina to the moon in under 70 years, it gives a whole new perspective to the idea of verification.

A lot of those successes resulted from early failures where engineers learned the hard way what not to do. World War II also gave us Murphy’s Law, which states that “if anything can go wrong, it will.” Compared to our historical colleagues, our lives as verification engineers today are much easier since we have the benefit of simulation and other verification tools to try things out virtually before we actually build them. But in a bow to Murphy, we know that it’s critical to make sure everything gets tested. A corollary to Murphy’s Law for verification could easily be “if it isn’t covered, it doesn’t work.”

And that’s why this issue of Verification Horizons is dedicated to the topic of coverage. There are, of course, many types of coverage and many tools that generate coverage data. The key to being able to measure all these different types of coverage is to have a common database to hold all of the data so it can be analyzed and correlated. Our first article this
month, “Mentor Has Accellera’s Latest Standard Covered,” describes the development of Accellera’s new Unified Coverage Interoperability Standard (UCIS) and how the Questa® Unified Coverage Database (UCDB) provides the necessary infrastructure to realize the kinds of analysis envisioned by the standard.

The notion of coverage itself has also changed over the years. Code coverage was a useful metric because it was easy to automate, but it didn’t really give enough information to know if your verification was really complete. As the use of constrained random stimulus took off, functional coverage became necessary as an application-specific way to measure whether certain scenarios were actually exercised. The problem with functional coverage is that, as the scenarios get more complex, it gets more difficult to specify what exactly you want to measure. This is where testbench automation comes in, and we have a trio of articles that show how Questa inFact Intelligent Testbench Automation can help you reach your coverage goals.

In “Is Intelligent Testbench Automation For You?” my colleague (and fellow MIT alum) Mark Olen explains the applications best suited to using inFact and where the tool will provide optimal results. If you see yourself in any of these use cases, you’ll definitely want to check out inFact. In “Automated Generation of Functional Coverage Metrics for Input Stimulus,” we explore how inFact can automatically create SystemVerilog covergroup definitions targeted to your stimulus. As anyone who has tried to define cross coverage of complex protocols knows, it’s often difficult to define your coverage properly to achieve 100% coverage of all possible scenarios. This is often due to the fact that some scenarios are actually not possible. Because inFact is based on a graphical representation of the protocols, it is possible to automatically generate the exclusion conditions, allowing you to reach 100% coverage without repetition.

Input scenario coverage is only part of the problem. There are many times when it is necessary to define input scenarios that will put the DUT in a specific internal state in order to reach a desired coverage goal. “Targeting Internal-State Scenarios in an Uncertain World” outlines the difficulty that constrained-random testing generally has, in being able to set up the preconditions required to hit these cases and shows how inFact can target them more efficiently. By taking information from elsewhere in the testbench and feeding it into the stimulus rule graph, inFact can automatically generate reactive stimulus sequences that achieve the desired coverage.

Emulation gives us the ability to run simulations on large designs that would be impractical to test with simulation alone. But the benefits of emulation have mostly been restricted to small teams who have access to the physical emulator in the lab to connect peripherals via “in circuit emulation” (“ICE”). Our next article, “Virtualization Delivers Total Verification of SoC Hardware, Software, and Interfaces,” shows how Mentor’s VirtuaLAB library of virtual peripherals can be used with Veloce to provide an easily configurable emulation environment to support total verification of the hardware, software and peripheral interfaces for teams around the world.

Switching gears a little, our next article comes from my colleague Mark Peryer who explains how to handle “On the Fly Reset” in a UVM testbench. While the Accellera UVM committee is still working on a phasing solution, Mark offers some concrete guidelines on how a well-planned testbench can handle this common scenario, which is actually one of the issues that phasing is intended to address. Since the solution is independent of the new run-time phases in UVM, it can be used in OVM testbenches, as well. I encourage you to take a look.
In our Partners’ Corner, we begin with a paper co-authored by our friends at Xilinx in Longmont, Colorado that was presented at this year’s DVCon. In “Relieving the Parameterized Coverage Headache,” the authors discuss the problems associated with gathering coverage on a design that may itself be parameterized, which requires the coverage collection to be modified based on the parameter values used to configure the design. Using UVM (or OVM) configuration to pass information on these parameter values lets you simplify the definition of covergroups in your testbench, thus customizing your coverage collection to match the DUT configuration.

FPGA prototyping is another tried and true lab-based approach to verification. The next article, from our friends at MathWorks, shows you “Four Best Practices for Prototyping MATLAB and Simulink Algorithms on FPGAs” that let you reuse your high-level models as you move towards implementation. Tightly linked to Questa (and ModelSim), their HDL co-simulation step lets you analyze system-level behavior in terms of your original model, closing the loop back to your original design goals.

Finally, as a special bonus, we’re including the runner-up Best Paper from this year’s DVCon, “Better Living Through Better Class-Based SystemVerilog Debug,” by my Mentor colleagues Rich Edelman, Raghu Ardeishar and John Amouroux. Congratulations, guys!

One more thought before I leave you: One of the jet airplanes they had at the Evergreen Museum was a Lear 24 from 1963. I hope you’ll permit me a bit of filial pride in pointing out that my dad designed several of the cockpit instruments, including the altimeter, artificial horizon and airspeed gauge. That got me thinking about the other electronics on display in everything from the Spruce Goose to the SR-71 Blackbird to the Lunar Module (including the TV camera, which, I must point out, Dave Rich’s dad helped design). Take a moment to consider how these instruments worked so well and accomplished so much without the “high tech” tools we sometimes take for granted today, such as those that are sure to be featured in the DAC exhibit hall. If you are attending, as always, please stop by the Mentor Graphics and Verification Academy booths and say hi.

Respectfully submitted,
Tom Fitzpatrick
Editor, Verification Horizons
Table of Contents June 2012 Issue

<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 6</td>
<td>Mentor Has Accellera’s Latest Standard Covered</td>
<td>by Darron May, Product Marketing Manager, Verification Mgmt, Mentor Graphics and Abigail Moorhouse, R &amp; D Manager, Mentor Graphics</td>
</tr>
<tr>
<td>Page 10</td>
<td>Is Intelligent Testbench Automation For You?</td>
<td>by Mark Olen, Product Marketing Manager, Mentor Graphics</td>
</tr>
<tr>
<td>Page 20</td>
<td>Targeting Internal-State Scenarios in an Uncertain World</td>
<td>by Matthew Ballance, Verification Technologist, Mentor Graphics</td>
</tr>
<tr>
<td>Page 25</td>
<td>Virtualization Delivers Total Verification of SoC Hardware, Software, and Interfaces</td>
<td>by Jim Kenney, Marketing Director, Emulation Division, Mentor Graphics</td>
</tr>
</tbody>
</table>
On The Fly Reset
by Mark Peryer, Verification Methodologist, DVT, Mentor Graphics

Relieving the Parameterized Coverage Headache
by Christine Lovett, Bryan Ramirez & Stacey Secatch, Xilinx, Inc.
and Michael Horn, Mentor Graphics

Four Best Practices for Prototyping MATLAB and Simulink Algorithms on FPGAs
by Stephan van Beek, Sudhir Sharma, and Sudeepa Prakash, MathWorks

Better Living Through Better Class-Based SystemVerilog Debug
by Rich Edelman, Raghu Ardeishar, John Amouroux, Mentor Graphics
Mentor Has Accellera’s Latest Standard Covered

by Darron May, Product Marketing Manager, Verification Mgmt, Mentor Graphics and Abigail Moorhouse, R & D Manager, Mentor Graphics

INTRODUCTION

If you can’t measure something, you can’t improve it. For years, verification engineers have used “coverage” as a way to measure completeness of the verification effort. Of course, there are many types of coverage, from different types of code coverage to functional coverage, as well as many tools, both dynamic and static, that provide coverage information. Simply put, coverage is a way to count interesting things that happen during verification and the measure of coverage is being able to correlate those things that happened back to a list of things you wanted to happen (also called a verification plan). Meaningful analysis requires a standard method of storing coverage data from multiple tools and languages, which is what Accellera’s Unified Coverage Interoperability Standard (UCIS) finally delivers.

Last month, after five and a half years of hard work, the UCIS technical subcommittee approved draft version 1.0 of the standard, which the Accellera board will likely approve at DAC 2012, thus officially providing the industry with its first standard method of storing, analyzing and reporting coverage across multiple tools and languages. UCIS is an unequivocal accomplishment and a massive step toward producing a unified way of handling coverage. What did the process involve? And what should be expected from the standard? For answers, it’s perhaps useful to look back at work done at Mentor Graphics to build a unified coverage database for our tools, work that took place long before the UCIS process started and that in fact has seeded Accellera’s march toward better coverage interoperability.

HISTORY OF THE UCIS

Accellera’s UCIS subcommittee was formed in November 2006 with two goals: identifying interoperability opportunities and defining coverage models and an interoperability standard for data exchange to encourage EDA innovation in the coverage space. The original group was made up of over 180 people from 18 member companies, including EDA companies and their users. There were many barriers to overcome, notably including aligning user and vendor requirements, and also dealing with nonstandard coverage models built with proprietary implementations of code coverage techniques and mixed languages. Even a coverage metric like SystemVerilog functional coverage, which is defined in a standard, differs slightly in implementation between vendors in identifying coverage in parameterizable classes. If coverage was ever to be unified between vendors, then every implementation would have to follow the same coverage representation.

Accordingly, during the first 18 months of the UCIS process, the subcommittee defined both a high level design of what a unified coverage database would need to look like and coverage terms. Next came a call for technology donations. Given the alignment between what the standard required and the capabilities of Mentor Graphics’ Unified Coverage Database (UCDB) technology, we decided to donate this technology to the industry. A year later, after a separate donation by Synopsys of their own Unified Coverage API (UCAPI) technology and the addition of UCDB extensions from Cadence to support their data models, Mentor Graphics’ UCDB was voted the basis of the UCIS standard. The last two years have been spent working through complex areas of the standard to ensure alignment with the initial user requirements and delivery of a standard that could be supported by all vendors.

HISTORY OF THE UCDB

Mentor Graphics’ UCDB was architected in 2005 to unify coverage collection within all the company’s verification tools. It was first released within Questa and ModelSim in early 2006 as a way of natively storing, analyzing and reporting on functional coverage, code coverage and assertions. The database has an open C Application Programming Interface (API) that has been used to develop all the UCDB utilities for Questa and ModelSim such as ranking, merging, analysis and reporting. The database not only ensured the definition of standard coverage models for all metrics generated by Mentor Graphics tools but also provided support of third party and user coverage, along with the ability to extend to coverage yet to be defined by any tools. This openness has allowed users to bring coverage data from other languages and vendors tools into the UCDB and thus use the database to completely unify
their coverage storage and analysis within their flows. After its initial release, other Mentor Graphics product groups added their support for the UCDB, including those focusing on formal verification, analog design and verification, and emulation.

The database was architected to support the management of verification and so, quite logically, became the basis of the Questa Verification Management (VM) tools. These tools allow data management, and also process and tool automation to improve verification productivity, specifically in the areas of capacity, throughput and turnaround time.

A complete verification management solution requires a database to store unified coverage; low level utilities to manage this data, including merging, ranking, analysis and reporting; and higher level applications, such as testplan tracking, trend analysis, results analysis and run management automation. While Mentor Graphics has been actively involved with the UCIS over the last few years, the company has been simultaneously building applications based around the UCDB.

UCDB OVERVIEW

From the users’ point of view, the database has three primary sections. One is the coverage data collection area. The second is for recording of test-specific information, such as test name, tool settings, CPU time, username and so on, and is extendable to store any user defined information. This section also includes the history of how the tests were generated, combined and/or merged together. The third section, used for testplan tracking, allows the storage of testplan items that can be linked to the coverage model and/or the test cases themselves.
Designs and testbenches are hierarchically organized. Design units (Verilog modules or VHDL entity/architectures) can be hierarchical, though they are not always. Test plans can be hierarchical. Even coverage data (of which the SystemVerilog covergroup is the best example) can be hierarchically organized.

The UCDB handles hierarchical structures using scopes (also referred to as hierarchical nodes), which store hierarchical structures (i.e., elements of a database that can have children). Coverage and assertion data are stored as counters, which indicate how many times something happens in the design.

The basic design of coverage hierarchy is shown below. Note that the counters count how many times a sequence completed, how many times a bin incremented or how many times a statement executed. In UCDB terminology, these types of counters and some associated data are called coveritems. These counters are database leaf nodes, which cannot have children.

One of the major benefits of the system is that it allows multiple tests to be merged together into a single database while still keeping an association between which testcase incremented which bin. This optimization reduces the amount of stored data but preserves the important test association information needed for analysis. It also allows a vendor to add great value in the way the database is organized and optimized. A prime example is Questa, which, after more than six years of real world optimizations, today delivers capacity and performance in a platform-independent database.

MENTOR GRAPHICS' SUPPORT FOR UCIS

Mentor Graphics today offers a prototype UCIS API library capable of running the examples in the UCIS draft standard and a lot more. We also have a clear migration path from the original proprietary UCDB to the UCIS API that will allow users to continue to access existing UCDB databases.

Indeed, we are already ahead of the UCIS with feature extensions like testplan management that will be donated to the standard in the next round (or in some cases, have already been donated, but were not included). These extensions are of course available now to Mentor Graphics' users. Both the UCIS and UCDB APIs provide views into and control of otherwise opaque databases. The same database can support both API views, although the APIs cannot be mixed within the same application.

Due to the source of the standard it is no surprise that Mentor Graphics also models coverage in the database using many of the data models published in the standard, sometimes with minor differences. Migration to complete data compliance is a longer process and will probably occur on an as-needed basis. The database does not inherently constrain the data that may be loaded into it so all models are technically UCIS-compliant. Universal Object Recognition (UOR) offers an additional guarantee that a specified naming convention has been applied though is not currently supported for all the coverage models.
SUMMARY
The release of UCIS 1.0 is the first important step towards unifying the coverage generated by multiple tools and vendors. Mentor Graphics has been trailblazing in this area for many years with its development of UCDB and applications built around the database that have become the basis of the standard. We will continue to support the UCIS efforts and ensure that other UCDB optimizations and features are put forward when the standard goes through its revisions. Questa users today have complete access to the UCIS API by the use of a prototype UCIS API library. And of course since Mentor Graphics complete verification management tool set is based around UCDB (and therefore UCIS), our users are already benefiting from infrastructure we built around the database even without using the UCIS API.
INTRODUCTION

Intelligent Testbench Automation (iTBA) is being successfully adopted by more verification teams every day. There have been multiple technical papers demonstrating successful verification applications and panel sessions comparing the merits to both Constrained Random Testing (CRT) and Directed Testing (DT) methods. Technical conferences including DAC, DVCon, and others have joined those interested in better understanding this new technology. An entire course curriculum is available at the Verification Academy. And many articles have been published by various technical journals, including in this and previous Verification Horizons editions. So with all of the activity, how do verification teams separate out the signal from the noise? How do they know whether Intelligent Testbench Automation is applicable to their designs? How do they know whether they will experience the advertised gains in verification productivity, as shown in Figure 1 to the right? This article does not discuss how Intelligent Testbench Automation works, as indeed there are several sources for that information just mentioned. This article discusses where iTBA is best applied and where it will produce optimal results, as well as where it is not and will not. As with most emerging technologies, iTBA yields better results in some applications than others.

VERIFICATION SPACE

The more choices to be made during verification the higher the impact realized from Intelligent Testbench Automation. A design that has a large number of configurations, modes, operations, commands, functions, variables, payloads, and other parameters, is the best application for iTBA. To effectively verify this type of design verification teams must test all important combinations of these choices. Examples would include SoC designs with multiple processors, busses, and peripheral interfaces; complex bus fabrics with multiple masters and slaves, and arbitration schemes; routers and more. iTBA rarely returns less than a 10X gain in productivity when presented with a large verification space with a large number of choices. Its natural ability to remove unwanted redundancy while preserving desired randomness maintains the Constrained Random Testing characteristic of generating tests for conditions not previously envisioned by verification engineers, while reducing the number of tests required to achieve or exceed coverage goals. In a perfectly balanced and symmetrical situation, iTBA achieves coverage goals faster than CRT according to the following equation -

\[
\text{Tests required by CRT to achieve coverage of } N \text{ scenarios} = N \ln N + \gamma \\
\text{Test required by iTBA to achieve coverage of } N \text{ scenarios} = N
\]

Where \( N \) equals the number of different combinations of conditions

And \( \gamma \) equals the Euler-Mascheroni1 constant = 0.57721…

According to the equation above, once the verification space exceeds 22,027 scenarios, iTBA achieves a minimum...
of 10X gain in productivity. As an illustration, please refer to Chart 1 below that shows a comparison between iTBA and CRT selection of a variable with 64 values. Chart 1 shows the selection distribution after 64 “tests”. The Graph-based iTBA selection achieves all 64 values in random order, while the Random CRT selection only reaches 42 values, missing 34% of the important values. Notice the uneven distribution of random selections in red, where 14 values have been tested multiple times, while 22 values have yet to be tested.

If the variable values were re-ordered in the chart, we would in fact see a red bell curve distribution across the range, compared to the flat blue distribution generated by iTBA.

However in general, it would certainly be pretty aggressive to expect to achieve 100% coverage of all important scenarios with perfect efficiency, so please refer next to Chart 2 below that shows what happens when the simulation is allowed to continue for another 64 “tests”. Notice that the Random CRT distribution continues to repeat as expected, and that even after running twice the number of tests needed, CRT still reaches only 83% coverage with 12 corner cases still not verified. Meanwhile, if iTBA is allowed to continue running, all values are now tested twice.

If you’re curious about how many tests it will take for Constrained Random Testing to finally reach those difficult to find corner cases, go back to the formula above and plug in N = 64. This experiment allowed the simulation to continue for 256 tests, at which point CRT achieved 96% coverage. So a single run of this simulation produced a result consistent with the formula above. And if the test were repeated several times with different seeds, the average result would converge to the formula’s calculation.

UNBALANCED CONDITIONS
While the testcases above are small in size, and most teams would not adopt a new technology for a 4x benefit, please keep in mind that this example depicts a perfectly balanced set of conditions. Each of the 64 values of our variable is equally likely to occur. For example, the protocol rule in Figure 2 generates transactions of varying burst lengths and payload sizes. The resulting protocol graph shown in Figure 3 shows a perfectly balanced protocol. If burst length is one, then there are three sizes of payload, for three equally weighted sub-combinations. And if burst length is two, three, or four, then there is only one valid size of payload, for three equally weighted sub-combinations, yielding six equally weighted total combinations.
Such a balanced protocol is very unlikely to exist (yet this one still yields a productivity gain of $N \ln N + \gamma$). However, most actual verification requirements include difficult-to-find corner-case scenarios caused by unbalanced conditions. These can be design-related or test-related. Some design functions require very specific pre-conditioning sequences to enable rarely encountered functions (not depicted in the protocol above). In addition, some test goals require cover points that vary in size by orders of magnitude. If verification engineers are spending lots of time writing directed tests to fill coverage holes not tested by Constrained Random Tests, then these conditions likely exist. Intelligent Testbench Automation can return upwards of 100X gain in productivity when presented with very unbalanced conditions.

TARGET COVERAGE

There are many different types of coverage to measure during verification, but they often can be grouped into two main categories - black-box coverage and white-box coverage. Questa is better targeted at black-box coverages, which includes design stimulus coverage and design response coverage. These coverages can be derived from functional specifications, test plans, or engineering know-how. They are all related to how a design functions, and can be observed at the inputs and outputs of a design. White-box coverages include RTL code coverage, internal state coverage, assertion coverage, and clock domain coverage. These are best addressed by other key technologies included in the Questa functional verification platform. RTL code coverage is monitored and reported by Questa Simulation, while Questa inFact generates stimulus. Internal state coverage and assertion coverage are achieved by Questa Formal. And clock domain coverage is modeled, measured, and reported by Questa CDC. And all of these coverages, both black-box and white-box are collected by Questa Verification Management for rapid merging and reporting. In the end it is the integration of several key technologies that enables the Questa Platform to completely verify complex designs.

