Adiel Khan, **Justin Refice,** **Warren Stapleton**

Version: Final 1.0

OVM TO UVM DEFINITIVE GUIDE PART 1

Abstract- The Universal Verification Methodology (UVM) is an industry standard maintained and developed by the Accellera technical subcommittee for verification intellectual property (TSC-VIP). The UVM standard consists of SystemVerilog IEEE 1800 compliant reference library along with a technical reference manual and a VIP engineer's user guide.

UVM is constantly maturing and as OVM based projects come to end everyone is asking how they should move to use UVM for the next project. At AMD there is a long term strategy to migrate all projects using OVM code to UVM. Based on the knowledge within the methodology team there has been several successful SoC code-upgrades each containing numerous subsystems with dozens of IP block projects.

During OVM's short life span, it went through a multitude of revisions and lots of bug fixes. Many other issues and bugs were never resolved in OVM which created divergence in the user community as verification teams created their own solutions. UVM not only fixes several open bugs from OVM but also draws from VMM to address many of the basic features required for verification that were left unanswered by OVM.

The definitive guide is split over two parts. This paper is part one, a practical aid for the users who are upgrading from OVM to UVM. There will be a subsequent paper namely "Part Two" which addresses methodological modifications required to upgrade from OVM to UVM.

I. INTRODUCTION

When upgrading from OVM to UVM there is more than just syntax and semantics that need to be modified. In this paper we will discuss for each major section of the UVM's base classes how to upgrade syntax from OVM to UVM, covering explanations as how to move forward with UVM.

The starting point of UVM code base was created by an automation script that performed a renaming of keywords, from the OVM base class library. The script was the first OVM to UVM renaming technology developed and was donated to Accellera. Subsequently, Accellera members have enhanced, modified, rewritten and renamed the original script, `OVM_UVM_Rename.pl` to `ovm2uvm.pl`. Unfortunately, these scripts quickly became outdated as UVM moved forward and they cannot take code from OVM to UVM-1.X version. There are other scripts and technologies in the industry that can automate much of the process. However, automation is not a substitute for knowing what and why changes are occurring. It is not the purpose of this paper to provide an automated script, but instead to provide the user with the knowledge as to what such an automated script might be doing.

The UVM library proved a switch to the user to ensure that users have not accidently overlooked an area which is deprecated or targeted for deprecation. `+define+UVM_NO_DEPRECATED` is a compilation option that enables error messages if deprecated features are used.

II. PRIOR TO OVM-2.1.2

The OVM-2.1.2 release notes list deprecated features in OVM. These need to be resolved before converting to UVM. Any items marked as deprecated prior to OVM-2.1.2 will not be available in UVM. As OVM went through rapid churning, not all projects were able to keep up to date and not all changes were easily identifiable as deprecated features or required code modification. Whilst many of the OVM deprecated features were indicated at runtime by a specific OVM_DEPRECATED warning message there were some that were not so easily identifiable by the end users. To ensure OVM migration to UVM was as smooth as possible AMD methodology team created OVM_2.1.2_TRANS version which was OVM-2.1.2 but with all deprecated features removed. This enabled project teams to identify areas where their code was using features not available in UVM.

A. *Typical deprecation issues not enforced in OVM*

For completeness this section outlines some of the more prevalent issues a user would need to resolve prior to upgrading from OVM to UVM. The issues include modification to the phasing API, the removal of semi-colons to macros and the replacement of ovm_threaded_component objects along with vendor specific classes.

1) Threaded Classes

In older versions of OVM library there was an `ovm_threaded_component` object that could be used by people to derive their own custom VIP components. The `ovm_threaded_component` was originally the class that held `run()` mechanism. By the time of UVM and OVM-2.1.2 the `ovm_threaded_component` had been completely deprecated and functionality merged into `ovm_component`.

