Using UVM

The Condensed Guide For Designers, Debuggers, Test-Writers And Other Skeptics

Gordon Allan
Mentor Graphics Corporation
Fremont, California, USA
gordon_allan@mentor.com

***Abstract*—**

This paper and presentation is for the users of a UVM testbench, rather than its architects and coders. It is intended for RTL design teams who might choose to use a UVM testbench provided by their verification team counterparts to test the code they write, and engineers involved in minor testcase coding and regression triage/debug. If you fall into this category, you typically interact with an existing or newly-developed UVM testbench, using it as a tool for debug, or validation of incremental design features, or 'what-if' testing.

Maybe you are skeptical of the costs/benefits of adopting UVM, or worried about the process of change and how it might affect the normal design, testing, debug flow you are used to. We'll equip you with some knowledge, some tools, and a little armor that may come in useful later in your chip project. You may be about to embark on a UVM-supported development for the first time, or trying to improve teamwork between design and verification personnel. UVM should help, rather than get in the way. It is a complex machine, but you should have the expectation of a simple user interface and feature set that works for you, without requiring the same deep OOP knowledge that the testbench architects may need. We provide you with the key knowledge to ease adoption, and gain an understanding of:

- **The UVM black box -**
  learning to love your UVM-based testbench
- **What UVM brings -**
  a pragmatic approach to UVM adoption
- **Testing with UVM -**
  a stimulus package for those hidden DUT bugs
- **Configuring UVM -**
  making the machine work for you, not against you
- **Debugging with UVM -**
  a structured approach to the assignment of blame
- **Speaking UVM's language -**
  how to train your verification team
- **UVM resources to help you -**
  where to go for assistance and therapy

The material is entirely technical content, with pragmatic descriptions of processes, problems and solutions; no tool marketing content is included. The paper and accompanying presentation includes a good in-depth look at the kind of SystemVerilog code you will typically encounter, wrapped around your DUT, and how to go about debugging the hookup of signals, config, and protocol-specific verification components.

Keywords—UVM, Universal Verification Methodology, design, verification, testing, test-writing, debugging, teamwork, regression, triage, concurrent engineering

I. INTRODUCTION - USING UVM

Historically the engineering functions of design and verification were indistinguishable. Design architects and engineers verified their own work as they created it, and co-verified the designs of their peers as they integrated their work. To achieve higher quality through independent scrutiny they may have simulated each others designs.

The consolidation and advancement of Verification Methodology into various specialized languages, testbench architectures, skill sets, and team organizations has both benefits and disadvantages. The approach is scalable, fits today's requirements for design reuse and integration, and enables contributors to excel through specialization. However, it also separates design and verification, team-wise and project-plan-wise, and this separation breeds new problems requiring new solutions.

If your verification team has decided to use UVM (Universal Verification Methodology) on your next project, what does that mean to you as designers and test writers? We aim to help in this paper by providing some useful preparation and an approach to working with the UVM testbench that might work for your team situation.

We are not going to teach you 'easier UVM' here - the premise of this paper is that the UVM testbench can be developed by UVM experts as a software product, and have usability as a black box by those who do not have, and do not need, any UVM expertise, beyond understanding of the API provided to them which uses domain-specific and protocol-specific terminology.

We focus here on **Usability**. Our goal is to provide enough knowledge for you to be dangerous, informed and productive.

Of course the real learning phase for you comes when you use the actual UVM testbench on your DUT, within your area of familiarity; when you need to debug that failing test, or find out why the testbench is broken. Then you can revisit this topic material and apply it to get the most out of your use of the tool, and get the most out of your verification team counterparts.

This is as much about peopleware as code and APIs - we encourage you to structure your design and verification activities in parallel rather than throw the design 'over the wall'.

Verification teams beware: treating your design team to this valuable information may increase their expectations!

II. THE UVM AS A BLACK BOX - LEARNING TO LOVE YOUR UVM-BASED TESTBENCH

We describe here an approach to making the design team - verification team relationship work; discussing how UVM testbenches, components, interfaces can be designed for reuse, and used in a black-box manner without requiring internal knowledge.

A. You are not the only customer - avoid irrelevant detail

While being involved in the specification and project planning is a good thing, when considering the interface shared between design and verification, the detail on either one side is irrelevant to the other.

Define and specify a good 'API' between teams just as you would for communicating design blocks. Decouple false dependencies and strengthen real dependencies. Keep those meetings and documents separate from those which go into detail of architecture or implementation or otherwise require special knowledge. This works both ways.

B. Connections between testbench and DUT

All signal communication between testbench and DUT should be done via SystemVerilog interfaces. These are named groups of signal ports that can save a lot of typing. They can also contain additional logic for smart interaction with the testbench.

1) Ownership of the SV interfaces

These belong to the verification team, not the design. If the DUT already uses SV interfaces, then the verification team will in most cases need to duplicate them. Typical reasons are:

(a) Off-the-shelf or reused testbench components may have different signal names or groups or naming conventions

(b) There may be assertions in the interface written by the verification team and used elsewhere in the testbench or crucial to the coverage gathering / test plan hookup elsewhere.

(c) There may be procedures in the interface to support the testbench's interactions with signals, e.g. synchronization.

This should not be treated as a problem, there are conflicting needs that need not be resolved - that's what an 'interface' means.

2) Clarity in the interface definition and hookup

The testbench's interfaces should be comprehensible by all. You should know where they live; they should be in identifiable files in a known location, which may by necessity vary for each protocol type on your DUT/testbench.

Their member signal names may differ slightly from the DUT port names due to the reasons above, but still be clear.

Merging unrelated signals together into one interface is not good. A good abstraction is to have one SV interface per DUT interface / protocol signal group / related functions. Not all signals together, or all inputs / outputs together. If necessary, have a 'misc' interface for the few remaining config/status pins and have the verification team hook them up to a model.

C. The DUT is a black box too.

The black box abstraction works both ways and benefits both teams - the verification want to treat your DUT as a black box too. Here is an example of how that can be implemented:

The DUT should be instantiated in a single wrapper SV module, containing the interface connections we described.

This is also the place to code any static or reactive stimulus i.e. inputs which are never affected by the testbench, they are either constants, or autonomous functions of other I/O signals.