EXISTING ENVIRONMENT

The type of verification methodology already in place should also be considered when adopting Intelligent Testbench Automation. In each case iTBA can provide a step-function gain in productivity, but the steps taken to maximize the gain will be different. Consider the following three cases -

**Case 1** - Constrained random test stimulus and functional coverage measurement are both in place, and simulation results and coverage reports are available. In this case the existing constraints and covergroups can be imported into the Questa inFact Intelligent Testbench Automation toolset. If the constrained random stimulus was achieving satisfactory coverage, then Questa inFact will be able to achieve the same coverage at least 10x faster as described...
above, enabling much shorter turnaround time, or enabling test expansion to cover functionality previously thought not testable. If the constrained random stimulus was not achieving satisfactory coverage, then Questa inFact will offer even more value by achieving the target coverage while still reducing the time by 10x and eliminating the need to write directed tests to target the random resistant corner-case scenarios.

**Case 2** - Constrained random test stimulus is in place, but functional coverage measurement is not. In this case the constraints can be imported into Questa inFact, and inFact’s coverage editor can be used to target stimulus generation and report coverage achieved. In addition, Questa inFact can generate SystemVerilog covergroups to be used to report and manage coverage results with Questa Verification Management. The SystemVerilog covergroups generated by Questa inFact can even be used by Questa Verification Management to measure the effectiveness of the original constrained random stimulus, validating the step-function gain realized when using Questa inFact to generate stimulus.

**Case 3** - Directed test stimulus is in place to select different test combinations and manage test sequencing. Typically there will be limited (or no) functional coverage measurement in place here. In this case the directed tests can be mapped into Questa inFact. Then inFact can automatically expand the number of test combinations without repetition. Questa inFact can also improve the test order, prioritizing important test combinations earlier in the simulation. In addition, Questa inFact can generate SystemVerilog covergroups to be used to report and manage coverage results with Questa Verification Management. Questa inFact will yield the best results when existing directed tests are short in duration, involving multiple variables with complex relationships. In these cases Questa inFact can quickly and efficiently multiply the directed tests by 100x or more, without repetition.

**VERTICAL MARKETS**
Design application segments have little direct impact on Intelligent Testbench Automation applicability. Telecom is not necessarily more applicable than consumer electronics, which is not necessarily more applicable than wireless, or mil-aero, or others. iTBA can be deployed at the block, subsystem, or system level to generate efficient stimulus and response that accelerates coverage closure, in spite of extreme design complexity. If there is any vertical market dependency, it would not be related directly to iTBA, but to other aspects of the verification environment. For example, iTBA is best employed with verification IP components to provide DUT-level interfacing such as bus masters and slaves, peripheral models, or transactors. Access to a ready supply of high quality verification IP will have more influence over vertical market applicability. That is one reason why Questa inFact is often deployed with Questa Verification IP, which provides high quality DUT-level interfacing, with multi-level viewing for ease of design debugging.

**SUMMARY**
Questa inFact’s Intelligent Testbench Automation is most appreciated by customers who use constrained random testing, but then have to write directed tests to cover those hard-to-find corner-case scenarios. That’s where Questa inFact shines. First, it helps verification engineers eliminate their wasted tests. Second, it helps them get to their coverage goals faster. Third, it relieves them from having to write directed tests to manually cover corner-case scenarios that are resistant to automated random testing. And fourth, it leaves them time to consider test expansion, thus enabling verification engineers to target even more functionality.

**END NOTES**
1 The Euler Mascheroni constant (also called Euler’s constant) is a mathematical constant recurring in analysis and number theory, usually denoted by the lowercase Greek letter γ (gamma). It is defined as the limiting difference between the harmonic series and the natural logarithm.
Automated Generation of Functional Coverage Metrics for Input Stimulus

by Mike Andrews, Verification Technologist, Mentor Graphics

Questa® inFact intelligent testbench automation has allowed many verification teams to achieve their initial coverage goals far more efficiently than would have been possible with traditional constrained random generation, providing an opportunity to expand those coverage goals to ensure a more comprehensive verification of the DUT. This shifts the verification engineer’s challenge from efficiently achieving coverage to efficiently and accurately defining the additional covergroups and coverpoints required. This process can be especially difficult when defining cross coverage goals where there are multiple constraints that limit the legal combinations of the variables concerned. Determining the correct expressions required to exclude the illegal combinations from the cross is both time consuming and error prone. Questa inFact allows for graphical definition of the coverage goals and can, with the 10.1 release, automatically generate SystemVerilog covergroups from this definition, including the exclusions needed to accurately represent the achievable coverage. This article describes how this capability can simplify the definition of more comprehensive stimulus coverage metrics.

INTRODUCTION

Verification teams are always under pressure to meet their project schedules, while at the same time the consequences of not adequately verifying the design can be severe. This puts the team between a rock and a hard place as they say. The main value of Questa inFact is to help with the problem of meeting the schedule requirements by more efficiently, and more predictably, generating the tests needed to meet coverage goals in the case where coverage metrics are being used to determine ‘completeness’ of the verification project. An indirect benefit of this has always been that, when planning to use this capability on a project, a verification team can expand the scope of the functional coverage metrics and therefore can expect to avoid the prior mentioned severe consequences of letting serious bugs slip through. This has however appeared to move the bottleneck in the process to the creation of these expanded coverage metrics. If many additional cross coverage goals are added this can be especially time consuming, and actually many bugs can be, and have been in my experience, introduced into the coverage scoreboard during this process. This issue has definitely slowed down the overall move to true coverage driven verification processes in some organizations, forcing them to fall back on easier, but less reliable, metrics such as code coverage.

DEFINING CROSS COVERAGE IN THE PRESENCE OF COMPLEX CONSTRAINTS

The real difficulty in defining cross coverage goals is to get the exclusions correct, and this can be really difficult when the variables in the cross have many complex constraints that determine the relationships that must be maintained between them. If these exclusions are not properly identified then the ability to accurately gauge the actual coverage achieved becomes impossible since the missing coverage may contain a significant percentage of illegal cases. Or, quite often, the exclusions defined are incorrect and important cross coverage combinations are not tracked by the metrics, further obfuscating the actual coverage as reported by the verification metrics.

To illustrate the difficulty let’s consider a very simple three variable example:

```verilog
class my_item extends uvm_sequence_item
    rand bit A;
    rand bit B;
    rand bit [1:0] C;

    constraint limC { C < 3; }
    constraint relate_vars {
        if (A == 0) {
            B == 0;
            C == 0;
        }
    }
endclass: my_item
```
Our coverage goal is to test all of the legal combinations of all three variables A, B and C. The definition of the legal cross coverage goals is not trivial even for this case, since we have to determine which combinations of A, B, C are unreachable. This can take a little thought to determine by hand. The illegal combinations turn out to be:

\[
A[0], B[1], C[0,1,2] \\
A[0], B[0], C[1,2]
\]

This leaves 7 valid combinations out of a total of 12, which would mean that if the exclusions are not specified then the maximum coverage attainable for the cross is about 58%. This is actually a fairly common percentage for most real testbenches that I have encountered, and verification engineers often struggle to determine if this result is actually a good level of coverage or whether the missing coverage contains a significant portion of important legal variable combinations. Questa inFact has been able to help with this problem for quite some time since it can provide an accurate count of the legal solutions in a cross, as shown in Figure 1 below.

In Figure 1 we have three views of the same stimulus item variables. The first (top left) shows the full state space with a count of the total number of combinations as described in the stimulus graph. The second view (top right) shows an annotation of a cross coverage goal on the graph. The third view (bottom) is a count of the legal combinations as determined from a combination of the variable types and the constraints placed on the combinations of those variables. From this an engineer can determine at least the actual achievable coverage for a particular cross. There is still a potential issue however if there is an error in the definition of the exclusions so that we end up with a similar count in the metrics, but we have inadvertently flipped a bit somewhere in our exclusion logic. If this is the case then we still end up with a mismatch in the coverage metrics score reported vs. the real situation, and the engineer still has to try to determine where the problem lies, i.e. in the stimulus generation process or the covergroup itself.

**AUTOMATED GENERATION OF SYSTEMVERILOG COVERGROUPS**

A recent enhancement to Questa inFact brings significant additional value for this kind of problem. The user now has the option to create a SystemVerilog covergroup that is based on the inFact coverage strategy defined in the tool’s IDE. The covergroup so created will include automation of the necessary exclusions based on the constraints defined on the variables in the stimulus class. Following is the actual coverage code as created by inFact for this simple case.

```
// Path Coverage cross_ABC
cross_ABC_A_cp : coverpoint m_cov_item.A {
  option.weight = 2;
  bins A[] = {0, 1};
}
cross_ABC_B_cp : coverpoint m_cov_item.B {
  option.weight = 2;
  bins B[] = {0, 1};
}
```
cross_ABC_C_cp : coverpoint m_cov_item.C {
    option.weight = 3;
    bins C[] = {0, 1, 2};
}
cross_ABC : cross cross_ABC_A_cp, cross_ABC_B_cp, cross_ABC_C_cp {
    option.weight = 7;
    ignore_bins unreachable_bins =
        (binsof(cross_ABC_A_cp) intersect {0} &&
         binsof(cross_ABC_B_cp) intersect {0} &&
         binsof(cross_ABC_C_cp) intersect {1,2}) ||
        (binsof(cross_ABC_A_cp) intersect {0} &&
         binsof(cross_ABC_B_cp) intersect {0} &&
         binsof(cross_ABC_C_cp) intersect {0,1,2})
    ;
}

Figure 2. inFact Generated Covergroup Code

In Figure 2, note the ignore_bins statement in the cross coverage definition. Having written some of these types of statements for more complex situations myself, I definitely appreciate the apprehension with which a verification engineer might have approached that task without tools to help. This fear may certainly have been weighed against the promise of taking advantage of the power of inFact's algorithms to confidently define more, and larger cross coverage goals in their functional coverage metrics.

SAVING TIME IN FUNCTIONAL COVERAGE CREATION

This new capability in Questa inFact opens up the possibility of an improved process for creating functional coverage metrics, at least for the covergroups that track stimulus coverage for the sequence_items in a testbench.

Once the sequence_item itself has been created, another facility of inFact can be used to import the information from the sequence_item, including the definition of the randomize-able variables, and the constraints placed upon them. From this imported information a graph is automatically created to generate valid values for that item.

A coverage strategy can then be determined based on the graph variables, using the tools in the inFact IDE to get an accurate picture of the number of legal combinations in the cross coverage goals being defined.

Figure 3 on the opposite page shows a more complex sequence_item example which is part of a testbench for an ethernet controller. This example shows a clearer picture of how these tools can be used.

Figure 4 on page 19 shows an inFact coverage strategy defined for a testbench generating ethernet traffic for a controller DUT.

In Figure 4, there are two cross coverage goals defined, each of which targets a different subset of the graph variables. For each cross coverage goal in the strategy, the colored region defines the variables to be considered, with some variables in the region defined as ‘Don’t Care’ as denoted by a white oval around the variable. For visual clarity, each of the goals can be viewed separately if required, which is important if there are many different combinations.

The specific bins for each of the cross coverage goals have been iteratively defined, using the path counting feature (shown at the bottom of the figure) to roughly balance the sizes of the goals. The largest of these goals gives a general idea of the number of items that need to be generated and simulated to meet the coverage goals for this particular item (although, depending on the constraints between all the variables, meeting all the goals may take a few more items than the 288 shown in Figure 4).

From the coverage strategy defined in inFact, a covergroup can now be automatically created that will track the actual coverage metrics as determined by the SystemVerilog coverage features of the user’s simulation environment. All this without the need to painstakingly analyze the constraints to manually determine the exclusions needed for accurate tracking of the coverage progress.

To get a sense of the potential time saved, take a look at the ‘eye-chart’ in Figure 5 on page 18, which is the actual generated coverage code that defines one of the crosses for this example, with the ignore_bins statements required to properly exclude illegal combinations.
typedef enum {
    TX,
    RX,
    RXTX
} ethmac_rxtx_scenario_t;

class ethmac_rxtx_seq_item extends uvm_sequence_item;

    `uvm_object_utils(ethmac_rxtx_seq_item)

rand ethmac_rxtx_scenario_t scenario_type;
// Enables
rand bit tx_crc;
rand bit tx_pad;
// Control the retry limit to program, and the
// number of retries the MII driver must provoke
// Retry limit to be programmed into the BD
rand bit[3:0] tx_rtry;
// Retry count for the MII driver
rand bit[4:0] tx_rtry_count;
rand bit[15:0] tx_payload_sz;
rand bit tx_irq_en;
// Provokes a CRC error
rand bit rx_crc;
rand bit[15:0] rx_payload_sz;
rand bit rx_irq_en;

// Constraints to zero some fields based on scenario type

c constraint scenario_fix_c {
    if (scenario_type == RX) {
        tx_crc == 0;
        tx_pad == 0;
        tx_rtry == 0;
        tx_rtry_count == 0;
        tx_payload_sz == 4;
        tx_irq_en == 0;
    } else if (scenario_type == TX) {
        rx_crc == 0;
        rx_payload_sz == 4;
        rx_irq_en == 0;
    }
}

c constraint bug_c {
    // Appear to be issues with enabling pad. TX State Machine hangs.
    tx_pad == 0;
}

c constraint valid_c {
    tx_payload_sz inside {[4:8192]};
    rx_payload_sz inside {[4:8192]};
}
endclass

Figure 3. Sequence Item
Example for Ethernet Testbench
cfg_cross : cross cfg_cross_tx_crc_cp, cfg_cross_tx_pad_cp, cfg_cross_tx_rtry_cp, cfg_cross_tx_rtry_count_cp, cfg_cross_tx_irq_en_cp, cfg_cross_rx_crc_cp {
    option.weight = 256;
    ignore_bins unreachable_bins =
        (binsof(cfg_cross_tx_crc_cp) intersect (0) & binsof(cfg_cross_tx_pad_cp) intersect (1) &
            binsof(cfg_cross_tx_irq_en_cp) intersect (0) &
            binsof(cfg_cross_rx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_rtry_cp) intersect
            ([0:3],[4:7],[8:11],[12:15]) &
            binsof(cfg_cross_tx_rtry_count_cp) intersect ([0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]) &
            binsof(cfg_cross_tx_crc_cp) intersect (0) &
            binsof(cfg_cross Tx_pad_count) intersect (1) &
            binsof(cfg_cross_tx_irq_en_cp) intersect (1) &
            binsof(cfg_cross_rx_crc_cp) intersect ([0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]) &
        || (binsof(cfg_cross_tx_crc_cp) intersect {0} &
            binsof(cfg_cross_tx_pad_cp) intersect {1} &
            binsof(cfg_cross_tx_irq_en_cp) intersect {0} &
            binsof(cfg_cross_rx_crc_cp) intersect {1} &
            binsof(cfg_cross_tx_rtry_cp) intersect
            ([0:3],[4:7],[8:11],[12:15]) &
            binsof(cfg_cross_tx_rtry_count_cp) intersect ([0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]) &
            binsof(cfg_cross_tx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_pad_cp) intersect (1) &
            binsof(cfg_cross_tx_irq_en_cp) intersect (1) &
            binsof(cfg_cross_rx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_rtry_cp) intersect
            ([0:3],[4:7],[8:11],[12:15]) &
            binsof(cfg_cross_tx_rtry_count_cp) intersect ([0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]) &
            binsof(cfg_cross_tx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_pad_cp) intersect (1) &
            binsof(cfg_cross_tx_irq_en_cp) intersect (1) &
            binsof(cfg_cross_rx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_rtry_cp) intersect
            ([0:3],[4:7],[8:11],[12:15]) &
            binsof(cfg_cross_tx_rtry_count_cp) intersect ([0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]) &
            binsof(cfg_cross_tx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_pad_cp) intersect (1) &
            binsof(cfg_cross_tx_irq_en_cp) intersect (1) &
            binsof(cfg_cross_rx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_rtry_cp) intersect
            ([0:3],[4:7],[8:11],[12:15]) &
            binsof(cfg_cross_tx_rtry_count_cp) intersect ([0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]) &
            binsof(cfg_cross_tx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_pad_cp) intersect (1) &
            binsof(cfg_cross_tx_irq_en_cp) intersect (1) &
            binsof(cfg_cross_rx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_rtry_cp) intersect
            ([0:3],[4:7],[8:11],[12:15]) &
            binsof(cfg_cross_tx_rtry_count_cp) intersect ([0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31]) &
            binsof(cfg_cross_tx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_pad_cp) intersect (1) &
            binsof(cfg_cross_tx_irq_en_cp) intersect (1) &
            binsof(cfg_cross_rx_crc_cp) intersect (0) &
            binsof(cfg_cross_tx_rtry_cp) intersect
            ([0:3],[4:7],[8:11],[12:15]) &
            binsof(cfg_cross_tx_rtry_count_cp) intersect ([0:3],[4:7],[8:11],[12:15],[16:19],[20:23],[24:27],[28:31])

    Figure 5. Cross Coverage Goal Definition For Ethernet Config
What would normally take a number of hours to define, and subsequently debug, can now be automatically generated, leaving time to focus on other aspects of coverage such as embedding assertions or more critical analysis of the overall verification goals.

**CONCLUSION**

If meeting coverage goals is one of the current pain points in a verification project, then intelligent testbench automation solutions such as Questa inFact can significantly help address this. This latest capability — the automated generation of coverage code — is intended to proactively attack what may be the next bottleneck, i.e. the efficient and accurate definition of more comprehensive coverage goals that the intelligent automation can then target (with equal efficiency).

*Figure 4. Coverage Strategy for Ethernet Rx/TX Sequence Item*
Targeting Internal-State Scenarios in an Uncertain World

by Matthew Ballance, Verification Technologist, Mentor Graphics

The challenges inherent in verifying today’s complex designs are widely understood. Just identifying and exercising all the operating modes of one of today’s complex designs can be challenging. Creating tests that will exercise all these input cases is, likewise, challenging and labor-intensive. Using directed-test methodology, it is extremely challenging to create sufficiently-comprehensive tests to ensure design quality, due to the amount of engineering effort needed to design, implement, and manage the test suite. Random test methodology helps to address the productivity and management challenges, since automation is leveraged more efficiently. However, ensuring that all critical cases are hit with random testing is difficult, due to the inherent redundancy of randomly-generated stimulus.

Questa inFact Intelligent Testbench Automation, part of the Questa verification platform, provides a comprehensive solution for efficiently and comprehensively exercising the functional input space of a design – the input commands and operating modes. Questa inFact’s efficient graph-based stimulus description enables 10-100x more unique stimulus to be created in a given period of time than could be created using directed tests. The advanced coverage-targeting algorithms within Questa inFact achieve input functional coverage 10-100x faster than random stimulus, and enable this benefit to be easily-scaled to the simulation farm.

Many of the most-interesting verification scenarios, however, are scenarios involving design-internal state. These internal-state scenarios often end up being verified with directed tests, due to the difficulty in coercing random tests to reliably target the desired scenarios. Often, the difficulty in exercising these internal-state scenarios lies in properly combining the inputs required to achieve the pre-conditions for the internal-state scenario with the stimulus required to make progress towards coverage of the scenario. For example, a customer I recently worked with found that in one case, their entire regression suite only covered 5% of a moderately-sized internal-state coverage due to the dual requirements of creating pre-conditions, then hitting an interesting internal-coverage case once the pre-conditions were met.

