First Reports from the UVM Trenches:

User-friendly, Versatile and Malleable, or just the Emperor's New Methodology?

Jonathan Bromley
Verilab Ltd, 272 Bath Street, Glasgow Scotland G2 4JR
jonathan.bromley@verilab.com

ABSTRACT

For us, UVM arrived at exactly the right time. Our large new project evidently called for more powerful verification techniques than its predecessors. At the same time, a change of tool chain was on the horizon, bringing cross-vendor compatibility concerns to the fore. Finally, many members of the team were beginning to take their first steps in OOP and constrained-random verification, making it essential to establish a robust framework that could be stable for the foreseeable future, while offering a growth path that would match our developing needs. This paper reports on some of the successes, pitfalls, unexpected problems and unanticipated delights of our UVM rollout.

Our project was making day-to-day use of UVM verification components within two weeks of the Early Adopter release, leveraging previous OVM experience of some members of the team. Several months on, our in-house UVM library is burgeoning in size and flexibility. Real bugs in our design have been found and fixed, and our first predominantly UVM-verified tapeouts are due as this paper goes to press. Starting from a plain-old-SystemVerilog methodology that worked well but had limited room to grow, numerous members of the team have transitioned to SystemVerilog OOP with the help of UVM, aided by a strong existing culture of re-use.

Not everything about our UVM experience was positive. Aside from the handful of minor shortcomings in the UVM library that any early adopter must expect to live with, we describe some important things that we had hoped UVM would provide but which we have been obliged to build from scratch. We also report on challenges we faced in ensuring consistency of approach among team members, incorporating existing verification assets into our UVM framework, and getting the best out of the whole team's skills.

In the paper we review key aspects of our UVM experience, giving special attention to the match (or mismatch) between UVM advocacy we've heard and the UVM features that we found were most productive. Examining several specific technical issues in detail, we highlight areas where we have chosen to abandon established or published recommendations in favour of a more radical approach. Finally we assess the overall productivity gains and losses that UVM brought, and identify key concerns that we believe the UVM community must soon address to avoid the unpleasant prospect of large numbers of users each with their own incompatible implementations.

KEYWORDS

UVM,verification, methodology deployment, verification IP

1 INTRODUCTION

This paper reports on our introduction of the Universal Verification Methodology, almost as soon as it became available, on an ASIC development project. Section 2 describes the motivation and rationale for this choice. Section 3 discusses some of the project management and technical strategy issues we encountered during our initial deployment.

In sections 4 and 5 we examine a selection of specific concerns with the design, documentation and presentation of the methodology, and indicate areas where we believe it could usefully be improved. It is important to be aware that this paper specifically describes the Early Adopter release, and some of the concerns have already been addressed by later developments from the Accellera committee responsible for the methodology.

Sections 6 and 8 describe specific techniques for connecting a SystemVerilog testbench to the Verilog static hierarchy. They have proved to be useful in our UVM environments and may be of interest to other users. Section 9 enlarges on these ideas to provide a novel method for integrating legacy verification components with a new UVM environment.

Finally, section 10 briefly discusses some anecdotal observations concerning other engineers' responses to the rollout of this new approach in the team.

2 BACKGROUND

The Early Adopter release of the Universal Verification Methodology (UVM) [1] was published by Accellera in May 2010. Our team's management agreed to adopt it for immediate use on new verification code for a current ASIC project. Although it might seem foolhardy to adopt a new and presumably untried methodology toolkit, there were excellent reasons for so doing in this case. The author and a colleague were tasked with facilitating the deployment of UVM on this project.

The team's existing verification activity made extensive use of SystemVerilog, but little use of classes or object-oriented programming. All verification components were captured as modules or interfaces as appropriate, with port connections hooking to DUT signals and SystemVerilog mailboxes used to provide communication channels between components. Simple classes were used to capture data objects (typically the contents of a single bus transaction such as a read or write cycle) so that they could be conveniently passed from one component to another through these mailboxes.

Extensive experience with this home-grown methodology had led to the creation of a wide selection of useful and reliable verification building blocks that could relatively easily be applied to the verification of new RTL both at block and at system level. However, any advocate of object-oriented programming (OOP) methodology would argue that the use of modules to encapsulate verification components is sure to make it difficult to extend existing components to meet new

requirements, and is likely to lead to an undesirable diversity of application interfaces to the components because there is no way to derive them all from a common set of base classes. These limitations were indeed becoming apparent.

A new project, with larger and more complex digital content than had been tackled before, led the team to consider adopting a mainstream OOP-based verification methodology. However, the risk of becoming locked into a single tool vendor's offering was a major obstacle to adoption of either VMM [4] or OVM [5] even though both those methodologies were clearly adequate for the task in hand.

We were carefully observing Accellera's VIP initiative [1] to develop a vendor-neutral verification methodology and toolkit, and so the timely announcement of the UVM Early Adopter release provided exactly the trigger we needed. It allowed us to proceed with confidence that UVM code we write today will continue to be useful, perhaps with minor modifications, in the future.