If the DUT consists of several toplevel modules or external standalone models, they should be integrated here, but any complexity in either DUT integration or model configuration (e.g. an entire array of configurable memory model instances) should be wrapped in a sub-module and pushed down a level and/or moved to a separate file.

D. Whitebox testing: Internal signal hookup / forcing

When white box testing is used, the wrapper should instantiate an SV interface for those internal signals to be monitored/forced, and do the hookup by downwards path assignment. In the case of VHDL DUTs, the assignment can be done using SignalSpy or an equivalent monitor/force hookup API.

The signals can then be treated as any other port by the testbench. Whether black-box ports or white-box internal test points, it is important to confine all the hookup to this one file, and not allow other bits of testbench to have downward path references that do not go via this wrapper. This will avoid surprises during debug.

E. VHDL DUTs

If the DUT is VHDL and has ports which use VHDL 'record' types, it must be wrapped twice - one inner VHDL wrapper to instantiate the dut and expand all VHDL records on DUT in/out ports to individual VHDL wires/buses, and then an outer SV layer which instantiates the VHDL wrapper and reconvenes signal groups into SV interfaces, as per the Verilog DUT wrapper. Any VHDL-specific needs can be taken care of in the inner wrapper - most UVM testbench developers will be less familiar with VHDL syntax.

F. Models in the black box

We discuss the UVM register model several times in this paper - this is one of the most useful features of UVM and will be exposed to you in the testcase API syntax. It is an easy way to interact with register values. As a black box strategy, it is important for the verification team to completely hide the creation and construction of the register model so that it is 'just there' for you to use, and documented sufficiently that you know exactly how to write or read registers and fields.

Behind the scenes, the verification team has to construct that model from the DUT specification, normally via some intermediate syntax such as IP-XACT or a spreadsheet. They should ensure that when the specification changes, the model is brought up to date. This flow can optionally generate the RTL for your registers as well as firmware headers and other views.

III. WHAT UVM BRINGS - A PRAGMATIC APPROACH TO UVM ADOPTION

We aim to provide an introduction to UVM features, terminology, architecture, and some common practice, pitfalls and solutions for this reboot of your verification flow - a taste of what UVM adoption will bring; the pains and the gains.

Overall we want to convey a pragmatic approach, elements of a flow from specification to development to debugging. This first section specifies some essential usability requirements.

A. *UVM provides a testbench you can use as a design tool*

Design and testing often proceed concurrently, or more than likely design-first test-later, with some overlap. Both those activities need a testbench:

- The verification team needs one to implement a regression that can find bugs, and to achieve closure of their coverage goals by exhaustively testing the design specification features and corner cases in their test plan
- The designer needs one to continue RTL development beyond initial bringup, to debug failing/broken tests, and to lessen risk when making design tradeoffs and refinements late in the flow, in order to achieve closure on features, timing, area, power, and testability goals.

It makes no sense to have two testbenches, one is enough.

B. *Specify your requirements as a testbench customer*

Features specific to the needs of the designer are often bolted on as an afterthought; it is better to specify APIs and boundaries up front. The verification team will have their own adoption / migration plan, important for you to understand that - know what you'll have early and what you won't have until later, and have some input into those requirements.

USABILITY REQUIREMENTS: involve the design team in the testbench specification process whenever the user interface is being discussed, defined, explained or reviewed. They are users too!

We define here a list of essential UVM testbench features to aid a designer's occasional or regular usage of the testbench, for continued refinement of the design, or debugging, or minor testcase creation or adaptation.

UVM supports all of the requirements listed here, and although it supports many different ways to solve the same problems, it is possible for the verification team to select solutions that address these requirements. They are often overlooked while lost in the detail, but are not rocket science.

USABILITY REQUIREMENT #1: a well-defined signal interface between DUT and Testbench. There should be a single SV file (e.g. top.sv or dut.sv) with a single module definition that defines everything about the DUT from the testbench's point of view. It should contain all interface hookup to DUT pins, including any internal test/force/backdoor connections made by downward paths.

UVM encourages an interface agent oriented approach, using relevant signal groups in interfaces, as a way of dividing and conquering the verification problem. Make this work for you by insisting on a clear, transparent interface layer.

This requirement also means that strictly no signal control or monitoring should occur in the testbench without going via this layer. There should be no 'magic' connectivity that occurs from deep within the testbench, no matter how convenient that may be to the authors, it is hard to comprehend what's going on while debugging a problem when hidden connections for cause and effect is employed. This includes 'force' or other back-door interfaces. Redirect them via this clear, observable layer.

USABILITY REQUIREMENT #2: locate all config parameter settings together in one file for easy reference and modification. This 'dashboard' file should include simple, clear documentation, explaining each group of config types, how they are set and where they are used, and documenting each setting, their defaults, their set of available values, whether they are randomized and constrained, where they are used, or point to specific additional files for detail.

A testbench typically has some settings to control how it behaves or to configure what kind of DUT topology is of interest. These settings come in several different forms, hard and soft, static and dynamic, described in more detail later in this paper: defines, parameters, testbench configuration, DUT config pin values, and DUT config register contents. Different types of configuration are set and used in different ways, sprinkled around the testbench code. We will cover dynamic config (e.g. registers) later but static config is important too.

USABILITY REQUIREMENT #3: a simple representation of the DUT registers and other dynamic config / state. This should be accessible to testcase writers for ease of setup of the DUT internals prior to injecting traffic stimulus, and for ease of determining or synchronizing to DUT internal state exposed via register bits.

One benefit of the UVM is the provision of a register model. This provides a simple programmatic representation of the DUT's programming model. It does a number of things. Firstly, it allows a testcase to do writes and reads in a symbolic manner, comparative to using modular C code and data structures rather than low level hex addresses. Registers once updated can be written to the DUT by the testbench driving the relevant DUT bus.

Secondly, it keeps a mirror/shadow copy of what are the current values in the DUT, for reference by any part of the testbench. It can keep that up to date by monitoring DUT buses. It is an example of good abstraction. It is also possible to use internal 'backdoor' signals instead of bus reads/writes for the updates and observed values; this method is not without problems and should only be used if necessary.

