Welcome to our special DAC 2009 edition of Verification Horizons! I really like having DAC in July every few years, and not just because it gives us another month to get all the presentations and other collateral together. I like DAC in July because then I don't have to miss my son's baseball games that always happen during DAC week in June.

Regular Horizons readers may recall that I coach my son's baseball team every year. He's now 11 years old, and this was my sixth year as his coach. It's been a great bonding experience for us, and it's also given me a chance to teach him and other boys about the importance of sportsmanship and teamwork — lessons I hope they'll keep with them for a long time. This year, teamwork became even more important for the boys, and that led me to make “teamwork” the theme for this issue.

For the Groton Angels, teamwork became an issue right away because we lost our best player to another division at the start of the season. To make it worse, we had several players who, to be kind, weren't as good as you might expect boys of this age to be. Always one to make the best of a difficult situation, I set the tone at our first practice explaining to the boys that they had to work together as teammates and that they needed to think on every play — themes that we continued to emphasize throughout the season. We won only three games during the regular season, but everyone on the team got at least one hit and everyone improved, which is really all a coach can ask for. Living in Massachusetts, every team in the league made the playoffs, and that's where things really started to click for us.

I won't finish the story just yet, but suffice it to say that teamwork can really make the whole greater than the sum of the parts. The same can be said for any endeavor, including verifying
a design. In this issue, we'll share several articles that show how we at Mentor and some of our partners are putting technology in place to help you and your team achieve more than you might have expected.

The key to teamwork is to have everyone working together. In verification that means starting with a plan so that everyone knows what they're doing and progress can be measured effectively. Our first article this issue, “Verification Teamwork Requires Process Automation,” looks at Questa's enhanced Verification Management features, which are targeted at achieving “Electronic Closure” by automating the planning, measuring, and analysis of results into a true closed-loop process. Automation not only makes each step more efficient, but also improves your ability to rerun specific tests targeted at specific failures to improve your time-to-debug.

Our next article discusses “Team-Driven OVM Testbench Creation” using Mentor’s new Certe™ Testbench Studio (CTS) tool, which enables correct-by-construction assembly of OVM-based verification environments. As you’ll see, CTS allows IP producers to create a library of reusable component templates that greatly simplify the task of creating OVM environments. CTS walks the testbench developer through the instantiation and connection of the basic components, allowing the developer to customize them while enforcing the required aspects of usage. The CTS also checks any custom code against a set of rules to ensure compliance. CTS allows you to visualize a fully-elaborated and configured OVM testbench so you can make sure that everything turned out the way it should have.

Remember how I mentioned that my baseball team improved this year? In baseball we have objective measurements, like runs, batting average, and wins, to measure progress. In verification there really hasn’t been a similarly objective measure of how good a given team is at doing verification. Our next article, from my friends and colleagues Harry Foster and Mike Warner of Mentor, introduces a straightforward process model that lets you see where you are on a scale of “Evolving Verification Capabilities.” It breaks the verification process into 12 separate aspects and provides assessment criteria for each, allowing you to see where you fall on the “crawl-walk-run” scale of process capabilities. If you’re not where you’d like to be, the article has some suggestions on how you can improve.

Teammates make each other better. In “The Dynamic Duo of Veloce and TBX,” you’ll see how to take advantage of transaction-level verification using our Veloce emulator. This lets you develop your verification environment at the transaction level, using OVM or other methodologies, and run your design, drivers, and monitors in the emulator. The communication to the rest of the testbench is handled automatically for you.

Sometimes teammates do things that you didn’t know they could do. As a coach, I know that it’s just a matter of getting the players to take advantage of their abilities. Our next article, “Maximizing Your Verification Throughput,” will give you an overview of the debug and performance features found in our recent Questa 6.5 release. Just remember that productivity is more than just how fast a particular testcase runs. True productivity comes from being able to run a set of tests, analyze the results, fix any bugs, and rerun the tests. The article will show you how these new features, and others that Questa has had for a while, will help improve your productivity by reducing the time spent in that loop.

There were a few games this season where we needed to call up a player from a lower division to fill in for us. We were fortunate that the player we called up was a great teammate in that he was not only a good player, but he did whatever we needed him to do. “Applying Scalable Verification Components to a USB2.0 HUB” gives you a good look at our new USB Multi-view Verification Component (MVC) and how you can use it to verify the USB protocol at multiple levels of abstraction throughout your environment.

In our Partners’ Corner, we have three articles from some or our Questa Vanguard Partners. These articles are indicative of the success we’ve had in partnering with other companies to expand the OVM ecosystem. Our friends at eInfochips share their experience in “Porting a Legacy Verification Environment for a Wireless SoC to OVM,” a case study in how to take advantage of OVM to improve your verification environment while reusing some aspects of your legacy testbench including SystemC and C tests.

Next, the folks at Agnisys explain how to achieve “Process Improvements Using OVM Registers and IDesignSpec.” Mentor recently released an OVM Register package, and this article shows how Agnisys’ IDesignSpec tool automates the process of generating customized register code from a single spec, including tests that
automatically read and write all registers. You’ll also see how this single-source approach simplifies the process of making changes as the design evolves. Last but not least, our partners from Duolog Technologies introduce their new Socrates tool in “Interface-Based OVM Environment Automation.” The tool allows you to specify your design interfaces, ports, and registers and will automatically generate an OVM testbench for you, including customizing key aspects of the testbench. All three of these articles illustrate Mentor’s commitment to being a team player in working with our partners to ensure a wider set of verification tools that you can use to achieve your goals.

And finally, in our Consultants Corner, you’ll read an all-too-common story of the perils of hardware-software verification. Our teammates in the Mentor Consulting Division will also provide some recommendations on how to take advantage of common register specifications, automation, verification IP, and other technologies we’ve discussed to solve many of those problems.

I know you’re all dying to know how my team did in the playoffs. With a lot of effort and some great teamwork, we won two playoff games and made it to the semifinals, where we lost. The most gratifying thing was that in all three games, the boys kept encouraging each other and really played like a team. As other teams were screaming at each other when things weren’t going their way, my boys stayed focused and kept a positive attitude. After we lost in the semifinals, we all went out for ice cream, so even that wasn’t so bad.

Respectfully submitted,

Tom Fitzpatrick
Verification Technologist
Mentor Graphics
TABLE OF CONTENTS

Page 7... Verification Teamwork Requires Process Automation
by Darron May, Product Marketing Manager, Verification Mgmt and Coverage, Mentor Graphics

Page 10... Team-Driven OVM Testbench Creation
by Tom Dewey, Technical Marketing Engineer, Mentor Graphics

Page 14... Evolving Verification Capabilities
by Harry Foster, Chief Verification Scientist, Design Verification Technology and Mike Warner, Worldwide Verification Practice Manager, Mentor Consulting, Mentor Graphics

Page 18... The Dynamic Duo of Veloce and TBX: Accelerating Transaction-Based Verification to the Next Level
by Bob Koehl, Tray Read, Senior Consultants, Mentor Consulting Division, and Vijay Chobisa, Product Marketing Manager, Mentor Emulation Division, Mentor Graphics

Page 22... Maximizing Your Verification Throughput
by Joe Rodriguez, Verification Product Marketing, Mentor Graphics

Page 25... Applying Scalable Verification Components to a USB 2.0 HUB
by Manoj Manu, Senior Member Technical Staff, Mentor Graphics
Partners’ Corner:

Page 28... Porting a Legacy Verification Environment for a Wireless SoC to OVM
by Nilesh Ranpura, Project Manager-ASIC and Pratik Vasavda, Project Leader-ASIC, eInfochips

Page 35... Process improvements using OVM Registers and IDesignSpec
by Anupam Bakshi, CEO, Agnisys Inc.

Page 39... Interface-based OVM Environment Automation
by Zoltán Sugár, David Murray, Mike Smith, Tamas Olaszi, Duolog Technologies

Consultant’s Corner:

Page 43... Owzthat?
—Or How I Was Nearly Out, Caught
by Mark Peryer, Verification Practice, Mentor Consulting
“...automating the planning, measuring, and analysis of results into a true closed-loop process. Automation not only makes each step more efficient, but also improves your ability to rerun specific tests targeted at specific failures to improve your time-to-debug.”
You’ve heard it all before: More to verify in less time.

Functional verification has to constantly ride the wave of increasing design sizes, competitive pressure demands and the limitations of resource and budgets. Development and verification of tomorrow’s technology often requires monitoring and controlling dispersed project teams across multiple time zones, which adds further to the complexity of these challenges. The ultimate price of not effectively managing this process is unplanned re-spins, which in turn leads to missed market windows or, even worse, field returns.

To achieve effective management and visibility of this process there must be comprehensive planning in a form that will allow a verification plan to drive the process. There needs to be accurate measurement of progress and effectiveness of the process and the automation to achieve “electronic closure” between these metrics and the plan. Run management is required to allow controllability, repeatability and further automation of the process. Automation of the complete verification process allows improvements in time to coverage and time to next bug, and it enables multiple dispersed project teams to accurately estimate the time to completion, allowing products to hit their market windows by avoiding unplanned respins and ensuring the necessary quality.

VERIFICATION MANAGEMENT PAIN POINTS

The problems within the verification management of a project fall into four major categories. The first lies in managing available resources: people, hardware, and verification methods. Planning the expectation and alignment of these available resources is the first step along the road to success. However the greatest unknown can be in the management of the data generated from all sources; this in itself can become a full time job.

The second category is the management of the overall verification quality. How do we know that we are focusing our efforts on the right areas? How do we know how close we are to being complete? Critical decisions are often made with risky ad-hoc data methods. There is normally very limited visibility between the actual process and the verification plan, which defines the required quality. This means that often when things change as the project progresses it is normal for the plan to become out of date and, due to tight schedules, often it gets abandoned completely. This means slight changes to the design specification simply never get reflected into the verification plan and bugs can easily escape identification until it’s just too late.

Schedules are the third category of pain felt within the process. There is often little visibility between time and the process itself. What kinds of risks are we taking if we have to tape-out tomorrow? This is where trade-offs between quality and schedule occur, without visibility the decision comes down completely to a gut judgement call.

Verification management requires resources, quality and schedule to be monitored so that the fourth category of pain—cost—can be controlled. The reality is that as geometries shrink project management costs are increasing and there needs to be a way to monitor the process to control these costs.

VERIFICATION MANAGEMENT PROCESS

The verification process has to start with a plan, which is derived from the design requirements in the functional specification. This plan needs to detail what is going to be tested and what will be sufficient to answer the question of “when are we done?” This plan needs to take into consideration resources and schedules and must keep track of the importance and priority of design requirements to keep the team on target.

Every design has requirements that are critical to the success of the project, features that are nice to have, and features that fall somewhere in-between. During the planning process, all requirements need to be prioritized for analysis and recording. This allows bugs
to be hunted in the most important product features before fixing problems in the less important features, if the project constraints allow.

A verification process is not useful without a measure of what has been tested and how well. There are many ways of measuring test completeness. A blend of the appropriate types enables informed decisions on where you are in the process. Code coverage is a good measure of structural coverage and measures how well the HDL code is executing. Assertion properties check internal design states and monitor boundary protocol conditions. Functional coverage gives control over developing customized coverage that measures whether each important feature of the design has been exercised. There are other metrics that can be used alongside these, including bug rates and code stability measurements.

The last part of the process is analysis of results, which allows the measurements to be linked back to the plan itself. Because of the detailed planning stage, it is possible to track down to the smallest detail of the plan, monitoring team resources, schedules and the overall quality of the verification process dynamically. Process optimization is also possible, using this information to react to changes over time, pinpoint coverage holes, eliminate redundant tests, and manage the impact of engineering changes.