In this article, we will look at how two capabilities of Questa inFact Intelligent Testbench Automation can be used to more-efficiently target verification scenarios involving design-internal state.

PIPELINED COMMAND PROCESSOR EXAMPLE

The example that we will examine in this article is a command-processing pipeline. The pipeline, in this case is a 5-stage pipeline that processes a command with operands. This particular processor supports eight commands – CMD1 through CMD8. Under ideal circumstances, a new input command is accepted by the pipeline every cycle, and a single command completes every cycle. As with all pipeline-based processors, however, stalls can occur in this pipeline when one of the stages takes longer than one cycle to complete.

One of our verification tasks for this pipeline involves ensuring that certain command sequences proceed through the pipeline. Specifically, we want to ensure that all combinations of back-to-back commands are exercised. For example, CMD1 followed by CMD1, CMD1 followed by CMD2, etc. We also want to exercise these same back-to-back command sequences with one, two, and three different commands in the middle. Figure 1 on the opposite page summarizes the sequences that we wish to verify. The blue-shaded commands below are the ones that we care about from a coverage perspective. The grey-shaded boxes are the commands whose specific value we don’t care about, apart from ensuring that these commands are different from the commands that begin and end the command sequence.

We are using a UVM environment for this block. Stimulus is described as a UVM sequence item that contains a field that specifies the command (cmd) as well as fields for both command operands (operand_a, operand_b). A UVM sequence running on the sequencer is responsible
This verification scenario presents a couple of challenges. First, describing the full set of desired sequences is a challenge. We could, of course, carefully create the scenario sequences using directed tests, but this would be a significant amount of work. We could leverage random generation on a single-command basis and hope to hit all the cases. However, the efficiency with which we can achieve our goals is hampered by the redundancy inherent in random generation and the fact that the constraint solver doesn’t comprehend the overall sequential goal that we are targeting. The second challenge involves the pipeline stalls. From our perspective as a test writer, these stalls are unpredictable. Despite our careful efforts to design a command sequence to apply to the pipeline, what is actually processed by the pipeline may be quite different than what we intended.

**DESCRIBING THE STIMULUS SPACE**

The task of describing and generating the command sequences is a classic input-stimulus problem. First, we create a set of inFact rules that describe the sequence of five commands. The rule description specifies the variables for which inFact will select values and the constraints between the variable values (in this case, for simplicity, there are no validity constraints).

At the top of the rule description, we declare graph variables, using the meta_action keyword, corresponding to the fields in the cmd_item sequence item: cmd, operand_a, operand_b. We also need to check the state of the pipeline when we issue a command. The cmdX_stall_i meta_action_import variables bring the current state of the

---

**Figure 1 - Command-Sequence Examples**

for generating this stimulus, while the driver converts the command described in the sequence item to signal-level transactions that are applied to the command-processor’s interface.

---

**Figure 2 - UVM Environment**
pipeline into inFact from the testbench. Since we are describing a sequence of five commands, we create five sets of variable declarations to represent cmd1 through cmd5.

We use symbols to group our variables together. Each symbol defined below in Figure 3 (Cmd1 through Cmd5) declares the sequence of operations needed to issue a single command to the command processor. Specifically: Call the UVM Sequence API start_item task, sample the stall state of the pipeline, select values for cmd, operand_a, and operand_b, then call the UVM Sequence API finish_item task to send the sequence item to the command processor.

Finally, at the bottom of the rule file we describe the top-level operation sequence. The most important aspect of this operation sequence is the repeat loop that contains references to the Cmd1 through Cmd5 symbols. During execution, this will cause inFact to repeatedly generate sequences of five commands.

Figure 4, shown opposite, provides a visual representation of our stimulus space. We can see the top-level sequence of operations described at the bottom of the rule file.

**TARGETING VERIFICATION GOALS**

Next, we need to describe the set of stimulus for Questa inFact to generate, which corresponds to the verification goals outlined above. At a high level, we are interested in crossing the following variables in order to realize the command sequences described earlier:

- Cmd1 x Cmd2
- Cmd1 x Cmd3
- Cmd1 x Cmd4
- Cmd1 x Cmd5

However, we also need to account for the requirement that commands in the middle of our command sequences must be different from the starting and ending commands in the sequence. Questa inFact provides a special type...
of constraint, called a coverage constraint, which provides an added level of flexibility and productivity when describing stimulus-creation scenarios like that above. A coverage constraint only applies when inFact is targeting a specific stimulus-generation goal, which enables stimulus to be very targeted for the portion of the simulation when inFact is targeting a specific goal, but revert to being less-constrained once inFact achieves that goal.

We create four constraints like the one shown below to describe the specific restriction that commands in the middle of a sequence must be different than the commands at the beginning and the end of the sequence.

The constraint shown in Figure 5 describes the restrictions on a three-command sequence. In this case, our verification goals call for the command in the middle of the sequence (cmd2) to be different than the command at the beginning of the sequence (cmd1) and the command at the end of the sequence (cmd3). This constraint, and the other three like it, are linked to the corresponding cross-coverage goals that describe our verification goals.

**REACTIVE STIMULUS**

Of course, our efficiently-described comprehensive stimulus isn’t much use if the design doesn’t actually accept what we apply. Fortunately, Questa inFact supports generating reactive stimulus based on state information from the environment. inFact is able to react to the current design state in cases where this must be done in order to create valid stimulus. In addition, inFact is also able to make coverage-targeting decisions based on the input from the environment. This enables inFact to take advantage of the current state of the environment to make rapid progress to coverage, even when inFact isn’t able to directly control the environment state. In other words, inFact is constantly looking for an opportunity to make progress towards the user-specified verification goals, and makes choices based on the current state to target verification goals that have not yet been satisfied.

In this case, the environment provides a way to query whether the first stage of the pipeline is stalled. Feeding this design-state information to inFact before each command is issued allows inFact to properly target our back-to-back command verification goals. Since the pipe-stage stall information tells us whether our verification scenario was properly applied, we reference the stall information in our coverage constraints. If the pipeline stalls during application of a command sequence, the coverage constraint will evaluate to false, causing inFact to retry that command sequence at a future time.
CONCLUSION
As we’ve seen from this example, Questa inFact provides specific features that simplify the process of targeting coverage involving design-internal states. Coverage constraints simplify the process of describing complex verification goals. Questa inFact’s reactive stimulus generation enables inFact to react to the design state and generate stimulus that makes progress towards the verification goals whenever possible. And, as always, inFact’s redundancy-eliminating algorithms enable efficient coverage closure for verification goals with and without design-state dependencies. The customer I mentioned at the beginning of the article applied inFact to the verification problem where their full regression achieved only 5% of a particular internal-state coverage goal. With a small amount of integration work and one short inFact simulation, they were able to achieve full coverage of that verification goal. For them, achieving this type of difficult-to-hit coverage goals was critical to the success of their project. The ability to achieve the verification goal efficiently – both in terms of engineering investment and simulation efficiency – was truly intelligent testbench automation.
With the majority of designs today containing one or more embedded processors, the verification landscape is transforming as more companies grapple with the limitations of traditional verification tools. Comprehensive verification of multi-core SoCs cannot be accomplished without including the software that will run on the hardware. Emulation has the speed and capacity to do this before the investment is made in prototypes or silicon.

This means that, theoretically speaking, emulation’s time has come. However, there has been an intractable and very practical barrier to that arrival. The high-cost of emulators has made them affordable only for companies with deep pockets. Fortunately, recently introduced virtualization technologies are changing that by demolishing this barrier. It won’t be long before emulators will be a common fixture at small and medium sized companies, as well as larger ones. In this article we will look at how and why that is.

THE RISE OF THE MACHINES

Full SoC verification, of hardware and software, requires connecting the chip to its end environment, including all of the peripheral devices; such as USB, Ethernet, displays, hard drives, and PCI Express. A device driver cannot be run against an SoC and fully tested unless the device driver has a device to talk to. As those devices are outside of the SoC, they need to be connected to it in order to be tested.

Typically, this has been done with physical hardware and speed bridges or speed adapters. For emulation, it’s been done in a mode called “in circuit emulation” (ICE), where physical devices are cabled to a speed adapter, which are in turn cabled into the emulator. This setup consumes expensive lab space and takes time to configure and debug. With all the cables and external hardware, it is the least reliable part of the emulation environment. And it’s difficult to debug if something goes wrong. These physical peripheral setups are not only inflexible but also costly to replicate. Each supports only a single user, locking the machine down to a given project or a single SoC. Emulators cost so much money that they need to support many users, yet ICE limits multi-user flexibility.

UNLOCKING THE DOOR TO GLOBAL EMULATION ENVIRONMENTS

The golden key is to virtualize all of this hardware. In this scenario, hardware accurate models of the peripherals run on a standard workstation, such as Linux, so that the SoC and software device drivers running in the emulator can interact with these hardware accurate models, just like with ICE, except that they are all implemented virtually. There is no physical hardware, no cables, and no hardware speed bridges for these devices. Because the virtual lab is entirely...
in software, a single emulator can support many users. The complete emulation environment can be instantly replicated and reconfigured.

The virtual lab setup enables emulators to be shared around the world and around the clock. The emulation environment is now data center compatible, as opposed to being confined to the role of a lab bench dedicated to a single project. Because of the ease of replication and configuration, engineers from anywhere in the world can take turns using the same emulator.

Each user needs only a single workstation connected to an emulator to verify their entire SoC environment. These workstations are highly reliable, low cost, and very compact. Instead of a mass of cables, speed adaptors, and a lab bench full of boards and stuff, there is only the emulator and a small rack of unit-high workstations.

With the emulation environment now housed in a data center format, it has to be reconfigurable by software, because users will not have physical access to it. With the virtual lab, teams around the world can reconfigure it via software. It is officially shared by multiple projects and geographies, and it can be reconfigured for another project with a different set of peripherals instantaneously; in the same amount of time it takes it to load a new design into the emulator.

**JUST AS ACCURATE, JUST AS FAST AS THE REAL THING**

Virtual models are hardware accurate because they are based on licensed design IP; therefore they are just as accurate as ICE peripherals. Virtual models use the actual, synthesizable RTL that SoC designers license and put into their chips. Because it is synthesizable RTL, the peripheral model can be compiled into the emulator, and the DUT can talk to a full, accurate, RTL hardware representation of the peripheral, such as a USB 3.0 controller that is in the emulator.

Continuing with USB 3.0 as an example, on the workstation side, there is a USB software stack and software that targets it for a mass storage function client or USB memory stick. The result is a functionally accurate USB memory stick; all done virtually between the RTL synthesized into the emulator linked over a co-model channel to the software stack and function client running on the workstation. This is why users do not lose any accuracy by going to the virtual approach.

Virtual devices are also very fast. They are not faster than ICE, but they are just as fast. Emulation speed is typically limited by how fast a design can run in the emulator, not by the virtual device or the communication to the workstation.

To demonstrate this, we can take a Veloce customer’s experience using Mentor’s VirtuaLAB Ethernet capability to generate Ethernet traffic to exercise an edge router chip. They activated twelve 10G ports on their chip, with a million packets passing through each port, for 12 million packets total. It took 45 seconds to emulate it on Veloce. The customer estimated that it would take about thirty days to run a single port, with one million packets, in simulation. Doing all twelve ports would have been impractical to simulate.
The customer’s edge router chip was configured by their router control application running on a Linux workstation, just like their routers are configured by the same application. Via the co-model host they linked the Linux box to the router chip design in the emulator. It looked exactly like running their chip in the lab, except that it was all done virtually and done long before the hardware prototypes or silicon was available. The chip was compiled into the emulator, and the Virtual Lab Ethernet capability was shared between the Ethernet transactors in Veloce—written in RTL and compiled in the emulator—and a traffic generator on the co-model host. This setup allowed them to do all kinds of analysis, make modifications, analyze performance and bottlenecks, and fix their chip months before silicon.

To do the same thing with the classic, physical ICE approach they would need to purchase Ethernet testers, which are fairly expensive. Then they would need a bank of speed bridges to act as the speed adapters between the Ethernet testers and the emulator. In order to support multiple users, as they could easily do with a virtual lab, they would have to duplicate that entire environment of Ethernet testers, speed bridges, and cables. When designs start getting up to 24, 48, 96, and even 128 ports, it becomes impractical to do this via ICE. The virtual lab is a much better approach.
Another benefit of the virtual approach pertains to how simulators and emulators support a capability called save and restore, or check point restart. With check point restart, a user does not have to re-emulate their RTOS every time they want to check out the device drivers and applications they wrote— as the RTOS only needs to be debugged once and it can take from one to ten hours to emulate the RTOS boot just to get to the device drivers. However, if physical peripherals are connected to the emulator via the classic in-circuit approach, they are not going to be able to checkpoint their state and they are not going to be able to restore or restart their state. Physical peripherals simply cannot save register and restore states.

Virtual lab peripherals support save and restore of the complete environment, including the peripherals connected to the SoC as it sits in the emulator. This makes save and restore practical on a full-chip, with peripherals, at the emulation level.

CONCLUSION

A virtual lab emulation environment is very cost effective, providing simultaneous access to a single emulator for many software engineers. Veloce and the VirtuaLAB technologies support total verification of the hardware, the software, and the peripheral interfaces— all in a reliable, flexible, reconfigurable, easily replicated virtual environment that is as fast and as accurate as hardware without the limitations. By delivering efficient multi-user support and data center compatibility, virtual peripherals will usher in emulation for companies large and small. Around the world. Around the clock.
INTRODUCTION
A common verification requirement is to reset a design part of the way through a simulation to check that it will come out of reset correctly and that any non-volatile settings survive the process. Almost all testbenches are designed to go through some form of reset and initialization process at their beginning, but applying reset at a mid-point in the simulation can be problematic. The Accellera UVM phasing sub-committee has been trying to resolve how to handle resets for a long time and has yet to reach a conclusion.

In a UVM testbench, stimulus is generated by sequences which create and shape sequence items which are sent to a driver for conversion into pin level activity compliant with a specific protocol. Sequences pass sequence items to drivers via a hand-shake and arbitration mechanism implemented in the sequencer. This mechanism is based on the assumption that each protocol transaction will complete. If a reset occurs, then a sequence will need to be terminated early. Killing a sequence abruptly potentially results in the driver trying to return a handshake or a response to a sequence that has been deleted from the sequencer, causing a fatal error in the simulation. One way to avoid this problem, is to implement the driver so that when it detects a reset it terminates the in-progress sequence items with a reset status so that the stimulus control process can handle the reset appropriately.

HOW TO HANDLE A RESET
A reset can be considered as a highly disruptive event that occurs unexpectedly. Typically, a hardware reset will cause a DUT to immediately stop what it is doing and re-apply default values to its register contents. On exit from reset the DUT will most likely need to be re-initialized or re-configured before it can start normal operations. There are two ways in which a reset can be generated in a UVM testbench; the first is when the UVM stimulus thread is in control of the reset and the second is when the static side of the testbench generates a reset asynchronously to the flow of the UVM stimulus process.

Generally speaking, if the reset is generated under the control of the UVM testbench, the stimulus control process can make sure that all of the stimulus generation machinery is in a quiescent state before asserting the reset signal. The control process would de-assert reset and then go through the required initialization process before resuming the rest of the test. In many circumstances this may be good enough, but there may be situations where an external asynchronous reset is necessary to check DUT behavior, or resets may be generated by the UVM testbench at random intervals.

An asynchronous reset places two requirements on the UVM testbench code – it must first detect that a reset has occurred and then it must react to the reset. In other words, a testbench needs to be coded so that components such as drivers, monitors and scoreboards are reset aware and the stimulus generation process needs to be able to react to the occurrence of a reset.

The stimulus generation process in a UVM testbench may well use a hierarchy of sequences and there could be multiple sequences sending sequence items to a driver at any point in time. At the top level of the sequence hierarchy there will be a master execution thread that is coordinating the execution of all the different sequence threads, usually in the test or in a virtual sequence. A hardware reset will be detected by the driver since it is driving and monitoring the pin level interface. At the start of a reset, the driver needs to halt its operations and then signal back to all the sequences that are sending sequence items that a reset is in progress. In turn the sequences need to terminate, returning a reset status. If the sequences are hierarchical, then they need to terminate with a reset status recursively until the main control loop is encountered. The control loop then needs to change the stimulus generation flow so that a new re-initialization process is started before re-applying traffic stimulus.

On the Fly Reset
by Mark Peryer, Verification Methodologist, DVT, Mentor Graphics
Figure 1 - Comparison of sequence execution flow for normal completion and a reset

If a driver is not active, it will detect a reset but it will not be able to return the reset status. If all the drivers, sequences and sequence items in the testbench are reset aware, then it is very likely that at least one of them will return a reset status that will allow the stimulus process to go into a reset handling thread. However, if this is not the case and the reset status is important, then the main stimulus generation loop should monitor the state of the reset signal in parallel with its normal activities and initiate a change in stimulus if a reset occurs.

IMPLEMENTING RESET HANDLING

The key to being able to handle an asynchronous reset in UVM testbenches is to be able to flag detection of a reset condition back to the process that controls the test and the stimulus generation. The most convenient way to do this is to add a status field to sequence items and sequences to communicate that a reset has been detected. This status field can then be tested as part of the stimulus generation process to determine what to do. The UVM base classes do not contain a suitable field, so this is something that the user has to add. The recommended approach is to add a variable called `status` of type `uvm_tlm_response_status_e` to sequence items and sequences. When a sequence item or a sequence completes successfully, the status field would be set to UVM_TLM_OK_RESPONSE. When a reset occurs then status value of UVM_TLM_INCOMPLETE_RESPONSE should be set.