USABILITY REQUIREMENT #4: an easy protocol-centric API for the creation of stimulus. Bring together in a single file a boilerplate of all the elements of specification of how to set up, what to do, what to test, simple sequential do-this-then-do-that and parallel do-this-simultaneously-with-that criteria.

UVM has SV class-based structures called sequences which contain the data for one or more stimulus transactions to be sent via UVM drivers to one or more particular DUT interface

port. They are protocol- or domain-specific and represent all the interesting chunks of stimulus you may want. They are hierarchical so you can build up complex sets of stimulus.

As a designer you have two requirements here: one is some input into the set of useful stimulus sequences required to exercise interesting corner case scenarios. Provide that input via the test plan / verification plan and associated brainstorm/spec process.

Your other input is to specify an easy framework for simple testcase comprehension, development, and modification. There are many different ways for the verification team to implement complex random stimulus requirements, but only a few of those architectures are comprehensible by non-experts. Sometimes the simplest of directed testcases is hard to obtain. By focusing on a simple solution will likely arrive at the best solution for both verification and design needs; the clarity and ease of use benefit all. This is a test-writers' API [1].

> USABILITY REQUIREMENT #5: a set of debug controls and helpful advice accessible from one location. Settings that can be made in SV code, or on UVM command line, or useful simulator-specific debug features, are all useful during debug. As with configuration, bring those together in one file to provide a one-stop diagnostic panel. Where not possible to provide actual settings, provide comments with advice on where to look and what to do.

A UVM testbench provides debug capability to: control how much information is output to a log file - this is called verbosity. For regression one normally wants less verbose output but for debug, the most verbose output is often useful. Verbosity can be set on specific portions of the testbench, i.e. to report what is observed on specific portions of the DUT.

Verbosity allows you to see exactly what transaction and configuration was involved in the test you are debugging, with a view to isolating the problem or narrowing down the testcase.

UVM also provides hooks to transaction recording which works in conjunction with the features of your simulator to display what is going on inside your testbench. These features are often switched off by default for highest regression performance, but again they should be easily controllable on your debug front panel, at your fingertips when required.

> USABILITY REQUIREMENT #6: a well-defined waveform recording configuration. To match the well-defined SV interface connections between DUT and TB, provide a shared waveform setup to accelerate debug productivity. Maintain it to match the evolving signal interfaces. Add protocol-specific interpretations. Provide visibility of key testbench activity and settings.

In addition to the debug control knobs above, debugging involves staring at waveforms and making sense of what is going on. Often there is duplication as multiple team members build their own waveform profiles, and often there is delay when debugging a problem as the waveforms are manually populated. A shared waveform setup can provide an easy window into the activity of a UVM testbench.

Provide also the waveform setup to get the most out of the transaction recording features, to debug some internal paths in the testbench. This lets you trace transactions visually from input to output on the DUT, and from driver to monitor to checker to coverage, in the testbench.

> USABILITY REQUIREMENT #7: zero UVM or OOP knowledge required to operate the testbench as a debug tool, for exploration of the design, or for minor testcase modification or development.

The testbench internals and functions should be hidden from the normal user, wrapped in simple APIs and sensible file organization, so that designers can run the testbench, control it, debug with it, do some testcase creation or modification, all without having to learn OOP, or class-based SV, or UVM.

Of course some basic Verilog knowledge is required: how to reference scopes (this dot that) and call tasks, passing parameters, or functions returning values, and how to do conditionals, loops, forks. Some design teams may use VHDL not Verilog for RTL coding, so some learning is required in those cases. But it should be possible to interact with the testbench as a user, without knowing the details, without having to learn all about inheritance, factories, constructors, polymorphism, etc. Use documented templates if necessary.

C. Use concurrent, phased development on both sides

Not all requirements for UVM testbench usability are features, some refer to the development flow or project planning aspects. The peopleware is important too [4].

> USABILITY REQUIREMENT #8: develop both the design and testbench in phases and deliver working releases incrementally, they can evolve together and both can be usable along the way.

Incremental development of both the RTL design and the UVM testbench is possible, so that a working setup can be integrated at several checkpoints along the way. There is no need for the one to await completion of the other, and in fact that strategy would most likely result in deadlock.

This way, the testbench can be a useful tool for the designer during the majority of the overall project time, and that helps avoid the 'two competing testbenches' outcome.

It is also possible to retrofit these requirements to an existing, inadequate testbench with only a little re-architecture.

> USABILITY REQUIREMENT #9: report and track bugs and defects in the testbench just as you would for the design.

The benefits of bug-tracking apply equally to verification code as to RTL. The team benefits from a working database of problems encountered and root causes / solutions to refer to during the project, and are left with a valuable record to inform successive projects. As with any bug-tracking activity, use the tool for productivity without fear of shame or retribution.

A well-managed project leads to a more usable testbench. Proper specification, managed development, clarity of issue tracking/resolution, will serve you well in your joint project.

Design and Verification teams are often separate entities, but work better when they have a common interchange of ideas and use this shared set of interfaces between RTL and UVM.

IV. TESTING WIH UVM - A STIMULUS PACKAGE FOR THOSE HIDDEN DUT BUGS

Our aim here is to make the switch from directed test to random sequences, using an test writing API approach to the stimulus layer, keeping the benefits and familiarity of directed testing.

A. *UVM is a framework for random testing*

The UVM is built around a fundamental base requirement: random test stimulus to explore a large feature/bug space, as opposed to traditional directed test stimulus. This in turn requires a self-checking testbench, with all the infrastructure that implies, as opposed to the traditional directed test which did a little checking after injecting some stimulus, or which relied on manual scrutiny of results to store golden reference outcomes for future runs of the same test.

Essentially, the old way was linear test vectors - set things up, do this, check that, at the pin/signal level. The new way is Sequences - do this, do that, do those, random variations of pin/signal stimulus, specified at a high level (the abstract transaction level), in conjunction with a model/scoreboard which checks results at that same high level, for any stimulus.

```
port1.drive_packet( .mode(MODE1),
                    .destaddr(1),
                    .len(128),
                    .payload('{128{'{8'hFF}})
                  );
do_some_checking( some_params );
```

Example 1. Before UVM - directed test task usage

B. *Random sequences are directed tests with added variety*

It would take a whole paper to discuss the relative merits of random and directed testing. Suffice to say that this author asserts that random testing leads to chaos and confusion unless it is done with a directed testing mindset, from the outset.