**ELECTRONIC CLOSURE**

Electronic closure is the ability to take each of the stages of the verification process and automate the planning, measuring, and analysis into a closed loop flow. This automation requires a database that provides the infrastructure for verification management. Test plan information is stored within the database and its infrastructure must include a linking mechanism that allows any coverage object within the database to be linked with any test plan object. This feature of the database allows test plan tracking, and, therefore, electronic closure. Having all the data in one place allows analysis tools and process optimization to be built around the database.

Questa Verification Management (VM) includes test plan tracking. This is the application layer that sits over the infrastructure built into Mentor’s Unified Coverage Database (UCDB) to give users the ability to close the loop between their planning stage and the measurement of progress. The plan can include engineering resources, schedules, milestones, priorities and any other attributes that the project needs to track. This document also includes linking information that links the test plan features to the objects that are being used to measure their progress. These objects can be any type of coverage that is stored within the data. Test plans can be hierarchical and merged from plans from individual team members or geographical locations into a single view. Once the data from multiple verification runs is merged with the test plan, the Questa Verification Management GUI and command line interface can be used to analyze, query, and react automatically to the changes in the environment. The tracker allows different filtered views to be dynamically generated based upon the attributes defined in the test plan. This attribute data can be used to get all kinds of filtered views of progress including but not limiting to, personal engineer views, verification methods, and team views. The data within Questa VM tracker is sharable in HTML format so that managers can view the data without access to the tools.

**FURTHER PROCESS AUTOMATION**

There are other ways to improve automation and, therefore, the process within verification management. The tracking and measuring of requirements improves the time to electronic closure by improving the time to coverage; however, a big improvement can also be made in the process by improving the time to finding the next bug.

The first part of this is providing the infrastructure to capture, configure, and run tests and regressions. Having a standard transparent method for controlling and dispatching tests allows many of the tasks during this process to be automated and leads to consistency and repeatability within the process. Tasks like re-running failures with full visibility, managing seeds within constrained
random environments, merging coverage results and analyzing failures across a complete regression run can be automated. Such a system should be powerful enough to capture the most complex requirements of running regressions on projects across multiple teams and geographical locations as well as having usability features which allow the everyday user to easily interact with the system and monitor progress.

Failure analysis is one of the most important aspects of improving the time to next bug. The ability to analyze messages from multiple tools over the complete set of regressions is necessary to triage failures. The ability to group like failures and have that information automatically sent to the person responsible means that less time is spent figuring out who should be fixing a particular bug or having multiple resources working on the same problem. Including this kind of analysis into the running of the regressions in an automated fashion ensures that the whole process of bug finding is as optimized as possible.

The last piece of the puzzle within the process is the ability to track verification progress over time to allow trends to be seen. The verification database requires the ability to save snapshots of the process over time so that it is easy to look at the history for a particular coverage object. This trending can often be used with the data from other metrics to give a better indication of the completeness of the verification process.
Team-Driven OVM Testbench Creation
by Tom Dewey, Technical Marketing Engineer, Mentor Graphics

INTRODUCTION
In the world of OVM testbench development there exist producers and consumers. Producers create and provide reusable OVM components and templates to consumers, who in turn employ these items to assemble testbenches and to write specific tests. While producers and consumers typically possess differing goals and skill sets, they must work together for success. This article discusses how these two groups of developers can interact efficiently to create testbenches, enabled by a new tool from Mentor Graphics: Certe™ Testbench Studio (CTS).

CORRECT-BY-CONSTRUCTION TECHNIQUES
Creating a testbench using correct-by-construction techniques means that the team is using provably-correct refinement steps as they write code. These steps provide standardized and automated code, such that the team can focus on adding value to that code. Correct-by-construction techniques can include: using templates, visualizing the testbench, and checking code against rules.

USING TEMPLATES
A template is a means to encapsulate and standardize best practices. The producer can create templates for environments, components, tests, sequences, agents, and classes, for example. These templates contain configurable variables that the consumer can then customize in order to get started with a correct framework of code. Figure 1 shows a partial template for a driver. In this example, variables are represented in all capital letters (such as TMPL_DRIVER).

```verbatim
class %TMPL_DRIVER% extends cvm_driver #( %TMPL_REQ_TON% );

// declaration macros
cvm_component_utils_begin(%TMPL_DRIVER%)
cvm_component_utils_end

// external interfaces
cvm_analysis_port #( %TMPL_REQ_TON% ) req_ap;

// variables
%TMPL_REQ_TON% req_txn, rsp_txn;
```

Figure 1: A Partial Driver Template

Nominally, a template contains boiler plate text with variables sprinkled throughout and the methods that the consumer needs to define are represented, as Figure 2 shows. However, templates can be as complex as desired. For example, they can be wrapped in Tcl code for programmable templates.

```verbatim
// new
function new(string name,
             cvm_component parent);
    super.new(name, parent);
endfunction: new

// build
virtual function void build();
    // IF Interface Object style
    cvm_object tmp;
    process v;
    super.build();
    // ELSEIF
    req_ap = new("req_ap",this);
    // ELSE IF Interface Object style
    if(!get_config_object("vif", tmp, 0)) begin
        cvm_report_error("build", "no virtual interface available");
    end
    else begin
        if(!cast(v, tmp)) begin
            cvm_report_error("build", "virtual interface is incorrect type");
        end
        else begin
            vif = v.vif;
            end
    end
    // ELSE IF Global Variable style
    vif = global_vif;
    super.build();
    // ELSEIF
endfunction: build

// run
virtual task run();
    // IF Blocking style
    drive_blocking();
    // ELSE
    drive_no_blocking();
    // ELSEIF
endtask: run
```

Figure 2: The Method Section of a Driver Template

The consumer chooses a template from a library created by the producer and CTS provides interactive help to fill it out, as Figure 3 shows on the following page.
The tool picks up all the variables defined in the template and provides them to the consumer to choose values. All the variables chosen are then replaced in the final code that is generated for the driver, as Figure 4 shows.

The consumer then adds appropriate content. Typically, this means filling out method functionality, adding additional OVM components via another template, and connecting up components. CTS provides auto-complete functionality to help correctly specify code constructs. As the consumer types, only the correct text or keyword allowed at that point in the code is presented for use. Figure 5 shows the progression of auto-complete steps for filling out the construction of an OVM component m_agent within a build method.

**USING AUTO-COMPLETE**

The consumer can then use the CTS auto-connect feature to correctly connect to a port or export. Figure 4 shows the only valid set of ports and exports to which to connect the req_ap analysis port to, within the m_agent component.
To connect the component, the consumer asks CTS for the legal connections for the connect statement. Figure 6 shows that only the listed exports are legal for the agent.

**Figure 6: Using Auto-Connect**

**VISUALIZING TO CHECK ASSEMBLY**

As the consumer adds the components provided by the producer and connects them up, the testbench can be visualized to check the progress of the assembly. Missing, unconnected, or miss-connected components can easily be identified. Figure 7 shows a fully-assembled environment class representing an OVM testbench for a router design.

**Figure 7: Visualizing Testbench Code**

**CHECKING CODE AGAINST RULES**

The consumer eventually types in code that is not guided by CTS correct-by-construction techniques. To check this code, the consumer can run the built-in checker against a set of rules that look for potential mistakes and simulation issues. The consumer interacts with the results to fix issues and can issue a checking report to the team. Both the producer and consumer agree on the rules that should be executed, what files should not be checked, and whether or not any violations are acceptable.

**BUILD MANAGEMENT**

Teams can standardize project builds and makefiles for simulation. This allows developers to select tests and environments and then run Questa™ using a standard set of commands and options on the correct set of testbench files. To minimize build mistakes, this build environment works in concert with the version management tool. The producer and the consumer typically work together in order to define a build management methodology. Once in place, CTS can be used to enforce this methodology. CTS offers a Build Manager that can be used to set up Questa commands and options for compilation and simulation. A makefile is generated automatically and executed when an OVM test is built for simulation.

OVM testbenches can be highly configurable. The actual topology of the testbench is not determined until Questa builds out the design. In order to check that the correct configuration of the testbench is being simulated, the consumer can obtain a dynamic visualization within CTS. Figure 8 shows that the p_driver component (circled in Figure 7) actually is configured for this simulation to use eight connected components.

**DESIGN REVIEWS**

Teams need to understand code relationships, navigate through references, and trace code through visualized environments for design reviews. Correct-by-construction techniques and automation allow the team to focus on important elements of the review, such as chosen testbench methodologies and effective sequences that impact simulation quality. Typically the team brings the following items to the review:

- **The static and dynamic visualizations of the design.** If viewed in CTS, the team can navigate to the source code using the static visualization.
- **The design checking reports.** The team can explain any existing violations or warnings within the review.
- **The class hierarchy and file references.** Review class inheritance and methods and use CTS to navigate the source file references.
- **Context-sensitive help.** If the team needs to look up details about an OVM class or concept, they can use CTS to research that information. For example, Figure 9 shows getting help on what an OVM agent is from the OVM Cookbook.
CONCLUSION

Using correct-by-construction and build management techniques, producers and consumers can quickly assemble accurate testbenches using Certe Testbench Studio. This allows the team to concentrate on productive testbench coding (as opposed to infrastructure code) and to achieve highly-focused design reviews.

For more information, please refer to: http://www.mentor.com/certe
The verification management problem can be described as a process of optimizing often conflicting goals, such as, project resources, quality objectives, schedule, and cost. However, a question frequently asked by project teams is: what is the best solution to this problem? Unfortunately, there are very few industry models for assessing an organization’s ability to achieve project goals. In this article, we propose a simple model that can be used by an organization to assess their advanced functional verification capabilities, while providing a roadmap for process improvement.

So what are the benefits of evolving an organization’s process capabilities? A good analogy to these benefits is the maturing process of a child. For example, when children come into this world, they are self-centered, that is, focused on satisfying their needs without any consideration for those around them. As they mature into adolescents, they recognize that their actions affect those around them. As a result, they start to consider the implications of their actions as they relate to the group to which they belong. In effect, they are optimizing their decisions for the immediate group around them. Then, as children grow into adults their actions (hopefully) evolve to a state where they consider the populations of the World and even future generations. What’s interesting here is that as our agendas become more outward-focused and dependent on others in the global community, the greater the return on investment we receive for the actions we take. The same holds true for an organization. When an organization moves from a state of crawl to walk to run, it moves from an individual focus to a group focus to an organizational/enterprise focus. Hence, evolving the organization ultimately enables it to optimize project resources, achieve its quality objectives, meet project schedule, and reduce costs.

THE NEED TO EVOLVE PROCESS CAPABILITIES

In the early 1990’s, the design community moved design up a level of abstraction from gate level to RT level. Evidence of this shift is shown in Figure 1, with the early increase in the curve representing the ability to design. [1,3] Yet, even with today’s synthesis breakthroughs in design productivity, designing and synthesizing RTL entirely from scratch cannot keep pace with what it is possible to fabricate (within a project’s schedule). Hence, solutions, such as the adoption of third-party IP, will be necessary to increase design productivity in the future.

A further examination of Figure 1, draws attention to a disparity between what can be designed and what it is possible to verify. Yet in many respects, the data in Figure 1 seems to defy reality. Project teams actually do successfully verify and tapeout complex chips today. Furthermore, today’s verification gap is not due to a lack of innovation in verification technology. In reality, what often differentiates a successful organization from an unsuccessful one is its ability to evolve process capabilities. Unsuccessful teams tend to approach development in an ad hoc fashion, while successful teams employ a more mature level of methodology that is systematic.