2) Reporting

An area that inevitably touches every part of a verification environment is the standard output text API. It has been well discussed about the limitations of using $display() within your code as it can add overhead for regression and lacks the controllability for different users. So, it was quite natural for users to want to adopt a better reporting system as provided by OVM. Most users got caught between different version of OVM having slightly different API to use as an alternative to $display(). i.e. `` `message() ``, `` `OVM_REPORT_INFO ``, `ovm_report_info` or `` `ovm_info ``. Also note, `` `message() `` was not part of OVM and will be discussed in vendor specific section.

a. Messages with Severities

By the time of OVM-2.1.2 users were expected to be using `` `ovm_info() ``, `` `ovm_warning() ``, `` `ovm_error() `` and `` `ovm_fatal() `` macros for reporting information. The capitalized variants were deprecated prior to OVM-2.1.2;

```
`OVM_REPORT_[info_WARNING|ERROR|FATAL]
```

This in UVM now becomes:

```
`uvm_[info_warning|error|fatal]
```

b. Message Macros with semi-colon

The semi-colon is a line terminator in Verilog and can cause functional differences in the code if not used correctly. For instance a semi-colon placed at the end of an already terminated line after an if-else clause will cause the "if" to be terminated unless it is followed by a "begin" block. Therefore, it is important to understand whether or not a semi-colon is included as part of the macro expanded code. Here is the code in question:

```
if (enter)
  if (somevar)
```

```
    `ovm_error(…);
  else ...
```

The issue in the above code is that the "`if (somevar)`" clause may be terminated by the semi-colon after the ovm_macro call.

Let us rewrite the above as some classic Verilog code.

```
if (enter)
  if (somevar)
    $display("ERROR Message output");
//`ovm_error(…)
    ;   //bad semi-colon after `ovm_error
  else ...  //Else is associated with "enter"
and not with "somevar"
```

Therefore,

```
`ovm_[info|warning|error|fatal](…);
```

becomes:

```
`ovm_[info|warning|error|fatal](…) `ifndef
BAD_semi ; `endif
```

3) *Sequence and Sequence_item Constructor*

The sequence and sequence_item constructor in OVM had 3 parameters passed in 2 of which were deprecated

i.e.:

```
ovm_sequence::new(arg1, arg2, arg3);
ovm_sequence_item::new(arg1, arg2, arg3);
```

Sample code is:

```
ovm_sequence::new(name, sequencer=null,
parent_sequence=null);
```

In UVM the constructor now only has the one supported parameter so the code becomes:

```
uvm_sequence::new(name);
```

4) *Phasing*

a. Terminating the Simulation

In OVM each component had a global_stop_request() method which has been removed from UVM. In UVM a global level (not component level) global_stop_request() method is present but marked as deprecated. The preferred mechanism in UVM is for components to participate in phasing and object to the current phase being executed from ending until the component no longer requires that phase to be executing. More on this topic in the methodology improvements section later in this paper.

i.e. OVM syntax:

```
mycomp::global_stop_request() or mycomp_inst.
global_stop_request();
```

Making the UVM syntax:

```
global_stop_request(); // to be deprecated
```

Component level global_stop_request() was deprecated prior to ovm-2.1.2 and removed in UVM from component into root space. In turn global_stop_request() in the root space is marked for deprecation.

b. Deprecated Phases

As OVM matured and learned what users needs were, several phases got deprecated along the way. Most notable `post_new()`, `configure()` and `pre_run()` phases should not be used in OVM-2.1.2 and are

completely removed in UVM-1.X. The expectation is that the functionality executed in these deprecated phases can be moved into a supported phase in UVM. For instance, `post_new()` functionality can often be moved into UVM's `build_phase()`. Similarly, `configure()` functionality can be collapsed backward into `connect_phase()` or it can pushed forward into `end_of_elaboration_phase()`. Lastly, procedural code executed in `pre_run()` can be mapped into `start_of_simulation_phase()`.

B. *Vendor Specific code in OVM*

There were numerous parts of users usage of OVM library that relied on code from vendor specific solutions. These have been removed so that UVM is vendor neutral and doesn't contain legacy non-agreed code in the standard. You may have some leftover pieces from a vendor specific library or vendor specific debug facility.