In the body method of a reset aware sequence, the status field of each sequence item should be checked after driver has completed its execution – usually on return from the `finish_item()` or the `get_response()` call. If the status returned is normal, then the sequence continues as usual. If the status field indicates that a reset has occurred then the sequence should stop what it is doing and return, having first set its status field to indicate that a reset has been encountered. The next level in the stimulus generation hierarchy must also test the status of the returning sequence and determine what to do with a reset status. For instance, instead of starting the next planned traffic sequence, it may start an initialization sequence which will start as soon as the hardware reset sequence has completed.
Both drivers and monitors need to be able to detect a reset condition and they need to be able to cope with the consequences. For a driver this means that a protocol pin level transfer has to be abandoned part way through, and that the status field of the current sequence item has to be updated to indicate that a reset has been encountered. The driver also needs to clear down any other sequence items that might be in the seq_item_port FIFO, marking them with a reset status. With more advanced protocols with out of order responses, there may be several outstanding sequence items queued up within the driver, and all of these will have to be terminated with a reset status. These requirements usually mean that the main driver loop has to

```verilog
class adpcm_tx_seq extends uvm_sequence
#(adpcm_seq_item);

   `uvm_object_utils(adpcm_tx_seq)

   // ADPCM sequence item
   adpcm_seq_item req;

   // Status – used to signal reset status
   back to originating thread:
   uvm_tlm_response_status_e status;

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

   task body;
   req = adpcm_seq_item::type_id::create("req");
   for(int i = 0; i < 10; i++) begin
     start_item(req);
     if(!req.randomize()) begin
       `uvm_error("body", "req randomization failure")
     end
     finish_item(req);
     status = req.status;
     if(status == UVM_TLM_INCOMPLETE_RESPONSE)
       begin
       `uvm_warning("body", "Interface reset occurred")
       return;
       end
     end
   endtask: body

   endclass: adpcm_tx_seq

Figure 3 - Reset aware sequence

Figure 4 – Test level control thread which restarts the sequence if it returns with a reset status
Figure 5 - Reset aware driver implemented using FSM

be implemented with a state machine so that it can treat the reset condition as a temporary state.

The purpose of a monitor is to track pin level activity against the protocol and to generate transactions which record completed transfers. One important monitor design decision is whether to send out a transaction when a reset is detected, even though a protocol transfer has not completed. Arguably, if the monitor has to handle a reset, then reset is part of the protocol and it should generate and broadcast a transaction with reset status at the end of the reset. Whether the reset transaction should contain information on the transaction that was in progress when the reset occurred is determined by the protocol and how much information the monitor had extracted from the pin level activity before the reset. Like the driver, the monitor

```
class adpcm_driver extends uvm_driver #(adpcm_seq_item);

\`uvm_component_utils(adpcm_driver)

virtual adpcm_if ADPCM;

typedef enum {IDLE, DELAY, FRAME} state_e;

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

task run_phase(uvm_phase phase);
int idx;
adpcm_seq_item req;
state_e state;

forever begin
ADPCM.frame <= 0; // Default conditions:
ADPCM.data <= 0;
@(posedge ADPCM.resetn); // Wait for reset to end
forever
begin
@((posedge ADPCM.clk or negedge ADPCM.resetn);
if(ADPCM.resetn == 0) begin // Reset detected:
ADPCM.frame <= 0;
ADPCM.data <= 0;
state = IDLE;
if(req != null) begin
req.status = UVM_TLM_INCOMPLETE_RESPONSE;
seq_item_port.item_done();
end
while(seq_item_port.has_do_available()) begin
seq_item_port.get_next_item(req);
req.status = UVM_TLM_INCOMPLETE_RESPONSE;
seq_item_port.item_done();
end
break;
end
else begin

endcase
end
end
end
endclass: adpcm_driver
```

```
request.delay--; 
end
else begin
state = FRAME;
idx = 0;
end
end
FRAME: begin // Send the ADPCM frame
if(idx < 7) begin
ADPCM.frame <= 1;
ADPCM.data <= req.data[3:0]; // Send nibbles
req.data = req.data >> 4;
idx++;
end
else begin
ADPCM.data <= req.data[3:0];
req.status = UVM_TLM_OK_RESPONSE;
seq_item_port.item_done();
req = null;
state = IDLE;
end
end
task: run_phase
endclass: adpcm_driver
```
Figure 6 - Reset aware monitor

will most likely have to be implemented using a state machine in order to handle resets.

Analysis components such as scoreboards and functional coverage monitors also need to be designed to handle resets. What a scoreboard should do in the case of a reset is determined by the behavior of the DUT. For instance, if the DUT flushes all of its data buffers on a reset, then the scoreboard will need to do the same, or it may mark outstanding transactions as flushed to check that the DUT has flushed its data. If recovery from an on the fly reset is important for certain conditions, then functional coverage monitors can be made reset aware by checking the state of the design when it entered reset.
HANDLING OTHER HARDWARE EVENTS
An asynchronous reset is probably the most disruptive type of hardware event that can happen to a DUT. Other types of hardware events can be handled in other ways or by adapting the techniques outlined, the exact requirements are dependent on device behavior and the nature of the event. The most common forms of disruptive hardware events are summarized in the table above.

CONCLUSION
On the fly resets can be made manageable within a UVM testbench by building reset awareness into the agent, the sequence item and sequences. If the driver detects a reset condition, then it can bring the current transaction to an early close and return reset status through the sequence hierarchy to ensure a managed change in the stimulus generation path. This approach works well in the situation when all the testbench components have been implemented in this way, but if there are components that ignore reset, then they will need to be handled separately, allowing currently running sequences to come to halt of their own accord and dealing with the consequences.

As the reset aware technique does not rely on multiple run-time phases, it can also be used with OVM testbenches.

FURTHER EXAMPLES
The code examples shown in this article have been taken from a relatively simple unidirectional agent, other types of agent with bidirectional and pipelined protocols require a similar implementation approach. For more details and code examples please refer to the UVM cookbook available on the Verification Academy.

<table>
<thead>
<tr>
<th>Event</th>
<th>Initiated by</th>
<th>Recommended Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>DUT</td>
<td>Implement a context switch in the stimulus generation, no disruption to protocol transfers. See: <a href="http://verificationacademy.com/uvm-ovm/Stimulus/Interrupts">http://verificationacademy.com/uvm-ovm/Stimulus/Interrupts</a></td>
</tr>
<tr>
<td>Error</td>
<td>DUT</td>
<td>Should be signalled as an error status, stimulus should have error handling embedded</td>
</tr>
<tr>
<td>Power Down</td>
<td>Testbench</td>
<td>Originated by the stimulus process, an orderly transition with protocol level transactions completing normally before entering power-down state.</td>
</tr>
<tr>
<td>Power Loss</td>
<td>DUT (Hardware)</td>
<td>Sudden loss of power is like an interrupt and should cause a context switch to enter a power-down state.</td>
</tr>
</tbody>
</table>
Modern FPGA and ASIC verification environments use coverage metrics to help determine how thorough the verification effort has been. Practices for creating, collecting, merging and analyzing this coverage information are well documented for designs that operate in a single configuration only. However, complications arise when parameters are introduced into the design, especially when creating customizable IP.

This article will discuss the coverage-related pitfalls and solutions when dealing with parameterized designs.

INTRODUCTION
Code reuse is increasingly common, mostly because it helps to make engineers more productive. Through reuse it is possible to use a single code base tuned to specific requirements in lieu of maintaining multiple code bases. A standard ASIC technique is to use strap signals along with register programming at the start of operation to generate a customized configuration. Designs on programmable fabric can take this one step further. The downloadable netlist can be configured by the use of parameters, which optimizes away unneeded logic at synthesis time. IP providers use this technique to generate optimized netlists for every customer with a single infrastructure.

The addition of parameters increases complexity compared to a non-parameterized design, but working from one code base instead of multiple customized solutions generally makes this complexity well worth it. However, compared to working on a single static configuration, verifying designs over a range of parameterizations can greatly increase the coverage space and, consequently, the effort required to reach coverage closure. The problem is compounded as the number of modifiable parameters increases. In this article, functional coverage of the parameter space will be referred to as “parameter coverage” and coverage of variable constants with functional coverage will be referred to as “parameterized coverage.” Parameter coverage appears to be just another coverage metric, except that it needs to be closed in the compile/optimization phase of simulation, instead of the run-time phase where normal functional coverage is closed.

The Verilog Language Reference Manual (LRM) describes in straightforward terms how to write parameterized Register Transfer Level (RTL). The addition of SystemVerilog onto the Verilog standard has made the language more powerful, allowing more advanced verification techniques to be used in developing testbenches.

Gaps still exist when trying to verify a parameterizable design within a SystemVerilog testbench, especially during the coverage merge step. The SystemVerilog standard doesn’t specify in detail how coverage is to be merged. For a design with a single parameterization, normal verification techniques generally lead to more or less normal merging of coverage. However, for multiple parameterizations it’s not clear what should be done when “constant” values in covergroups are no longer so constant. Likewise, classes that are parameterized with different values are now different types, and don’t appear to be part of the same merge hierarchy.

Along with the new requirement of parameter coverage, standard functional coverage metrics
including coverage groups, coverage properties, assertions, and code coverage are still used. However, workarounds are needed to handle the non-constant constant problem that parameters add to functional coverage. This article will focus on ways to make coverage useable for a parameterized design.

**COVERAGE FOR REUSABLE CODE Bases**

The parameterized coverage techniques described in this article developed as a result of Xilinx’s verification of its Serial RapidIO (SRIO) Gen2 IP core. The Serial RapidIO protocol is a serial standard supporting numerous line rates and multiple link widths. The Xilinx SRIO Gen2 IP is highly configurable, built to provide customers with flexibility while limiting the resources required for each specific application. This configuration is done with parameters, thus allowing the synthesis tools to provide an optimized solution for every design permutation. There are currently 77 parameters in the design, which control everything from register defaults to the existence of code to the changes in the protocol behavior or bus widths.

The issues described in this article may plague anyone creating reusable designs. It was particularly important during the development of the SRIO Gen2 IP to ensure the core was tested to some degree across the various supported permutations. However, this could also apply to anyone trying to future-proof their designs by improving maintainability. Any design that uses a single code base for multiple synthesized netlists will find these techniques useful. The two most common issues with coverage that arise from using parameters are varying bus widths and generated code. As bus widths vary, it makes it difficult to create meaningful coverage for the current parameterization being targeted. Generated code is an issue because code may or may not exist.

One of the problems with parameterized coverage is determining when to merge a certain set of coverage and when to keep it separate. Unfortunately, there are no clear rules dictating when to merge and when not to. Merging coverage is usually preferred as it enables quicker coverage closure. However, before completing the merge, the user must evaluate the goal of the coverage and determine if it must be kept separate. Specific examples of this will be shown later in the article.

**Parameter Coverage**

In an ideal world, all parameters would be fully crossed with each other to create parameter coverage, and then that would also be completely crossed with all functional coverage. This would ensure that each functional coverage item was tested against each specific parameterization. As the coverage space grows exponentially, and additional parameters or parameter values are added, this comprehensive approach quickly becomes infeasible. For example, if a design has two parameters, each with two different values, crossing the parameters would result in four different parameterizations, which translates to four times the amount of functional coverage. What happens if the design has more parameters?

Assume now the design has ten parameters, again with only two possible values each. This would translate to $2^{10}$ (1024) different parameterizations. Closing on functional coverage is often difficult enough, but it would take an unreasonable amount of time if it had to be closed over 1000 times. Imagine the problem if a design has 50 to 100 parameters with more than two values for each parameter. The same techniques to close functional coverage can be used to close parameter coverage. The verification engineer crosses parameters that interrelate just like regular functional coverage, leaving parameters that don’t interfere with each other as stand-alone coverpoints.

During coverage closure on one of the SRIO Gen2 testbenches, there were a set of parameters that were particularly difficult to close by pure constrained random techniques. The parameter coverage controlled what types of transactions could be initiated and targeted on the user interfaces and resulted in 512 unique parameterization crosses. Relying solely on constrained random would have taken weeks to close this coverage. Realizing this amount of time was unacceptable, randomizations were directed to generate all 512 parameterizations, enabling coverage to be closed in a weekend. Closing holes in parameter coverage is generally not a difficult problem.

While closing the parameter coverage space is a straightforward problem, crossing each point in that space with the complete functional coverage space is not realistic. Therefore, a different approach is needed. A couple of options exist to ensure functional coverage is adequately achieved relative to the various parameterizations. The
first step is to cover all parameter settings, but without crossing them. This verifies that all parameter values have been tested. To further validate the parameter coverage, only important parameters or parameters that have a direct effect on another should be crossed. Finally, specific functional coverage items should be crossed with any parameters that affect that functionality.

**Formal Verification with Parameterized IP**

Formal verification is another popular technique that can be used to close coverage on a design. Formal excels at arbitration and other state-deep problems that are difficult to cover with functional coverage. Additionally, most vendors provide static checks that check for code liveness by locating dead code or unhittable states in state machines that simplify code coverage analysis.

However, parameterizable code is not currently a good fit for this method. Normally, any design is constrained into a specific configuration, and then functionality and code liveness are mathematically proven for that mode. The process can then be repeated if the constraints need to be changed for a different configuration. However, for parameterized designs, every possible permutation needs to be proven in order to get the same level of confidence as is possible with a non-parameterized design. Like parameter coverage, a subset of parameter crosses can be made here, but with 77 parameters, covering all the parameter interactions still expand the number of configurations into a completely infeasible number.

While formal property checking is a poor fit for parameterized code, formal liveness checks aren't even possible. One of the purposes of parameters is to have a code base for different functionalities. This means for any given parameterization there will be dead code as illustrated by the two examples in Fig. 1 below.

```
// Example 1: Explicitly declaring dead code
// using a generate statement
generate if (MODE1) begin
  // Active when MODE1 is 1, dead code when
  // MODE1 is 0
  end else begin
  // Active when MODE1 is 0, dead code when
  // MODE1 is 1
  end

// Example 2: Implicit dead code using an
// if statement
always @(posedge clk) begin
  if (MODE1) begin
    // Dead code when parameter MODE1 is 0
    end else begin
    // Dead code when parameter MODE1 is 1
    end
end
```

**Figure 1. Parameterized Dead Code**

**Parameters and Functional Coverage**

Even in the case of a single design configuration, functional coverage requires a lot of time and effort to do correctly in the case of a single design configuration. When parameters and a parameterized design are factored into the equation, additional considerations arise.

**Parameters and SystemVerilog**

**Covergroups Within Classes**

Any value in a coverage bin definition must be constant and defined when the coverage group is constructed. Occasionally these are literal values, but parameterized designs also require parameter values in these definitions. Therefore, there needs to be some way to get parameters down into coverage groups.

One way would be to declare a covergroup inside a parameterized class and use the parameters of the parent class. The biggest issue when dealing with parameterized classes is that, as far as SystemVerilog is concerned, every different parameterization of a class results in a different type. Unlike class extensions, multiple parameterizations of a parameterized class (specializations) are not compatible through polymorphism[5]. The original class with default parameters can’t be used as a handle as it still creates a unique default specialization. As a result, coverage merging tools see different types when they are trying to merge, and are unable to do so. If a covergroup is defined within a parameterized class and different simulations are run with coverage results saved out, the coverage results contained within each parameterized class instance will not be able to merge together.
Our preferred solution is to not define covergroups within a parameterized class but rather in a non-parameterized class, which is then instantiated within the parameterized parent class. Two variations exist for passing parameter values to the non-parameterized class:

**Generalized solution for getting parameters into covergroups**

The first option, which works regardless of the methodology used, is to pass parameter values from the parameterized class as constructor arguments. This exposes the information needed to collect useful coverage, but without affecting post-simulation coverage merging. A simple example of what this would look like in code is shown in Fig. 2

```vhdl
// The coverage_collector class contains all
// the coverage to be sampled from the
// my_agent class
class coverage_collector;
  int full_count; // Indicates fifo fullness

  covergroup cg_fifo_count(int fifo_depth);
    coverpoint (full_count) {
      bins empty = {0};
      bins in_use = {[1:fifo_depth-2]};
      bins full = {fifo_depth-1};
    }
  endgroup

  function new(int fifo_depth = 16);
    cg_fifo_count = new(fifo_depth);
  endfunction : new
endclass : coverage_collector
```

**Figure 2. Parameter Passing Using New**

In this example, arguments are added to the constructor (one per parameter). Once this class is defined, then it could be used in a parameterized class as shown in Fig. 3.

```vhdl
// my_agent instantiates the
// coverage_collector
class my_agent #(int FIFO_DEPTH = 16);
  coverage_collector coverage_col;
  ...

  function void build();
    coverage_col = new(.fifo_depth (FIFO_DEPTH));
  endfunction : build
  ...
endclass : my_agent
```

**Figure 3. Parameterized Class Providing Parameters to Coverage Class**

Fig. 4 below shows this example where my_agent is a parameterized class passing FIFO_DEPTH through the constructor to the coverage_collector. The coverage_collector then passes this on to the covergroup cg_fifo_count.

```vhdl
my_agent
  FIFO_DEPTH
  ...
coverage_collector
  fifo_depth
  ...
cg_fifo_count
  fifo_depth
```

**Figure 4. Parameters passed through the hierarchy**

By extending the constructor in the non-parameterized class, parameters can be used to collect and affect coverage. This technique is useful for simple cases and can easily be extended to deal with multiple parameters.
However, consider a real world design such as SRIO where 77 parameters are used. These parameters would have to be passed down through every constructor, a tedious requirement that would make change difficult.

**Getting parameters into covergroups for a OVM/UVM methodology**

Our preferred solution for any OVM[6] or UVM[7] based environment uses a variant of this technique that is easier to maintain for a highly parameterizable design. An object should be created containing all of the possible parameters for a design. The object can also contain the parameter coverage for the design, since all variables are present in one location. The object is then placed into the configuration space, making it available to any object anywhere in the object hierarchy. Note that this means that some unnecessary parameters are in the object; but this is overshadowed by the simplification of having all parameters available in a common location. In the case where large numbers of parameters are used, such as in SRIO, using the parameter object provides an easy, reusable way to get the parameter information to coverage classes. Covergroups must be created in the constructor of their parent class. However, since the parent class isn’t constructed until the build phase, the configuration class is already available to the covergroup when it is constructed inside the parent. This is not a problem since the build phase is top down and has already completed for the top level test which places the parameter object into the configuration space. Using this variation, a configuration object is created to hold the parameter values. An example container is shown in Fig. 5 below.

```plaintext
// The param_container class hold the values
// for all parameters in the design
class param_container extends uvm_object;
  `uvm_object_utils(param_container)

  // Data members to hold parameter values
  // Each parameter is set to a default value
  // these values are overwritten in the
  // parameterized test base when the object
  // is created.
  int fifo_depth = 16;
  // other parameter values
  ...

  // constructor, etc.
  endclass : param_container
```

**Figure 5. Parameter Container Class**

The coverage_collector class then gets the parameter container prior to constructing the covergroup as shown in Fig. 6. Note that the coverage object must still pass in every parameter that is required.

```plaintext
// The coverage_collector class contains all
// the coverage to be sampled from the
// my_agent class
class coverage_collector
  extends uvm_subscriber #(transaction_class);

  function new(string name, uvm_component parent);
    param_container params;
    ...

    // Call into the configuration space
    // to get the FIFO_DEPTH value
    void(uvm_config_db #(param_container)::
      get(this, "", "param_container", params);

    cg_fifo_count = new(params.fifo_depth);
  endfunction : new

  endclass : coverage_collector
```

**Figure 6. Parameter Passing Using the Configuration Space**
This method for parameter access is shown in Fig. 7 below.

![Diagram showing parameter access using configuration space](image)

**Figure 7. Parameter Accesses Using the Configuration Space**

A single instance of the `param_container` class is created, has its parameter values set and is then placed into the configuration space in the UVM test. Using a configuration coverage object to encapsulate parameters is described in the DVCon 2011 paper “Parameters and OVM: Can’t They Just Get Along”.

**CLASS-VERSUS MODULE-BASED COVERAGE**

When writing functional coverage, it is important to determine where it should live within the test hierarchy. Two options to consider are class-and module-based coverage.

Class-based coverage is when `covergroups` are defined within a class such as a UVM subscriber. Module-based coverage has `covergroups` defined within `modules`. The `modules` could be standard RTL code, or SystemVerilog `interfaces` or `modules` that are instantiated using `bind` statements. Note that SystemVerilog requires cover properties and assert properties to be located within `modules`. Regardless of where the coverage is placed, tradeoffs must be considered when parameters are involved.

1) **Class-based Coverage**

Protocol level coverage should be isolated within a transaction class that is used across interfaces in a design. Coverage for the protocol can then be reused across all interfaces handling that transaction. Some examples of protocol coverage for SRIO include packet types, valid byte sizes, packet priorities, and other applicable packet fields. Fortunately, protocol coverage should have no need for parameterized code. All parameters should be isolated within the layers around the coverage class as described above. In the event the coverage class does require a parameter value, workarounds are required in order to access those parameters, such as the previously shown configuration object.

In order to write a `covergroup` covering a signal with a parameterized width in a class, the minimum and maximum values should be considered based on all possible configurations and the bins must be appropriate for all configurations. Some coverage is not possible for a given configuration, which will result in run-time warnings. However, all coverage will be closed over a full regression run. The following example covers a counter which iterates though packet identifiers (IDs) and verifies all IDs are used. The `ID_SIZE` parameter is used to set the number of available IDs to the 0 to 31 range when `ID_SIZE=SMALL` or the 0 to 63 range when `ID_SIZE=LARGE`. To save resources, the width of the counter is parameterized to 5 or 6, respectively. If this coverage was written in a single `coverpoint`, all bins could not be covered in only one configuration. Once multiple `ID_SIZE` values are merged, coverage could still be closed over the complete solution space. In this case, two `covergroups` are required in order to make sure all the values are seen in both configurations since coverage cannot be disabled on a bin basis due to restrictions in SystemVerilog. This could also be done using cross coverage, but then many illegal bins would have to be specified for `ID_SIZE=SMALL` and a value of 31 cannot be grouped into multiple bin types. The cleanest solution is shown in Fig. 8.

