OVM to UVM Migration

or There and Back Again, a Consultant’s Tale

Mark Litterick
Verification Consultant, Verilab GmbH, Germany
Email: mark.litterick@verilab.com

Abstract

Many companies have a goal to migrate to UVM but this must be achieved without disrupting on-going projects and environments. In our current clients we have all possible combinations of migration paths, a desire to move new projects to UVM and a definite need to support OVM for a good few years. This article presents an interesting OVM to UVM migration story where we successfully translated a whole family of verification components from OVM 2.1.2 to UVM, assessed the impact, and then reworked the original OVM code, which was still live in a series of ongoing derivative projects, to make the ongoing translations totally automatic and part of the project release mechanism.

The key was to push all functional modifications possible into the OVM world including code that is deprecated in the UVM like the old sequence library utilities, and to perform minimal UVM work in the very last layer. This approach means that the ongoing project can still endure with OVM, but new projects and near future derivatives can go straight to UVM. In fact the translation layer is such that with only a few limitations we can actually translate back from UVM to OVM in order to deliver VIP to other projects that do not yet have the ability to migrate to UVM.

Keywords – OVM, UVM

Introduction

While it is generally accepted that the Universal Verification Methodology (UVM) is the way forward for modern SystemVerilog based verification environments, many companies have an extensive legacy of verification components and environments based on the predecessor, the Open Verification Methodology (OVM). Furthermore single push migration of all block-level and top-level environments between the two methodologies may be considered too disruptive for most companies taking into account families of complex derivative under development and scheduled for the near future. Several of our major clients are in the transition phase of initiating new product developments in UVM while having to maintain ongoing development in OVM. Furthermore it is imperative for the new development that we migrate the vast majority of existing verification components from OVM to UVM even though the projects providing the VIP remain with OVM for the time being.
This transition between OVM and UVM is represented by Figure 1, which shows overlapping development cycles for multiple projects and derivatives that share a common VIP database. In this case some projects remain on OVM for some time moving forward, but they must supply VIP, environments and tests to UVM for use in the new project families currently evolving in UVM.

This scenario is a bit different to that described in [1], resulting in additional restrictions and goals for our migration flow, that can be summed-up as follows:

- Support ongoing development in OVM for established projects with scheduled derivatives and intense reuse
- Validate all functional changes and enhancements to VIP using regressions environments for established projects
- Provide a path for all VIP to migrate to UVM in order to provide a comprehensive library and support new project development in UVM
- Demonstrate a mechanism to move entire project derivatives to UVM when appropriate for specific projects
- Ideally allow for limited backward translation of some new UVM VIP into ongoing legacy projects and for other OVM-based groups in the company

The process presented in this article was established by performing two full translations from OVM to UVM using a top-level environment with many in-house proprietary OVCs. The aim of this first pass was to establish if there were any major gotcha’s in our code-base and to establish credibility for further work. The purpose of the second pass was to optimize the flow to provide an automatic and reliable translation process that would support our multi-methodology requirement without compromising current ongoing project derivative or future new developments.
Starting Point

It is important to clarify that our starting point for this process enhancement was a family of OVCs that were developed from scratch using a clean OVM-2.1.2 implementation. The OVCs are for fundamentally different application protocols but they have architectural and coding similarities since they were developed together by the same team of people. There was no AVM or URM legacy and no previous code translation or inter-operation compromises.

This has two bearings on the discussion that follows. Firstly, this is probably as easy as it gets! Specifically, the careful adherence to a common coding style and specific attempt to keep the VIP architectures similar for the different protocols means that our translation scripts are more effective and less general-purpose. Secondly, it means that the reader may have to adapt the findings slightly to suit their own circumstances, but we hope that the information provided here will still be of benefit in assisting with that process.

First Pass – Early UVM Translation

For our first pass we performed full translation of all the OVCs and the environment to UVM essentially using the process outlined in the Verification Academy [2]. The main purpose here was to determine if we had any unexpected problems in our database and to establish where the major effort would appear. We wanted to know if we could fully automate the translation process to allow continued development in OVM but reliable delivery to UVM from the parent project.