The 'constrained' part of 'constrained random' is the most important - it is not possible to achieve incremental bringup of a new design with a new testbench, using overly random stimulus. Instead, start with highly constrained stimulus.

```
mypacket pkt = mypacket::create();
pkt.randomize() with { mode==MODE1;
                       destaddr==1;
                       len==128;
                       foreach(payload[i]) payload[i]==8'hFF;
                     };
pkt.start(port1);
```

Example 2. After UVM - constrained random sequence usage

We recommend a designer-friendly testcase flow where testing starts with random stimulus that is so constrained it is essentially 'directed', and then other testcases successively relax those constraints to explore the design space incrementally.

In the example above, each constraint can be present or absent, depending on the level of 'don't care' variety required. The programmatic nature of the constraints also makes it easy to provide complex data values or relationships between successive inputs, which is not so easy to do with task calls.

C. *Get the most out of Hierachical Sequences*

UVM stimulus is defined and used in a SystemVerilog class wrapper called a Sequence. Although we do not need to know how that works inside the testbench, it provides a simple interface for building up complex random-friendly scenarios.

```
class hdmi_control_field extends hdmi_sequence;
   rand ...
endclass

class hdmi_data_field extends hdmi_sequence;
   rand ...
endclass

class hdmi_frame extends hdmi_sequence;
   rand hdmi_control_field ctrl;
   rand hdmi_data_field data;
   ...
endclass

class hdmi_3d_frame extends hdmi_sequence;
   rand hdmi_frame left;
   rand hdmi_frame right;
   ...
endclass

class hdmi_3d_series extends hdmi_sequence;
   rand hdmi_3d_frame frames[20];
   ...
endclass
```

Example 3. building a hierarchy of reusable sequences

Sequences can call other sequences by instantiating them. This allows us to define pre-canned stimulus, suitable for randomization, and call it from a toplevel test task just as we would a call a directed test built from sequential task calls.

The 'class'..'endclass' SV construct has one similarity to an RTL 'module'..'endmodule' - it is a group of functionality and data contained in a wrapper with a well-defined interface. There are many similarities - but the main difference is that modules are static (they have one or more instances that exist throughout the whole simulation run) and classes are dynamic - instances can come and go as required. Thus they are suitable for modeling data, like packets, or frames, or bursts on a bus.

D. *Requirement: a testcase template with an easy API*

Our goal here might be to learn the minimum necessary to write meaningful stimulus, or adapt existing stimulus to do what-if experiments or during debug of failing testcases.

One useful approach is for the verification team to provide a starting point, consisting of three things:

1) *A test-writers' API of useful utilities*

One of our usability goals for your verification team was to provide a designer-friendly testcase API to all the capabilities of the testbench that a testcase may need, from configuration to DUT setup to stimulus specification and application [1].

The API should be domain-specific - including protocol-centric common functions or design-specific utilities, such as:

- If your design has N channels of a protocol input, include a mechanism to blast parallel random stimulus on all ports together, as well as individual ports.
- For protocols with reactive elements: interrupt-driven calibration/training requirements in addition to regular traffic; provide an API to control and respond to them.

- If your DUT has addressable registers, provide an easy API to the underlying UVM register model to let you set up the registers quickly and concisely.

2) A template for creating new testcases, using the API

A standard format for the majority of tests, and a clone-and-go starting point for a new test, will benefit test-writers experienced in UVM/OOP or not. This also serves as a self-documenting source for the API use model. Each test would follow the same pattern: config, setup DUT, run traffic, done.

3) A growing library of testsbased on the template

By necessity these features are ultimately implemented by calls to UVM internal objects and methods. Providing an API can help to hide the complexity and make for a more regular look and feel to all tests in the library. Advanced users can use internal functions but the majority of tests can use the API.

Having tests that exist and which can be used as examples or better still cloned and modified to meet testing needs, will save time and leverage the reusable template / API.

E. Test suite - group tests according to degree of stress

Just as the design team's stress levels increase towards tapeout time, so should the level of stress provided by the stimulus in the test suite. First basic, then explore, then stress.

Start small [5], with a group of very basic bringup tests. In constrained random terms, these tests are virtually completely directed; there is no need for variety just yet. A good set of bringup tests are short runs that do one type of thing only, and use the most basic of DUT configurations.

Later, explore the feature space of the DUT, by combining test stimulus on various inputs, by varying the DUT configuration from the default, by adding random freedom to some attributes of the stimulus traffic.

Finally, stress the DUT as much as possible, with blasts of concurrent random stimulus on all ports, relentless back-to-back traffic with enough random variation to explore many internal corners. Keep doing that until it is time for tapeout.

A useful approach to managing this kind of test suite is to:

- put each testcase in a separate named/numbered file
- name each one meaningfully, maintain groupings
- encourage developers to clone-and-go e.g. for debug
- don't consolidate small tests into one random one
- refer to tests by name in bug reports, verification plans

```
% ls
test_101_basic_tx_packets.svh           // basic bringup tests
test_102_basic_rx_packets.svh
test_103_basic_loopback.svh
test_201_explore_packet_sizes.svh       // some randomization
test_202_explore_control_scenarios.svh
test_301_stress_random_packets.svh      // longer stress tests
test_302_stress_fast_back2back.svh
test_303_stress_random_allchannels.svh
test_bug1234_fifo_overflow.svh          // bug-specific tests
test_bug2345_bad_checksum.svh
test_req123_sleep_mode_while_busy       // specific feature req
test_req456_fifo_corner_case.svh
test_template.svh                       // boilerplate to clone
tests.sv                                // `includes all tests
%
```

Example 4. Simple Testcase Naming Convention

Keep each of these three groups of test for the entire project and reuse them on the next project - the basic 'directed' tests have value in their own right - they should not be 'retired' in favor of more random tests once design and testbench reaches some maturity level.