Aside from today’s productivity gap challenges, ensuring functional correctness on RTL designs continues to plague many organizations. In fact, an often-cited 2004 industry study revealed that 35 percent of the total development effort was spent in verification. [1] More recent studies indicate that the verification effort has risen to 46 percent of the total development effort. [2] Unfortunately, with the increase in verification effort, the industry has not experienced a measurable increase in quality of results. For example, only 28 percent of projects developing silicon designs are able to achieve first silicon success, as shown on the following page in Figure 2. [2]
To make matters worse, the industry is witnessing increasing pressure to shorten the overall development cycle. However, only about a third of the industry has been able to meet schedule, as shown if Figure 3.[2]

Clearly, new functional verification techniques, combined with a focus on evolving process capabilities within an organization, are required to help close today’s verification gap.

A MODEL FOR MATURING AN ORGANIZATION’S PROCESS CAPABILITIES

In the early 1980’s, the United States Department of Defense established the Software Engineering Institute at Carnegie Mellon University in response to a perceived software development crisis related to escalating development costs and quality problems. One of the key contributions resulting from this effort was the published work titled The Capability Maturity Model: Guidelines for Improving the Software Process. The Capability Maturity Model (CMM), which consists of five maturity levels, as illustrated in Figure 4, is a framework for assessing effective software process. It provides an evolutionary path for improving an organization’s processes from ad hoc, immature processes to developed, mature, disciplined ones.

Unfortunately, the CMM does not currently address expertise required by an organization for a particular application domain, such as hardware verification. Nor does it advocate specific process technologies related to a particular discipline. Furthermore, even though the CMM advocates the importance of defining metrics-driven processes, it offers no guidance on what those metrics should be. Finally, many experts believe that the overhead incurred in applying the CMM to today’s hardware projects with short development cycles is prohibitive. Hence, to help close today’s verification gap, a new and simplified model is needed for assessing an organization’s functional verification process capabilities, which we refer to as the Evolving Capabilities Model (illustrated in Figure 5).

THE NEED FOR A SIMPLER MODEL

So, what characterizes an ad hoc organization? First, it is important to understand that an ad hoc organization can accomplish amazing work—but not predictably and repeatedly, and their success is often on the backs of the superstars or firefighters within the organization. For an ad hoc organization, most of processes tend to be improvised by practitioners and their managers, thus lacking any formal definition, documentation, and unified understanding within the organization. Ad hoc organizations tend to be reactionary and focused on solving the immediate crisis. Furthermore, the organization lacks quantitative and objective bases for solving product or process problems. Hence, an ad hoc organization has a difficult time predicting the project outcome in terms of schedule, resources, and quality—and often will compromise functionality and quality to meet schedule.

Figure 2. First silicon success continues to decrease

Figure 3. Two-thirds of all designs slip schedule

Figure 4. Traditional CMM

Figure 5. Traditional CMM
Adopting the Evolving Capabilities Model provides improved visibility into an organization’s verification processes using a metrics-driven approach. This enables the organization to identify risk while providing feedback for optimizing processes. For example, Table 1 provides a high-level set of metrics that can be used to assess an organization's ability to predictably execute in terms of cost, schedule, and quality objectives. For instance, an organization that crawls, with respect to the Evolving Capabilities Model, typically deviates from its original plan by greater than 50 percent. They often experience three or more respins, they are likely to overrun their cost projections in proportion to schedule overruns, and they deploy tools, methodologies and flows in an inconsistent manner.

<table>
<thead>
<tr>
<th>Crawl</th>
<th>Walk</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution: &gt;50% Deviation</td>
<td>Execution: 20-50% Deviation</td>
<td>Execution: +/- 20% Deviation</td>
</tr>
<tr>
<td>Quality: 3+ Spins</td>
<td>Quality: 1-2 Spins</td>
<td>Quality: First Pass Success</td>
</tr>
<tr>
<td>Cost: &gt;50% Unbudgeted</td>
<td>Cost: 20-50% Unbudgeted</td>
<td>Cost: &lt;20% Unbudgeted</td>
</tr>
<tr>
<td>Deployment: 5-25%</td>
<td>Deployment: 25-50%</td>
<td>Deployment: 50-100%</td>
</tr>
</tbody>
</table>

Table 1. Assessing an organization’s overall execution

The next level of assessing an organization’s evolving capabilities focuses on specific aspects of the verification process. For example, Table 2 provides a set of processes typically associated with different verification capability levels.

<table>
<thead>
<tr>
<th>Crawl</th>
<th>Walk</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stimulus: Directed</td>
<td>Stimulus: Constraint random</td>
<td>Stimulus: Scenario/Use case based</td>
</tr>
<tr>
<td>Correctness: Manual</td>
<td>Correctness: Automated</td>
<td>Correctness: Requirements driven</td>
</tr>
<tr>
<td>Coverage: Prioritized checklist</td>
<td>Coverage: Structural</td>
<td>Coverage: Functional</td>
</tr>
<tr>
<td>Methodology: None or ad hoc</td>
<td>Methodology: Repeatable</td>
<td>Methodology: Unified</td>
</tr>
<tr>
<td>Reuse: Accidental</td>
<td>Reuse: Planned</td>
<td>Reuse: Standards based</td>
</tr>
<tr>
<td>Debugging: Visual inspection</td>
<td>Debugging: Ref Model/Scoreboard</td>
<td>Debugging: Automated Analysis</td>
</tr>
<tr>
<td>ABV: None or ad hoc</td>
<td>ABV: Hot Spots</td>
<td>ABV: Comprehensive and full proofs</td>
</tr>
<tr>
<td>Reviews: On demand</td>
<td>Reviews: Planned but unstructured</td>
<td>Reviews: Structured, periodic, &amp; cross functional</td>
</tr>
<tr>
<td>Planning and Scheduling: You, you, and you!</td>
<td>Planning and scheduling: Historical</td>
<td>Planning and scheduling: Statistical</td>
</tr>
<tr>
<td>Experience: &lt; 4000 hours</td>
<td>Experience: 4000 – 10,000 hours</td>
<td>Experience: &gt;10,000 hours</td>
</tr>
<tr>
<td>IT: Use what’s available</td>
<td>IT: Peak planning</td>
<td>IT: Forecasted</td>
</tr>
<tr>
<td>Metrics: Ad hoc</td>
<td>Metrics: Defined and tracked</td>
<td>Metrics: Used to optimize processes</td>
</tr>
</tbody>
</table>

Table 2. Assessing an organization’s verification process capabilities

Table 2 provides assessment criteria for many aspects of the functional verification process such as: stimulus generation, correctness modeling, coverage analysis, methodology adoption, reuse strategy, debugging approach, assertion-based verification adoption, review process, verification planning, time that the group has worked as a team, the project’s IT plan, and metrics that have been implemented to measure success.

For example, an organization that crawls generally thinks about stimulus only in terms of a set of directed test. The problem with
this approach is that it becomes unmanageable as the number of scenarios increases, often due to concurrency. As an organization evolves its verification process capabilities, it typically introduces constraint-random stimulus generation approaches into its flow. An advanced organization tends to think of stimulus generation at a high level of abstraction—such as transactions and scenarios. Each key aspect area listed in Table 2 can be used to help an organization assess current verification process capabilities.

EVOLVING YOUR ORGANIZATION

The fist step in evolving process capabilities involves the following:

a. understanding the organization's existing technologies, methodologies, and flows
b. identifying opportunities to introduce disruptive technologies, methodologies, and flows

The first point requires the implementation of process metrics to identify existing strengths and weaknesses, while the second requires innovation, creative thought, and industry perspective. Both vectors must be pushed, along with management buy-in, to successfully drive an organization’s evolving process capabilities.

The Evolving Capabilities Model is both a tool for assessing an organization’s advanced functional verification process capabilities, as well as a roadmap for process improvement. For example, an organization that plans to evolve its process from one level of process capability to the next needs to consider the resources, skills, tools, and metrics it must implement to succeed. To assist in this effort, Mentor Graphics provides the following resources:

• The Verification Academy (www.verificationacademy.org), which is an online resource for maturing skills needed to adopt advanced functional verification techniques. This free resource provides a methodology bridge between advanced functional verification high-level value propositions and low-level tool operation and language details.

• Mentor Training, which offers a wide range of onsite classes and web-based events designed to help you apply advanced functional verification tools and languages.

• Mentor Consulting Services, which offer excellence and expertise in the deployment, adoption, and knowledge transfer for advanced verification techniques.

SUMMARY

Addressing today’s functional closure and productivity gap challenges makes it necessary for an organization’s verification process capabilities to evolve. In this article, we introduced a simple model for assessing an organization’s advanced functional verification process capabilities while providing a roadmap for process evolution.

REFERENCES


System-on-Chip (SoC) designs continue to grow in size and complexity. Traditional verification techniques are rapidly becoming inadequate to the task of performing system-level verification. This problem is further amplified by the need to verify enormous amounts of embedded software which is more prevalent in today’s SoC designs. To combat this burgeoning verification challenge transaction based verification has been introduced. But is it enough?

SO WHAT ABOUT EMULATION?

Since its inception emulation technology has experienced extraordinary progress and improvement (See Not Your Father’s Emulator example). Not only have emulation speed and capacities increased, but techniques used to control the emulator, simulate the design, and debug the results have also changed radically. The advent of Mentor Graphics’ TestBench-Xpress (TBX) tool combined with the Veloce emulator opens an entirely new chapter in emulation use.

NOT YOUR FATHER’S EMULATOR

From the very beginning, in-circuit emulation (ICE) consisted of specialized hardware designed to act as the circuit so that native code could be developed and debugged in the environment where the circuit in question was actually used. An example of this was the Intel™ 8085 ICE85 Development Kit. A cable from the ICE85 emulator box was plugged into the micro-processor connector in the actual circuit board device under test (DUT) and the assembly code executed in the emulator. The ability to set breakpoints and to single step proved to be a huge benefit for the software developer.

In the late 1980s emulation took a step forward with the introduction of the Quickturn Rapid Prototype Machine (RPM). The use of FPGA technology allowed the RPM to be configured to model any design in one or more RPMs at 50K gates per RPM running at emulation speeds of 200 kHz. Quickturn was later purchased by Cadence Design Systems and continued to increase the capacity of their emulators. Cadence later dropped the product line in lieu of processor based technology when it became apparent that the I/O limitation of available FPGAs could not be overcome.

Mentor Graphics Corporation first entered the emulation market in the mid 1990s by purchasing the French company Meta. At about the same time, the Massachusetts Institute of Technology developed VirtualWires Technology and spun off a company called Virtual Machine Works (VMW). With VirtualWires, VMW solved the FPGA I/O limitation problem by automating signal multiplexing between FPGA partitions. In 1996, Ikos Systems purchased VMW and continued to develop the VirtualWires Technology, increasing both the size and speed of their emulators. In 2001, Ikos offered the first VStation emulator with 15M gate capacity and later increased that to 50M gates. Mentor Graphics purchased Ikos in 2002 and combined it with their Meta team to form Mentor Emulation Division (MED). Incorporating the best traits of the Meta products and Ikos products, MED developed the Veloce product line. Veloce, a modular system built around a customized FPGA card with internal memory and external I/O connections, can emulate up to a half billion gates and 32Gbytes of memory at speeds up to 1.2MHz.

TBX ADVANTAGES

The Veloce TBX transaction-based accelerator utilizes high levels of abstraction modeling (e.g. models written in C, C++, SystemC, and SystemVerilog) to verify the RTL inside the Veloce emulator. These languages can be used in conjunction with the emerging OVM, VMM, and TLM standards to create a truly flexible transaction-based testbench methodology. TBX software completely automates the emulator clocking controls necessary to pause the emulation while the software (C, C++, SystemC, or SystemVerilog) side is accessed. The greatest advantage of TBX is that the transactors using TBX’s standards-based communication interface and the Extended RTL (XRTL) modeling subset can be run on any SystemVerilog compliant industry leading software simulator, in addition to the Veloce emulator. Another important benefit to using TBX is it allows verification engineers to utilize the power of the Veloce emulator to debug at the...
transaction level without the need to build costly and time consuming ICE hardware target systems.