In UVM the transaction recording has been abstracted away from the base-class library and EDA vendors have an API which allows them to provide tool-specific transaction recording without affecting the UVM library code. Historically, many OVM users migrated from the now defunct URM or AVM methodologies.

As noted previously you may have unknowingly been using reporting features like `` `message() `` and `` `dut_error() `` which were never actually part of OVM. `` `message() `` and `` `dut_error() `` should be replaced with `` `ovm_info() `` and `` `ovm_error() `` respectively. It is important that your code be clean of all `urm_*` and `avm_*` code before attempting to upgrade to UVM. In a similar but much more subtle vain there was a macro that had the `ovm` prefix but clearly leveraged the `urm` code infrastructure, although it may not have been quite so clearly stated as deprecated in OVM it definitely had no place in UVM. The macro in question is `` `ovm_msg_define(<verbosity_arg>) `` which users performing the upgrade are expected to replace with `ovm_report_enabled(<verbosity_arg>)` prior to moving to UVM. The main functionality is to test what level the verbosity has been set to and the result will return "`1`" if the configured verbosity is greater than the `verbosity_arg` supplied. Another popular method that is should be kept upto date with OVM advancements to reduce issues in migrating to UVM is `set_global_verbosity()` which in OVM-2.1.2 becomes `set_report_verbosity_level_hier()`.

III. OVM TO UVM

One could imagine since UVM was derived from an OVM code base that by performing an "O" to "U" translation on all scripts and code should suffice to enable UVM compilation rather than OVM. This is not true as OVM had some files and members that were not prefixed with `ovm`. Many of these discrepancies have been addressed in UVM. However, this does mean that the user needs to upgrade their scripts and top-to-bottom code usage appropriately.

In the following section let us review what is required to have correct UVM functionality for each of the major building blocks within the UVM library. This covers the general base class usage, the infamous configuration and factory schemes, a review of the reporting structure, what is involved in switching to UVM phasing, leveraging the new UVM sequence infrastructure and lastly modification in the TLM domain.

A. *Base Classes*

For the majority of the classes you will find `OVM_` or `ovm_` has become `UVM_` or `uvm_` respectively. The leads us to look at areas where simple conversion is not possible such as how users access the library code, changes in

temporary workarounds, how access to the top level arrays have been modified, discrepancies in macros that have been fixed, and modification to the comparator operations that could catch you out.

1) Class Inclusion Scheme

The first place to start with the OVM is how users were accessing it from their verification code. A large portion of users were using `` `include "ovm.svh" ``. Well in UVM there is no `uvm.svh` and if you pull in uvm.sv then there will be duplicate entries of the uvm library compiled. The correct mechanism is to use `import uvm_pkg::*;` which also enables users to pre-compile the library into its own precompiled library for all teams to use. Of course package imports do not bring in macro definitions therefore `` `include "uvm_macro.svh" `` is expected where macro usage occurs.

2) Parameter Class Workaround

The file `ovm_template.svh` was used to workaround simulator deficiencies. OVM usage of this file should have been restricted to simulators which did not fully support template types in separate scopes. The file enabled template objects to be available in multiple scopes where the definitions and specializations did not need to be shared between the scopes. For UVM this file does not exist and all references in the users code must be removed.

3) Top_levels Array

The Top_levels array is very useful for viewing all the classes that are used in your UVM testbench. If a uvm class has been constructed and not passed a "parent" hierarchy handle then it will be a top_level component. The access to the `top_levels` array is now contained within the `uvm_root` object as a static element. Whereas in OVM one could access using `uvm_toplevels`, now in UVM the access is `uvm_root::top_levels`.