The reason for their value is simple: when things go wrong, after a DUT change or a new configuration of reuse, all bets are off. Some basic functionality could be compromised. So it's important to bootstrap the DUT again during your debug with basic tests first, stress tests later. Regression debug is more productive with clear indicators.

F. Steps to migrate from a legacy directed test suite to UVM

You have a set of NNN legacy test cases each consisting of: set up the DUT, provide some specific inputs, wait for it to be processed, check some specific outputs, and indicate pass or fail. The migration process your verification team will use is as follows:

1) implement a passive testbench with monitors

These connect to each DUT port to monitor the protocol transactions they see by reassembling high level transaction representations of signals and checking the protocol. Run the existing test suite on that testbench to debug them.

2) implement a model of the configuration space

Code a UVM register model to mirror the internal DUT configuration, capable of tracking the setup phase of each test in the suite. This can be used as part of the checking algorithm.

3) implement a self-checking testbench (scoreboarding)

This involves coding models and checkers (AKA scoreboards) which can derive expected outputs from monitored inputs of arbitrary value, and compare the actual monitored outputs against the expected ones, and pass or fail accordingly. Each kind of check that was performed manually in the directed tests must be incorporated, and more checks are likely added. Again, run the test suite, with deliberate DUT errors injected as required, to prove that each checker works.

4) implement coverage of configuration, stimulus, checks

Measure the overall coverage of the test suite - which configurations are set up by the various tests as mirrored by the register model, which values or combinations of stimulus and response are measured by the monitors, and which checks were triggered by the combination of activity.

5) replace the directed tests with equivalent random ones

The measured coverage profile is now the baseline. Create random sequences and constrain them as required until the equivalent coverage is reached or more likely surpassed. And enjoy the reduced regression runtime. Once you trust these results, the legacy test suite can be archived.

The process above would be implemented by a verification team, developing the testbench as they go. This is an effective way to reuse the value of legacy tests, by using them to calibrate and validate the new development, ultimately ensuring equivalence.

We have insufficient space here to include the complete code examples; these are available for download separately.

V. CONFIGURING UVM - MAKING THE MACHINE WORK FOR YOU, NOT AGAINST YOU

We describe the mechanisms available to you for configuration, binding of signals and interfaces between DUT and testbench, stimulus selection, control of a simulation run, its inputs, process and outputs, and expansion/reuse.

A. *Several different levels of configuration*

We will talk about different kinds of configuration, from hard parameters to soft options, and how they are used with UVM, and some example implementation patterns.

Designs are often highly configurable - this configuration controlled by defines / parameters / settings, and so testbenches often have to mirror that configuration.

The different kinds of configuration available all have applicability to different problem domains, they all have their merits. We list them here each with some discussion and some UVM examples.

B. *Verilog `define macros*

```
`define MAX_A_WIDTH 32
...
`ifdef PORTA_HAS_FEATURE1
  ...
`endif
...
bit [`MAX_A_WIDTH-1:0] address;
...
% <verilog-compiler> +define+MAX_A_WIDTH=64
```

Example 5. Verilog define macro usage

This is a compile-time configuration method: it has a high cost (recompile necessary when changes), and has low flexibility. In particular it is not scalable as parameters are global to the compile unit in scope, not localized to an instance or a hierarchy.

The UVM uses several `define constructs and several more elaborate macros internally.

```
+define+UVM_NO_DEPRECATED
+define+UVM_REPORT_DISABLE_FILE_LINE
+define+UVM_MAX_STREAMBITS=4096
```

Example 6. UVM config options set at compile time

Although some of the define values can be set from the simulator Verilog compilation command line, note that if using a UVM kit that is 'built in' to your simulator, it will have been compiled a certain way, and using other `define values will require you to do your own compilation.

C. *Verilog Parameters (and VHDL Generics)*

```
module dut #(int A_WIDTH) (/*ports*/);
...
dut #(32) mydut;
```

Example 7. Verilog elaboration parameters usage

Elaboration-time parameters have a lower cost than compile-time `defines, and there are some optimizations possible here, but they impose an overhead on a UVM testbench. Parameters can be integers, strings, classes, and also types:

```
class CLASS #(TYPE NAME=DEFAULT,...) extends BASE;
```

They are resolved after compile and import, and before run. They can be used with Verilog generate blocks allowing conditional and iterated instantiations and procedural code.

1) *Parameters considered harmful? Sometimes*

Parameters are often overused. Parameterized SystemVerilog classes can become a toxic complexity and performance problem area that needs to be confined to the component structures and not allowed to bleed into the data classes (including testbench internals such as analysis ports, analysis components such as scoreboards, functional coverage, checkers, and also stimulus: transaction ports, sequences and transactions themselves).

More appropriate abstraction is always the right way to both restrict the proliferation of parameters to where they are actually needed, and to meet the needs of abstraction without such being tied unavoidably to low level concerns. From the outside user's point of view, if you perceive that there is a lot of non-intuitive typing involved just to declare and invoke a testcase calling some sequences, then there is probably some misuse of parameters going on.

D. *Verilog runtime plusarg options*

```
status = $value$plusarg("VARNAME=%FORMAT", varname);
...
% <simulator> +VARNAME=VALUE
```

Example 8. Verilog runtime plusarg usage

Runtime plusargs have the lowest cost of usage, no recompilation is necessary, but they can set only 'soft' dynamic data values.

UVM uses plusarg configuration for the following options, most of which are internal but which may appear in scripts or Makefiles provided by the testbench developers:

```
+UVM_TESTNAME                 // which named test class to run
+UVM_VERBOSITY                // how verbose the log messages
+UVM_MAX_QUIT_COUNT

+UVM_SET_CONFIG_INT           // set config from command line
+UVM_SET_CONFIG_STRING
+UVM_SET_INST_OVERRIDE
+UVM_SET_TYPE_OVERRIDE

+UVM_OBJECTION_TRACE          // Various internal debug settings
+UVM_PHASE_TRACE              // and features which allow debug
+UVM_CONFIG_DB_TRACE          // of the testbench
+UVM_RESOURCE_DB_TRACE
+UVM_DUMP_CMDLINE_ARGS

+UVM_NO_RELNOTES                 // internal compatibility modes
+UVM_USE_OVM_RUNTIME_SEMANTIC   // for migration OVM->UVM
```