**FLEXIBLE TESTBENCH SOLUTIONS**

With TBX, the time required for building and maintaining a PCB ICE target, with its limited visibility, cost, and inflexible design, is now replaced with a much more flexible solution. TBX allows for multiple re-spins on a daily basis simply by tweaking the DUT and the testbench components. Existing C BFM's from the verification or modeling teams can also be used in emulation with testbench and DUT visibility greatly enhanced. TBX facilitates a huge amount of re-use among both verification and emulation teams and allows for instant environment replication when deploying on multiple emulators compared to the much more labor intensive ICE emulator replication process.

**THE C, C++, AND SYSTEMC TESTBENCH ENVIRONMENT**

A verification environment developed using C, C++, SystemC untimed testbenches along with a timed DUT written in Verilog, VHDL, SystemVerilog or mixed languages, which communicate via SV-DPI, is interoperable between TBX (for acceleration) and software simulators (for simulation). This allows users to keep one common verification environment and source files for simulation and acceleration. Though TBX does not require a simulator engine to run this environment, the testbench can execute on an OSCI open source SystemC kernel, or alternatively, on Mentor’s Questa advanced verification environment running only SystemC, as seen in Figure 1.

**THE SYSTEMVERILOG TESTBENCH ENVIRONMENT**

A verification environment developed using SystemVerilog untimed testbenches and a timed DUT written in Verilog, VHDL, SystemVerilog or mixed languages, which communicate via TLM FIFOs/Ports, is interoperable between Veloce TBX (for acceleration) and software simulators (for simulation). In this case, TBX requires a Questa simulator to execute a SystemVerilog testbench on a Linux host (Figure 2). The testbench running on the Linux host communicates to the DUT in the Veloce emulator via standard TLM channels. Users can select their choice of simulator for simulation-based verification which could be VCS, NC, or Questa; however, when users move to acceleration, TBX will use Questa (transparent to the user) as an underlying simulation engine.

Mentor Graphics is fully committed to working with the evolving open standards and interoperable modeling methodologies.

**OVM METHODOLOGY**

The Open Verification Methodology (OVM) facilitates writing structured, interoperable, and reusable verification components and is adopted by both Mentor Graphics and Cadence. By virtue of its support for untimed SystemC and untimed SystemVerilog, TBX enables users to build verification environments based on OVM. The testbench components depicted in purple in Figure 3 run on a Linux host. The timed logic (DUT + Transactors) depicted in green in Figure 3 run on Veloce, and interact via a transaction-based communication mechanism using SV-DPI, transaction pipes, or TLM. TBX delivers...
accelerated TLM FIFO, which is an accelerate-able version of the tlm_fifo, allowing the user to build a TBX-compliant OVM environment for acceleration. The accelerated TLM FIFO is extended from tlm_fifo, and its usage is identical to tlm_fifo and functions within native software simulators as well. An OVM environment can be accelerated as long as certain guidelines are adhered to making it compliant with Veloce TBX as well as the native software simulator.

These guidelines include:

- Transactors (monitors, drivers, and responders in Figure 3 below) need to be coded in the eXtended RTL (XRTL) subset provided by TBX
- The communication between the emulator side components and testbench components is done by using SCE-MI 2 compliant function calls, transaction pipes, or accelerated TLM FIFOs and ports
- Testbenches components should be untimed

![Figure 3. The OVM Veloce TBX acceleration environment.](image)

**VMM AND VERA RVM METHODOLOGY**

VMM and Vera RVM based environments designed with the guidelines outlined in OVM section can be accelerated using Veloce TBX. Users will be able to use the source code (DUT + Testbench + Transactors) with VCS or Questa for simulation, and then use the same source code with Veloce TBX. TBX uses Questa as the underlying simulation engine for the untimed VMM or Vera/VMM test bench execution. Use of the OVM overlay package from Mentor Graphics is required.

![Figure 4. Vera/VMM Veloce TBX acceleration environment.](image)

**WHAT THE HECK IS A TRANSACTION ANYWAY?**

To get the most performance out of any transaction based model, it is best to capture multiple clock cycle events and repackage those as transactions. The code to handle this optimization is typically called a transactor. Simply put, a transactor is code added for the purpose of communicating between an untimed testbench and a timed DUT. As mentioned earlier, the communication between the testbench and the DUT can run in an industry standard simulator or in the Veloce hardware. Transactors can be written using synthesizable RTL, or a larger synthesizable subset of Verilog/SystemVerilog referred to as eXtended RTL (XRTL). This makes transactors more powerful and the task of writing the transactors simple. Figure 5 lists a number of the constructs that XRTL provides.

![Figure 5. List of XRTL constructs.](image)
**TRANSACTION MODIFICATIONS EASY TO CODE**

As previously stated, TBX can communicate using SV_DPI calls, transaction pipes, or TLM facilities. This has greatly simplified the verification engineer’s job. The TBX software completely automates the control necessary to communicate with the Hardware Verification Language (HVL) application. The HDL required is as simple as declaring a DPI function call and using it as in Figure 6. Figure 7 shows the necessary code for a transaction pipe. The user does not have to worry about controlling the emulator clocks while the HVL application executes, nor does he need to understand any of the underlying mechanics involved in communicating between the emulator and the HVL application.

```
import "DPI-C" context function void dpisramread256
  (input bit [26:0] read_addr,
   output bit [255:0] read_data);

import "DPI-C" context function void dpisramwrite256
  (input bit [26:0] write_addr,
   input bit [31:0] write_data,
   input bit [255:0] write_enable);

always @(posedge clk)
  if (read_strobe==1'b1)
    dpisramread256(read_addr[31:5],read_data[255:0]);

always @(posedge clk)
  if (write_strobe==1'b1)
    dpisramwrite256(write_addr[31:5],write_data[255:0]);
```

*Figure 6. Verilog DPI source code.*

```
#include "scemi_pipe.h"
void TB::tp_thread() {
  ...
  void *handle_in = scemi_pipe_c_handle("top.inputPipe");
  ...
  void scemi_pipe_c_send( void* handle_in, (svBitVecVal *) &b, 0);
  ...
  void scemi_pipe_c_send( void *pipe_handle,
    int num_elements, const svBitVecVal * data,
    module top,
    scemi_input_pipe #($1, 40) inputPipe();
  always @(posedge clock) begin
    ...
    if(tem) begin
      inputPipe.realtime(, no_valid, data, eom);
      ins byte < data;
      end
    end
  ...
  scemi_input_pipe #($BYTES_PER_ELEMENT(I),
    PAYLOAD_MAX_ELEMENTS(I),
    BUFFER_MAX_ELEMENTS(40)));
```

*Figure 7. Transaction Pipe code.*

**CONCLUSION**

Though emulation has been around for decades, the advent of Mentor’s Veloce/TBX duo allows users to incorporate into their verification strategy the fastest, highest capacity emulation engine on the market. With only minimal code changes, TBX can pair the Veloce emulator with C, C++, SystemC, and SystemVerilog testbenches which can then be extended to an OVM environment. Additionally, only one flow and one common source need to be maintained for simulation and TBX acceleration because Veloce TBX is a standards-compliant accelerator.
The amount of new verification technology released each day can be overwhelming even for the most technically voracious company. With so many choices the opportunities for improving performance get lost, especially when most people are comfortable with the age old view that performance is a simple measure of how long one or two target simulations takes to complete. Adding memory footprint to the performance datapoint is a fairly new way to to add an extra measure of comfort, measure once and assume the metric will be valid thought the life of your project current and future. Although this view is completely valid, these are not the only throughput metric that should be considered. Throughput is also the time to run a regression environment, the time to coverage, and the time to resolve a bug. All are equally important in the lifetime of a project. Projects are seldom static: workforce balancing and market influences on project specification can stress the most experienced teams.

How you solve your throughput challenges may be a matter of knowing what potential opportunities exist. There is a cast of technologies that are broadly relevant, including regression suite throughput, coverage closure techniques and time to bug resolution. There are many other opportunities for throughput, however we will focus on two in this article.

REGRESSION SUITE THROUGHPUT

Consider typical design and verification components: Testbench, RTL DUT, CPU core model(s), memory, SystemC, C/C++, verification IP, assertions, debug database, coverage database, multiple clock domains, scan chain logic. Mixing and matching during block and system testing is very common. Testbenches, MVC verification IP, CPU core model support have all been evolving rapidly. SystemVerilog constrained random test stimulus and functional coverage, Assertions and libraries of assertions, Codelink application code debug technology for validation of reusable application code, verification IP with TLM based analysis and ready to use verification plan. Each brings significant ROI, partly by reducing the risk inherent in any verification project. What has not changed as rapidly is the rethinking of what constitutes verification throughput. Does measuring simulation run time under serve your verification goals? For many, the answer has evolved along with the verification technology they have deployed.

The time to run a regression suite, measure the results, and the time to the next revision of design is the real goal. The time to the next version of a chip is a natural response to time to market pressures. This places a premium on your ability to leverage as much from existing resources in order to contend with a growing design and associated verification task. With this perspective the time to run one test really is a subset of the overall time to release a product plan.

As a quick reference an abridged history of Questa; global optimizations were introduced in 2005, threaded waveform logging in 2007, threaded SystemC compile in 2008. The opportunities for more global and threaded performance are far from exhausted. The reality is new hardware advances have provided rich opportunities. In addition to the “traditional” simulation flow, there exists a novel time to next simulation technology that has been deployed with dramatic regression suite improvements in both time and disk resources. The basic idea is to reuse pre-optimized portions of the design. In many environments the Device Under Test (DUT) remains constant while the testbench changes from one run to the next. Even if the DUT has some

**Figure 1: Typical Design Components**

*by Joe Rodriguez, Verification Product Marketing, Mentor Graphics*
parameters or generics that change from run to run they can remain flexible and this type of DUT can become pre-optimized reusable “black boxes”. The black boxes (bbox) have two lasting benefits, eliminating the time to optimize the DUT and their associated disk image. See Figure 2 on the following page.

![Typical Optimized Flow](image)

**Figure 2: Time to next simulation benefits**

For some users this documented flow’s improvement has been dramatic. This flow is posing new questions. The prime target environments are those where the optimization phase (vopt) time is significant versus the run time. As mentioned, verification technology has changed the way tests are created and their associated impact to run time needs to be re-analyzed. Where you were ten years ago is most likely not where you are today, given the profile of your regression suite optimization time and regression test suite run time. There may be some long running tests that may skew the average, but chances are you have an opportunity to examine and update your regression test harness to take advantage of this. With a shortened time to run a test suite many customers focus on additional testing that had been rejected due to time constraints, or simply to improve time to next version of the design, ultimately time to market. To best answer the resource question you need visibility into your coverage environment.

**Coverage Closure Techniques**

Coverage closure is a common technique for analyzing when you have eliminated enough risk to sign-off your golden RTL design. The development of verification plans focus on areas of the design that are new or interfaces that are higher risk. How true your risk elimination aim is normally directly related to the resources and time available during validation. This is actually one driver behind constrained random test stimulus. In addition, the availability of off the shelf Verification IP like Mentor Graphics MVC with TLM for performance and understanding is verification test plan ready so that you can easily focus on protocols you plan on supporting. Yet for many there is still a persistent lack of verification process visibility and access to the information needed to know when you’re done. The ability to transition from verification planning, verification process measurement and ability to analyze and react exists today, but are not yet fully utilized outside of the most advanced verification teams.