3 OUR INITIAL UVM ROLLOUT

Much of the UVM's base class library is strongly based on OVM 2.0 [5]. This made it relatively easy for us to begin work, as the author and some colleagues already had extensive experience with OVM and VMM. Consequently we were able to create some key UVM components (in particular, agents for some proprietary bus structures that are widely used in the DUT) within only a couple of weeks of the Early Adopter release. Although it took a little longer before we had working verification environments doing useful testing, this initial success gave us confidence that we could indeed roll out UVM across much of our verification effort.

3.1 Infrastructure

Creation of some initial UVM agents was quite straightforward. It was more challenging to establish a common framework (directory structure, naming conventions, etc) so that the UVM components we created would be accessible to the verification team in a way that did not disrupt existing practice. The team has an elaborate and project-proven scheme for organizing files and directories, using Subversion [7] for revision control, and we felt it was very important to respect and build on that tradition. We now have a system whereby an environment can have access to any UVM component simply by adding just one include file to its master list of files; that include file then does whatever hierarchical includes are required to compile all parts of the component. Furthermore, the directory structure associated with any UVM component or sub-environment is consistently named and easy to navigate.

3.2 A Problem of Proliferation

An obvious early step in our UVM rollout was to provide UVM wrappers for some of the large and valuable collection of existing module-based verification IP. Most of these blocks took the form of traditional bus functional models (BFMs) with the usual pattern of signal connections through module ports, and tasks designed to be called from the verification environment to get the BFM to perform various operations. Clearly this has strong resemblance to a UVM *driver* or *monitor* class. However, we were not sufficiently proactive in setting up a framework and guidelines for creating these BFM-to-UVM gaskets. Consequently we found ourselves facing *embarras de richesses* with more than a dozen such components available for use, but very little consistency of application-programming interface (API) among them. In many cases the user's interface to these objects was entirely through task calls and there was no randomizable transaction class to capture a unit of activity that could be performed by the component. The result was a UVM quasi-component that suffers exactly the same shortcomings as the original BFM: it is easy to use when creating simple directed stimulus, but cannot be used with the sequences mechanism and does not support any kind of transaction-level (TLM) connection with the remainder of the testbench.

3.3 User Reluctance

Our user base of verification engineers – many of whom are primarily RTL designers who also have excellent verification skills – varied greatly in their enthusiasm for UVM. Some were very positive about the new approach, often seeing it as an excellent opportunity to develop their SystemVerilog OOP skills. Others saw our growing UVM codebase as an obstacle to their understanding and progress, pointing out that UVM made it harder to do things that they could do rather easily with their existing techniques.

3.4 Lessons Learnt

With hindsight there are some clear lessons from this experience. Those who got involved with the development of new UVM code, early in the rollout, were much more likely to be positive about UVM adoption than those to whom the UVM was presented as a *fait accompli*. It is clear that we should have taken greater care to design a progressive rollout plan that engaged all team members effectively. We underestimated the importance of shared understanding and shared decision-making across the team.

For experienced OVM users on the team, it was very easy to forget the learning curve associated with UVM adoption. For example, the challenge of gaining familiarity and confidence with the huge portfolio of reporting control methods and options is a big disincentive to making proper use of the UVM.

These human factors are discussed more fully in section 11.

4 SHORTCOMINGS OF THE EARLY ADOPTER RELEASE

This section presents some concerns about the UVM Early Adopter release (1.0 EA). It is probable that there will be a new and very much enhanced production-quality release of the UVM available by the time this paper is published, and the author is confident that many of the concerns described here will have been addressed by it. However, he feels it is useful to record them so that developers may continue to bear them in mind as the UVM moves forward.

4.1 Documentation

There is little doubt that the documentation associated with UVM could usefully be improved. The extensive use of NaturalDocs [11] as a tool to generate publishable documentation from structured comments in the source code is welcome, and generally has led to a thorough and useful reference document [2]. However, it is at the mercy of the quality of the original source code's comments. We were sorely disappointed by some of the material, such as this example describing `uvm_object::copy`:

```
function void copy (uvm_object rhs)
```
The copy method returns a deep copy of this object.

That statement is remarkably unhelpful. The method doesn't return anything at all, it copies `rhs` rather than the current object, and it updates the contents of the current object `this` as a side effect. Although this is a particularly grotesque example of poor internal documentation, there are many other cases where careful review with the *reader's* needs in mind would be most welcome.

The reference documentation is also flawed by the omission of various important details. For example, argument lists of the `uvm_do_*` family of sequence macros are nowhere described. It is tiresome to be obliged to study the source code, or to search through informal user-guide documents, in order to locate such missing details.

4.2 Handling Low Levels of Abstraction

A fundamental goal of any sophisticated verification methodology, including the UVM, is to raise the level of abstraction at which verification can be done. By expressing stimulus and responses as transactions, rather than signal transitions, the verification engineer can operate at a level that more closely relates to the design specifications, and better reflects the description of device activity typically found in requirements documents and use-case scenarios. SystemVerilog OOP somewhat forces the verification engineer's hand in this respect, making it remarkably difficult to gain access to DUT signals directly from code in a class – especially if the class is defined in a SystemVerilog package for ease of later re-use.