4) Field macro usage

OVM, unintentionally, supported the declaration of an enumeration as `int` within the field automation macros.

ie:

```
typedef enum {FOO, BAR} foo_e;foo_e my_foo;
`ovm_object_utils_begin(my_object_type)
    `ovm_field_int(my_foo, OVM_ALL_ON)
`ovm_object_utils_end
```

UVM has specializations within `` `uvm_field_int `` which makes this code erroneous. The corrected code becomes:

```
typedef enum {FOO, BAR} foo_e;foo_e my_foo;
`uvm_object_utils_begin(my_object_type)
   `uvm_field_enum(foo_e, my_foo, UVM_ALL_ON)
`uvm_object_utils_end
```

5) Comparator

It is generally accepted that the actionable methods in UVM users need to override should be `do_*()`. In OVM there was a `comp()` method that users could override for performing the compare functionality. With UVM one should override `do_compare()` rather than `comp()` when using `uvm_class_comparator`.

i.e: `ovm_class_comparator::comp();`

is now: `uvm_class_comparator::do_compare();`

B. *Configuration and Factories*

The OVM mechanism for applying configurations has been significantly revamped and improved in UVM. At the time of writing the OVM `set/get*()` API had been requested to be deprecated. Hence in the later section of upgrading methodological usage we will discuss how to move from the old OVM API to the improved UVM API. In this section we will cover the minimal subset of changes required to make OVM configuration and factory API's operate with UVM code.

1) Set/get config_object()

There was an ambiguity that required clarification between OVM documentation versus OVM code versus UVM documentation versus UVM code implementations of `set/get_config_object()` API. The following is an amended extract from Accellera's UVM bug tracking system Mantis with reference bug number 3731.

http://eda.org/svdb/view.php?id=3731

In OVM-2.1.2, cloning would occur on a `get_config_object()` only if the clone bit for **and** the corresponding `set_config_object()` were set to 1. This prevented inadvertent cloning when `get_config_object()` was called without specifying the clone argument, which defaults to 1. One could argue the default for clone should have been 0.

Set	Get	affect
0	0	no cloning at all
0	1	no cloning at all
1	0	cloning on set
1	1	cloning on set (useful for multi-gets)

In UVM-1.0, the clone bit for the set is not saved, so the truth table is different, and not backward compatible.

Set	Get	affect
0	0	no cloning at all
0	1	**cloning on get**
1	0	cloning on set
1	1	cloning on set, **cloning on get**

Thus discussion of Mantis 3731 ensued. Resulting in, auto-configuration method, `apply_config_settings()` being fixed in UVM-1.1b to honor the clone bit provided with the `set_config_object()` method per the Accellera standard specification. The `get_config_object()` has intentionally been left as implemented in UVM-1.0 (not following the Accellera standard specification) in order to preserve current library implementation semantics.

2) Printing Factory & Override

Code modifications that were performed in other areas of OVM were also applied to the factory section. Some areas are high likely to affect users such as the modification in how to print the factory and how to print all the overrides in the factory. Previously, an OVM user would have called `ovm_factory::print()` and to output all the override, `ovm_factory::print_all_overrides()`. Whereas now in UVM the code becomes `uvm_factory::get.print()`.

To perform override calls to enable the factory to replace objects the API has gone from `ovm_factory::set_type_override[_by_type|_by_name]`

Thus in UVM becomes:

```
my_obj::type_id::set_type_override[_by_type|_
by_name];
```

C. *Reporting and Controlling Messages*

1) Command line message control

Converting OVM Command line options cause UVM warnings. The format of the example below is an incorrect translation from "`O`"vm to "`U`"vm. The verbosity options are reordered under UVM.

i.e:

```
+uvm_set_verbosity=start_of_simulation,uvm_te
st_top.t0.ioe_s.slave[0].*,UVM_HIGH
```

becomes :

```
+uvm_set_verbosity=uvm_test_top.t0.ioe_s.sla
ve[0].*,_ALL_,UVM_HIGH,start_of_simulation
```

2) `uvm_info inside of sequence static methods

In OVM, sequences did not have implementations of the report methods, and as such relied soley on the ovm_pkg scope functions.

In UVM, sequences do implement these methods, allowing for `` `uvm_info() `` (et al.) to be called from within the sequence without needing to point to the sequence's parent sequencer. These implementations are not static however, which means that any static methods defined inside of a sequence must not use `` `uvm_info `` calls.