Arguably knowing “when you’re done” and “not done”, where your coverage holes are and what resources are aiding coverage closure are more important than regression suite throughput. There are short term tactics that can aid your throughput. The Unified Coverage DataBase (UCDB) within the Questa environment provides the infrastructure for collection of all coverage information, the ability to link test plans to simulation coverage results and the ability to share and analyze the results. See Figure 3. Even with code coverage you can merge and rank your test suite results and rank them to determine how many of your tests are actually contributing to your coverage goals. Why rerun tests that are not contributing to coverage closure? Instead trim your regression suite throughput by eliminating non-contributing or invest in new contributing tests. Higher value is taking the time to map your test plan to the simulation results with Verification management.

Verification Management is an effective methodology for gaining control of your coverage closure challenges.

A successful verification process will identify bugs. Using Verification Management technology you can leverage the UCDB to identify and organize failures and automate the ability to rerun interactive analysis of the failed tests. This is the first step in finding and fixing the root cause of the failure. If you have invested in The Open Verification Methodology you realize that debug of SystemVerilog brings new challenges. Viewing OVM hierarchy,
assertion debugging and dynamic object debugging are unique challenges that Questa has already enabled. Other high value debug items such as rendering design intent similar to Finite State machine (FSM) extraction and animation, source code hyperlinking and causality traceback, can make significant differences when engaged in bug hunting. Just like performance and coverage technologies have evolved, so is debugging.

Verification technology enables an expanded definition of through-put. Run time is still very important and the subject of continued investment by the EDA community, but is no longer the only valid metric to the design and verification community.

For more information on Questa and the techniques discussed in this article, please visit, www.mentor.com/questa.
Verifying a USB HUB is always a challenge because you need to reuse the same environment at the system level. The verification environment should facilitate the use of multiple TLM components when only the HUB is ready, and, as the RTL host and devices are created, it should replace the TLM components with the RTL. It should also gather TLM activities at all interfaces in order to analyze them and make sure that all the necessary transactions have been run successfully.

The verification of a USB 2.0 HUB requires that the transactions are run at high, full, and low speeds so as to verify the HUB’s capability of handling split transactions targeted to full/low speed devices in parallel with high speed transactions. In this article, we will use the verification of a USB 2.0 HUB as an example to show how this can be done. The same approach can be applied to other protocol components that assume similar top-level functionality as a USB 2.0 HUB.

There are a few points where verification components will be used while verifying a HUB-based system.

1. To verify a stand alone USB 2.0 HUB when no USB 2.0 host and devices are ready.
2. To verify an individual USB 2.0 host and devices.
3. To verify a USB 2.0 HUB when either the host or few or no devices are ready.
4. To gather the TLM activity at all the interfaces for a transaction displaying functional coverage/transaction logging when all the RTL host and devices are connected with the HUB. In this case, verification components would be working in passive mode.

While defining the architecture you must make sure that: an exhaustive test suite can be generated, functional coverage is collected at all the interfaces, protocol compliance checks exist at all the interfaces, and, most importantly, it is easy to reuse this environment at the system level when the host and devices are ready.

This can be easily achieved with Mentor’s USB Multi View Verification Component (MVC), which works at multiple abstraction levels. You only need to replace TLM host/devices with RTL host/devices once the design is ready.

MVCs are SystemVerilog-based and OVM 2.0 compliant. They provide the flexibility of running stimulus at any abstraction level, making it much easier for users to move from one abstraction level to another. For example, in USB there are two abstraction levels: packet (token, data, and handshake) and stage (setup, in data stage, out data stage, start split stage, complete split stage). This provides other kinds of flexibility as well. For example, you can provide different patterns of delay between packets by running stimulus at the packet level. You can run transactions at different abstraction levels, you can view them in Questa’s waveform display, and you can log them in a log file. In other words, it provides a direct view of transactions run at different abstraction levels.

Figure 1: Waveform snapshots of both packet and stage level transactions as well as the parameters of the respective transactions can be easily seen in the waveform window.
Figure 2 shows a USB 2.0 HUB with four ports. Each port is connected to a device (two high-speed devices, one full-speed device, and one low-speed device).

To verify a USB 2.0 HUB similar to this one, MVC can be used at each port (upstream and downstream ports), yielding the following features:

1. Each interface will have a different configuration. The upstream port has to be configured as the high-speed upstream port of the USB 2.0 HUB while the downstream ports are configured as high / full / low speed device ports, as the case may be. Thus the environment would comprise of more than one MVC getting instantiated.

2. Since each device as well as the hub will have a unique address, the host can be constrained to generate legal addresses only.

3. The host can be constrained to initiate legal transactions, depending upon the type of the device. For example, the host would do a split transaction when targeting transactions to full / low speed devices.

4. Functional coverage will be collected at each interface level. MVC comes with an XML verification plan which can be used along with the Questa Verification Management feature to automatically track the final coverage with respect to the XML verification plan.

5. A protocol checker will be instantiated at each interface so that if any illegal behavior is observed an assertion is fired.

6. Transaction level displays in the waveform window as well as in the log file.

Figure 3 and Figure 4 illustrates the MVC architecture for a host and device interface that replaces the host in Figure 2.

In order to verify the USB 2.0 HUB, the MVC architecture shown for the RTL host can be directly connected to downstream ports of the HUB. For the upstream port of the HUB, the architecture shown in Figure 2 can be used. The only difference it will have is the TLM host sequence. Earlier the TLM host sequence would have been targeting USB peripheral devices only, now it will be targeting the HUB as well as more than one other device.
Once tests are run then functional coverage can be saved, and it can be merged with XML provided verification to view how far one has gone in their verification goals. Figure 5 shows a snapshot of functional coverage.

![Functional coverage report](image)

**Figure 5.** Functional coverage report shows verification holes.

**CONCLUSION**

To verify a block or a system, an environment must be created which can be easily moved from the block to system level. The verification components used must have a protocol compliance checking mechanism which should be measured precisely. Verification components must have a functional coverage mechanism along with a detailed verification plan so that verification progress can be tracked.

MVCs come with all of the above features and also can work at any abstraction level, making it much easier to move from one abstraction level to another. An analysis component (e.g., functional coverage) is a SystemVerilog-based unencrypted component; therefore, it can be changed if a block supports only a subset of a complete protocol.

**REFERENCES**

For more information about MVC, please refer to the following web page:

- “SystemVerilog Testbench Constructs”
- “Open Verification Methodology” – http://www.oviworld.org
ABSTRACT:

This article focuses on techniques to port a normal verification environment to a SystemVerilog based environment with details from the verification of a real-world wireless SoC project. Emphasis will be on using three major strengths of OVM based SystemVerilog environment: (1) SystemVerilog modeling with assertions (2) Functional coverage and (3) Simplify the block to top flow using standardized components, directory and flow.

Key techniques mentioned are:
1. Deciding on the right transaction boundary.
2. Migration to OVM while reusing some existing infrastructure, i.e. C tests.
3. Creating a coverage driven approach using OVM-SystemVerilog-SystemC.
4. Other implementation aspects like: register space implementation in OVM and hardware-software sequences for system testing.

Verification environment creation is a critical phase of project execution as it defines the features, schedule, and completeness of the verification task. Different SoCs require different approaches for verification. In the case of a wireless SoC, multiple aspects of wireless communication have to be taken care of by the verification environment. The SystemVerilog DPI supports interfaces to C but not C++ classes. Many environments have C++, C, VHDL, and Verilog modules. Binding their features together and making a “methodology driven” environment is a key requirement in a complex SoC project execution cycle.

An interesting project we recently finished at eInfochips had a tightly coupled verification infrastructure comprised of the following.
A. Infrastructure: Common files make user interfaces for build and scripts used by multiple test directories. Automatic detection of out-of-date source files and forced recompile.
B. Test case writing process: C test writing capability with familiar user commands providing higher-level extensions like flow control, looping, array indexing, etc.
C. Simulation and dump process: Test and case-specific filenames created for all generated files to support batch mode processing of simulations. RTL, gate-level best case, and gate-level worst case simulations performed in same test directory using same testbench and test stimulus files. Automatic comparison of analyse.txt result files with corresponding “golden” files as part of pass/fail determination. analyse.txt. parsed file created from analyse.txt. This shows the register names instead of the register addresses. Data is shown in a concise bin/hex format instead of 32-bit binary strings. Time stamps are in ms, and a delta-time is provided. Automatic scanning of simulation log files for known
character strings (e.g. ERROR, *E, etc) that indicate failure of the simulation.

D. Analysis phase: Automatic check for test completion flag to prevent stalled tests from indicating a false pass indication. Comprehensive error summary and pass/fail report generated for each test.

As shown in figure 2, the complete SoC RTL was defined as Chip_top and integrated with the testbench infrastructure. Behavioral models were designed in VHDL and C. We wrote C programs on an ARC compiler and mapped hex files to the RTL memory. C programs were executed on every simulation for verification of a different block. A low level “results monitor” was designed to watch clk, GPIO, and Memory transactions to perform low level protocol checks. Most of simulation checks occurred through software programs.

The testbench component at the IP level was comprised of a C model or VHDL model for respective protocol blocks, such as WLAN and HSDPA module. AHB monitor was a VHDL model to detect AHB protocol violations. Thus, the testbench flow created a complex link structure as shown below.

Init IP level BFM and chip_top RTL sequence of parameter config for each BFM, limited constraints

Full chip_tb drives/initiates RTL at interface level

ARC C code (ROM image) is dumped through Verilog file structs inside ARC Core RTL

On the fly test execution switch start (makes rest to full chip RTL and testbench and initializes)

Software sequence runs on ARC and RTL simulations starts

**ANALYSIS FLOW**

Current log file:

```
message: ASSERT/FAILURE * from process :Readlazy:lazy
case: rtl gatebc gatewc
timewindow: 0 end US
-------- entry # 8 ---------
message: Testbench stopped
case: rtl gatebc gatewc
timewindow: 0 end US
-------- entry # 9 ---------
message: SIMULATION FINISHED WITHOUT ERRORS
case: rtl gatebc gatewc
timewindow: 0 end US
```
Tests are mainly written in “C” programs that run on ARC and perform verification as shown below.

```c
extern void (*intisr_10_p) (void);

void int_isr_10 (void)
{
    display (NOTE, "Entered into ISR 10 for MT calib check");
    if (((32)(CLK+CLK_T1) & 0x00001000) != 0)
        display(WARN,"Interrupt generated when MK is in progress");
    if ((RegC(ICU, IRQ_ENA_0)) & 0x10000000)
        RegD(ICU, IRQ_CLEAR_0, 0x10000000);
    RegE(ICU, IRQ_ENA_0, 0x00000000);
}
```

**MIGRATION REQUIREMENTS**

- Legacy environment was in VHDL, Verilog, C, and different language sub-sets
- Multiple IP with different verification approaches developed across the years
- No strong focus on advance verification methodology, features, and reuse
- Time to market versus execution time ↔ Maintenance of verification environment plus extensibility

**WHY OVM?**

Legacy environment was created by layer after layer implementation by designers, verification team and system validation team to cater to their individual requirements. There was no coherency about “architecture, flow, debug methods” at various levels, i.e Debug requirements were completely different for the designer and validation teams. This means, methodology was missing...!! When you avoid “methodology”, you end up in making environment bulky, tough and bug driven. OVM brings that discipline of methodology, flow and re-use. The OVM class library is very structured, derived from experience, and fully tested, and provides the following:

1. Environment configuration and multi-channel sequencer
2. Protocol layering with ovm_sequencer and API
3. Open class library enables multi-vendor support
4. TLM communication and built-in messages using ovm_transactions class and standard interfaces
5. Smaller test writing using inheritance; multiple test combining using factory; tests writers do not need to know OOP details of testbench, as it is separated from testbench.