This raising of abstraction level is unquestionably a powerful approach, allowing code to be written that models complex behaviors without becoming mired in the irrelevant details of pin-level or clock-by-clock activity. Unfortunately, though, it is not always possible. Even the largest, most complex ASIC is nevertheless a piece of digital hardware with clock, reset and enable signals. For some verification activity, the detailed behavior of certain signals at a very low level of abstraction is critically important. The UVM, and its associated documentation, fails to provide adequate guidance to users faced with this kind of concern.

4.2.1 Events

For example, our ASIC uses a common timebase signal (typically running at some tens of kHz) to synchronize major activities across various parts of the design. Almost every design block, and therefore almost every verification component, needs to be aware of this timebase for purposes such as grouping a series of data samples according to the timebase slot in which they fall. To capture the transitions of this timebase as UVM transaction objects is an unnecessarily heavyweight mechanism. More importantly, it is the wrong level of abstraction. The timebase conveys no information except that it has pulsed, and it is conceptually inappropriate to represent that as a transaction. The `uvm_event` mechanism is clearly a useful candidate, but it does not fit smoothly into the rest of the methodology. For example, what does it mean to "connect" an event from one UVM component to another? There are, of course, many straightforward ways to make such a connection, but whatever method one uses there is an uncomfortable sense that it is outside the methodology.

4.2.2 Signal Access

We have many critical verification requirements that depend on parts of the testbench having detailed knowledge of the real-time state of certain specific signals. A typical situation is that the meaning of a transaction may vary depending on the value of some control signal at the moment the transaction occurred. If the signal is not part of a standard interface protocol, but instead is a global control signal in the design, then it does not form part of the transaction and must be sampled by other means, while maintaining knowledge of the relative timing of that sampling and the protocol transactions that the signal affects.

Sampling and driving such arbitrary "one-off" signals is unreasonably troublesome in the UVM. We soon decided to create a special UVM agent, with the usual monitor/driver/sequencer architecture, to handle signals of this kind. However, it was unreasonably difficult to design a meaningful and useful transaction class that made sense for all the varied situations in which such signals are used. This is a clear example of *abstraction inversion*: the methodology obliges us to use an inappropriately abstract representation (transactions) for something that inherently demands a rather low level of abstraction. It has led to the creation of verification components that are difficult to understand and deploy, and suffer unnecessary runtime performance overhead. More recently we have learnt to approach this problem in a more satisfying way (described fully in section 8) but it has taken us away from mainstream UVM technique, leaving us fearful that we may have "broken the rules" and created architectures that will not match other UVM users' best practice.

4.3 Underspecified Data Comparators

The portfolio of comparator components found in the UVM library is disappointingly inadequate to support real verification problems. Although the *algorithmic comparator* with its *transformer* class provides an interesting tutorial in object-oriented programming, none of the provided comparators has the flexibility that we need. Whenever we tried to use them we were obliged to add preprocessing to the input data streams to support skipping of samples, duplicated samples, ignoring a certain number of samples after a reset, and suchlike real-world issues. It is these concerns that dominate the coding effort. By contrast, the UVM-provided behavior of matching data at the output end of a pair of FIFOs is somewhat trivial, and we soon chose to abandon the standard comparators in favor of our own designs that better fit our purpose.

The fate of the UVM standard comparators was sealed because of a bizarre oversight in their implementation: there is no way to clear the contents of their FIFOs because their FIFO data members are declared to be local. Modeling of reset and mode changes is therefore intractable, and requires so much rework of the original code that there is little value in using the provided classes.

4.4 Register Modeling

As already mentioned, we enthusiastically took up the UVM Early Adopter release because it offered the promise of vendor neutrality within a familiar framework. Beyond our selfish local concerns, though, it was unfortunate that UVM was released without including an Accellera-mandated register abstraction package. It is hard to imagine any non-trivial project that does not require such a feature.

Our team's existing SystemVerilog verification framework included useful and mature tools for register modeling, but they were not easy to adapt for the dynamically-created UVM verification environment. Instead we tried to use one of the register packages that had been contributed to the UVM World website [10]. We were aware of, and troubled by, the fact that

this package was not standardized and might become effectively deprecated at any time. What we were not prepared for was the rather large amount of work required to massage our existing register descriptions (derived from a spreadsheet by means of various Perl scripts) into the IP-XACT XML format required for the register package we tried to adopt. Consequently we have invested a non-trivial amount of effort into support for a register package that is now effectively deprecated thanks to Accellera's blessing of a different register abstraction mechanism that will form part of the first production release of the UVM [3]. We welcome the newly-standardized package, but regret our inappropriate choice and the wasted work that it brought us.

4.5 Lack of Temporal Assertions

Although this is a SystemVerilog issue that could never have been solved by the UVM, it seems appropriate to mention here a serious limitation of SystemVerilog's testbench facilities: the lack of temporal assertions for use in classes. SystemVerilog Assertions (SVA) [6] provide a powerful, concise and intuitive way to describe possible design behaviors over time, and to have those behaviors monitored for checking and coverage. Sadly (although for entirely valid reasons) the SVA temporal language can be used only in static Verilog design elements such as modules and interfaces. Consequently, temporal assertions cannot easily be added to UVM verification components. Instead the verification engineer must fall back on traditional techniques for coding temporal checks, such as state machine descriptions or ad hoc mechanisms.