3) Reporting Classes

The `ovm_reporter` has been superseded by `uvm_report_object` and the `ovm_report_global_server` class is replaced with a proper implementation of a singleton in the `uvm_report_server` class. The following statements

```
ovm_report_server srvr;
ovm_report_global_server glbl= new();
srvr = glbl.get_server();
GLBL.SET_SERVER(MY_SRVR);
```

should be modified as follows:

```
uvm_report_server srvr;
srvr = uvm_report_server::get_server();
uvm_report_server::set_server(my_srvr);
```

4) Topology Printing

The printing of the topology is now scoped to the uvm_root object constructed as uvm_top. Therefore, `ovm_print_topology()` now becomes `uvm_top.print()`. The same modification needs to be performed for `ovm_enable_print_topology()` for it to become in UVM `uvm_top.enable_print_topology()`.

Based on component context this should be reviewed by user to ensure valid modification is performed during upgrade.

5) Printer API's

Several printer knobs are deprecated and no longer have any effects.

In OVM the printer knobs were located in:

```
ovm_default_printer.knobs
```

these in UVM 1.0 became:

```
uvm_printer_knobs
```

subsequently in UVM 1.1 the printer knobs were reviewed and many where removed, such as:

```
uvm_printer_knobs::max_width
uvm_printer_knobs::truncation
uvm_printer_knobs::name_width
uvm_printer_knobs::type_width
```

```
uvm_printer_knobs::value_width
uvm_printer_knobs::sprint
uvm_component::print_config_settings
```

One of the knob members was renamed from:

```
ovm_printer_knobs::global_indent
```

to

```
uvm_printer_knobs::indent
```

The `uvm_component::print_config_settings()` method is deprecated in favor of the method `uvm_component::print_config()`.

The following statement:

```
comp.print_config_settings("", null, 1);
print_config_settings("", comp, 0);
```

Hence, should be modified as follows:

```
comp.print_config(1);
comp.print_config(0);
```

The other notable point with printers is that in OVM the `down()` method had two arguments whereas in UVM there is only one argument. See the following example:

```
printer.m_scope.down(get_name(), null);
```

Is now in UVM as:

```
printer.m_scope.down(get_name());
```

D. *Phasing*

UVM provides a run time switch allowing the phasing to use the scheduling as ordained by the OVM run_phase() , the switch is `+UVM_USE_OVM_RUN_SEMANTIC`. Usage of this switch should be temporary to ease the migration process, once fully conversant in UVM the phasing should be implemented correctly and remove usage of the switch. An obvious change that you will notice is that the phase names have been enhanced to explicitly state <method>_phase and pass the phase_name as an argument to the method. By doing so it give easy access to phase methods such as objection interactions. This correlates with how various timeouts have been modified.

1) *Phase names, calls and super accesses.*

Even though the same OVM phase methods are in the UVM library they are marked for deprecation and it is strong advised that all users implement using the new phase names and API.

This means OVM code

```
<phase_name>();
```

Becomes:

```
<phase_name>_phase (uvm_phase phase);
```

The signature of all pre-defined phase implementation methods are modified, `build`, `connect`, `end_of_elaboration`, `start_of_simulation`, `run`, `stop`, `extract`, `check` and `report`.

Due to the change in UVM if there was code in the environment that was directly calling a phase method such as `<mycomp>.build()` this will now cause an error condition to be signified. Direct calls to `<anyusercomponent>.build()` are illegal and should be removed from the code.

Similarly, the phase name modifications need to be applied to all `super.<phase_name>()` where there are intermediate classes used for derivation functionality

containment these now need to use `super.<phase_name>_phase (phase)`.