6. Built-in phases to handle all phases in run-order, as shown below
   a. new()
   b. build()
   c. connect()
   d. end_of_elaboration()
   e. start_of_simulation()
   f. run()
   g. extract()
   h. check()
   i. report()

**OVM BASED SoC ENVIRONMENT**

The OVM testbench environment consists of reusable verification environment/components. Full chip verification was partitioned as (a) IP verification at the block level, (b) sub-system verification, and (c) full chip (system) verification. We designed them to be encapsulated and ready to use with a consistent architecture. Three verification components were mapped to OVM as block level testbenches, as shown in table 1.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WLAN block</td>
<td>SystemVerilog VIP</td>
</tr>
<tr>
<td>2</td>
<td>HSDPA block</td>
<td>Verilog Model</td>
</tr>
<tr>
<td>3</td>
<td>AHB block</td>
<td>SystemVerilog VIP</td>
</tr>
<tr>
<td>4</td>
<td>Peripheral block (I2C, SPI, GPIO, UART)</td>
<td>SystemVerilog VIP</td>
</tr>
<tr>
<td>5</td>
<td>ARC CPU</td>
<td>ARC model</td>
</tr>
</tbody>
</table>

Table 1: VCs that were mapped to OVM
A typical AHB verification component complying with the OVM will look like the diagram in figure 7. We have three main agents in the system (as already described): Master, Slave, and Bus [Arbiter + Decoder]. To combine them we use a wrapper on the top. This is our environment. Please note that each of them has its own configuration class. Assertions are mainly used to know whether the DUT is driving the signals properly or not. Functional checkers are employed to check the overall functionality of the DUT. It will trigger when the DUT or any module performs the wrong transaction. Please note that this will be active only and only if you keep the `CHECKER_EN` active in the `ei_abh_user_define.sv`. Env is the top level component of Verification IP.

**TYPICAL USAGE**

Following are difference examples of an AHB verification component that was set of compiler directives to set the below parameters.

- Enables for Master, Slave, Bus, checker and Assertions
- Enables AHB, AHB LITE topology and a number of agents (Slaves, Master)

A WLAN and peripheral block testbench was designed using the same approach. HSDPA and other block level testbenches followed the same approach with a SystemVerilog layer. Build() and create_component() functions are important ones to ensure the OVM multi-sequencer is the right combination for the OVM factory. In fact, the HSDPA block testbench was a typical Verilog model to generate base traffic and model the HSDPA functionality. The OVM library on top of it added “messaging”, “component structure”, ovm_threadhandler, ovm_sequencer. The HSDPA verification IP was modeled as an ovm_component with all the OVM features.

ARC sub-system testing was created reusing a legacy environment. ARC sub-system comprised of ARC CPU core, AHB matrix, WLAN device, HSDPA device, and peripherals like UART, I2C etc. ARC sub-system was verified using normal C (firmware programs) running on ARC CPU using ROM image. This C program performs (a) the initialization of memory, registers, and devices, (b) basic data read/write, and (c) dumping data values to memory to compare with actual values. Thus score boarding was done at the memory data level by ARC. This is called a “Soft-Scoreboard.” An OVM compliant AHB monitor is hooked with AHB RTL, while this ARC sub-system verification is performed. This ensures that if anything goes wrong on AHB bus, assertions will be triggered. All C programs ran on ARC reference models, and then the actual ARC sub-system and results were compared through a script in batch mode. Each RTL block on ARC-Soft-Scoreboard was verified using this mode. The top level scoreboard was linked with the scoreboard of the peripheral verification environment and Soft-Scoreboard derived from the ARC ROM.
OVM BASED FULL CHIP VERIFICATION ENVIRONMENT:

Register/Memory model

This model behaves almost the same way as the DUV. It will have the same address mapping, access, and update facility as the DUV. This model will be helpful in verifying the registers access and data validity. Along with register model, the memory model will be present which will have almost the same access as the DUV. There are a couple of extra methods available for back-door writing and reading.

UART Master

It is an interface OVM Verification Component (OVC) used to verify UART bus protocol.

BFM-Driver

It generates UART data with different configurations and error injection and detection. From UART basic item, different constraints will be given to inject correct and erroneous data/control. More than that, tasks and events are used to inject and get the data and address of the packet. Basic sequences are available by which different and complex kinds of scenarios are possible to implement. UART is the main component, as in the DUT. Only UART can operate in the debug mode while other modules are in the normal mode. So, UART can get all internal registers’ data and memory values.

Typical UART Environment code:

```plaintext
class ei_uart_env_c extends ovm_env;
  // Virtual Interface variable
  virtual interface ei_uart_if.driver_if driver_if;
  // Control properties
  protected int unsigned num_drivers = 1;
  // Check enable
  bit intf_checks_enable = 1;
  bit intf_coverage_enable = 1;
  // Components of the environment
  ei_uart_agent_c driver;

  // Provide implementations of virtual methods
  `ovm_component_utils_begin(ei_uart_env_c)
    `ovm_field_int(num_drivers, OVM_ALL_ON)
    `ovm_field_int(intf_checks_enable, OVM_ALL_ON)
    `ovm_field_int(intf_coverage_enable, OVM_ALL_ON)
  `ovm_component_utils_end

  function void build();
    string inst_name;
    super.build();
    $display("Inside env build");
    driver = ei_uart_agent_c::type_id::create("driver", this);
  endfunction : build

endclass : ei_uart_env_c
```

Bus Monitor

This monitor is used to monitor all master and slave behavior. It is also useful to check the data transfer from all slaves to the master. Here one instance of the slave and one instance of the master is used. The DUV behaves as a slave component.

RF/Audio

It is an interface task used to verify RF/Audio data. There are separate fields available for I and Q data. User interface signals/task/function is developed to inject and get data. Sample rate field is available to make sample rate configurable.
Monitor

Monitor will collect the signal line and emit the coverage on data reception. On this event, the data checker will compare the data with expected data. For checking, data collected on this event will be passed to local variables and then after collecting required amount of samples, data comparison is done.

Predictor

Predictor will get the HSDPA memory code, and it will behave the same way as the DUV. Action command block will issue some commands to read/write the required status, memory locations, and data from peripheral interfaces. It will also issue commands for data comparison and expected/actual data collection. From a control point of view, this is the most useful block.

Configuration

This system configuration unit is used to constrain top level parameters. You can also overwrite configurable parameters of interface OVCs in this unit.

VSD (Virtual Sequence Driver)

Virtual sequence is used to constrain I2C, I2S, parallel audio, and sequences. We can constrain virtual sequences in the test, and it will drive these sequences.

Reset and Clocks

Reset is an input to DUV. Reset is generated in SystemVerilog and driven to DUV and subsequent classes of the SystemVerilog environment. There is a “reset (unit: no_of_cycles)” function defined in top environment “ei_mk_wlan_clocks”. You can assert the reset for defined number of clock cycles.

Clock is also an input to DUV. Clocks are generated in SystemVerilog clock file and driven to both DUV and SystemVerilog environment’s subsequent modules. Other clocks, like DSP clock, AGC, FLL, VCO and other miscellaneous signals, are also controlled in this interface.

Stop mechanism

Stop mechanism is implemented using objections. Once reset is detected in top environment, “TEST_DONE” objection is raised. Once reset is de-asserted, one more objection is raised in the virtual sequence driver. Once the sequence is finished, one objection is dropped. Now there is a test_done flag. Once this flag is set, another objection is dropped. Once all the objections are dropped, task “all_objections_dropped ()” is called. This task starts one task “postpone_end_of_test ()” to drain the test for a number of system clock cycles. In this task, a new objection is raised, and it waits for a number of configuration cycles before it drops it again. Once all objections are dropped, the test is stopped.

Fig 6: OVM compliant SoC Verification Environment

CONCLUSION:

The Open Verification Methodology helps to build an environment in which different language/modules work together more effectively. i.e. interoperability in terms of multi-language interaction and reuse of existing infrastructure. With a unique class library support, layered test stimulus, and phase methods, porting SoC level verification environments for a simple user interface is a time-saving task. Utility classes, connect, and configure, in particular, allow generous reuse and concealment of OOP concepts below the user level. It is really architected for reuse and extensibility!
ABOUT THE AUTHORS:

Nilesh Ranpura, Project Manager-ASIC
10+ years of experience in VLSI Design Cycle, Methodology and Verification in the areas of Bus Interfaces and Networking devices. Participated on multiple projects for Multi-million gate count SoC verification architecture definition and complete product development.

Pratik Vasavda, Project Leader-ASIC
8+ years of experience in verification of ASIC/IP using HDL and HVL like Vera, SystemVerilog, e, and C++. Has done multiple projects in the area of Mobile TV, Storage Area Networking, Optical Networking. His key area includes development of Verification Environment using re-useable verification methodology.
ABSTRACT:
This paper covers the benefits of deploying an OVM-IDesignSpec based methodology for design and verification. It discusses the testbench automation provided by the OVM Register package. It shows the advantages of using the new libraries from Mentor in conjunction with a new tool called IDesignSpec from Agnisys.

This combination of new technologies enables design teams to realize cost savings, quality improvement and TTM benefits across the engineering organization.

THE PROBLEM:
SoC design is fraught with many challenges, the most important among them being IP integration and verification. The IP integration challenge is compounded due to the complexity of multi-million gate designs with a variety of programming modes. This is exacerbated when the various IP providers, be they in-house or external, provide inadequate programming guidelines and documentation. The growing trend to make the IPs more reusable and configurable adds to the complexity. At the hardware level, this programming configurability is achieved by an ever increasing number of programmable registers.

While designing such a device becomes a challenge, verification becomes an even more daunting task since there could be 100s if not 1000s of registers that require verification.

Another daunting challenge is managing changes through the design process. It is typically very troublesome. Without automation, design teams fear change so much that any deemed change has to go through a formal approval process which in turn causes numerous schedule slips.

Proper specification documentation isn’t just required for design but for verification, firmware, device driver, lab debug, diagnostics, application software and even technical publication. Keeping up-to-date documentation for the design is a non-trivial task. Yet it’s vital to have documentation that correctly describes the device behavior.

CAN GOOD DOCUMENTATION SOLVE THE PROBLEM?
Yes, but designers want to design and not document. Let’s face it, documentation is a chore no one really wants to do -- but someone has to. After all who wants to spend time creating a document that becomes outdated the moment coding starts!

CAN DESIGNERS BE FORCED TO PRODUCE GOOD DOCUMENTATION AND KEEP IT IN SYNC WITH THE CODE?
Yes, but the reason designers resent documentation is because of duplicated work. Why should they be asked to document and code the same information at the same time? Not to mention the added responsibility of managing the two separate information sources as changes take place through the design cycle.

THE SOLUTION
The solution is to have a single source for the functional or design specification along with the register information. Based on our experience in working in design and verification projects we found that its no fun creating registers by hand. It is a laborious, error prone process. We use IDesignSpec™ to capture the specification and...
the register information and generate all design and OVM based verification code from it. It is available as a plug-in for editors; it enables users to embed register information right inside the functional specification.

**DOCUMENT DRIVEN VERIFICATION METHODOLOGY**

IDesignSpec provides design teams a way to capture and extract register specifications within a design specification document. This methodology fits right into any system that the users may already have in place. Most IP providers already create IP-XACT based register information. IDesignSpec can read in IP-XACT and CSV files with Register data and generate SystemVerilog files that form input to the OVM based register verification environment.

The following figure shows what a specification with embedded registers may look like in IDesignSpec. Note that the text is free form and the user is able to add any formatting or graphics enabling free expression of ideas. The proximity of the specification to the register means there is no need for excessive cross referencing.