5 INSUFFICIENT GUIDANCE FOR USERS

Section 4 could be read as simply a catalog of accusations against the UVM. That is not the intent; UVM has brought great benefits to our project and we will continue to use and value it. However, it is a recurring theme in section 4 that the methodology should provide a supportive framework, guiding users' implementation decisions when faced with common architecture problems. In some areas, the UVM meets this challenge superbly well. Transaction-level modeling and the associated connection arrangements, analysis ports, the object factory, and the conventional agent architecture are all fine examples of the UVM's contribution to a consistent, easy-to-follow implementation framework. There are, though, some equally important concerns that the UVM does not address satisfactorily. This section outlines the issues that caused us greatest pain, and for which we would value robust guidance to reduce the risk of users adopting widely divergent approaches.

5.1 Sharing of Globally Important Objects

Every test environment has information that must be shared by many different parts of the testbench. Typical examples of such globally significant information include:

- timebase and other major synchronization events
- DUT configuration such as address maps, memory sizes
- test case configuration
- reporting and verbosity options

The UVM configuration mechanism works well for shared objects that can be created at the outset, and then shared around the environment by top-down configuration. Often, though, shared global objects such as test setup control cannot be constructed until creation of the verification environment is largely complete – too late for the automatic configuration mechanism. The UVM lacks a uniform mechanism for sharing of such late-generated objects. It has no shortage of techniques – the pool classes, built-in tools for navigation of the instance hierarchy – but users would benefit from more specific guidance on how to deal with this kind of issue. Some of the responsibility for this guidance must fall not on the UVM's implementers but on the user community as a whole (and, more specifically, on book authors, trainers and tool vendors' customer-facing applications engineers).

5.2 Getting the Nuts and Bolts Right

Published material on the UVM's predecessor OVM [12] has generally been somewhat dismissive of the problem of how to connect an OOP verification environment to the DUT's Verilog signals. The author believes this to be misguided. Linking UVM classes to a test harness or DUT may be beneath the dignity of expert OOP practitioners, but it is vital to the success of a verification effort and users of the UVM deserve to have clear, practical guidelines for doing it. We have discovered to our cost that, lacking such guidelines, there will be as many different ways of implementing it as there are verification engineers on the project.

The problem is exacerbated by the current widespread enthusiasm for SystemVerilog's virtual interfaces, whose shortcomings the author has already lamented elsewhere [8].

Attaching a collection of virtual interfaces to their proper places in a test harness usually requires that at least some UVM classes be coded not in a package, where we prefer to put them for ease of re-use, but in a module that is instanced somewhere in the Verilog hierarchy. From classes defined in such a module, the user can make direct access to specific signals and interface instances, making the UVM-to-signals connection possible. However, our verification engineers soon discovered the enormous convenience of making direct access to signals from code in their test classes. Before long an unfortunate habit had developed of writing the whole of a top-level UVM environment and its test case classes in a module rather than a package. Such test cases readily degenerate into an orgy of raw signal manipulation, wiping out many of the key advantages of an OOP verification methodology.

5.3 Register Modeling

As already noted, the lack of register modeling facilities was a significant drawback for us. The first production release of the UVM will fill this gap with an Accellera-standardized register modeling framework, but we note with some concern that its code generators (which take a description of the DUT's register set, and from it generate SystemVerilog classes and other code to support the model) will be provided by tool vendors and therefore may diverge. Users will be able to minimize that divergence by adopting a widely-supported standard format such as IP-XACT for their register descriptions, but even that format requires vendor extensions to support the full set of register functionality that almost every user will need.

From our experience, we urge future users not to underestimate the work required to integrate and configure any register modeling package, especially if they already have in-house register modeling in place that must be aligned with the new UVM machinery.

5.4 Slave Sequences

The conventional UVM agent architecture of monitor, driver and sequencer works well for passive agents (monitor-only) and for

active agents (stimulus generator using sequences). There is, however, a third and equally important use case: the slave agent. In this scenario the agent's driver is used to drive response values (typically a READY signal, or read data) on to an interface, in real-time response to some transaction on that interface. The UVM agent in this case is acting as a bus slave rather than as a bus master.

In this situation, it is almost always necessary for the response to be controlled in some way by the details of the request – for example, a read cycle should provide data that is controlled by the observed read address. The standard UVM sequencer/driver interface does not support this requirement well. The most challenging problem is that the sequencer should respond in zero time, so that the driver is not stalled in mid-transaction by its sequencer. But this cannot be done reliably, because the TLM connection between driver and sequencer is a blocking one and so is implemented as a task. The base class library's sequencer/driver interface supports this zero-time requirement in a fragile and unsatisfactory way by introducing a number of `#0` delays in its `wait_for_sequences` method. We found it necessary to use nonzero time delays in our driver's synchronous sampling/driving loop, so that the sequencer could always be sure to respond soon enough for the driver to be able to take the response without stalling and therefore introducing an unwanted idle cycle or wait state. This nonzero time delay is extremely unsatisfying (sequences should, ideally, be completely decoupled from details of driver timing) and it probably degrades performance somewhat.