```vhdl
// Covering packet ID values within a class
// when packet_ids is a parameterized width
covergroup cg_packet_ids;
  cp_small_pid: coverpoint (packet_ids) iff (id_size == small) {
    bins min = (0);
    bins mid = ([1:30]);
    bins max = (31);
  }
```
2) Module-based Coverage

Verification of behavior that affects internal logic and is not visible on an external bus level should be covered within a *module*. The reason: higher level classes do not know or care about these details. An example of this, which is seen frequently in SRIO, is handshaking between internal *modules* or across clock domains. When block A asserts a VALID flag, VALID cannot deassert until a READY response is asserted from block B back to block A. One required coverage point is that a stall condition is seen where VALID is asserted without the assertion of READY. This functionality is not visible on a transaction level and should be written in the *module* to verify this expected functionality occurs.

Figure 10. Covering Packet IDs in a Module

![Handshaking Between Two Modules](image)

Writing coverage within a *module* provides access to all parameters in the *module*. This provides the benefit of being able to write only valid coverage based on the current configuration and the ability to merge coverage across *modules* depending on vendor support. Merging across *modules* is possible because a union of all coverage pieces may be available (again, the merge algorithm is not defined in SystemVerilog). In this case, when covering a parameterized bus the width is known and the coverage can be written using the parameters such that only valid bins are created. This eliminates unnecessary warnings. The example in Fig. 10 below shows the same packet ID coverage described above, but where the *covergroup* only contains valid bins based on the ID_SIZE parameter. Since it is within a *module*, the *generate* keyword is available to exploit.

```vhdl
// Covering packet ID values within a
// module when packet_ids is a parameterized
// width
generate if (ID_SIZE == SMALL) begin
    covergroup cg_packet_ids;
    cp_small_pid: coverpoint (packet_ids){
        bins min = {0};
        bins mid = {[1:30]};
        bins max = {31};
    }
endgroup
end else if (ID_SIZE == LARGE) begin
    covergroup cg_packet_ids;
    cp_large_pid: coverpoint (packet_ids){
        bins min = {0};
        bins mid = {[1:62]};
        bins max = {63};
    }
endgroup
end endgenerate
```

Figure 9. Handshaking Between Two Modules

3) Placement Recommendation for coverage

Understanding the issues with both module and class coverage can help determine where coverage should be located. When covering protocol requirements, it is best to cover in a class because this enables easy reuse of the coverage. The reason: parameter workarounds aren’t required. Coverage of implementation-specific design
choices, independent of the testbench, should be located in a module where it has access to parameters and does not need to be reusable.

USE CASES FOR PARAMETERIZED COVERAGE CLASSES

During development of the SRIO core, a range of problems and limitations were seen with SystemVerilog when covergroups needed to be parameter aware. The next group of examples illustrates the basic set of problems and shows the associated solutions.

1) Parameterized Number of Bins

One common deficiency is how to use parameterized values as bins within a covergroup. In general, if a single maximum value is used, a parameter can be inserted into the bin definition. However, there is no clean way to code the coverage if the parameter affects the number of desired bins. For example, for a buffer structure, it is easy enough to specify full and empty as coverage values. This can even be generalized to any number of specific values, such as empty, quarter, half, three-quarters and full. If the number of values to cover based on the parameterized depth needs to be specified, there is no clean way to do this within a single covergroup because the number of values to cover is not known ahead of time. One possible solution is to create a series of covergroups within a generate block (for modules) or within the new function (for classes) as demonstrated in Fig. 11.

```verilog
// Break a collection of values into single values. Determine which values are active and new the correct groups
covergroup watermark(int value);
    coverpoint (curr_watermark) {
        bins hit = {value};
    }
endcovergroup

// Create a placeholder for the maximum possible values
watermark cg_watermark[MAXDEPTH/16];

// Create a covergroup for each value function new(string name, uvm_component parent);
```

Note that the solution to this problem (determining which covergroups to construct), is very similar to the solution for the ID_SIZE parameter example depicted above. Whenever parameters dictate which bins are active, this solution can be used.

2) Parameterized Bin Values

The SRIO core uses the AXI4-Streaming protocol for ports and internal connections. A verification component is used for every interface for improved reuse. While SRIO only has a single bus width, the verification component is designed to be valid for any possible configuration; bus widths may vary from 8 bits to 4096 bits. There is not a clean way to cover a range of specific literal values when dealing with a variable width bus. One requirement is to cover all possible combinations of the byte enables where the enable bits are packed to the right. These bins are easily coded for a constant width signal. For example, a 32-bit bus would have bins of `{4'b1111, 4'b0111, 4'b0111, 4'b0001}` as valid byte enables. Figures 12-15 show the valid data bits in green for each of the valid byte_enable values.

![Figure 12. Valid Data Bits When byte Enables is 4'b1111](image1)

![Figure 13. Valid Data Bits When byte Enables is 4'b0111](image2)
Coverage within the verification component needs to handle up to 4096-bit data widths. The solution used for this is to directly declare every possible value as shown in Fig. 16. The tools will throw an elaboration warning for bins that are too large and cover the rest normally. Unlike the packet ID example above, the widths that are too large will never be covered for a merged run since the bus width is known in advance. This is not a coverage hole since the excess bins are automatically removed because the literals are not possible given the signal width being covered.

```systemverilog
// cg_byte_enable_cov contains all the coverage to sample from the byte_enable signal
covergroup cg_byte_enable_cov;

coverpoint (byte_enable) {
    bins align[] = bins{2'b01,
                        2'b11,
                        4'b0111,
                        4'b1111,
                        ...
                        512'b1111_1111...};
}

endgroup
```

In the actual code for this verification component, this covergroup has 102 lines of hand-typed code, with error-prone literals. Using the new syntax, there are only 10 lines.
of code to maintain. Additionally, only the correct bins are created removing all of the warnings generated by the work-around version.

PARAMETERIZED BEHAVIOR
Protocol coverage is best written in a class which allows for reuse throughout the testbench. This will almost always be the case but complications can arise when protocol functionality is parameterized.

Consider the case of an arbiter where eight different packet types can be issued on each input port and the number of ports is parameterized by value N_PORTS. This is a simplified version of a problem encountered in the SRIO design. These packets can also have variable lengths. In the common use case, a shared class would exist on all N_PORTS and cover all packet types, then cross this with all the valid packet lengths to guarantee all expected functionality is transmitted. Now consider that each of the eight packet types is enabled or disabled through a parameter independently for each port defined. For example, if N_PORTS is one, port A can have 2 to the power of 8 combinations of supported packet types (8 packet types by 2 modes (on and off)). For SRIO, this arbiter has the added complexity that some ports are not allowed to send certain packet types. If packet types were not restricted to certain ports, all types would be covered over a merged run and a shared covergroup could be used. With type restrictions on certain ports, sharing a covergroup would result in coverage gaps on restricted ports even with merged coverage.

Given the limitations of SystemVerilog, it’s difficult to find a clean solution to this problem since covergroups cannot be generated inside of a class. One solution is adding configuration settings for the coverage class. Based on these settings, the appropriate coverage is created that only covers the supported transactions. The shared class is still reusable across ports. Note this is similar to the FIFO example where the number of covergroups was dynamically selected. The downside to this is that multiple covergroups now have to exist, one for each possible configuration. For SRIO, packet type coverage needed to be crossed with various other metrics like packet length and spacing between packets. Combining all these types in one covergroup would result in gaps when a packet type did not exist on a port. To account for this problem, each packet type was defined in its own covergroup and the configuration setting determined which covergroups exist.

This quickly becomes a burden as N_PORTS increases and additional requirements are added to each port. The complexity can be seen in the example below where, based on the additional configuration setting on the coverage class, only the appropriate coverage should be constructed. Fig. 18 shows an example of one such SRIO configuration where N_PORTS=3. Port A can only send READS, port B can only send WRITES and port C has to handle the remaining six packet types. When port A is created in the testbench environment, the coverage configuration information will need to indicate READs are supported and disable all the other packet types. Port B will need to enable only WRITE support and disable all other types and port C disables READs and WRITEs with all other types enabled.

![Figure 18. Arbiter with N_PORTS=3](image)

Fig. 19 shows the logic used to create only valid packet coverage based on the configuration settings.

```vhdl
// Construct the appropriate covergroups
// within the new function
function new(string name, uvm_component parent);

  // configured_types will be set for
  // interfaces which disable some packet
```
Figure 19. Conditional Covergroup Creation

Originally, the SRIO transaction coverage was ~190 lines of code prior to packet configuration requirements. After parameterization allowing disabled packet types was added, coverage alone was ~600 lines of code. This number was increased even more to configure the coverage class for each port and select only appropriate coverage for a total of ~1200 lines of code.

**PROPERTY COVERAGE CONSIDERATIONS**

Cover directives and assertions (properties), according to SystemVerilog, must be located within a module or interface. The module or interface provides direct access to parameters as static values. Although parameter workarounds are not required, additional work is needed up front in order to correctly write property coverage. This affects how RTL is written. Generate/if blocks that exist in parameterized designs must be written correctly to avoid bogus assertions and false coverage.

If a bind file is used, the RTL must be written such that the bind has access to the signals being covered. Generates will often be coded with local variables. This is great for coding but hides access to these signals from the bind file because the generate statement makes a new scope. The bind is creating a new instance of a module or interface with the module that had the generate statements. This means there are now two parallel scopes which can’t see each other. If these signals are moved outside the generate block and thus made global variables, they are now visible to the bind file and can be covered. Fig. 20 below shows module logic_block with logic_block_bind bound to it. The bind file is able to access the send_ccomp signal located in logic_block. When localized to generate ccomp_logic_gen the bind cannot access it since it is in a parallel scope.

Figure 20. Signal Access Allowed From A Bind File

Care must be taken to ensure signals are not accidentally covered in an invalid configuration. Undriven signals lead to assertion failures and can trigger false or illegal coverage. This can be fixed by either initializing these signals in the else condition or disabling the cover property based on the generating parameter as done in the examples in Fig. 21 below.

```
// Example 1: Clear undriven signals
// for invalid configurations.
// A clock compensation is only sent if
// CCOMP_EN is 1
wire send_ccomp;
```
generate if (CCOMP_EN == 1) begin:
    ccomp_logic_gen
    assign send_ccomp =
        (ccomp_ctr == MAX_CCOMP_CNT);
end else begin: no_ccomp_logic_gen
// zero out when CCOMP_EN is 0 so the
// bind file only samples valid values
// and assertions on this signal do not
// fire incorrectly
assign send_ccomp = 0;
end endgenerate

// Example 2: Disable properties for invalid configurations.
cover property
    @(posedge clk) disable iff (!CCOMP_EN)
    (send_ccomp);

Figure 21. Global Variables for Coverage

GENERATE BLOCK CONSIDERATIONS
Another problem with generated code is that merging coverage for a signal across all configurations of a design is not always wanted. For example, consider the case where a FIFO has a parameterized depth which can be set to 8 or 32 and a full condition needs to be covered. Covering that the full signal was asserted might not be sufficient because it will be merged whether full is seen with a depth of 8 or 32; it is possible full might have only ever occurred when the depth was 8. If the parameterization affects the implementation, it is important to see both settings to ensure full correct functionality. These decisions must be made by someone familiar with the design architecture. A simple solution for this example is to cross the values of FIFO depth with the full flag set to 1. This is shown below in Fig. 22, which is also an example of replicating parameter coverage simply to provide access to a cross.

// Based on the buffer depth, see each
// buffer location as full and empty
reg [BUF_DEPTH-1:0] free_locations;
...

cg_fifo_full covers the full signal for
// each valid value of FIFO_DEPTH
covergroup cg_fifo_full;
    cp_fifo_full : coverpoint (fifo_full);
// Coverage on FIFO_DEPTH is already
// managed within parameter coverage but
// is duplicated here to allow for use
// in the cp_fifo_full cross.
    cp_fifo_depth: coverpoint (FIFO_DEPTH) {
        bins min = (8);
        bins max = (32);
    }

cross cp_fifo_full, cp_fifo_depth {
    ignore_bins ignore =
        binof(cp_fifo_full) intersect {0};
}
endgroup

Figure 22. Crossing Parameters with Functional Coverage

Writing properties within modules covering parameterized code can be greatly simplified by the use of generate statements, which are also available. Note that generated code creates implicit hierarchy. This leads to unique coverage spaces that will not be merged. For busses where each bit needs to be covered independently, only one cover property needs to be written in a generate statement that can be reused on all bits. The example in Fig. 23 shows a signal with a width of BUF_DEPTH which indicates free locations in a buffer. Each location needs to be detected as full and empty.
Another use case for a `generate` statement is for multi-dimensional arrays. The SRIO core can have a serial link width generated based on a user’s needs. For this, the `LINK_WIDTH` parameter indicates how many lanes are valid. In this example, each lane needs to maintain a count of how many bit errors were detected on the link and thus a multidimensional array of counters is used. A cover property can be generated for only valid links to cover the error count reaches its max value as done in Fig. 24.

```
// Check the counter reaches its max value
// on each lane.
reg [3:0] error_cnt [LINK_WIDTH-1:0];
...