![Figure 2: Specification with embedded registers in IDesignSpec.](image)

Here is part of the OVM output generated by IDesignSpec

```verbatim
//*** This file is auto generated by IDesignSpec (http://www.IDesignSpec.com). Please do not edit this file. ***
// generated by : anupam
// IDesignSpec rev : 1.1.0

// block-name: memcontroller
typedef struct packed {
    bit [31:19] field1;
    bit [18:13] padding13;
    bit [12:7] field2;
    bit padding0;
} memcontroller_Controlreg_t;

typedef struct packed {
    bit [31:0] vctl;
} memcontroller_CTLR_ID_t;

class memcontroller_Controlreg extends ovm_register
    #(memcontroller_Controlreg_t);
covergroup c;
    field1 : coverpoint data.field1;
    field2 : coverpoint data.field2;
endgroup

function void sample();
    c.sample();
endfunction

function new(string name, ovm_component p);
    super.new(name, p);
    WMASK = 'b1111111111111000000;
endfunction
endclass
```

![Figure 3: Generated OVM code fragment.](image)

This methodology works to improve the productivity of the company at three levels. Rather than working to achieve a local optimal operational efficiency for one particular group, it ensures a complete product-team wide optima.

1. Engineers become more effective as they don't have to spend time working on mundane activity like creating and verifying registers manually.
2. Design and Verification Process becomes more effective as redundancy is removed from the system and change is managed elegantly.

3. The technologies that enables this methodology (OVM and IDesignSpec) are lightweight, easy to learn and make simple things easy and difficult things (like customized OVM output) possible.

DIGGING DEEPER

IDesignSpec generates so called “redef” files for the OVM register verification environment. Each block/IP in the specification document is transformed into a SystemVerilog Package. Each of the packages contains packed register structures, the register definitions and register instantiation. This can be used by the verification team to create register aware tests, drivers, monitors and scoreboards.

The OVM Register verification environment provides a lot of useful functionality right out of the box. Additionally, it is possible to extend the functionality.

• Register types
  - Normal registers are supported out-of-the-box, and other “quirky” registers are also supported out of the box – for example, an ID register, a coherent register and a model register are all modeled in the OVM Register kit. Additional customer specific behaviors can be easily created by simple extensions of the basic OVM Register API.

The following figure shows how a supervisory register, and model register can be specified in IDesignSpec:

- Register Tests
  - Write – read per register
  - Writes followed by all reads

Customized tests are easy to write – either from scratch or by extending and enhancing the existing functionality. The user has easy access to the underlying data model – the register maps, the register file and the registers themselves.

• Register Coverage
  - Register coverage is available automatically on read and write, and is customizable by the user for other interesting functional relationships.

• The OVM Register package can be used to describe various register behaviors and access policies, including:
  - Read Writable (R/W)
  - Read Only (RO)
  - Write Only (WO)
  - Read-to-Clear (CR), Read-to-Set(RS)
  - Write-1-to-Clear(RW1C), Write-1-to-Set(RW1S)

In IDesignSpec, this is simply selectable on a per-field basis.

• Hierarchical Register specification
  - Blocks
  - Register files
  - Multidimensional register structures

IDesignSpec’s Chip and Block structures enable users to create a hierarchy in a linear document. RegGroup adds a dimension to the registers and one can specify a RegGroup inside another ad-infinitum. This is basically a shortcut for specifying arrays which translates into multi-dimensional register structures which are great for Video and Image processing filter applications.

The following figure on the next page shows how a register bank with 512 copies of registers is created.
• User customizations
  - The generated “regdef” file is typically not edited manually, as doing so would violate the single-source principle, which in turn will cause the user to manually edit every time changes are made to the specification document.

IDesignSpec provides mechanisms by which users’ customizations are made to the generate code. For example, users can specify that an OVM register should be extended from a class other than the default “ovm_register” class. Despite this, if absolutely required, the “regdef” file is a readable file that can be edited and extended as needed.

**BENEFITS**

Using this tool set has enabled us to:

- **Specify Registers within functional specification**
  Keeping the register specification within the functional or design specification is very important. Changes to the functional specification usually means changes to the registers and vice versa. Having a separate tool for register management would simply exasperate the problem as cross references would need to be manually maintained and the two would constantly need to be kept in sync.

- **Dismantle information silos**
  Designers and architects are the producers of Register information, but the entire engineering team is the consumer. Starting out with a spec that is shared by the entire team is a good start, adding register information to the spec is better as the two go together and there is less need for cross referencing. Keeping a consistent methodology helps not just one product team but is fundamental to design and spec reuse.

  - **Achieve global process optimization minima rather than local group minima**
    The pressure for meeting deadlines is such that in many cases each engineering team could focus on their deliverables and not think about the overall productivity improvements that could be had by having a common system in place that produces an overall advantage at the cost of one team having to go the extra mile.

  - **Adapt to change rather than fight it**
    Change is inevitable in digital design. Rather than coming up with stringent regulations for avoiding it, the system should be able to accommodate it without excessive re-work. For example, if a register location changes, or its bit fields etc changes, it should be possible to recreate the tests and run them without any issue.

  - **Create a single information source**
    A single source of information eliminates the need for duplication and synchronization and saves time and resources while improving quality.

**SUMMARY**

This paper has shown why it’s important to have a consistent register methodology for getting significant ROI. It has also shown what the new additions to the latest OVM register packages are, and how to use OVM and IDesignSpec to streamline the register specification and testbench generation process.

**ABOUT THE AUTHOR:**

Anupam Bakshi is Founder and CEO at Agnisys Inc. He can be reached at ab@agnisys.us
INTRODUCTION

The increasing size and complexity of today’s Systems-on-Chip is driving the adoption of modular, IP-centric design and verification flows. Increasingly complex IPs, sub-systems and systems require high levels of modularization, standardization and re-use and must be comprehensively verified as quickly and efficiently as possible. Advanced verification methodologies such as OVM provide important verification capabilities, such as coverage-driven verification and reusable verification components, that leverage the benefits of object-oriented development. In order to quickly access these verification methodologies and apply them to the task at hand, it is important to automate the creation of as much of the verification infrastructure as possible. This paper presents a methodology for auto-generating much of the OVM verification environment for an IP using an interface-based executable specification, combined with a flexible generator framework. This methodology provides IP verification engineers with immediate access to advanced verification capabilities.

HOLISTIC IP VERIFICATION FLOW

Today’s IP verification teams face ever-increasing IP complexity and configurability. SoC platforms with software-centric flows require extensive HW/SW interface verification. Aggressive schedules demand more efficient verification, while QA mandates more quantitative and predictive metrics. Traditional HDL verification flows are not scaling, leading to poor productivity, inefficiency and unpredictability.

New methodologies are emerging to address this verification crisis. Verification-friendly languages such as SystemVerilog allow a wide degree of flexibility regarding verification stimulus generation, response monitoring and coverage measurement. Emerging best-in-class verification procedures are being consolidated into standardized, open methodologies such as the Open Verification Methodology (OVM). OVM promotes Open Verification Components (OVCs), which provide high-level verification programming interfaces for standard protocols such as AHB and OCP. OVM also supports HW/SW interface verification by standardizing the definition of IP registers and bitfields in an OVM register package. This provides a mechanism for rapid and efficient verification of the HW/SW interface. OVM also has strong EDA vendor support and products such as Mentor’s Questa verification platform provide high-performance verification and debug environments.

Using advanced methodologies such as OVM is key to addressing the current verification crisis. However, verification teams can face some challenges when attempting to leverage the full power of OVM. Many verification engineers are unfamiliar with the SystemVerilog language and object-oriented methodologies. While many facets of an OVM environment can be reused, each IP interface may require different hook-up and configurations. It may also be difficult to maintain alignment between the design and verification views, especially regarding the HW/SW interface where registers and bitfields may be added to the design but not to the verification view, leading to synchronization bugs.

Automated generation of an OVM environment from the source IP metadata enables instant access to the many benefits of OVM as well as ensuring alignment between design and verification views. Duolog Technologies has developed an ‘interface-based’ OVM automation flow using their Socrates integration platform. Socrates is a chip integration platform that supports IP to full chip-level integration across multiple disciplines. Socrates hosts three primary tools to address the different application domains:

• Socrates-Weaver: IP interface packaging and system assembly
• Socrates-Bitwise: HW/SW interface (registers, memories, memory-maps)
• Socrates-Spinner: I/O layer auto-generation

Socrates employs a three-step methodology, Capture, Validate & Generate, to deliver correct-by-construction views of IPs and systems. In the following case study, Weaver is used to package an IP’s interfaces while Bitwise is used to capture its registers and bitfields. Once the captured specification has been validated, the tools can generate the OVM infrastructure for the IP. Through a combination of Duolog’s Socrates integration platform and Mentor’s Questa verification platform, verification teams can gain immediate access to all the benefits of SystemVerilog and OVM without having to spend valuable time hand-coding the verification infrastructure.
CASE STUDY: USING SOCRATES TO AUTO-GENERATE THE OVM ENVIRONMENT

This case study details the automated creation of an OVM environment for a legacy IP module - an LCD controller block (CLCDC). The required infrastructure is illustrated in Figure 1:

The LCD Controller has the following interfaces:

- AHB slave interface for register and internal memory access
- AHB master interface for the external screen memory accesses
- IRQ signals that define standard and IP-specific IRQ interfaces
- A system interface that groups IP reset and clock signals
- External LCD interface

In an OVM environment each of these interfaces is connected to an OVC, which includes a driver, an optional monitor and a sequencer. These OVCs are controlled by the top-level virtual sequencer, which extracts test cases from the test case library. The LCD controller has just 14 base registers. However, when arrayed registers are expanded, there are over 530 actual registers. All of these must be defined in an OVM compliant register verification environment using the OVM register package. Finally, scoreboard and functional coverage management are also provided.

To achieve the maximum level of automation of this environment, it is necessary to analyze the specific aspects of the LCD controller IP that can be used to guide the automation. Firstly, there are some generic interfaces on the IP such as the AHB master and slave, which could use off-the-shelf OVC components. IP-specific interfaces, such as the external LCD interface, will require new OVCs to be created. The software-accessible registers for the IP are defined in a specification document.

The presented solution employs a methodology where the user defines high-level IP metadata which is used as the basis for auto-generating much of the OVM infrastructure. This metadata consists of design information such as interfaces, ports and registers, and also constraints that will guide the automation process. The main elements of the flow are illustrated in Figure 2 below:

Formal packaging of the IP’s interface and register information in Socrates enables the automation of the OVM infrastructure. The Weaver tool, which is part of the Socrates platform, can import port information from a VHDL, Verilog or IP-XACT file. Using a GUI, the user can quickly aggregate these ports to create interfaces. These interfaces can be further annotated to define what OVC is required, whether they require OVM drivers and monitors, or any other verification-specific information. Once captured in Weaver, this information can be used to generate the SystemVerilog wrapper for the IP and to auto-generate the main OVM infrastructure for the OVC as well as the hook-up to the wrapper. For instance, in the LCD case, the LCD OVC infrastructure can be almost completely automated. For hooking up to pre-existing OVCs, there may be naming mismatches between the ports used in the OVC driver and the ports on the DUT. It is also possible within Weaver to define this port mapping.

Figure 1: LCD Controller OVM verification environment, running on Mentor Questa
The Socrates Bitwise tool is used to manage the full HW/SW interface for an IP or system, including registers, bitfields, memories and memory maps. HW/SW interface definitions can be imported from IP-XACT but for the LCD controller they are captured manually. Bitwise validates the integrity of the captured or imported information and auto-generates various views including documentation, RTL code, C code and, in this case, SystemVerilog HVL.

As with all Socrates tools, Bitwise generators are template-based and new generators can be quickly and easily added. While all standard register and bitfield attributes are supported in Bitwise, the tool also supports extensive customization so that special access types and user-defined attributes can be added to extend the basic functionality. Bitwise also explicitly handles ID Registers and modal behavior as outlined in the OVM 1.0 Register Package.