5.4.1 Slave Sequences and Callbacks

The new callback mechanism in the UVM provides an alternative solution to this blocking response problem. We have used callbacks with some success for this purpose. They are easier and more natural to use than the sequence request/response mechanism, which presents many pitfalls for users.

5.4.2 The Problem Remains Unsolved for Users

Callbacks cannot sidestep the problem that getting a sequence item from a sequence requires a blocking task call. It is completely unacceptable for a driver (or a callback) to place such a call and assume that it will return in zero time, even though this is precisely what will happen in most practical situations. The XBus example helpfully provided with the UVM kit presents one possible solution for this issue, but – like most such solutions – it seems clumsy and is not in any way standardized, and does not form part of any written recommendation.

6 HOOKUP TO THE VERILOG HIERARCHY

There are, in essence, two ways in which class-based SystemVerilog code can gain access to signals and other static objects in the SystemVerilog module instance hierarchy:

- Code in any class that is declared within a module or interface can directly access anything declared in that design element, because it is in the same scope. From there it can reach out to anywhere in the Verilog instance hierarchy.
- Virtual interface variables provide a reference or pointer to an interface instance in the Verilog hierarchy. Any class, even if declared in a package, can have a data member of virtual interface type. Code in the Verilog instance hierarchy can then populate that data member with a reference to an interface instance. Code in the SystemVerilog class can now reach through the virtual interface reference and access anything declared in the target interface.

6.1 Direct Access from Classes in a Module

At a glance, the first of these mechanisms seems more flexible because it gives a class fuss-free and unfettered access to the Verilog hierarchy. If the class in question is declared in a module that is near the top of the instance hierarchy – for example, in the test harness – it becomes very straightforward for code in the class to reach down through the hierarchy to any point in the DUT. However, this convenience comes at a very high price: the code so written is no longer portable to even a slightly different verification environment. Consequently, this technique seems to be appropriate only for "one-off" test case code such as occasional driving or reading of DUT signals during debugging. Experience suggests that it is best avoided even in those cases, because of its extreme fragility when details of the DUT or test harness structure are changed.

6.2 Virtual Interface Connection

Virtual interfaces, by contrast, allow for complete decoupling of any UVM class from the signals that it will manipulate. The class can now be placed in a package, with no direct access to signals, but it can reach the real world of SystemVerilog signals through a virtual interface variable. Consequently an instance of the class can now be used with any interface instance whose type matches its virtual interface variable, and it is therefore portable from one verification environment to another.

To use virtual interfaces, though, some additional code is needed. The concrete interface definition should normally be provided as part of the code that is distributed as a UVM verification component, because it is tightly coupled with the virtual interface variable that will reference it. However, the test harness (or similar code) must now include an instance of this interface, with its signals appropriately wired to the DUT signals of interest. (A thoughtful verification component author will have provided ports on the interface to make this task as simple as possible). Finally, procedural code somewhere must make an assignment to the object's virtual interface variable, so that it references the appropriate physical interface instance.

7 AWKWARDNESS OF VIRTUAL INTERFACE CONNECTION

The final step described above, of assigning to the UVM component's virtual interface, is remarkably troublesome in practice and causes much confusion to novice UVM users. The code that constructs the component is likely also to be in a package, so cannot reach into the Verilog hierarchy to find its interface instance. The configuration mechanism can be used, with code in a top-level module creating a wrapper object that is then written into the UVM's global configuration table for later interrogation by the UVM component that needs it. Production releases of UVM will offer a *resources* mechanism allowing some simplification of this chain, but it remains messy with a bewildering range of possible options for the organization of top-level code.

7.1 Embedded Classes Simplify Connection

We have increasingly adopted an alternative approach that eschews the use of virtual interfaces altogether and instead is based on writing a UVM class definition in the body of an

interface. If this embedded class definition is derived from another class that the user has created in a package, it becomes possible to define an API to the class (a set of virtual methods) without requiring any other code to have sight of the embedded class definition. Furthermore, if the common base class is itself derived from `uvm_component` then it automatically has access to the phasing mechanism and so its internal activity can be synchronized with the rest of the UVM environment. Finally, a reference (handle) to this embedded class can easily be obtained by hierarchical reference in the code that launches the UVM test, and then placed into the UVM global configuration table for easy access by other components. Reference [9] describes this approach in more detail.

8 EMBEDDED CLASSES FOR AD HOC DUT CONNECTION

As indicated in section 6, we have begun to adopt an alternative style of connection between UVM classes and the Verilog static hierarchy. After using this approach for signal-level connection to our UVM agents, we also noted that it provides a convenient methodology for making *ad hoc* connections to a DUT or test harness.

8.1 A Very Simple Example

As already mentioned, verification of our DUTs often calls for inspection of the instantaneous values of individual signals that do not form part of a standard protocol or transaction. To provide a compact and simple illustration, we consider the problem of exposing a single-bit status signal to the UVM testbench. This is a good example of real-time information that is too simple to justify the overhead of a transaction, but nevertheless needs to be observed from within the UVM testbench.