This process involves executing the ovm2uvm translation script provided with the UVM release early on in the flow as shown in Figure 2.
During the execution of this flow we performed regression checks at each stage of the process, and compilation and sanity checks for each of the minor steps. In order to efficiently convert our whole family of OVCs we generated, or enhanced, automatic scripts for each stage of the process. Some of these scripts were fully automatic as shown in Figure 2, but others required additional work after execution (e.g. all of the items in the OVM deprecated.txt file are identified by a simple script, but the user has to implement any code changes). Figure 3 provides a table of the script setup and usage for the first pass translation.
In fact the translation process using this flow went quite well. We could successfully run UVM-based sanity tests quite early on and validate the subsequent enhancements incrementally. The only real issue was that we had too much work to do after the translation to UVM which would have to be done repetitively since, in our case, the providers of the VIP are live OVM projects.

**Push Back to OVM**

During the execution of the first pass stage it became clear to us that some of the steps in the process could be pushed back into the OVM world so that they were only executed once and did not become part of the translation flow. In particular, because the OVCs were developed with a clean OVM-2.1.2 implementation, the following features are already in the original code:

- only objection handling was used for end-of-test control
- improved message macros used instead of methods
- using back-ported `uvm_reg` in OVM environment

Furthermore we realized that if we invested some effort up front in OVM we could greatly simplify the translation of two key steps, which provided the most manual repair work in our first pass flow:

- modify the virtual interface configuration setup
- deprecate the old OVM sequence utils and sequence library in OVM
Making these changes was also implemented with automatic scripts, followed by manual repair work, but the key point is that this was done only once in the actual OVM project codebase. The modifications then became part of the normal OVC behavior for the live OVM project and all related regression activity. In other words we prepared the OVM codebase for UVM translation as part of our standardized OVM coding style guidelines, this allowed the provider OVM project to continue and simplified subsequent OVM to UVM translation.

Objection Handling

Our original OVCs use objections to handle end-of-test scheduling so there were no issues deprecating \textit{stop()} and \textit{global\_stop\_request()} from the code-base. But one important point to note here is that although we do translate from \textit{uvm\_test\_done} to phase-based objection handling, we currently discourage the use of multiple run phases. The main reason here is that in line with the rest of the industry, we are waiting for stability in the proposed UVM phasing mechanism and in fact we have an adequate and flexible sequence-based phasing solution currently in use where required. One advantage of the sequence-based phasing solution is that it does not put any additional demands on the OVM to UVM translation process.

Message Format

The message macros, which improve the performance of the OVM and UVM messages by evaluating the verbosity settings before calculating formatted string parameters, are already available in OVM-2.1.2 and are used throughout OVCs.

Register Model

During the lifetime of the original project development Mentor contributed a back-ported version of the UVM register model (or register layer) to the OVM community. At that stage we modified our register generation tools to switch to \textit{uvm\_reg} with a specific goal of forward compatibility with UVM. Almost two years later we get payback! We extended the original register model to add several new features, but the translation from OVM to UVM is currently trivial for the register code and user environments.

Virtual Interface Configuration

It was only really at the UVM translation stage that we realized we had an opportunity for improving the virtual interface configuration in OVM in such a way that it would allow for easier conversion to UVM syntax. Specifically we replaced our previous implementation that used an explicit interface wrapper class instance with a much more generic container solution, very similar to that proposed in [3]. By providing static methods \textit{for set} and \textit{get} in the container that closely match the signature of the \textit{uvm\_config\_db} API, we can create an easy translation path for going from OVM to UVM as shown in the following code snippets:
Sequence Library Changes