Example 9. UVM config options set at run time

E. *UVM Configuration Database / Resources*

UVM has various set/get APIs for passing configuration values around inside the testbench. Users need to know a little about this, as it inevitably appears at the outside - in the toplevel Verilog module that instantiates the DUT, its

interfaces, and invokes the testbench. There is normally a little configuration to be done here to pass in options, and hook up the interfaces. Normally this hookup is coded by the verification team, so the only important piece to learn as a user of the testbench, is where to set the configuration values and what possible values there are.

```
module top;
   ...
   axi_if axi0, axi1;                    // DUT has two AXI ports
   mydut dut(....);
   ...
   testbench_config cfg;                 // setup all TB config
   ...
   uvm_config_db#(virtual axi_if)::set(null,"*","port0", axi0);
   uvm_config_db#(virtual axi_if)::set(null,"*","port1", axi1);
   ...
   cfg.no_of_ports = 2;                  // choose some config
   cfg.enable_data_checking = 1;
   ...
   uvm_config_db#(testbench_config)::set(null,"*","tbcfg",cfg);
   ...
endmodule
```

Example 10. UVM config object setting example

The above code shows two kinds of item being passed into the testbench world from the Verilog module world: (1) an interface to the DUT pins, and (2) a configuration object containing a bunch of settings.

The simple example is only to illustrate some syntax, but we would recommend that the config settings are kept in a separate file, all together, with comments that document how they can be used, any restrictions or useful information.

A good test of success of the whole 'testbench as a black box' approach is as follows: the designer knows where to go to find and change a config setting, is able to do so without fear and trepidation, and when the test is re-run with the new config it behaves as expected (apart from any bugs it finds of course!)

F. UVM Register Model configuration register setup

Once we have taken care of configuring the parameters of what shape of DUT we have, and set up the config options for the testbench, the next kind of configuration is to set up the DUT internal behavior. This is normally done by addressable registers and/or configuration pins.

UVM provides a register model feature which makes this very easy to do in a testcase - it provides an abstract way to refer to registers, fields, values. We recommend you insist upon its use if you have registers, it makes programming them so much easier in a testcase, and will also help the testbench writer who can refer to the model values when doing checks.

```
// setting a register in the model
   uart1.CONFIG.set(16'h0104);

// setting register by named field
   uart1.CONFIG.PARITY.set(UART_PARITY_EVEN);

// flush all writes to registers via the bus
   uart1.update();
```

Example 11. UVM register model and setup sequence example

Your testbench writer can add support for easy access to the register model into your testcase API, so that you can refer to groups of registers like 'uart1' in the example above, without needing to use 'env.cfg.regs.uart1' (or similar) all the time.

G. Randomizing the configuration

The configuration of a testbench (this includes the register model) is held in SystemVerilog class containers, just like the transactions to be sequenced on each DUT port. Consider it as part of your stimulus - still dynamic, perhaps not as dynamic as the packet-by-packet sequence of traffic.

Therefore, you can randomize the configuration just as you would the traffic. Constrain it in such a way that it is always a valid combination of settings.

```
   testbench_config cfg;
   ...
   cfg.randomize();
   ...
   uvm_config_db#(testbench_config)::set(null,"*","tbcfg",cfg);
   ...
```

Example 12. UVM randomization of testbench config

As mentioned previously - in your list of testcases, start with the more 'directed' ones, and this means fixed, preset, 'safe' config values, before branching out into the random ones.

Registers can also be randomized, which is a useful way to find bugs. You might anticipate that registers are set up a particular way or in a particular order, but your firmware team may have other ideas. Randomization lets you try many different combos of register settings in your testing.

H. Reuse and configuration

As mentioned during our list of essential usability requirements, all available configuration settings for a given UVM testbench should be brought out together to a known file, which is well commented and self-documenting to the extent that it can be comprehended by the design team as well as its writers in the verification team, and altered without trepidation.

A UVM testbench typically has configuration settings to control its structure, operation and enable reuse. DUTs which are reused across projects in different configurations (horizontal reuse) typically have some parameterization - different bus widths, or channel counts, or optional features. Their testbench needs to have those degrees of freedom also.

For vertical reuse (cases where a testbench or portion of a testbench can be reused in a larger context, for a DUT which is instantiated in a larger context subsystem) there is a different set of reuse requirements affecting UVM configuration.

Where UVM VIP components may have been active (responsible for driving values into the DUT pin interface) at the unit level testbench, a vertical reuse of that same testbench at system level requires those UVM components to be passive (only monitoring what is going on on the DUT pin interface) because what was the whole DUT is now a module, integrated along with others, with its pins driven by some RTL around it.

When reuse of this nature is anticipated in advance, configuration options can be put in place to better support it. For example, the UVM register model of the unit DUT may be integrated as part of a larger system-level register model, requiring an override of its base address, which was zero at unit level, and is now in some context of the system address map.

VI. DEBUGGING WITH UVM - A STRUCTURED APPROACH TO THE ASSIGNMENT OF BLAME

We discuss interpreting and adjusting UVM testbench output, tracing from signal to driver to transaction to sequence, various transcript- and code-based debugging techniques discussed, also early-warning techniques, efficiency, roles and responsibilities for debug, assertions, root cause analysis. The goal is to find a practical and efficient way to answer the question: 'DUT bug or testbench bug'

A. *Does UVM help or hinder our debug productivity?*

Of course, UVM is just a framework - how it is used, to create the 'product' that is your testbench, is what determines the ultimate level of usability, and debuggability. We assume here that the essential recommendations made earlier in this paper have all been implemented by your verification team. Some of their benefits will become apparent during debug.

> Use of UVM benefits debug productivity in two ways: it provides a regular, predictable, scalable structure to a testbench, and it provides mechanisms for structured reporting (to logfiles, transaction recording, simulator built-in debug) of what's going on.

The first of those should help us answer the 'DUT bug or testbench bug' question more quickly - the regular structure lends itself to simple tracking of stimulus and response and how they are processed within the testbench - and in turn a more organized and scientific approach to debug. Having implemented a standard waveform setup for each DUT interface, we can quickly assimilate signal group activity and reference against transaction activity. Your verification team can consult the transaction/class debug capabilities of your toolchain to set this up for productive debug.