8.2 Define the API as a Base Class

Our signal-probing class will appear as a `uvm_component` with a `uvm_event` to notify signal transitions, and an access method `get_value()` to return the current value of the signal. We capture this as a base class `probe_base` derived from `uvm_component`, adding our special access method and event member. This class definition goes in a package that encapsulates our new component.

```
package pkg_probe;

  class probe_base extends uvm_component;

    `uvm_component_utils(probe_base)

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

    uvm_event ev_value_change;
    virtual function logic get_value();

  endclass

endpackage
```

Code Example 8-1

8.2.1 Factory Registration

Ideally, `probe_base` would be coded as an abstract (virtual) class, since it has no implementation of its `get_value()` method and therefore cannot usefully be instantiated. However, we wish to register it with the UVM factory like any other component, and this registration does not work for an abstract class.

8.3 Create Interface with Embedded Class

To provide a physical hook to the Verilog hierarchy we next implement an interface that contains an embedded class derived from `probe_base`, as indicated in Code Example 8-2.

```
interface i_probe (input sig);
  import pkg_probe::*;

  class concrete_probe extends probe_base;

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

    function logic get_value();
      return sig;
    endfunction

    task run();
      forever @(sig)
        ev_value_change.trigger();
    endtask

  endclass

  concrete_probe PROBE;

  function automatic probe_base get_probe;
    if (PROBE == null)
      PROBE = new($psprintf("%m.PROBE"));
    return PROBE;
  endfunction

endinterface
```

Code Example 8-2

It would be equally effective to use a module, but an interface has the advantage that it can be compiled unconditionally (along with the package) with no risk of the simulator instantiating it as an unwanted top-level module if it is not used elsewhere in the simulation. By simple instantiation and port connection, this interface can be connected to any chosen signals in the Verilog hierarchy.

The embedded derived class appears within the scope of the interface definition, and therefore has full access to static properties of the interface, making signal access straightforward. The embedded class provides concrete implementations of virtual methods in the base class. The class is a `uvm_component` and so can use the standard UVM phase methods to build its internal structure, launch a processing loop in its `run()` task and so forth.

Although this class is a `uvm_component`, it is important that it should not be registered with the UVM factory. It will never be created by the factory, and the possible existence of the same-named class in more than one instance of the interface would cause serious problems for the factory's type registration system.

Finally, the function `get_probe()` provides easy access to the embedded UVM component, first creating it as a child of `uvm_top` if it does not already exist. The component is given a UVM instance name that is conveniently related to its Verilog

hierarchy location. This function returns a base class reference. The concrete derived class is irrelevant to the UVM verification environment. This gives a pleasing separation of concerns, with signal connection and manipulation details localized in the interface, but with the functional behavior (API) fully defined by the base class in a package.

8.4 Instantiate and Connect the Interface

To monitor a signal it is necessary to create an instance of the interface we just defined and connect its port to the appropriate signal. For the sake of our example we will assume that the interface is instantiated with instance name `probe_intf` inside module harness. If the signals to be probed are inside the DUT hierarchy, it may be appropriate to use `bind` to create this instance without disturbing existing code at the instantiation site; we discuss this idea more fully in a later section.

8.5 Integrate Using UVM Configuration

Finally we must get a reference to our probe class and pass it to appropriate places in the UVM testbench. The standard UVM configuration mechanism is a perfect fit for this, allowing code outside the UVM class structure to plant a reference into the global configuration table where it can later be retrieved by any UVM component. Thanks to our `get_probe()` function in the interface, this reference can be obtained in a very straightforward way. Code Example 8-3 shows an example of how code in the top UVM module could do this configuration, just before launching the UVM test.

```
module UVM_topmost_module;
  import uvm_pkg::*;

  initial begin
    set_config_object(
      "*.some_component", "signal_probe",
      harness.probe_intf.get_probe(), 0);
    run_test();
  end

endmodule
```

Code Example 8-3

The call to `set_config_object` plants a configuration table entry that will be accessible to any UVM component with an instance name matching `*.some_component`. The configuration entry is named `signal_probe`. To use UVM's automatic application of configuration settings, it is of course necessary that the target component have a data member named `signal_probe` that has been registered using the `uvm_field_object` macro and has the appropriate data type `probe_base`. Through this data member, the target component can easily read the probed signal's value and respond to events on it.

8.6 Advantages of This Approach

This technique for linking UVM classes to Verilog hierarchy is in most respects superior to the commonly described virtual interfaces approach.

- It allows the connection to be configured into UVM with only a single line of code. There is no need to declare a wrapper class derived from `uvm_object` and then encapsulate the virtual interface in it. Instead you are working with an object (the interface's embedded class) that is already derived from `uvm_object`, and therefore can be passed directly to the configuration mechanism.
- It allows UVM phasing to be applied to code within the physical interface. This greatly eases various concerns about the order of construction of objects, synchronization of startup activity, reporting control, and collection of information at the end of simulation.
- It provides a convenient point at which class-based code can be given direct access to signals and other things in the Verilog static hierarchy, without creating a free-for-all of signal accesses at the top level of the UVM testbench.
- It allows diagnostic messages from code in the interface to be properly rooted in the UVM hierarchy, rather than coming from the global reporter.
- Issues relating to type parameters are much simplified because the interface's parameters do not propagate into any data types seen by UVM.

