UVM Do's and Don'ts for Effective Verification

Kathleen Meade
Solutions Architect
Cadence Design Systems
meade@cadence.com

Sharon Rosenberg
Sr Solutions Architect
Cadence Design Systems
sharonr@cadence.com

ABSTRACT

With more than a year of production use, the Accellera Systems Initiative UVM is now clearly the methodology of choice for verification. The rush to adopt UVM has both matured the BCL quickly, producing UVM 1.1 and 1.1a bug-fix versions, as well as created a wealth of institutional know-how. Of course, the challenge with know-how is that it tends to be distributed among all the members of the community with little pearls appearing in various forums and contributions. While many sessions introduce the UVM to new users or specific aspects of it for advanced users, the critical tips and best practices are often diffused throughout that material if they are presented at all. So for all of the verification engineers that have been working this year and thought "I wonder if this is the best approach" or "should I use this UVM feature", this presentation cuts right to the answer with specific pointers and code examples, gathered from live projects worldwide, that you can use immediately for more effective verification.

Some of the topics covered in the paper include, but are not limited to, the following:

• Configuration: key features, which verification elements to configure in certain test phases, configuring hierarchically, and more
• How and where to use the objection mechanism
• Best practices for using the register package itself and using it with IP-XACT
• How and where to use TLM2 in UVM

Readers may find concepts in this expert-to-expert paper that challenge their thinking and trigger interesting discussions leading to better utilization of UVM. To further that goal, the code examples discussed in the paper will be contributed to the UVM World for the whole community to use.

Keywords
UVM, Functional Verification, Accellera Systems Initiative, Phasing, Register Package, IEEE 1800, TLM2, IP-XACT

1. INTRODUCTION

The UVM was introduced March 2010 and the verification community declared "We now have one methodology on all simulators. Hurrah!" Soon after books, tutorials, training classes became available and engineers started to apply it. The simulator vendors incorporated UVM in their product portfolios and added support, including debug, to the library.

While the UVM is built on a solid, proven code foundation many users have questions about the new features introduced in UVM 1.0 and 1.1. This paper focuses on those new features.

2. UVM Objection Mechanism

The UVM objection mechanism is designed to coordinate activities such as the test termination. For example, a master agenda may need to complete its entire read and write operations before it can be considered complete. In this example, the various UVM verification components (UVCs) can raise objections assuring that they are able to complete before the master agenda declares that it is done. When those other UVCs complete their work, they clear their objection flags enabling the simulation to continue.

When the objections are implemented, there are a few things to be sure to do. A simplistic way to use objections is to raise these in the pre_body() before starting the sequence and dropped in the post_body(). For easier debugging and possibly better performance, the creation of a sequencer root sequence is suggested. Some users create a base sequence that implements the raise/drop logic and have active sequences extend from this to simplify the derivative sequences implementation. Associating the raise description string with the same string for drop is a good practice for debug. Finally, note that slave agent sequences typically do not object to end-of-test because they are merely serving requests as they appear.

3. UVM Configuration Mechanism

The configuration mechanism is a powerful means to attribute configurations to the verification environment. The mechanism is hierarchical so an upper component can override lower-level values without changing files or having to derive a new component. The configuration can occur at any point in the hierarchy and the values are saved in a side-storage so they can be fetched as needed. Wild cards and regular expressions can be used as can user defined types. With these benefits, the engineer needs to also be aware that the configuration is type sensitive implying that the types need to be communicated carefully between the UVC developer and the end user.

When implementing the configuration, there are some points to note. All of the uvm_config_db functions are static so they must be called using the :: operator. The virtual interfaces should be configured in the connect phase and the uvm_error (not fatal) should be used to get all of the connectivity errors in one simulation run. The exact the same parameter type must be used in the set and get calls or the configuration will not work. For users familiar with the methodologies leading to the UVM, the new API for loosely typed strings should be used. Note that using wildcard can impact the environment performances. Finally, don't use the resource_db interface for hierarchical configuration because db semantics are not fully defined and are still being evaluated for a future UVM revision.

4. UVM Register Package UVM_REG

The UVM_REG memory and register package was introduced as part of the UVM 1.0 release to streamline and automate register related activities. The package derives the register level API from VMM and the use model, register sequenced, register operation items, layering concepts, and more from the UVM_RGM package contributed to UVM World. It is important to note that the SystemVerilog code generators typically associated with a register package are not provided by Accellera.

The register package will meet the needs of most verification environments and it is extensible. It is recommended that users capture register specifications using Accellera IP-XACT standard for portability and to drive the SystemVerilog code generators. The code generators can be obtained from multiple sources including Cadence. It is important to note that the checking and coverage in sequences does not support all of the modes, including passive. Engineers should update and compare design values against the mirror value in the predictor and randomized register values should be copied to prevent collisions.

5. TLM2

A partial implementation of the TLM2 standard is implemented in the UVM to support the high-speed connection to SystemC and to implement both bi-directional ports and generic payloads. For the SystemC connections, the current solution is a partial implementation and does required proprietary simulator support. Future work will be done in Accellera to enable more interoperable

multi-language support. With that said, the bi-directional ports and generic payload is interoperable and should be used as needed.

6. Run-Time Phases

Run-time phases were added to the UVM to simplify the integration of verification IP that require distinct segmentation of the time-consuming portion of the verification run. To enable this, a dozen built-in phases were defined, but the implementation also provides support for complex phasing operations. The API to the more complex operations are the subject of discussion for a future release of the UVM so that API will likely change. If run-time phases are needed, be sure to carefully review the implementation and the project requirements.

7. ACKNOWLEDGMENTS

The authors would like to thank the many users that are working with the UVM for their deep technical discussions and recommendations.