generate
  for (int ii=0; ii < LINK_WIDTH; ii++) begin
    cover property
      (@(posedge clk)
       (error_cnt[ii] == MAX_ERRORS));
  end
endgenerate
```

**Figure 24. Generate Loop for Multi-Dimensional Signal Coverage**

For both examples described, it is desired to not merge coverage. By using `generate` loops, hierarchy will be provided and results in not merging the coverage.

**PARAMETERS & CODE COVERAGE**

Normally, code coverage is used as a cross-check for functional coverage to ensure that all coverage points have been defined. Gaps in code coverage without any relevant functional coverage can also identify dead code. As mentioned earlier, while formal can detect this condition early in the design process for standard designs, this is not possible for non-parameterized designs. The only way to locate dead code or unhittable states in a parameterized code base is to run code coverage and review the output. This makes the code coverage review phase even more critical for a parameterized design.

One additional problem for code coverage is that it is additional functionality provided by the vendors and not specified by the SystemVerilog standard. Vendors are free to specify code coverage as they see fit including any parameterized merging behavior. Users must examine all merged code coverage results in order to ensure that results provide the information they need for their design.

We suggest code coverage tools merge gathered coverage metrics across `generate/for` loops. For most situations, the code’s functionality is what is being tested; multiple copies don’t provide more information about the correctness of the code. If it is the case that specific behavior matters per loop, embedded functional coverage can be bound in to track this. Note that this requires that `generate` signals be declared as global variables as described above.

For `generate/if` or `generate/case` constructs, we suggest that vendors don’t merge the branches, even when the `generate` labels are identical for each branch. Because custom code appears in each branch, every branch needs to be fully exercised in order to ensure that the code is well tested.

**CONCLUSION**

Parameters are well integrated into the RTL design subset of SystemVerilog. This enables a clean methodology for designing and maintaining a single code base, while still delivering multiple custom netlists. The addition of keywords like `generate` have eased the burden of designing with parameters.
The solutions are not as straightforward within the verification space of SystemVerilog. The good news is that testbenches can still completely verify these parameterized designs; the bad news is that solutions still require a lot of overhead. The availability of the configuration space in OVM/UVM environments has simplified parameter passing and access. Using a container object to collect all parameter values makes it easy to access these values anywhere in the design, including functional coverage groups.

However, there is still work to be done, particularly for functional coverage, where handling variable code causes additional overhead, as shown previously. When verifying a parameterized design, coverage constructs grow in size. Additionally, work must be duplicated in order to correctly ensure that every variant is covered within a configurable environment.

New additions to the SystemVerilog standard will greatly help with the problem. Until then, the tips and techniques outlined above will reduce the headache from a migraine to a minor annoyance.

ACKNOWLEDGMENTS
We would like to thank Adam Rose from Mentor Graphics for some excellent technical guidance. We would also like to thank Geoff Koch for helping to edit this article.

REFERENCES

ENDNOTES
1 Design parameters will be written in all capital letters throughout this article.
2 Parameters which are passed as arguments to children or covergroups will be denoted with the same parameter name, but in lowercase letters.
3 Note the use of “ii” for loop variables rather than “i”. The amount of loops is very high in parameterized code. This coding guideline simplifies searching for loop variable use within the code since patterns like “ii”, “jj”, etc, are unlikely to be found within the text. At the same time, variable patterns like “ii” keep it very clear that the variable is loop-based rather than a signal-based.
Chip design and verification engineers often write as many as ten lines of test-bench code for every line of RTL code that is implemented in silicon. They can spend 50% or more of the design cycle on verification tasks. Despite this level of effort, nearly 60% of chips contain functional flaws and require re-spins. Because HDL simulation is not sufficient to catch system-level errors, chip designers now employ FPGAs to accelerate algorithm creation and prototyping.

Using FPGAs to process large test data sets enables engineers to rapidly evaluate algorithm and architecture tradeoffs quickly. They can also test designs under real-world scenarios without incurring the heavy time penalty associated with HDL simulators. System-level design and verification tools such as MATLAB® and Simulink® help engineers realize these benefits by rapidly prototyping algorithms on FPGAs.

This article describes best practices for creating FPGA prototypes with MATLAB and Simulink. The best practices are listed below and highlighted in Figure 1.

1. Analyze the effects of fixed-point quantization early in the design process and optimize the word length to yield smaller and more power-efficient implementations
2. Use automatic HDL code generation to produce FPGA prototypes faster
3. Reuse system-level test benches with HDL co-simulation to analyze HDL implementations using system-level metrics
4. Accelerate verification with FPGA-in-the-loop simulation

**WHY PROTOTYPE ON FPGAS?**

Prototyping algorithms on FPGAs gives engineers increased confidence that their algorithm will behave as expected in the real world. In addition to running test vectors and simulation scenarios at high speed, engineers can use FPGA prototypes to exercise software functionality and adjacent system-level functions, such as RF and analog subsystems. Moreover, because FPGA prototypes run faster, larger data sets can be used, potentially exposing bugs that would not be uncovered by a simulation model.

Model-Based Design using HDL code generation enables engineers to efficiently produce FPGA prototypes, as shown in Figure 2. This figure illustrates the practical reality that engineers often abbreviate the detailed design phase in an attempt to begin the hardware development phase to meet development schedules. In practice, engineers will revisit the detailed-design phase during the HDL creation phase as they discover that the fixed-point algorithm is not meeting system requirements. This overlap contributes to an elongated HDL creation phase as depicted by the long purple bar and may result in design compromises, such as glue logic or design patches.

Because automatic HDL code generation is a faster process than hand-coding, engineers can invest some of the time savings to produce higher quality fixed-point algorithms in the detailed design phase. This approach enables engineers to produce higher quality FPGA prototypes faster than with a manual workflow.
DIGITAL DOWN CONVERTER CASE STUDY

To illustrate best practices for FPGA prototyping using Model-Based Design, a digital down converter (DDC) serves as a useful case study. A DDC is a common building block in many communications systems (see Figure 3). It is used to transform high-rate passband input into low-rate baseband output, which can be processed using lower sample rate clocks. This results in lower-power, lower-resource hardware implementation.

The main components of a DDC are those shown in Figure 4:

- Numerical controlled oscillator (NCO)
- Mixer
- Digital filter chain

BEST PRACTICE #1 – ANALYZE THE EFFECT OF FIXED-POINT QUANTIZATION EARLY IN THE DESIGN PROCESS

Engineers typically test new ideas and develop initial algorithms using floating-point data types. Hardware implementation in FPGAs and ASICs, however, requires conversion to fixed-point data types, which often introduces quantization errors. In a manual workflow, fixed-point quantization is usually performed during the HDL coding process. In this workflow, an engineer cannot easily quantify the effect of fixed-point quantization by comparing the fixed-point representation to a floating-point reference. Nor is it easy to analyze the HDL implementation for overflows.

To make intelligent decisions on the required fraction lengths, engineers need a way to compare the floating-point simulation results against fixed-point simulation results before starting the HDL coding process. Increasing the fraction length reduces quantization errors;
however, such increases mean that word length needs to be increased (for more area and more power consumption).

For example, Figure 5 illustrates the differences between the floating-point and fixed-point simulation results for stage one of the low-pass filter in the DDC filter chain. These differences are due to fixed-point quantization. The top graph shows an overlay of floating-point and fixed-point simulation results. The bottom graph shows the quantization error at every point in the plot. Depending on the design specification, engineers may need to increase fraction lengths to reduce the introduced quantization error.

In addition to selecting a fraction length, engineers must optimize the word length to achieve low-power and area-efficient designs.

In the DDC case study, engineers use Simulink Fixed Point™ to reduce the word length of parts of the digital filter chain by as many as 8 bits (see Figure 6).

**BEST PRACTICE #2 — USE AUTOMATIC HDL CODE GENERATION TO PRODUCE FPGA PROTOTYPES FASTER**

HDL code is required to produce an FPGA prototype. Engineers have written Verilog or VHDL code by hand. As an alternative, generating HDL code automatically using HDL Coder™ offers several important benefits. Engineers can:

- Quickly assess if the algorithm can be implemented in hardware
- Rapidly evaluate different algorithm implementations and choose the best one
- Prototype algorithms on FPGAs faster

For the DDC case study, we generated 5,780 lines of HDL code within 55 seconds. Engineers can read and readily understand the code (see Figure 7). Automatic code generation lets engineers make changes in the system-level model, and produce an updated HDL implementation in minutes by regenerating the HDL code.

**Figure 5 - Quantifying the effect of fixed-point quantization using Simulink Fixed Point.**

**Figure 6 - Optimizing fixed-point data types using Simulink Fixed Point.**
BEST PRACTICE #3 – REUSE SYSTEM LEVEL TEST BENCHES WITH CO-SIMULATION FOR HDL VERIFICATION

Functional Verification
HDL co-simulation enables engineers to reuse Simulink models to drive stimuli into the HDL simulator and perform system-level analysis of the simulation output interactively (Figure 8).

While HDL simulation provides only digital waveform output, HDL co-simulation provides complete visibility into the HDL code, as well as access to the full suite of system-level analysis tools of Simulink. When engineers observe a difference between expected results and HDL simulation results, co-simulation helps them to better understand the system-level effect of the mismatch.

For example, in Figure 9, on the following page, the spectrum scope view enables an engineer to make an informed decision to ignore the mismatch between the expected results and HDL simulation results because the differences lie in the stop-band. The digital waveform output, in contrast, merely flags the mismatch in expected results and HDL simulation results as an error. The engineer might eventually arrive at the same conclusion, but it would take more time to complete the required analysis.

Test Coverage
Engineers can use HDL Verifier, Simulink Design Verifier, and Questa / ModelSim to automate code coverage analysis. In this workflow, Simulink Design Verifier produces a suite of test cases for model coverage. HDL Verifier automatically runs Questa / ModelSim with this test suite to gather code coverage data for a complete analysis of the generated code.

BEST PRACTICE #4 – ACCELERATE VERIFICATION WITH FPGA-IN-THE-LOOP SIMULATION

Having verified the DDC algorithm using system-level simulations and HDL co-simulations, you can now deploy the DDC algorithm on an FPGA target platform. FPGA-based verification (also referred to as FPGA-in-the-loop simulation) of the algorithm increases confidence that the algorithm will work in the real world. This enables engineers to run test scenarios faster than with host-based HDL simulation.
For the DDC algorithm, you use a Simulink model to drive FPGA input stimuli and to analyze the output of the FPGA (Figure 10). As with HDL co-simulation, the results are always available in Simulink for analysis.

Figure 11 compares the two verification methods, HDL co-simulation and FPGA-in-the-loop simulation, used for the DDC design. In this case, FPGA-in-the-loop simulation is 23 times faster than HDL co-simulation. Such speed increases enable engineers to run more extensive sets of test cases and to perform regression tests on their designs. This lets them identify potential problem areas that need more detailed analysis.

Although it is slower, HDL co-simulation provides more visibility into the HDL code. It is, therefore, well suited for more detailed analysis of the problem areas found during FPGA-in-the-loop simulation.

**SUMMARY**

Following the four best practices outlined in this article enables engineers to develop FPGA prototypes much faster and with a greater degree of confidence than a traditional, manual workflow. In addition, engineers can continue to refine their models throughout development and rapidly regenerate code for FPGA implementation. This capability enables much shorter design iterations than a traditional workflow that relies on hand-written HDL. To learn more about the workflow outlined here or to download a technical kit, visit http://www.mathworks.com/programs/techkits/techkit_asic_response.html

**ENDNOTES**

1 Hardware/Software Co-verification, by Dr. Jack Horgan. http://alturl.com/gfzsf

<table>
<thead>
<tr>
<th>Verification Method</th>
<th>Simulation Performance</th>
<th>Visibility into HDL Code</th>
<th>System-Level Analysis Possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDL Co-simulation</td>
<td>46 sec</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>FPGA-in-the-Loop</td>
<td>2 sec</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
SystemVerilog [1] UVM [2] class-based testbenches have become as complex as the hardware under test, and are evolving into large object-oriented software designs. The usual RTL debugging techniques must be updated to match this new complexity. Debugging tools are addressing these complexities, but this article will describe techniques and approaches that can be used to help debug these complex environments without advanced debug tools.

INTRODUCTION

Debugging large testbenches has changed recently. The testbenches are larger than they used to be, and they are more like software than they used to be. In addition, the testbench language features use object-oriented constructs, and may use a new library of verification components. Each of these characteristics adds to the pain of debugging the testbench, which must be done before debugging of the actual device-under-test can begin.

This article covers simulator independent debugging techniques that can be adopted as aids during the testbench debug period. Each technique is a small step for better debug, but when considered together can compensate for tool limitations, staff training issues, or other issues with adopting the SystemVerilog language and verification libraries. The code and techniques are standard SystemVerilog and should be available on all modern, compliant simulators.

We discuss simple organizational techniques for debug like naming conventions for files, directories, verification IP, class names and class member variables.

Breakpoints are important to track flow of control, but can be hard to use in dynamic, class based testbenches. We discuss breakpoint pools, and a small infrastructure that can be used to turn breakpoints on and off using the command line. This technique requires a little bit of planning, but can aid flow control debugging. These breakpoints can be class based or even instance based.

Verification libraries, like the UVM, contain many debug statements and aids that can be either compiled on, or turned on from the command line. The UVM itself contains dump() and print() routines that can be used to explore the UVM data structures. We describe easy ways to build compiled-in debug and dynamically controlled debug for user testbenches.

Most large UVM based testbenches are transaction based. We discuss using transactions for debug, including simple convert2string() usage, and recommendations against using any of the field automation macros. Although not fatal, these macros can hide problems, and create an unnecessary distance between the testbench transaction and the verification engineer.

Transactions can be recorded, but usually printing transactions can be just as useful. Using the messaging system and convert2string(), transaction data and control flow can be debugged.

We recommend using one single simulation log file, which provides instant correlation of the entire system. With one log file, tools like sed, awk, grep and perl can be used to extract information, create reports and produce debug summaries. Using a single log file allows all the time correlated data produced by the simulator to be recorded as it was produced. If that information is split into multiple log files from the simulator, then it is hard to correlate events after the fact. Use post-processing tools to extract interesting data; don’t use the simulator to do that.

The UVM reporting system (messaging) has many abilities, including verbosity control, message ID filtering and hierarchical property setting. This system is large and complex, and although it has been available for years, it is mostly opaque to all but the most energetic engineer. We’ll show how to control this system with command line options, with a small message control block. Finally, we’ll build a custom reporting environment with a user defined report server and report catcher.
STRATEGY
Any testbench must be planned, and a testbench which supports debug is no exception. Debug does not come for free, and must be designed in. Debug tools are available, and can minimize the need for planning and design-in, but even they can require command line switches, or require a different kind of compilation flow.

For SystemVerilog UVM class based debug, we are going to assume full SystemVerilog LRM support, and support for the latest UVM release.

This article assumes there is no special debug tool being used. The only debugging available is debugging that comes from the simulator running.

NAMING CONVENTIONS
Naming conventions [3] [4] are often overlooked as either too simplistic or onerous, but they can often simplify debug by making it easier to understand a new testbench. The simplest conventions are usually the best, and should be followed where possible, but should not be used rigidly.

The best naming conventions are those that you agree to and that you use consistently. Some recommendations or suggestions are listed below – but they are just that – suggestions.

Naming conventions help eliminate mistakes by being consistent and simple to understand. Reading the code during debug is easy. Furthermore, finding things becomes easy, since they are where they should be. Finding all the axi_scoreboard factory creation calls in the testbench source code is easy with grep:

```bash
grep -r axi_scoreboard::type_id::create .
```

Names – Files, Classes, Member Variables

Generally speaking you should not use the `include construct, except to include a file in just one place. A `include should be used to build a package file (see below). If you need to reference a type or other definition, then use `import to bring the definition into scope. Do not use `include for general solutions.

```
package abc_pkg;
    `include “abc_macros.svh”
    `include “abc_monitor.svh”
    `include “abc_driver.svh”
    ... 
endpackage
```

Files should be named after what they are - for example, ahb_scoreboard.svh will contain the AHB scoreboard, while the axi_vip_pkg.sv contains the AXI Verification IP.

Create a class to implement functionality with a name that describes the functionality. For example, a scoreboard for the ABC verification IP would be named - abc_scoreboard.

Use a prefix in the class name for the kind of object this class is associated with. In this case `class abc_scoreboard` is a scoreboard associated with an `abc` kind of verification component.

Any member that is meant to be private should be named with a `m_` prefix, and should be made local or protected. Any member that will be randomized should not be local or protected. The `m_` prefix is a simple reminder that this variable or function is not publically available.

Directories and Packages
Testbenches are constructed of SystemVerilog UVM code organized as packages, collections of verification IP organized as packages and a description of the hardware to be tested. Other files such as C models are also to be organized.

Each package should exist in its own directory. Each of these package directories should have one file that gets compiled - a file with the extension .sv

```
abc_pkg.sv
```
Each package should have at most one file that may be included in other code. This file may define macros.

    abc_macros.svh

For a complex package that may contain tests, examples and documentation, create subdirectories:

    abc_pkg/examples
    abc_pkg/docs
    abc_pkg/tests
    abc_pkg/src/abc_pkg.sv

Sample file listing

    abc_pkg/src
    abc_pkg/src/abc_pkg.sv

    abc_pkg/src/abc_macros.svh

    abc_pkg/src/abc_env.svh
    abc_pkg/src/abc_interface.sv

    abc_pkg/src/abc_driver.svh
    abc_pkg/src/abc_monitor.svh
    abc_pkg/src/abc_scoreboard.svh

    abc_pkg/src/abc_sequence_item.svh
    abc_pkg/src/abc_sequencer.svh
    abc_pkg/src/abc_sequences.svh

    abc_pkg/docs/
    abc_pkg/docs/abc_user_guide.docx

    abc_pkg-tests/......

Using these suggestions will help you get organized, and keep you out of trouble.

DESIGNING THE TESTBENCH

The testbench should be designed with debug in mind. The main elements that will be used in debugging will be transactions, sequences and phased components. Building each of these with debug in mind is important.

Transactions should be designed so that you can figure out what data is stored in the transaction. For example a routine named ‘convert2string’ should be provided that creates a string representing the current “value” of the transaction. A transaction may also want to register the fact that is has been constructed.

Sequences should be designed to print information when they start and when they end. They should also have a way to breakpoint. They may also want to register the fact that a sequence has been constructed.

Phased components should be designed to print information when each phase starts and ends. They should also have a way to breakpoint.

DO IT YOURSELF TOOLS

Breadcrumbs

In order to understand control flow, or function call order, the user might insert a print statement that printed the file and line number when executed.

    $display("DBG: %s:%0d", `__FILE__, `__LINE__);

Each time this line is executed, it leaves a trail of where it has been. Instrumenting each line in a difficult section of code may help debug problems. The last line printed is where the problem is.

In addition to a simple file and line number, in class based debug, we can print the contents of the class. A SystemVerilog based solution is just to use ‘%p’ with a $display or $sformatf:

    $display("sequence_item=%0p", seq_item_handle);
A UVM based solution is just to use convert2string():

```verilog
$display("sequence_item=%s", seq_item_handle.convert2string());
```

These basic print statements can provide a good amount of debug. They provide visibility into control flow, and visibility into data. If your code is stuck in a loop somewhere, and there are many conditions for exiting the loop, you could set a breakpoint at the top of the loop, and single step through the loop. Sometimes this could take a long time before the problem appeared, and it relies on the debug tool providing breakpoints and single stepping. A crude but effective alternative is to sprinkle the $display that uses FILE and LINE on most lines in the loop or function. Then just run simulation, allowing the FILE and LINE messages to fill up your transcript. Eventually you may see a pattern. Certain lines may be repeating, and other lines may never be executed. With this information you can infer which if-then-else is failing or what expected exit condition is not being met.

This kind of breadcrumb trail can be extended to many situations. For example, at the beginning of every function or task call, you can add a $display, and print a “start” message

```verilog
$display("Function %s starting", "FunctionName");
```

With the UVM more information is available. The function or task in a UVM based class can include information from itself, its containing class, or any ancestor class. For example a function in a uvm_component could print the name of the class (xyz_h), the type of the class (xyz), the full name of the class (top.env. xyz_h).

```verilog
class xyz extends uvm_component;
  `uvm_component_utils(xyz)
  task run_phase(uvm_phase phase);
    $display("%s %s %s", get_name(), get_type_name(), get_full_name());
  endfunction
endclass
```

Under normal circumstances, the debug messages are not needed or are undesirable. They can be deleted from the source code, but usually they get put back in to help on the next phase of debugging. Those kinds of messages can be conditionally turned off and on using a simple command line switch. Don’t leave breadcrumb messages turned on when you don’t need them. Either delete them, compile them out, or turn them off with a run time switch.

```verilog
if ($test$plusargs("debug")) begin
  $display("%s %s %s", get_name(), get_type_name(), get_full_name());
end

Or
if (debug) begin
  $display("%s %s %s", get_name(), get_type_name(), get_full_name());
end

Or
ifndef NO_DEBUG
  if (debug) begin
    $display("%s %s %s", get_name(), get_type_name(), get_full_name());
  end
  endif
```

The UVM also provides a powerful message logging or reporting system that can be used in place of the simple $display. (`uvm_info).

**Threads**

Debugging threaded programs is difficult. Instrumenting each thread is important. Each branch should be instrumented – each decision point. The start and end of the thread should be instrumented. There will be many messages generated and they will be time correlated. It will be easy to see which thread is running, which thread is doing what.
A good example of a common problem is two threads communicating through an infinite size fifo. The producer thread runs, but the consumer thread never runs. By instrumenting the threads it will be apparent that the producer thread is always ready to run, so it continues to run, starving the consumer. By conditioning the producer on a clock edge, or to yield with a #0 after each 10 transactions are generated, the consumer will be allowed to run.

Threads exist in SystemVerilog and the UVM in initial and always blocks, in fork/join constructs, in UVM phases (run_phase(), build_phase(), etc) and in UVM sequences (body() tasks). There are many opportunities to have threads running.

**Plus args control**

Plus args are command line switches supported by the simulator. They are "application specific" – they can be defined for each application. The SystemVerilog LRM defines that arguments beginning with the '+' character will be available using the $test$plusargs and $value$plusargs PLI APIs. Using plusargs is a good way to set a mode – like the 'debug' mode. Or to set a value – like the 'debug_level'.

Using plusargs allows one setting to be made. In the case of multiple settings (like setting many breakpoints using the +bp switch), a different solution is needed.

The SystemVerilog LRM specifies that the vpi_get_vlog_info function returns information about the tool invocation, including the command line arguments. In the UVM, an interface has been defined to this vpi_get_vlog_info function. It is called the command line processor (clp). Using the clp.get_arg_values(+bp=", list), a list of all the +bp= settings is returned in the array 'list'. Below, 'list' is iterated, each iteration being a +bp= usage from the simulator command line.

```verilog
begin
  bit debug;
  debug = 0;
  if ($test$plusargs("d"))
    debug = 1;
  $display("Debug = %0d", debug);
end

begin
  int debug_level;
  if ($value$plusargs("debug=%d", debug_level)==0)
    debug_level = 1000;
  $display("DebugLevel = %0d", debug_level);
end
```

Usage:

```bash
<simulator> +d +debug_level=42
```

**Quieting Debug Messages**

Sprinkling code with debug messages makes an easy to debug, verbose testbench. But eventually the testbench will be debugged, and those messages should get turned off. They can be turned on and off with either compile switches and/or run time switches.

```verilog
ifndef NO_DEBUG
`define DEBUG(msg) \\
`uvm_info("DBG", msg, UVM_MEDIUM)
#else
`define DEBUG(msg)
`endif
```

Usage:

```bash
<simulator> +bp=.*seqA.* +bp=.*seqB.*
```

This is a useful technique to grab all the plusargs of a certain kind from the command line.
In this case, `DEBUG("abc")` is used in the testbench code to call `uvm_info` and produce messages. When it is time to turn these messages off, a `+define+NO_DEBUG` can be used on the compile invocation.

Runtime control is also possible for example, by re-defining the macro to be

```c
ifndef NO_DEBUG
#define DEBUG(msg) \
  if (debug) `uvm_info("DBG", msg, UVM_MEDIUM)
  else
  `define DEBUG(msg)
  `endif
```

Elsewhere in the testbench, we declared `debug` and set it from the simulation command line.

```c
int debug = 0;
...
if ($test$plusargs("debug") == 1)
  debug = 1;
else
  debug = 0;
```

Note that this solution has both compile time and run time options.

**Transaction Debug**

A transaction is a piece of data that is being communicated. It can contain many different kinds of information – but for debug purposes a function like convert2string should be defined that returns the important state information in a formatted string. Creating convert2string in the base class means that any class based debug can use any class handle, and call

```c
$display("%%s", transaction_handle.convert2string());
```

In the code snippet below, convert2string is defined to format the id and data values. Additionally, a function call like `get_type_name()` can provide useful information. For the code snippet, `get_type_name` is defined by the `uvm_object_utils` macro.

Another useful feature to embed in a transaction is a unique id. That way, if you ever see another transaction with the same id, then you know that it is really the same transaction – the same object. The id below is such a unique id – it is incremented for each call to new(). The UVM also has a unique id built in for each `uvm_object` that is created. In the UVM, it is called `m_inst_id`, and it can be accessed by calling `get_inst_id()`.

```c
class my_transaction extends uvm_transaction;
  `uvm_object_utils(my_transaction)
static int g_id;
int id;
int data;

function new(string name = "my_transaction");
  super.new(name);
  `ALLOCATED_TYPE
  id = g_id++;
endfunction

function string convert2string();
  return $psprintf("id=%0d, data=%0d", id, data);
endfunction
endclass
```

THE UVM

**UVM Breadcrumbs**

Instead of `$display`, use `uvm_info`, `uvm_warning`, `uvm_error` or `uvm_fatal`. Additional information is built-in, like filename, line number, time, and verbosity control. These breadcrumbs come with their own controls and capabilities.

**UVM Macros**

Don't use macros that generate complex code. It is very hard to debug complex code, and doubly hard when that code is hidden in a macro. There are useful macros, and macros to avoid [8].
Really Easy UVM Debug

Interested in what's in the factory?