The original sequence utility macros and sequence library are deprecated in UVM partly because they were badly conceived. Specifically the use-case of having a default sequence and count is not generally appropriate to the way we construct interesting scenarios for most verification environments, especially proactive master agents. Analysis of the solution proposed in [4] led us to realize that in fact we can live without the library setup entirely in OVM and UVM for this family of verification components. This was first tested in UVM and then the scripts were modified to perform the same changes in OVM. The main steps are to replace the sequence and sequencer utils with object and component utils respectively, and to deprecate the default_sequence and count setup. Since many of our sequences access sequencer attributes via the p_sequencer field, we added this to the base sequences for each “library” (i.e. collection of sequences targeted for a particular sequencer). The following Perl script extract provides an example of the level of complexity for the sequence translation step in OVM:

```perl
#!/usr/bin/perl
while(<>) {  # convert all sequencer utils
    s/ovm_sequencer_utils/ovm_component_utils/;
    # convert base_seq and declare the p_sequencer type
    s/\(\$\{1\}\)\`ovm_sequence_utils\(.*\w*\)\W\s*(\w\.*_sequencer)\+$\{1\}\`ovm_declare_p_sequencer\+$\{4\}\n\n\n$\{1\}`ovm_object_utils$\{3\};/
    # convert derived sequences
    s/\(\$\{3\}\)\`ovm_sequence_utils\(.*\w*\)\W\s*(\w\.*_sequencer)+\+$\{3\};/
    # convert remaining seq and utils_end macros
    s/ovm_sequence_utils/ovm_object_utils/;
    # remove all lines with ovm_update_sequence_lib(_and_item)
    s/\s*`ovm_update_sequence_lib\(_\w*\)_and_item\)+\n$/
    # remove all lines with *set_config_int*sequencer*count
    # (needs post tidy of env)
    s/\s*set_config_int\.*sequencer\.*count\.*\n$/
    print;
}
```
After running this script we do have manual repair work to do, but because this is done in the OVM code-base, the repairs are a one-off effort. Specifically we have a few empty methods lying around that we wish to remove for maintenance reasons, but more importantly a few of our OVCs are reactive slaves which generally require a default sequence in order to provide responses to the DUT. In this case we chose to start a default sequence explicitly in the run phase for the slave agent. This run method can be overloaded using the factory mechanism by tests that wish to select a different default sequence for the slave (e.g. an error response sequence) or call-backs can be built into the driver for error injection if required.

Second Pass – Late UVM Translation

With the major functional changes successfully implemented in the OVM source we are now in a better position to handle fully automatic OVM to UVM conversions using the flow shown in Figure 4.

![Figure 4: Full Process for Late UVM Translation](image)

Notice that Figure 4 shows the full process, but in fact the initial steps (inside audit OVM code) were only performed once, for all the OVCs in the original OVM source and never repeated. Therefore these steps are part of the overall migration to UVM but not part of the translation process; in fact the audit step can now be replaced with a script that simply identifies any possible hazards to the user. In addition convert steps do not need any significant manual repair.
work and can therefore now be bundled into a combined script. The resulting conversion from OVM to UVM can now be performed automatically as shown in Figure 5, whenever a new version is released in the OVM domain.

![Figure 5: Slim OVM to UVM Translation](image)

**UVM to OVM**

With such a slim OVM to UVM conversion process it is possible to re-engineer the scripts to provide translation in the opposite direction. This potential path is required at some clients to support delivery of UVM verification components back into an OVM-based project. In order to validate the feasibility of this approach we successfully modified scripts and back-ported a couple of UVCs that originally came from OVM and managed to compile and use them successfully. What we have not attempted at this stage is to take a pure UVC, which was conceived in UVM, and apply these same scripts to translate it to OVM. Obviously this path is less critical than the forward path from live OVM to UVM and is therefore a lower priority; nonetheless analysis suggests it is possible to back-port (like the UVM register model) provided the following limitations are considered.

**UVM Limitations**

If the OVM to UVM transition period is expected to endure for a long period of time (or many project and reuse lifetimes) then it is worth considering limiting the use of the following UVM constructs to ensure backward compatibility:

- avoid run-time phases (use sequence-based phasing instead)
- avoid TLM 2.0 (or use only where required and encapsulate well)

Note that the goal here is not to cripple the UVM implementation capability, but rather provide a relatively straightforward back-porting path for UVCs which have been identified as valuable VIP in the planned derivatives for the OVM-based environments during the transition phase. Depending on the prevailing conditions at your company, these guidelines may not be required at all. Some of these points also reflect a state of flux in the UVM user community (e.g. the run-
time phase discussions) and should probably be avoided until a consensus or more flexible solution is achieved.

Other improved aspects of UVM (such as the modified end-of-test objection handling using phases, updated phase methods, command line processor, and config.db changes) will require a little more effort to translate back to OVM but should be used as standard practice for UVM development.

References