Figure 4 shows the LCD controller example in the Bitwise user interface. In this example, register attributes are added to assist with OVM generation, including clocking, coverage and backdoor access information. The attribute editor is shown where all predefined attributes can be set by the user. The user can extend this to handle other attributes if needed. Once all verification attributes are defined, Bitwise generates the SystemVerilog register and memory map files, and the shadow register class definition and instantiation, based on the standard OVM register extension package.
The Socrates OVM flow produces three levels of auto-generated components:

- **Fully automated:**
  - SystemVerilog DUT wrapper generation including aggregation of ports to interfaces
  - Register package based on IP register definition
  - Standard register test sequences

- **Semi automated:**
  - Vendor-specific interface component infrastructure
  - Generic parts of the test case libraries
  - Compilation and simulation scripts based on the internal OVM class and package dependencies

- **Placeholder:**
  - OVM class definitions are provided as well as the generic part of the SystemVerilog code that the user should extend and customize for the given IP functionality
  - All verification components where the content is defined by the functional behavior of the IP

The auto-generation results in the creation of thousands of lines of both syntactically and semantically correct SystemVerilog code that is OVM-compliant. It is clearly more advantageous than a tedious and error-prone manual approach. It is also possible to auto-generate a suite of register and memory map related standard test sequences, as well as documentation and block diagrams detailing the structure of the verification environment. All generation is controlled by Socrates’ customizable generator templates, which allow for easy environment extension and maintenance. If required, the auto-generation of each specific part of the environment can be modified or switched to a simple placeholder generation that the user can complete with hand written code.

**CONCLUSIONS**

While it is certainly desirable to generate the register package information for an OVM environment, this is only one part of the overall infrastructure. Recognizing the registers as simply another DUT interface and implementing a full interface-based automation provides the best possible mechanism for automated OVM environment generation. This liberates valuable engineering resources and allows verification engineers to concentrate on writing verification application code rather than assembling and debugging the verification infrastructure. Socrates provides an efficient capture mechanism for interfaces, ports, registers, bitfields and memories and supports the extension of the specification to capture verification-oriented attributes and constraints. Socrates also ensures fully synchronized views between all design, verification and software development teams at IP and chip-level. As more and more IP vendors publish IP-XACT descriptions of their IP interfaces, it will be possible to enhance the automation even more as ports, interfaces and registers can be directly imported into the Socrates environment. With a combination of the Socrates and Questa platforms, users can immediately reap the benefits that OVM can bring and make the most of valuable verification resources. Let the real work begin!

**REFERENCES**

[3] ovm_register-1.0 Beta 7 March 26, 2009
One of the defining moments in my career could easily have been one of my last. I had developed an ASIC that was an interface between a low cost microcontroller and a complex daisy wheel printer mechanism. The ASIC provided a memory mapped register based software interface that had been carefully specified by me, the printer manufacturer and our software engineer. The whole project had been done on a crash timescale; the software had been developed concurrently with both the ASIC hardware and the mechanism. On the day the risk wafer ASIC samples arrived, we had a factory in Asia full of printers ready for the electronics and we needed to ship units to meet customer orders. Meanwhile we were in the UK with a prototype board bringing the software, the electronic hardware and the mechanism together for the first time. We had a few hours to check that everything worked before the Asian factory started work. Within a couple of hours it became evident that we had a problem.

As we started to investigate we found a number of differences in the bit map between the hardware and the header file that the software was compiled against. We got that sorted with some basic read-write testing, then as the software started to run we found a hardware-software-mechanical interaction that stubbornly refused to work. We solved that problem by correcting the interrupt table so that microcontroller understood the vectors being returned by the hardware, the mechanism started to do something. Then came the killer, the print hammer solenoid was firing at the wrong time causing the mechanism to chew up daisy wheels. This time the problem was due to a subtle combination of bugs in the software, the ASIC hardware and some undocumented microcontroller behaviour. Fortunately, we were able to work around these in the software.

By noon, Asia time, we had the first working release of the firmware which enabled the factory to start rolling.

The root cause of these problems was not the absence of a hardware software interface specification, nor a lack of teamwork. It was simply that our manual checks of the hardware software interface had let us down. We had got away with it with this project, but we knew that it had been a near miss, and this started me down the road of finding better ways to do things.

---

1 "Owzthat!?" is a call that comes from the sport of Cricket and is made by the fielding side to the umpire when there is an element of doubt about whether a batsman from the batting side is out or not. The call is not considered necessary if the dismissal is obvious since a gentleman knows when he is out and will do the honourable thing.
My experience will sound very familiar to anyone who has had to get new hardware and software to work, virtually every customer I have worked with since has encountered the same combination of issues. Although leading edge at the time, my ASIC design was tiny by the standards of today. It had about 20 registers and we had just got away with 19 of them working as expected. Most customers that I have worked with recently are integrating blocks of IP which might contain several hundred registers into a SoC with a combined register count into the tens of thousands, with variable address maps caused by using multiple CPUs, bus bridges and bus fabrics. It should be obvious from my pre-ramble that a holistic, scalable solution is needed to verify that both the hardware and the software view of the register interface is consistent and that ideally other system level interfaces should be taken into account.

There are several fundamental things that you need to be able to do to validate a hardware-software interface and to get the whole development team involved in making sure that it works before silicon gets cast. The most fundamental is that everyone in the team needs to be working from the same reference point, and the most usual form is either a document or a spreadsheet that specifies the register interface. Once the document is in place you need to be able to auto-generate the views that the various team members are going to use – the software header files, the register verification models and, in some simple cases, the actual register hardware. However, there will be more than one document, there will be a collection of documents, at least one for each sub-component and another specifying the mapping of the base addresses, and these could come in a variety of formats. This implies that the auto-generation mechanism has to support multiple input formats and has to be able to build aggregated, hierarchical, address maps. Depending on the complexity of the system, the address map will also have different view points based on which processor is using which access path.

You need to be able to regenerate the downstream views quickly when changes occur either because a hardware component has been added or removed from the design or an address map change has occurred. This is important to keep everyone in the team up to date with changes and configuration control should be built into the pre-regression checks to make sure that all the generated views are correctly aligned.

Two other vital components that need to be in place are a verification model of the register hardware and a means of verifying the hardware software interface. Ideally, these should be linked.

The hardware register model needs to be able to model register behaviour hierarchically from the bit level up to the system level address map. Registers should have individual bit fields mapped with attributes denoting read, write, and other specialised behaviour. Each register should have an offset address. Each set of registers belonging to an IP block should be relocateable. Each system level view of the register map should be relocateable. Over the years, many register models have been developed, some recent examples include the vr_ad register package and the open-source OVM register package.

Although useful, the register model may not be able to handle everything you need out of the box. Therefore, you need to ensure that the model format is extensible to allow modelling of non-trivial register behaviour or to model linkage between register bits. Another consideration is whether to snoop the behaviour of the hardware by some kind of backdoor method or to rely on front door accesses to keep track of the hardware state.

It is important to realise that the register model is only for the hardware-software interface. The hardware behind the register interface is usually much more complex and its behaviour may be difficult to predict. It may be beneficial to link a behavioural model up to the hardware side of the register model since it potentially allows the software team to start checking their low level software before hardware is available, but it comes at the cost of implementing the model and hooking it up.
The hardware register model can be used in several ways. One approach is to use it as a way to configure the design into a constrained random initial state before starting a test case. Another is to use it as a scoreboard to keep track of expected register states, but this use model can start to break down if the hardware behaviour is not predictable. Another is to use the model as an indirection mechanism to make register accesses relocateable and therefore reusable between block level and system level. In other words using the register model to return addresses and bit locations by name rather than by using fixed values. Often, the actual register use model is some combination of these methods.

With a hardware register model you should be able to automate some simple test bench driven hardware tests to check that the SoC interconnect is working correctly. Examples include reset tests and bit level read-write tests which allow most mapping and wiring errors to be detected. These tests work by looping through an array of registers and using the addresses and expected register bit states to check actual behaviour. Using this approach with hardware simulation can be done without having to run software on a CPU model, which frees up the software part of the verification team to work on more complex test cases.

The 5 basic rules of the Hardware-Software team game

1. Have a common reference for all members of the team
2. Be able to generate the views required by all the players, and be able to regenerate them quickly when something in the reference changes
3. Make sure that the view generation can create an aggregated, hierarchical, address map
4. Use a hardware register model that:
   - Models register field behaviour
   - Supports indirection
   - Supports hierarchical addressing
   - Supports back-door accesses
   - Is extensible to model complex behaviour
5. Hardware and Software functional coverage should be inter-dependent

The other side of the hardware software interface is the software and there are two basic techniques for verifying the interface from the software perspective. Either the software is compiled and executed on a model of the target processor, or the software is executed on a bus functional model which generates bus transactions. The trade offs are that the processor model is accurate but it will have performance limitations, whereas the bus transactor model will have higher performance but at the expense of accuracy. Since most blocks are implemented with on-chip bus interfaces, using a bus transactor VIP, such as an MVC, with built in abilities to exercise all the corner cases of the bus-interface is arguably the best choice for block level verification. System level co-simulation may initially be faster using bus transactors, but inevitably there comes a time when a CPU model has to be used in order to validate that the CPU behaviour has been correctly anticipated.

The software side of the interface is compiled using the header files generated from the reference documentation. The low-level interfaces can be validated against the hardware at an early stage in the block level development process and interactions with the real hardware can be explored, possibly even allowing the hardware and the software interface to be optimised. Depending on the nature of the block, the low level software may form part of the verification code and may be reusable at higher levels of integration. The ultimate goal is to reuse block level software on the real hardware accelerating bring up.

Getting hardware and software teams to work together can be something of a challenge. Using a single point of reference cuts down the misunderstandings, but it can also mean that the two teams are able to go their separate ways and ignore each other. Another strategy that needs to be put into place is that of the interdependent verification plan. Simply put, the hardware team needs to rely on the software team for part of its verification coverage and the software team needs to rely on the hardware team for part of its verification coverage.

With the advent of the SystemVerilog DPI, it has become easier for software teams to use a hardware register model based scoreboard resource to check that register reads were correct and to use hardware verification coverage metrics to understand under which hardware conditions a particular software routine was run. For instance, if an interrupt handler needs to be exhaustively tested then it needs to be proven that the hardware interrupt controller generated all types of possible interrupts in all combinations of priority and that the interrupt latency was within bounds. In a simulation, this needs to be tested using coverage groups and assertions along with pseudo random interrupt stimulus generation in the verification environment and then the interrupt handler needs to be run until a cross of all possible software paths has been taken for each of the appropriate hardware conditions.
The mirror of this is that the hardware verification needs to ensure that all of the applicable low level routines have been seen to run in combination with all of the appropriate hardware event scenarios. This requires the software to be compiled with instrumentation that allows it to flag to hardware simulation functional coverage code that a particular function call is in progress. The hardware functional coverage loops then take the software state into account when sampling covergroups and assertions.

The final step in the process is to get the software to request specific hardware scenarios during software orientated test cases or for the hardware to request specific software handlers are run when an interesting hardware event has been orchestrated. This level of synchronisation can focus the verification generation process onto the important areas rather than running many hours of simulation in the hope that something interesting shows up.

In conclusion, I am thankful for the developments in verification technology and methodology that have taken place since that stressful day with the Daisy Wheel printer. Creating a verification environment where a team that combines software, hardware and system disciplines can interact and fix issues earlier in the development process has become a mainstream activity. However, there is no room for complacency; increasing design complexity will always drive more innovation in verification techniques. In the final analysis, the thing that really makes the difference is to get the development team pulling together, covering all the angles to make sure that no one gets caught out and the role of the verification environment is to facilitate that teamwork.

1 “Owzthat!?" is a call that comes from the sport of Cricket and is made by the fielding side to the umpire when there is an element of doubt about whether a batsman from the batting side is out or not. The call is not considered necessary if the dismissal is obvious since a gentleman knows when he is out and will do the honourable thing.