```java
uvm_factory factory= uvm_factory::get();
factory.print();

#### Factory Configuration (*)
#
# No instance overrides are registered with this factory
#
# Type Overrides:
#
#   Requested Type  Override Type
#   --------------  -------------
#   driverA         driver2A
#   <unknown>       <unknown>
#
# All types registered with the factory: 57 total
# (types without type names will not be printed)
#
#   Type Name
#   ---------
#   driver2A
#   driverA
#   envA
#   questa_uvm_recorder
#   sequence_A
#   sequence_A_grandparent
#   sequence_A_greatgrandparent
#   sequence_A_parent
#   sequence_item_A
#   sequence_item_B
#   test
# (*) Types with no associated type name will be printed
#     as <unknown>
```

Interested in which resource [9] (which configuration) setting was set, but never used? Check it with 'check_config_usage()'.

```java
uvm_top.check_config_usage(1);
#
# :::: The following resources have at least
#     one write and no reads ::::
# config_object [^test\:e2\:a\:.*$] : ?
```

Less useful, simply for the quantity and style of information produced, print the configurations available at each hierarchy level.

```java
uvm_top.print_config(1, 1);
resources that are visible in recording_detail [^\*\*]$ : ?
-

test reads: 0 @ 0 writes: 2 @ 0
test.e1_a reads: 1 @ 0 writes: 0 @ 0
test.e1_a.driver reads: 1 @ 0 writes: 0 @ 0
test.e1_a.driver.rsp_port reads: 1 @ 0 writes: 0 @ 0
test.e1_a.sequencer reads: 1 @ 0 writes: 0 @ 0
```

To see the UVM topology from the top use print_topology().

```java
uvm_top.print_topology();
```

```
# UVM_INFO @ 0: reporter [UVMTOP] UVM testbench
# -----------------------------------------------
# Name           Type
# -----------------------------------------------
# test           test
# e1_a           envA
# driver         driver2A
# rsp_port       uvm_analysis_port
# sqr_pull_port  uvm_seq_item_pull_port
# sequencer      uvm_sequencer
# rsp_export     uvm_analysis_export
# seq_item_export uvm_seq_item_pull_imp
```
Use one of the library calls mentioned in this article to print a complete netlist.

```verilog
e1_b # uvm_env
driver # driver2#(sequence_item_A)
rsp_port # uvm_analysis_port
sqr_pull_port # uvm_seq_item_pull_port
sequencer # uvm_sequencer
rsp_export # uvm_analysis_export
seq_item_export # uvm_seq_item_pull_imp
e2_a # envA
driver # driver2A
rsp_port # uvm_analysis_port
sqr_pull_port # uvm_seq_item_pull_port
sequencer # uvm_sequencer
rsp_export # uvm_analysis_export
seq_item_export # uvm_seq_item_pull_imp
e2_b # uvm_env
driver # driver2#(sequence_item_A)
rsp_port # uvm_analysis_port
sqr_pull_port # uvm_seq_item_pull_port
sequencer # uvm_sequencer
rsp_export # uvm_analysis_export
seq_item_export # uvm_seq_item_pull_imp
```

Use one of the library calls mentioned in this article to print a complete netlist.

```verilog
show_connectivity(uvm_top, 0);
+ UVM Root: uvm_top
+ Test: test
e1_a + Env: test.e1_a
    + Component: test.e1_a.driver
        Port: test.e1_a.driver.sqr_pull_port
            Connected to Port:
                test.e1_a.sequencer.seq_item_export
e1_a + Sequencer: test.e1_a.sequencer
        Port: test.e1_a.sequencer.seq_item_export
            Provided to Port:
                test.e1_a.driver.sqr_pull_port
```

**Reporting System Control**

The reporting system has many features, but the most useful and easiest to use is verbosity. Planning your testbench, you should liberally sprinkle calls to the reporting system in, so that you can be made aware of process flow and current state. Normally most of these debug and status statements are quiet – they are turned off. They can be turned on from the command line in a variety of ways.

If you want to turn them all on, then you can just run simulation with verbosity turned up very high – that way your testbench will be very verbose.

If you want to turn on just a few components, then you can control the instances which will be verbose.

Simulating with

```
+UVM_VERBOSITY=UVM_HIGH
```

or

```
+UVM_VERBOSITY=UVM_DEBUG
```

will produce many messages – likely some detailed messages from the UVM itself. Your output will be very detailed, and very verbose.

Instead of turning the verbosity up for the entire testbench, the verbosity for a small part can be increased using

```
+uvm_set_verbosity=e.consumer*,_ALL_,UVM_HIGH.
```

This causes the string ‘e.consumer*’ to get matched, and all IDs at that level get verbosity set to UVM_HIGH. In our producer/consumer example this makes the consumer verbose, while the producer stays quiet.

It is good debug practice to introduce a debug statement in interesting places, for example when a phase starts (begin_phase(), run_phase() especially) or when a sequence begins or ends (at the start and end of the body() task). These messages remain quiet if they are using a verbosity higher than the normal verbosity. Normal verbosity is UVM_MEDIUM. The verbosity choices are

```
UVM_NONE,
UVM_LOW,
UVM_MEDIUM,
UVM_HIGH,
UVM_FULL,
UVM_DEBUG.
```
Normally simulations run at UVM\_MEDIUM, and debug messages are UVM\_HIGH or UVM\_FULL.

For example, a run\_phase instrumentation:

```
  task run\_phase(uvm\_phase phase);
  ...
  `uvm\_info(get\_type\_name(), "Starting... ",
            UVM\_HIGH)
```

Or a sequence body task instrumentation:

```
  task body();
  T t;
  `uvm\_info(get\_type\_name(), "Starting... ",
            UVM\_HIGH)
```

These messages won’t be printed unless the verbosity is at UVM\_HIGH or higher.

The first argument in the `uvm\_info macro is the ID. It’s a good practice to use the type\_name – that way you can control all the messages from a certain type by causing the reporting system to change verbosity or other attributes based on the ID.

**Data Structures**

The UVM Class data structures can be explored to provide information about the testbench structure, its current state, and the state of any generated stimulus. There are many APIs available to choose from – the uvm-1.1a contains 316 class definitions in 134 files for a total of 67,298 lines of code. Testbenches can contain hundreds or thousands of instances of class based components. They can generate thousands or millions of transactions or tests. They can run for hours or days. These are complex systems described by complex data structures.

**Netlist**

A simple netlist program can be built to traverse the UVM component hierarchy and print the connections. In Figure 1 a simple UVM producer is connected to a fifo. The fifo is connected to a UVM consumer. The netlist produced shows that the environment ‘e’ contains three components (consumer, fifo and producer). The consumer has a port named ‘port’, which is connected to a port on the fifo named ‘get\_peek\_export’.

```
  class env extends uvm\_env;
    `uvm\_component\_utils(env)
    producer p;
    consumer c;
    uvm\_tlm\_fifo #(<my\_transaction>) fifo;
    function new(string name = "env",
                 uvm\_component parent = null);
      super\_new(name, parent);
      endfunction
    function void build\_phase(uvm\_phase phase);
      p = new("producer", this);
      c = new("consumer", this);
      fifo = new("fifo", this);
      endfunction
    function void connect\_phase(uvm\_phase phase);
      p\_port\_connect(fifo\_put\_export);
      c\_port\_connect(fifo\_get\_export);
      endfunction
    endclass
```

![Figure 1 - Simple Producer / Consumer](image)
A more interesting netlist from a UVM perspective is the netlist from Figure 2. This diagram shows a test containing an environment, containing a sequencer and driver. The netlist shows the driver sqr_pull_port connected to the sequencer seq_item_export. It also shows the reverse – the sequencer seq_item_export provided to the driver sqr_pull_port.

Figure 2 - Sequencer / Driver Connection

The netlisting code is shown abbreviated below. The function show_connectivity() is called with a component ‘c’. First all the direct children are collected into an array named ‘children’. Each child is visited using the foreach(children[]) loop. A child is either a port or a component; the $cast() determines which. If this child is a port, then use ‘get_connected_to()’ and ‘get_provided_to()’ to fill in any connections. Next print the port name, and iterate the connections lists, printing connected-to and provided-to information. If this child was not a port, then it is a component, and show_connectivity() is called recursively with this child and depth incremented by 2. In this way the entire component hierarchy from the original component ‘c’ down to the last leaf will be visited.

```plaintext
function automatic void show_connectivity(uvm_component c = null, int depth = 0);
// Array of the children to iterate through.
uvm_component children[$];
// Nothing to do.
if (c == null)
  return;
begin
  ...  
  type_name = "Component";
  $display("%s+ %s: %s", {depth[" "]},
              type_name, name);
end
// Fill up the array with all the children of ’c’.
c.get_children(children);
// Iterate for each child.
foreach (children[i]) begin
  foreach (children[j]) begin
    if (c == null)
      return;
    begin
      ...  
      type_name = "Component";
      $display("%s+ %s: %s", {depth[" "]},
                  type_name, name);
    end
    // Cast to port_component_base to see if
    // this is a port.
    uvm_port_component_base port_component_base;
```

Netlist output:

```
+ Env: e
  + Component: e.consumer
    Port: e.consumer.port
      Connected to Port: e.fifo.get_peek_export
  + Component: e.fifo
    Port: e.fifo.get_peek_export
      Provided to Port: e.consumer.port
    Port: e.fifo.put_export
      Provided to Port: e.producer.port
  + Component: e.producer
    Port: e.producer.port
      Connected to Port: e.fifo.put_export
+ Seq: test
  + Test: test
    + Env: test.e1_a
      sequencer
      driver
      + Component: test.e1_a.driver
        Port: test.e1_a.driver.sqr_pull_port
          Connected to Port:
            test.e1_a.sequencer.seq_item_export
+ Seq: test.e1_a.sequencer
  Port: test.e1_a.sequencer.seq_item_export
    Provided to Port:
      test.e1_a.driver.sqr_pull_port
```
// Test to see if this handle is a port.
if ($cast(port_component_base, children[i]))
begin
  uvm_port_list connected_to_list;
  uvm_port_list provided_to_list;

  // See what is connected.
  port_component_base.get_connected_to(
    connected_to_list);

  // See what is connected.
  port_component_base.get_provided_to(
    provided_to_list);

  // If there is something connected,
  // print the port name and the connections.
  if ((connected_to_list.size() > 0) ||
      (provided_to_list.size() > 0)) begin
    // Print the name of the port.
    $display("%s Port: %s",
      (depth+2+2{" "}),
      children[i].get_full_name());
  end

  // Iterate across all the connections,
  // printing the thing this port is
  // connected to.
  foreach (connected_to_list[i])
    $display("%s Connected to Port: %s",
      (depth+2+2{" "}),
      connected_to_list[i].get_full_name());

  // Iterate across all the connections,
  // printing the thing this port is
  // provided to.
  foreach (provided_to_list[i])
    $display("%s Provided to Port: %s",
      (depth+2+2{" "}),
      provided_to_list[i].get_full_name());
end

else begin
  // It's a component. Recurse.
  show_connectivity(children[i], depth+2);
end
endfunction

This code is simple but powerful. It recursively traverses the component and its children finding the connections to any ports, printing what they are connected to. It uses four API calls (‘get_children()’, ‘get_connected_to()’, ‘get_provided_to()’) to produce a useful list of all components, their children and connections.

The netlisting code visits a potentially large data structure, printing it in human readable form. Data structures can also be explored one at a time. The code below has two ports, and just asks what they are connected to using the debug_connected_to() and debug_provided_to() calls.

```
fifo.put_export.debug_connected_to();
fifo.put_export.debug_provided_to();
fifo.get_export.debug_connected_to();
fifo.get_export.debug_provided_to();
```

These two snippets of code; a full netlister, and a partial netlister have explored just the basic “children” data structure in the uvm_component, and have used ports to show connectivity. The UVM has many other useful data structures waiting to be explored as a debug exercise for the reader.

**Active Sequences**

Most UVM testbenches will use sequences. The sequences will be used as stimulus generators. Each agent is likely to have at least one active sequence, if not more. These sequences are started, and work together to generate interesting stimulus conditions for testing. It is hard to know what sequences are running at any given time. An easy solution is to register the sequence when it is started, and then de-register it when it ends. A sequence has just the hooks needed – the pre_body() and post_body() tasks. By instrumenting the pre_body and post_body, we can keep track of what sequences are currently active.

The class ‘active_sequences’ below is a simple wrapper around an associative array indexed by sequence handle. When a sequence is started the function active_sequences::add(seq) should be called to register the sequence. When a sequence is ending the function active_sequences::remove(seq) should be called to de-register the sequence. At any given time, calling the function active_sequences::show() will print the list of currently active (currently registered) sequences.

This code is simple but powerful. It recursively traverses the component and its children finding the connections to any ports, printing what they are connected to. It uses four API calls (‘get_children()’, ‘get_connected_to()’, ‘get_full_
class active_sequences;
    // A list of sequences.
    static uvm_sequence_base
        list[uvm_sequence_base];

    // Add 'seq' to the list. If it is already in
    // the list, just override the existing one.
    static function void add(
        uvm_sequence_base seq);
        list[seq] = seq;
    endfunction

    // Remove 'seq' from the list. If it is not
    // in the list, just ignore it.
    static function void remove(
        uvm_sequence_base seq);
        if (list.exists(seq))
            list.delete(seq);
    endfunction

    static function void show(
        string file = `__FILE__,
        int line = `__LINE__);
        uvm_report_info("Seq",
            "Active Sequences", UVM_MEDIUM,
            file, line);
        foreach (list[seq])
            uvm_report_info("Seq",
                $sformatf("  %s", seq.get_full_name()),
                UVM_MEDIUM, file, line);
    endfunction

endclass

The active_sequences class and the functions used to register and de-register can be used without any further support. They can be used in any location where the functionality is useful. The tasks pre_body and post_body are two such places.

Instrumenting pre_body and post_body can be done manually, inserting the calls to active_sequences:add() and active_sequences::remove() in each sequence. This is tedious and error prone. Putting this kind of functionality in a base class is a better idea. Many times there is a common base class that all sequences in the testbench derive from.

task pre_body();
    // Begin a sequence
    active_sequences::add(this);
    super.pre_body();
endtask

task post_body();
    // End a sequence
    super.post_body();
    active_sequences::remove(this);
endtask

If no common sequence base class exists, one can be created – the uvm_debug_sequence. All sequences should derive from this common sequence, and will inherit the pre_body() and post_body() functionality that calls keeps track of active sequences.

class uvm_debug_sequence #(
    type REQ = uvm_sequence_item,
    type RSP = REQ) extends uvm_sequence#(REQ, RSP);

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

    task pre_body();
        // Begin a sequence
        active_sequences::add(this);
        super.pre_body();
    endtask

    task post_body();
        // End a sequence
        super.post_body();
        active_sequences::remove(this);
    endtask

endclass

Using the new base class takes a little planning, changing what class is extended from.
At any time in the simulation, a list of the active sequences can be printed by calling active_sequences::show.

active_sequences::show();

A report will be generated like the report below. The report below shows four sequencers with sequences running.

e2_b.sequencer@@ggp_seq test.e2_a.sequencer.ggp_seq
e2_b.sequencer@@ggp_seq test.e1_b.sequencer.ggp_seq
e2_b.sequencer@@ggp_seq test.e2_b.sequencer.ggp_seq
e2_b.sequencer@@ggp_seq test.e1_b.sequencer.seq#(T)
e2_b.sequencer@@ggp_seq test.e2_b.sequencer.seq#(T)
e2_b.sequencer@@ggp_seq test.e1_b.sequencer.seq

Objects Allocated
In many testbenches transactions – classes – are created dynamically and created based on randomized control. It is not always possible to predict either how many classes will be constructed or what types they will be. Many problems can occur in such a testbench, including allocating too many classes or too many classes of a specific type.

Adding simple instrumentation each time a class is constructed; we can keep track of each allocation, and print a report when needed. This simple technique will tell us the maximum allocated, but won’t tell us the maximum that actually was allocated at any given time. We’re not able to track the garbage collection – the destruction of objects, so we don’t know when objects are “unallocated”. This kind of information may become available in the future, but for now even just the total ever allocated and type is useful for tracking runaway allocations, expensive construction of already existing classes, or the uneven distribution of allocating one type over another.

When a class is allocated (constructed) we’ll instrument the code with a call to register that fact:

function new(string name = "sequence_item_B");
    super.new(name);
    objects::add(get_type_name());
    id = g_id++;
endfunction

Calling `objects::add(get_type_name())` in each constructor of our simple sequences example, increments the count for this type. Calling `objects::show()` at the end of simulation produces the following report:

# Objects Allocated (by type)
#   324: sequence_item_A
#    80: sequence_A
#    80: sequence_B#(sequence_item_A)
#    26: sequence_B_parent#(sequence_item_A)
#    26: sequence_A_parent
#    ... 
#    4: driver2#(sequence_item_A)
#     2: driverA
#    ...
#    2: sequence_A_greatgrandparent
#    1: test

The underlying data structure is a simple associative array, indexed by a string representing the type name of the allocated object. Each call to `object::add()` increments the corresponding array element. The `show()` puts the associative array elements into a SystemVerilog queue, and sorts it. The code then iterates the sorted queue, printing the allocation report seen above.
class objects;
static int obj[string];

static function void add(
    string type_name = "object");
if (obj.exists(type_name))
    obj[type_name] += 1;
else
    obj[type_name] = 1;
endfunction

static function void show();
typedef struct {
    int count;
    string name;} item;
item q[$];
item i;

$display("Objects Allocated (by type)");
// Put each item in a queue.
foreach (obj[type_name]) begin
    i.count = obj[type_name];
    i.name = type_name;
    q.push_front(i);
end
// Sort the queue.
q.rsort with (item.count);
// Print the items in descending order.
foreach (q[x])
    $display("  %6d: %s", q[x].count, q[x].name);
endfunction
endclass

'define ALLOCATED_TYPE()
    objects::add(get_type_name());

Conditional Breakpoint
When running simulation it is hard to stop at exactly the right place. This is especially hard for dynamic classes.

Such a conditional breakpoint might be "stop in packet processing after the ten thousandth packet has been processed if the current packet is of type ETH, and there are at least 10 errors". These kinds of conditional breakpoints are sometimes most easily hard-coded into the SystemVerilog test – for example:

if ((packet_count > 10000) &&
    (packet_type == ETH) &&
    (error_count > 10))
$stop();

Once simulation stops, you can examine the state of the packet processing. If stopping and examining is not possible, a summary report could be written out to the logfile, and simulation continued.

Dynamic message control
Sometimes certain debugging should start after many clocks, or after some condition is detected. This is a case where we want to run quietly – with little or no messages until the condition is reached. Once reached, we want to change the debug level or the verbosity level, producing many log messages. For example, after 100000 clocks, turn on debug messages in the AXI agent:

repeat (100000) @(posedge clk);
    axi_agent_h.set_report_verbosity_hier(…);

Breakpoints in sequences
Sequences can be created at any time during simulation. Sequences can have short or long lifespans. Many instances of the same sequence can be constructed. Many different debug scenarios could be imagined, but with a small breakpoint library, sequences can be stopped based on their name, their type or their instance name (full path name).

The breakpoint library is a collection of code which uses the built-in UVM regular expression interface to match desired breakpoints with current sequence execution. For example, if the sequence named "seq" hits the "breakpoint", then the breakpoint library will search to see if any regular expression matches the sequence name "seq" (or the type or the full name).
Using the breakpoint library is easy. Just call the breakpoint from the place where the check should occur. In a sequence, this is most likely at the start of the body() routine. Alternatively, the pre_body() could be instrumented to get all sequences instrumented.

Breakpoints are specified as plusargs on the simulation command line:

```
+bp=gp_.*
+bp:type=.*greatgrand.*
```

The ‘greatgrandparent’ sequence is instrumented with `BP:

```java
class sequence_B_greatgrandparent #(type T = int) 
extends sequence_B_parent #(T);
`uvm_object_param_utils(sequence_B_greatgrandparent#(T))
...
    task body();
    ...
    `BP                      // The Breakpoint
    for(int i = 0; i < 3; i++) begin
    ...
    end
    ...
    endtask
    ...
endclass
```

When the body() task is called, the code from the BP macro will be executed. The definition of BP is a call to the check function in the breakpoint pool library. (BPP::check()). Three arguments are passed in – the current values that are going to be checked against the regular expressions that are currently set as breakpoints. The three current values are get_full_name, get_name and get_type_name. Additionally, the file and line number are passed in for better debug messages.