Finally we observe that this design pattern is applicable to any situation in which a connection must be established between a generic UVM component (which itself has no knowledge of where it will connect in the Verilog hierarchy) and the specific structure of your Verilog design and testbench. It works well for typical UVM monitor/driver connections to a set of bus signals, and for access to individual signals as in our example. It also provides a convenient way to add UVM capability to existing module-based verification IP, as described in section 9 below.

8.7 Automatic Register Model Updating

The technique described in this section has also proved valuable in implementing the automatic updating of register model images in response to the value of status signals within the DUT. The name, or other specification, of a register field can be provided as a parameter to the interface instance. Code in the interface then locates the desired field image in a register model, and arranges for it to be updated automatically from the signal whose value the register reflects.

9 EMBEDDED CLASSES FOR LEGACY VERIFICATION COMPONENTS

We have found the embedded derived class approach, as described in the previous section, to be especially useful when integrating existing verification IP into our UVM testbenches. To illustrate this we take a very simple example of a pulse generator BFM written in plain Verilog, shown in Code Example 9-1.

```
module legacyPulseGen
            (output logic sig = 1'b0);

  task uvm_pulse(time tH, time tL = 0);
         sig = 1'b1;
    #(tH) sig = 1'b0;
    #(tL);
  endtask : uvm_pulse

endmodule
```

Code Example 9-1

We first define a suitable UVM API to this legacy module, in the form of a base class.

```
package pkg_uvm_pulsegen;

  class pulsegen_base
                extends uvm_component;

    function new(
        string name,
        uvm_component parent = null);
    endfunction : new
    virtual task pulse (
            time tH, time tL = 0);

  endclass

endpackage
```

Code Example 9-2

Next, as before, we create an interface that contains and instantiates an embedded derived class. In this case the interface has no ports because our API to the legacy BFM requires only task calls, not signal connections.

```
interface i_uvm_pulsegen;
  import pkg_uvm_pulsegen::*;

  class concrete_pg extends pulsegen_base;
    function new(string name,
        uvm_component parent = null);
      super.new(name, parent);
    endfunction

    task pulse(time tH, time tL = 0);
      legacyPulseGen.pulse(tH, tL);
    endtask

  endclass

  concrete_pg PG;;

  function automatic pulsegen_base get_pg;
    if (PG == null)
      PG = new($psprintf("%m.PG"));
    return PG;
  endfunction

endinterface
```

Code Example 9-3

Code Example 9-3 is straightforward, with the exception of the body of the embedded class's method `pulse`. This task has the interesting feature that it calls, by hierarchical reference, task `pulse` in the legacy BFM module. However, it does so by using the module's name, *not* an instance name, as a prefix. Because the code makes no reference to specifics of any instance hierarchy, this gasket interface is completely generic.

9.1 Bind an Instance of the Interface

We now create an instance of this interface inside our chosen instance of module `PulseGen`. Code Example 9-4 shows how that arrangement might appear in a test harness:

```
module TestHarness;

  wire test_pulse;

  legacyPulseGen test_pg(test_pulse);

  bind test_pg i_uvm_pulsegen gasket();

  ...
```

Code Example 9-4

The `bind` statement effectively creates an instance of the interface, with hierarchical name `test_pg.gasket`, as a bound child of the BFM instance `test_pg`. Consequently, hierarchical reference `legacyPulseGen.pulse` found in the interface's `uvm_pulse` task now floats out of the gasket interface and refers to task `pulse` in the specific instance `test_pg`.

9.2 Inform the UVM Environment about the Gasket Object

Elsewhere in the test harness, UVM configuration is used to push a reference to the gasket object into an appropriate UVM component:

```
  initial
    uvm_pkg::set_config_object(
      "*some.uvm.path", "pulsegen_gasket",
      test_pg.gasket.get_pg(), 0);
```

Code Example 9-5

It is now straightforward for code in a UVM component to get a reference to this gasket instance and access its members, including (in this specific example) its `pulse` task.

9.3 More Realistic Applications

This technique for embedding a UVM class in an existing Verilog module provides a very convenient way to make use of existing legacy verification IP. Thanks to the use of `bind`, it is applicable even when you are unable or unwilling to modify the legacy code.

Because the embedded class is a `uvm_component`, its API need not be restricted to simple task calls and signal access. The abstract base class can have TLM ports. In this way, large parts of a UVM agent component can be implemented using existing module-based IP, with a gasket class in a bound interface providing TLM connections such as an analysis port (for a monitor) or a sequence item pull port (for a driver).

9.4 Deployment of This Approach

The author has experimented with this approach (using TLM connections) for integration of legacy BFMs into a UVM environment, with useful results. It is unfortunate that we did not develop this methodology earlier in our UVM rollout process. Numerous UVM adapters were written, by various members of the team, to allow our UVM testbenches to use existing BFMs. There was little coordination of this activity, and therefore almost no consistency of API among the various adapters. Reworking them has, even at this early stage, become a dauntingly large task and it is a source of some regret that we did not identify this convenient and straightforward approach until too late.