It is implicit that we do not wish to spend time on debugging the internals of the testbench, only on the design. So a debug flow should quickly help us answer this question.

With ad hoc testbench architecture there are more variables, more things that can go wrong. UVM has more complexity but imposes a more regular, predictable structure which we can view as a black box.

B. *Methods for effective bug triage*

The first step in debug of failing tests is triage: the grouping of problems according to cause or severity, with initial quick diagnostics, before dispatching them for more detailed analysis and solutions.

> We propose a method for addressing the 'Monday morning' problem - the team arrives at the office, and looks at all the failing tests in the weekend regression run, and decides what to do, who does what, and how to be most productive. The above benefits of UVM can help. Here is an example flow for triage:

1) Prepare - get familiar with your testbench messaging

Log messages from UVM are very structured, and should identify the portion of the testbench involved precisely, along with (hopefully) useful information from the testbench writer.

2) Initial categorization of failing tests by error message

This first step will group our 'fails' according to the error message reported. The first error message in the log is the one that applies here (subsequent messages can of course be of value but are normally secondary effect rather than root cause).

3) Rerun representative failing tests with debug turned on

If the failing tests are indistinguishable due to too little information, this is a serious defect in the testbench to be fixed urgently but easily. For each group of 'like' fails, choose one test and rerun it, exactly the same testcase, stimulus, configuration and random seed as before, but with all the debug switches turned on.

4) Know the debug settings at your disposal and use them

Refer to the file we specified in the first section, which contains all the debug settings and help information on additional simulator or environment settings to make. A good simulation / regression environment will make that knob easy to flip: performance or debug.

C. *Rerunning failing tests with more debug visibility*

UVM provides several controls that we can use to tradeoff fastest batch performance vs fullest debug visibility.

> UVM supports performance optimization in logfile messaging by inhibiting the processing and display of messages above a certain controlled integer level of verbosity. This control also enables easy rerun with more message detail or with selective detail. Ensure that the testbench uses this UVM feature in order to assist with debug.

In order to maximize simulation runtime performance, and hence minimize overall regression time, a regression run will typically turn the 'visibility' settings towards zero. The only visibility we need from a batch regression run is 'did it pass or did it fail' and perhaps a measure of coverage. Of course some logfile information about 'what it did' at a high level is also useful, but in general, we want to turn all of that off.

This means that when we get a failing test, it's time to turn it all back on again and rerun. Of course this takes time, but it is an investment of time that we make in order to get the most productivity out of the time we spend afterwards debugging.

```
// UVM commandline control switches example
+UVM_VERBOSITY=UVM_FULL                    // show all messages
+uvm_set_verbosity=port0,_ALL_,UVM_HIGH    // selective (port0)

+UVM_CONFIG_DB_TRACE          // Various internal debug switches
+UVM_OBJECTION_TRACE          // not usually used in DUT debug
+UVM_PHASE_TRACE
+UVM_RESOURCE_DB_TRACE
+UVM_DUMP_CMDLINE_ARGS

uvm_top.print_topology();      // internal UVM testbench debug
uvm_top.check_config_usage();  // not usually used in DUT debug
uvm_factory::get().print();
```

Example 13. UVM controls for debug visibility

UVM allows very fine control over which portions of the testbench get more detail in logging messages or transaction recording - and that detail may be useful in large designs where every debug switch can hurt performance. Otherwise, turn it all on, rerun the test without further ado.

Ideally the testbench component which emitted the error message that indicated 'fail' will be a hint to which functional area may need debug, and hence which portions of the design to open up for example to record waveform information for graphical debug. It is not always possible to record 'all' signals, but ensure that signals in the pin interface layer are recorded.

D. Isolate the fail - rerun with more specificity

Our regression tests may be longer tests that do more, i.e. run a greater number of iterations of constrained random stimulus on the DUT. By the nature of randomized testing - a particular 'seed' value caused a particular combination of stimulus to occur which exposed the problem at some point in the testcase.

It is desirable to narrow that testcase down, to make successive iterations for debug or tryout of a fix shorter, and to leave behind an isolated testcase for future regression.

The UVM class-based random sequences create an obstacle to doing this - random stability. Different runs with different stimulus or changes to some settings may behave differently in the random generation. In the worst case, changes made in order to help debug may lead to different random stimulus (which may mask or not exercise the original failure mode).

> UVM has some support to minimize the random stability problem, so it is prudent to ensure that your verification team's testbench code addresses this requirement, including ensuring that each portion of the testbench has unique UVM names [2].

Note that our ability to cut down the testcase to isolate the problem is greatly enhanced by the regular testcase structure and test-writers API we discussed earlier. The general flow to follow is:

1) Clone the failing testcase, to work on it safely

Give the new testcase a temporary name - we will subsequently name it after the bug number when we do a bug report, and check it in for regression if it finds a DUT bug.

2) Identify the position of the failing stimulus

Decide whether to cut down the test to size by truncating the test just after the point of failure. Fixing the problem will let the test proceed beyond that point, but it may fail at a later point, but that is as likely to be a different problem altogether, to be triaged and debugged separately. Chopping down the test lets us focus on one bug, hopefully one sequence of stimulus.

3) Identify what matters and what does not matter

It is useful to look at the configuration of that test, determine anything unusual or common about that configuration relating to others that have failed, or others that are assumed bug-free. A brief analysis can help inform our efforts - maybe to fix N failing tests together if we identify the common problem.