```
// Macro: BP
// Must only be used in the context where the
// functions get_name(), get_full_name() and
// get_type_name() exist.
// For example, from within a sequence or within
// a component.
// Other macros can be created for other contexts.

`define BP if (BPP::check( get_full_name(), \ 
    get_name(), \ 
    get_type_name(), \ 
    `__FILE__, \ 
    `__LINE__))   $stop;
```

Evaluating breakpoints, writing regular expressions, and getting the UVM names right is hard. The breakpoint pool comes with tracing and debug messages.

```
% grep BPP::trace transcript
[BPP::trace matched] .*greatgrand.* matched
```

We instrumented our body() task with a macro - `BP. We supplied command line arguments +bp:type=.*greatgrand.*. When we ran simulation we stopped at the exact line were BP was placed in the body() task – line 220:

```
# Break in Task
# envB_pkg/sequence_B_greatgrandparent::body at
# envB.sv line 220
# Stopped at envB.sv line 220
196   class sequence_B_greatgrandparent
...
217     task body();
...
220       `BP                      // The Breakpoint
221         for(int i = 0; i < 3; i++) begin
```

```
% grep BPP::debug transcript
[BPP::debug - full_name] test.e2_b.sqr.seq#(T)
[BPP::debug - name] seq#(T)
[BPP::debug - type_name] seq_B_greatgrand#(seq_A)
```

We instrumented our body() task with a macro - `BP. We supplied command line arguments +bp:type=.*greatgrand.*. When we ran simulation we stopped at the exact line were BP was placed in the body() task – line 220:
Random stability in debug
Any debug solution must be randomly stable. A randomly stable solution for debug means that when debug is “turned on”, the simulation results will be the same as without.

The following code is unstable with UVM 1.1. It is simple code that calls `uvm_info when the command line contains a non-zero setting for ‘x’ (+x=1). In the code, when x takes the value zero, then the if-statement will be true, and the macro `uvm_info will be called. This call to `uvm_info is what makes this code unstable – it changes the random number order by creating a new class object.

```verilog
import uvm_pkg::*;
include "uvm_macros.svh"
module top();
int x, r;
initial begin
if ($value$plusargs("x=%0d", x) == 0)
x = 0;
if (x == 0)
' uvm_info("DBG", "x is zero", UVM_MEDIUM)
r = $urandom();
$display("x=%0d, r=%0d", x, r);
end
endmodule
```

In UVM 1.1a this issue is fixed.

For user code, sometimes you must ensure your code stays randomly stable, by capturing the random state before a “de-stabilizing” event. Then after the destabilizing event, restore the random state. The following code is stable using UVM 1.1, adding the lines commented ‘ADDED’:

```verilog
import uvm_pkg::*;
include "uvm_macros.svh"
module top();
int x, r;
initial begin
automatic process p; // Process - ADDED
string p_rand; // State - ADDED
if ($value$plusargs("x=%0d", x) == 0)
x = 0;
p = process::self(); // Capture - ADDED
p_rand = p.get_randstate(); // Capture – ADDED
if (x == 0)
' uvm_info("DBG", "x is zero", UVM_MEDIUM)
p.set_randstate(p_rand); // Restore – ADDED
r = $urandom();
$display("x=%0d, r=%0d", x, r);
end
endmodule
```

Using a report catcher
A report catcher is a UVM construct which allows a message for the messaging system to be “caught”. For example, a report_catcher could be implemented to catch messages for a specific ID.

Once the report catcher is executing, it can choose to ignore or catch the message. If THROW is used, then the next report_catcher will process this message. If CATCH is used, then this message will not be processed by any other report_catcher – the message has been caught.

A simple report catcher skeleton is listed below. It simply uses the API to retrieve message attributes (get_id(), get_fname(), etc), then it returns “THROW”.

```verilog
class my_report_catcher
extends uvm_report_catcher;

string id;
string filename;
string name;
string message;
int line;
int verbosity;
uvm_severity severity;
uvm_report_object client;

function new(string name = "my_report_catcher");
super.new(name);
endfunction
```
function action_e catch();
    uvm_severity_type usv;

    id = get_id();
    filename = get_fname();
    line = get_line();
    severity = get_severity();
    verbosity = get_verbosity();
    message = get_message();

    client = get_client();
    name = client.get_full_name();

    usv = uvm_severity_type'(severity);

    // Process this message.
    // Decide THROW or CATCH.
    ...
    return THROW;
endfunction
endclass

Using a report_catcher is easy. It must be constructed, and then it must be added to the list of report catchers managed by the reporting system (uvm_report_cb::add()). A report catcher can be prepended (UVM_PREPEND) or appended (UVM_APPEND) to the existing report catcher list.

class test extends uvm_test;
...
    my_report_catcher my_report_catcher_inst;
    my_report_catcher_inst = new("my_report_catcher_inst");
    uvm_report_cb::add(NULL,
            my_report_catcher_inst, UVM_APPEND);
...

Using a report server
A report server is a UVM construct which formats the reports in the messaging system. The report server below formats the name and id to fixed widths. The widths are programmable.

class my_report_server extends uvm_report_server;
    int name_width = 20;
    int id_width = 20;

    function string pad(string s, int width);
        if ( s.len() == width )
            return s;
        if ( s.len() < width )
            return {s, { (width - s.len()){ " "} } };
        else
            return s.substr(s.len()-width, s.len()-1);
    endfunction

    function string compose_message(uvm_severity severity,
            string name,
            string id,
            string message,
            string filename,
            int    line);
        // Make the width be exactly name_width
        // and id_width.
        name = pad(name, name_width);
        id = pad(id, id_width);

        return super.compose_message(severity, name, id, message, filename, line);
    endfunction
endclass
Using a report server is easy. It must be constructed. After it is constructed it can be configured. In this case, the configuration consists of setting the name and id widths. Name is set to 28 characters, and id is set to 20 in this example. Finally, the new report server is registered using `set_server()`.

```verilog
class test extends uvm_test;
    // Make my report server.
    begin
        my_report_server my_report_server_inst;
        my_report_server_inst = new();

        // Configure.
        my_report_server_inst.name_width = 28;
        my_report_server_inst.id_width   = 20;

        // Set.
        uvm_report_server::set_server(
            my_report_server_inst);
    end
```

Reports can be generated gathering statistics about the distribution of packet type over certain periods of time. Transfer rates can be calculated. The possibilities are limitless, but first you have to get organized, and plan your debug.

**REFERENCES**

[5] Stu Sutherland, SystemVerilog for Design

**UVM CLASS LIBRARY**

The UVM Class library is large. There are many classes, with many controls and options (The uvm-1.1a contains 316 class definitions in 134 files for a total of 67,298 lines of code). Some of the most interesting options, switches, flags and functions are listed below by UVM class name. This is not an exhaustive list, nor necessarily a useful list; just an interesting list for future investigation in the workings of the UVM.

A. uvm_root

```
+UVM_TESTNAME
+uvm_set_verbosity
+UVM_TIMEOUT
+UVM_SET_INST_OVERRIDE
+UVM_SET_TYPE_OVERRIDE
+UVM_SET_CONFIG_INT
+UVM_SET_CONFIG_STRING
+UVM_MAX_QUIT_COUNT
+UVM_DUMP_CMDLINE_ARGS
+UVM_VERBOSITY
uvm_root::find()
    uvm_root::find_all()
    uvm_root::print_topology()
```
B. UVM_COMPONENT

+uvm_set_action=\langle comp, id, severity, action\rangle
+uvm_set_severity=\langle comp, id, orig severity, new severity\rangle
uvm_component::get_parent()
uvm_component::get_children()
uvm_component::lookup()
uvm_component::get_depth()
uvm_component::phase_started()
uvm_component::phase_ready_to_end()
uvm_component::phase_ended()
uvm_component::check_config_usage()
uvm_component::print_config_settings()
uvm_component::print_config()
uvm_component::print_config_with_audit()
uvm_component::raised()
uvm_component::dropped()
uvm_component::all_dropped()
uvm_component::print_override_info()
uvm_component::set_report_id_verboisity_hier()
uvm_component::set_report_severity_id_verboisity_hier
uvm_component::set_report_severity_action_hier()
uvm_component::set_report_id_action_hier()
uvm_component::set_report_severity_id_action_hier()
uvm_component::set_report_severity_file_hier()
uvm_component::set_report_default_file_hier()
uvm_component::set_report_id_file_hier()
uvm_component::set_report_severity_file_hier()
uvm_component::set_report_default_file_hier()
uvm_component::set_report_severity_level_hier()
uvm_component::pre_abort()
uvm_component::set_config_object()
uvm_component::check_config_usage()
uvm_component::print_config

C. uvm_object

uvm_object::get_inst_id() (incremented on new object)
uvm_object::get_inst_count() (total count until now)
uvm_object::get_type_name()

D. uvm_CONFIG_DB

+UVM_CONFIG_DB_TRACE
uvm_resource_db_options::is_tracing()
  ( uvm_info(..., UVM_LOW)
uvm_config_db_options::turn_on_tracing()
uvm_config_db_options::turn_off_tracing();

E. Uvm_resource_db

+UVM_RESOURCE_DB_TRACE
dump()

F. uvm_resource

uvm_resource_optjons::trace_on_auditing
uvm_resource::print_resources()
uvm_resource::print_accessors()

G. uvm_resource_pool

uvm_resource_pool::dump_get_records()
uvm_resource_pool::find_unused_resources()
uvm_resource_pool::print_resources()
uvm_resource_pool::dump()

H. uvm_callback

uvm_callback::display()
  `uvm_cb_trace_noobj()

I. uvm_event

uvm_event::add_callback()
uvm_event::do_print()

J. uvm_factory

uvm_factory::debug_create_by_type
uvm_factory::debug_create_by_name
uvm_factory::m_debug_pass
uvm_factory::print

K. uvm_objections

uvm_objections::trace_mode
Report ID: OBJTN_TRC
uvm_objections::raised()
uvm_objections::dropped()
uvm_objections::all_dropped()
uvm_objections::get_objectors()
uvm_objections::display_objections()

L. uvm_phase

`UVM_PH_TRACE
+UVM_PHASE_TRACE → m_phase_trace
  `uvm_info(..., UVM_DEBUG)
uvm_phase::get_state()
  wait(get_state() ... ) → change phase state
uvm_phase::wait_for_state()
M. uvm_sequence_base

uvm_sequence_base::m_sequence_state
uvm_sequence_base::get_sequence_state()
uvm_sequence_base::wait_for_sequence_state()
uvm_sequence_base::pre_start()
uvm_sequence_base::pre_body()
uvm_sequence_base::pre_do()
uvm_sequence_base::mid_do()
uvm_sequence_base::body()
uvm_sequence_base::post_do()
uvm_sequence_base::post_body()
uvm_sequence_base::post_start()

N. uvm_sequencer_base

uvm_sequencer_base::print()
uvm_sequencer_base::convert2string()
`uvm_info("PHASESEQ", ..., UVM_FULL)

EXAMPLE SEQUENCE INSTRUMENTATION
The sequence 'sequenceB' below is an example sequence with instrumentations. Not all instrumentations shown here are required for good debug, but these are listed as potential sequence debug instrumentation points.

First, the sequence extends the uvm_debug_sequence, inheriting any instrumentation from it.

The sequence defines its type_name, which is hard for parameterized objects. The sequence uses the `ALLOCATED_TYPE macro to register the construction of this object. The sequence is managing an id for each different parameterized sequence that is created. The sequence defines pre_body() and post_body(), using the `DEBUG macro for output. The sequence uses the `uvm_info macros and $display in the task body(), providing breadcrumbs to trace execution flow. The sequence uses the `BP macro to allow breakpoint access to this sequence. Finally, the sequence defines do_record in a way that will record the actual types of the variables into the transaction database, instead of recording a "large enough" bit vector as the type.

class sequence_B #(type T = int)
  extends uvm_debug_sequence #(T);
  `uvm_object_param_utils(sequence_B#(T))

  const static string type_name =
    ("sequence_B#("T::type_name,""));

  virtual function string get_type_name();
    return type_name;
  endfunction

  int id;
  int var_sequence_B;
  int iteration;
  static int g_id = 0;

  function new(string name = "sequence_B");
    super.new(name);
    `ALLOCATED_TYPE
    id = g_id++;
  endfunction

  // `uvm_debug_sequence_body
  virtual task pre_body();
    `DEBUG("pre_body")
    super.pre_body();
  endtask

  virtual task post_body();
    `DEBUG("post_body")
    super.post_body();
  endtask

  task body();
    T t;
    `uvm_info(get_type_name(),
      "Starting... ", UVM_MEDIUM)
    `DEBUG($psprintf("My type_name is %s", get_type_name()))
    `BP

    begin
      int simple_int = -1;
      if (!m_sequencer.get_config_int( 
        "simple_int", simple_int))
        $display("GET CONFIG FAILED");
      else
        $display("GET CONFIG OK");
    
    `uvm_info("SEQ_B", 
      $psprintf("simple_int=%0d", simple_int), UVM_MEDIUM)
  end

  for(int i = 0; i < 3; i++) begin
    t = new($psprintf("t%0d", i));
var_sequence_B = t.id;
start_item(t);
finish_item(t);
end
`uvm_info(get_type_name(),
"Finishing...", UVM_MEDIUM)
endtask

function void do_record(uvm_recorder recorder);
super.do_record(recorder);
`uvm_record_field("id", id);
`uvm_record_field("iteration", iteration);
`uvm_record_field("var_sequence_B",
var_sequence_B);
endfunction
endclass

BREAKPOINT POOL LIBRARY

The breakpoint pool library, the other classes and macros mentioned in this article are available. Please contact the authors.

// Class: BP
//
// A breakpoint. Really a list of strings
// (regular expressions) that will be used to
// match against. When a match occurs that
// means a breakpoint has been “hit”.
//
// This class is not normally used directly,
// but rather from the BPP class.
//
class BP;
typedef struct { string name; } bp_t;
bp_t breakpoints[$];

// add_breakpoint()
// Put the argument in our list.
function void add_breakpoint(string str);
bp_t bp;
bp.name = str;
breakpoints.push_front(bp);
endfunction

// check()
//
// Returns 1 if we matched; the ‘match’
// string has matched something in the
// breakpoint list.
// Returns 0 if nothing matched.
// Return the regular expression that matched
// in the argument ‘thing_that_matched’.

// Uses regular expression matching.
//
function int check(string match = "",
output string thing_that_matched);
int nomatch;
if (match == "")
    return 0; // We didn’t match
foreach (breakpoints[j]) begin
    nomatch = uvm_re_match(breakpoints[j].name,
match);
    if (nomatch == 0) begin
        thing_that_matched = breakpoints[j].name;
        return 1; // Meaning we matched...
    end
end
return 0; // We didn’t match.
endfunction

// print()
// Print all the breakpoints.
function void print();
foreach (breakpoints[j])
    $display(" Breakpoint ‘%s’",
breakpoints[j].name);
endfunction
endclass

// Class: BPP
//
// A breakpoint-pool.
//
// Collects together three sets of breakpoints.
// Name, Type and Full name.
//
// This is a static class, and is normally
// used from macros.
//
class BPP;
static bit trace = 1;
static bit debug = 1;
static BP by_type      = new();
static BP by_name      = new();
static BP by_full_name = new();

// Fill in the by type
static function void set_by_type(
    string breakpoint);

// Displays trace messages for check().
static bit trace = 1;
// Displays debug messages for check().
static bit debug = 1;
static BP by_type      = new();
static BP by_name      = new();
static BP by_full_name = new();

by_type.add_breakpoint(breakpoint);
endfunction

// Fill in by name
static function void set_by_name(
    string breakpoint);
    by_name.add_breakpoint(breakpoint);
endfunction

// Fill in by full name.
static function void set_by_full_name(
    string breakpoint);
    by_full_name.add_breakpoint(breakpoint);
endfunction

// Calling trace_on causes trace messages to
// come out.
static function void trace_on();
    trace = 1;
endfunction

// Calling trace_off turns trace messages off.
static function void trace_off();
    trace = 0;
endfunction

// Calling debug_on causes debug messages to
// come out.
static function void debug_on();
    debug = 1;
endfunction

// Calling debug_off turns debug messages off.
static function void debug_off();
    debug = 0;
endfunction

// Trick to get the command line parsed
// without any intervention from the user.
static bit args_parsed = parse_args();

// parse_args()

// Legal syntax:
// +bp:type=
// +bp:
// +bp:

static function bit parse_args();
    string list[ ];
    uvm_cmdline_processor clp;
    clp = uvm_cmdline_processor::get_inst();

    // By Type => +bp:type=
    clp.get_arg_values("+bp:type=", list);
    foreach (list[i])
        BPP::set_by_type(list[i]);

    // By Full Name => +bp:full=
    clp.get_arg_values("+bp:full=", list);
    foreach (list[i])
        BPP::set_by_full_name(list[i]);

    // By Name => +bp=
    clp.get_arg_values("+bp=", list);
    foreach (list[i])
        BPP::set_by_name(list[i]);

    return 1;
endfunction

// check()

// returns 0 for no match.
// returns 1 for full name match.
// returns 2 for name match.
// returns 3 for type name match.
//
// Usually called from a macro:
//
// check(get_full_name(),
//      get_name(),
//      get_type_name(),
//      __FILE__,
//      __LINE__)

static function int check(
    string full_name = ",
    string name = ",
    string type_name = ",
    string file = ",
    int line = 0);

    string thing_that_matched;
    int ret;

    if (debug) begin
        `uvm_info("BPP::debug - full_name",
            full_name, UVM_MEDIUM)
        `uvm_info("BPP::debug - name",
            name, UVM_MEDIUM)
        `uvm_info("BPP::debug - type_name",
            type_name, UVM_MEDIUM)
    end
if (by_full_name.check(
    full_name, thing_that_matched))
    ret = 1;
else if (by_name.check(
    name, thing_that_matched))
    ret = 2;
else if (by_type.check(
    type_name, thing_that_matched))
    ret = 3;
else
    ret = 0;

// Tell what matched
if (((trace) && (ret != 0))
    'uvm_info("BPP::trace matched",
        $sformatf("%s matched", 
            thing_that_matched), UVM_MEDIUM)
    return ret;
endfunction

// print()
//
// Debug/information.
//
static function void print();
    $display("By Type");
    by_type.print();
    $display("By Name");
    by_name.print();
    $display("By Full Name");
    by_full_name.print();
endfunction
endclass
Now you don’t have to wait for the next printed issue to get the latest.

Hear from the Verification Horizons Team every week at VerificationHorizonsBlog.com