10 HUMAN FACTORS

The author finds himself in stark disagreement with the style of presentation of UVM (and, indeed, of other solutions with similar purpose) that says "here is a complete solution; all you need to do is to use it and press these buttons". This approach is condescending to users who, for the most part, are highly skilled verification engineers and programmers. It leaves new users with the problem of learning not only a methodology but a huge body of detail. Implementers of the UVM itself, and users such as the author who are pioneers of UVM rollout within an organization, benefit from the experience gained by implementing large amounts of infrastructure code. This experience is by far the most effective way to learn and internalize the architecture, rationale and details of the base class library and other facilities. By contrast, engineers cast in the role of "UVM user" are presented with a large body of code that is imperfectly documented and that they are expected to use without spending much time investigating its internals. Not surprisingly, many such engineers feel that the methodology is being thrust upon them without their understanding or consent. In this atmosphere users are unlikely to be motivated to make consistent, creative and effective use of UVM. Instead they tend to distort it to accommodate their familiar ways of working, and the human problem is thereby compounded because they gain little benefit from UVM and instead find that it merely presents them with obstacles to achieving results that they could have more effectively obtained an easier way.

11 CONCLUSIONS

This paper has described a number of shortcomings of the UVM, and discussed ways to work around them. In addition it is important to note the imminent release of a production version of the UVM, which will have many new facilities and better usability. Nevertheless, there are some difficulties that must be overcome by users' own efforts, perhaps with the help of techniques described here or perhaps by using other techniques. It is this diversity of solutions that most troubles the author, because it threatens to undermine one of the UVM's most powerful advantages: the promise of true interoperability of verification IP and infrastructure among suppliers and users.

In addition, section 10 highlights some of the challenges faced when deploying the UVM (or, indeed, any other advanced OOP verification methodology) in an organization where not all the engineers are familiar with the underlying techniques. The UVM provides toolkit, documentation and ecosystem to help users with this transition, but despite this our experience shows that adoption is unlikely to be painless. In particular, users who lack prior experience with OOP verification methodology are unlikely to be involved with early deployment of the UVM in an organization, and this puts them at a double disadvantage: not only must they learn a new approach rapidly, but also they must do so without having been not engaged with the rollout and in-house development, leaving them feeling little sense of ownership.

To ease these problems we must find ways to make UVM more accessible, and more exciting and attractive to users. Better documentation will surely help. Exposure of the UVM through conference papers, textbooks and verification IP will progressively bring it into the mainstream of verification culture. There is a pressing need for clearer guidelines on use of the UVM in commonly encountered practical situations and some of those issues have been raised in this paper. Finally there are some shortcomings in the UVM itself, although many of the most important issues (handling of reset and test iteration, register abstraction, interactive command-line access) are expected to be addressed by the production release of the UVM in early 2011.

12 ACKNOWLEDGMENTS

The author wishes to thank his employer Verilab, and many colleagues both at Verilab and at our clients, for insightful and encouraging discussions. Along with many other users, he is grateful to Accellera, Inc. and the members of its VIP Technical Committee for their work in bringing the UVM to fruition.

13 REFERENCES

[1] Accellera Organization Inc Verification Intellectual Property Technical Subcommittee. http://www.accellera.org/activities/vip/

[2] Accellera Organization Inc. Universal Verification Methodology (UVM) 1.0 EA Class Reference. May 2010. Distributed at http://www.accellera.org/activities/vip/

[3] Alsop, T. Accellera's Verification Intellectual Property (VIP) and Universal Verification Methodology (UVM). Accellera Organization Inc, 2010. Available at www.accellera.org/home/VIP_TSC_2010_article_121710.pdf

[4] Bergeron J, Cerny E, Hunter A, Nightingale A. Verification Methodology Manual for SystemVerilog. ISBN 0387-25538-9. Springer 2005.

[5] Cadence Design Systems, Inc; Mentor Graphics, Inc. Open Verification Methodology version 2.0.1. Available from OVM World http://www.ovmworld.org/

[6] Cerny E, Dudani S, Havlicek J, Korchemny D. The Power of Assertions in SystemVerilog. ISBN 978-1441965998. Springer 2010.

[7] Collins-Sussman B, Fitzpatrick B, Pilato CM. Version Control with Subversion. Available at http://svnbook.red-bean.com/

[8] Gran A, Vreugdenhil G, Bromley J. SystemVerilog Virtual Interfaces and Design Verification. User-To-User Conference, Mentor Graphics Inc, San Jose 2008.

[9] Rich D, Bromley J. Abstract BFMs Outshine Virtual Interfaces for SystemVerilog Testbenches. DVCon 2008.

[10] http://www.uvmworld.org/

[11] Valure G. NaturalDocs. Available at http://www.naturaldocs.org/

[12] Glasser, M. Open Verification Methodology Cookbook. ISBN 978-1-4419-0968-8. Springer 2009.