4) At each step of analysis, rerun the test to ensure it fails

If this is your first time working on a project with a structured verification environment such as UVM, and a structured, repeatable regression suite, you may be unfamiliar with this: the goals of a successful debug and repair session for a failing test are:

> (a) Identify a test which fails, and shows the problem
> (b) Debug and find the root cause of the problem
> (c) Fix the problem
> (d) Ensure that the test which once failed now passes
> (e) Document everything so that you learn for later

E. Methods for comprehension - tracing transactions

There are two kinds of tracing that you may have to do when debugging bad DUT or bad testbench behavior, or determining which of these is the cause of the failing test:

1) Trace the drive path from testcase to DUT signals

- For example, start at the signals and work back. Use the standard waveform file to look at the signals within the SV interface shared by the DUT pins and the testbench VIP drivers, and identify the time bound of the protocol exchange of interest.
- Identify the transaction passing internally in the UVM testbench to the driver of that interface port. There are two ways to do this: (1) have verification team provide transaction recording setup for you to switch on and visualize in your wave browser, and (2) identify trace prints in your logfile coming from the driver.
- Same for the sequencer. This is the part that feeds the driver one transaction at a time from a sequence. Your goal is to identify (1) the name of the sequence being run that caused the transaction of interest on the driver and pins, and (2) where in your sequence or testcase code that sequence was invoked
- Or, start at the testcase and work forward through the same steps, depending on what you are debugging, so that you can investigate further or alter the behavior of either testcase or DUT experimentally as required.

2) Trace the monitor path from DUT signals to testbench

- This is the reverse path to the above - monitoring rather than driving the pins - the testbench is making sense of the signals and reconstituting a transaction. The monitor writes out that transaction internally in the testbench on a port called an analysis port, and normally other parts of the testbench take that and do more checking on it, comparing with expected values.
- Start at either the pins or the monitor transaction report that you see in the logfile. There are fewer stops along the way on this path - just DUT pins, interface, and monitor transaction report. If you have to go further (into the testbench scoreboard, you will need assistance from the UVM expert. Do not hesitate to ask for more lucid debug messages in the logfile if it lacks clarity.

F. Managing waivers for failing tests and checks

At some point in the development, debug may identify a problem that cannot be fixed until some later time. This is when to put a 'waiver' mechanism in place. Comment out or otherwise disable a particular testcase, or (in the testbench) a particular kind of check or assertion. But make a note of that (a 'waiver') so that it is reviewed, and never forgotten about.

VII. SPEAKING UVM'S LANGUAGE - HOW TO TRAIN YOUR VERIFICATION TEAM

It is useful to know how to specify improvements, new features, new behavior, checks, and kinds of stimulus, informed by some knowledge of how the testbench operates.

A. *UVM brings a dictionary of new words for old things*

Today's advanced testbenches are architected, coded, resourced as if they were standalone software products.

Many coding techniques are borrowed from the world of Software Engineering to achieve this, and so this has given rise to some new vocabulary, including some debatably arbitrary different terminology.

Here is a UVM secret decode ring for some terms you will encounter, any more precise definitions are unnecessary detail:

- ***Verification***, ***Validation***, ***Testing*** are all the same thing
- ***Sequences*** code stimulus values driven as DUT input
- ***Monitor*** and ***Analysis*** interpret and check DUT output
- ***Agent*** or ***VIP*** is a bundle which drives/monitors a port
- ***Scoreboard*** is code that validates DUT output vs input
- ***Config*** is some settings to control how testbench works
- ***Resources*** are just addressable global ***Config*** variables
- ***Coverage*** is a count of how much of the spec is tested
- ***Abstraction*** is a high-level interpretation of something
- ***Layering***, ***Hierarchy***, ***Composition*** are the same thing
- ***Factory*** is an arrangement to substitute code at runtime
- ***Transaction*** is one protocol exchange in abstract form
- ***Objection*** is a flag raised which blocks another process
- ***Verification Environment*** is a testbench, some stimulus

B. *Find a common language but keep the detail separate*

Some verification professionals are perfectionists or pedants and may disagree with the above simple definitions, or talk in terms that are not in design teams' vocabularies, and that do not need to be. Discourage such behavior when possible.

It is important to find a common language that both parties can understand, during specification, enhancement requests, project planning and other joint activity. Avoid wasting time on detail that obscures and hence devalues the shared communication, rather than enhances it.

Remember the DUT is the most important thing - the most useful common language is terminology specific to the relevant application domain and the protocols used in the design.

C. *Continuous improvement*

As with any product, the UVM testbench can evolve over time, during the various phases of the project and from project to project. Keep track (in bug reports, for example) of what worked well and what did not, and participate in code reviews or project wrap-ups to ensure your feedback gets taken into account. It helps overall productivity if you act as a customer.

VIII. CONCLUSION

This paper is completely different from those that seek to teach designers and testbench users a little of how UVM works. We advocate instead a black box flow, with the UVM testbench as a product with a well-specified defined interface and a happy user community among designers, debuggers, those who write testcases or port legacy tests over.

This is one approach that may be the right fit for your team.

Provided you have a verification team or resource that takes care of the testbench architecture and coding, and can implement the requirements we list here, there's no need for design teams to learn 'easier UVM' or 'express UVM' or 'bite at a time UVM', any more than there is a need for the verification guy to learn complex design RTL and synthesis techniques.

As an advanced user of an advanced UVM testbench, you can be very productive without knowing anything about how sequences work or how the factory works, or how TLM hookups and analysis ports work.

Of course you can choose to learn if you want to, and there are many good resources to help you do that, if you want to get involved in verification. We list some of those below. But sometimes the most productive solution for the whole project is the black box solution. Treat the testbench as a tool - have input into its specification - and use the tool productively.

IX. UVM RESOURCES TO HELP YOU - WHERE TO GO FOR ASSISTANCE AND THERAPY

There are various information resources that can help you get the most out of your UVM testbench and improve usage from chip to chip. The ones we would recommend are:

Verification Academy - a comprehensive set of resources including video training classes - from Mentor Graphics. http://verificationacademy.com

The UVM/OVM Cookbook and other verification cookbooks - hundreds of pages of knowledge and proven methodology, useful for verification and design teams, http://verificationacademy.com/cookbook

See also the DvCon papers referenced below.

REFERENCES

[1] B. Ramirez, C. Lovett and S. Secatch, "OVM testbench API for accelerating coverage closure," Tech Design Forum, February 2011

[2] A. Efody, "UVM Random Stability: Don't leave it to chance," DvCon Proceedings, February 2012

[3] R. Edelman, "UVM Express", http://VerificationAcademy.com, February 2012

[4] N. Johnson, "UVM Is Not A Methodology", http://AgileSoC.com, March 2011

[5] N. Johnson, "Functional Verification Doesn't Have To Be A Sideshow". http://AgileSoC.com, May 2012