2) Timeouts and Termination Criteria

Timeouts were implemented in OVM and in UVM to prevent endless runaway simulations. They were never intended to be used as functional mechanisms to determine when a users verification of a design was complete. Moving from OVM to UVM the timeouts were streamlined, the OVM `set_global_timeout` and `ovm_top.stop_timeout` were removed from UVM in preference for the objection mechanism. Also, the `` `UVM_DEFAULT_TIMEOUT `` which in OVM was typically used as "`` `UVM_DEFAULT_TIMEOUT - $time `" still exists although the usage now is simply `` `UVM_DEFAULT_TIMEOUT `` and it will issue a `UVM_FATAL` message when reached from UVM-1.1c onwards.

Users wishing to prolong or terminate their testbenches must use the objection mechanism to determine when the simulation should end. There has been a change in the syntax which for OVM use to be: `ovm_test_done.[raise|drop]_objection(args1)`

Now with UVM-1.X the syntax, specifically when within a phase method context has become:

```
phase.[raise|drop]_objection(args1);
```

The *hammer* approach of `global_stop_request()` has been deprecated as it was not reusable and caused many issues when integrating code that incorrectly and ungracefully terminated the simulation. Later, in the methodology section we will cover a full example of how to properly terminate the simulation.

E. Sequence Control and Data Objects

The sequence API in UVM has been improved by adding more debug features and aligned the methods with other parts of UVM also removed unnecessary variables such as `count`. The OVM sequence library had several problems and in UVM-1.1 there is a new sequence library that solves many of the use and reuse factos faced previously.

1) Sequence Raise Objection with debug

UVM add a description field to the raise and drop objections functions. This provides a convenient way to view and debug when there are multiple interleaved objections within the simulation. The OVM syntax was:

```
raise_objection(object, count);
```

```
drop_objection(object, count);
```

Whereas UVM uses:

```
raise_objection(object, description, count);
```

```
drop_objection(object, description, count) ;
```

2) Sequence Objections

To correctly upgrade the syntactical usage of objections within sequences one must also address how they are used. In OVM the policy was to leverage the ovm_test_done object and use that as a container to raise and drop the objections controlling the end proloning or termination of the simulation. That is in OVM syntax one would have coded :

```
uvm_test_done.[raise|drop]_objection(<args>)
```

Whereas in UVM the policy is to raise and drop objections per phase which makes the syntax usage within a sequence as follows:

```
if (starting_phase !=null)
starting_phase.[raise|drop]_objection(<args>)
```

This direct translation between what is expected syntax previously to what is UVM syntax does not address the

complete usage. Most users had implemented a style of objection usage that one can call pessimistic mode. In many verification environments the objection usage is per-sequence placed, at worst within the sequence or more common within `pre/post_body()` for raising then dropping objections. It is probably overkill and engineering paranoia wanting to ensure each sequence can prolong or protect against simulation terminating during sequence execution. The code can be optimized by refactoring for an optimistic mode, whereby only the highest level sequences need to raise/drop objections from within `pre/post_start()`. For the pedantic, prior to UVM-1.1 it would have been within `pre/post_body()`. See section on body() callbacks in the methodology upgrade chapter to learn about how `pre/post_body()` are now callbacks for every sequence and `pre/post_start()` are only root sequence activated.

3) Sequence Item Port Connections

The pseudo TLM-1 OVM sequence API has been cleaned to align more closely with other API's within UVM. Namely the `*_if` has been dropped meaning OVM API's:

```
seq_item_prod_if, seq_item_cons_if,
seq_item_port.connect_if
```

Translates to::

```
seq_item_port, seq_item_export,
seq_item_port.connect
```

4) Data objects for constrained random verification

Some parts of OVM were more trial and error than driven by definitive users criterion. One such area was the creation of `ovm_transaction` class, it was considered useful at one point to have a lower level object that did not have the overhead of `sequence_item` members yet could be used for transaction recording and derived from for monitoring purposes. At the time of UVM review it was noted that overhead of using `sequence_item` object rather than `transaction` was minimal and created less divergent code as most users would anyway derive from `sequence_item` to add in their object members required for constrained random and coverage driven verification. The `ovm_transaction` class is deprecated and should be replaced with `uvm_sequence_item`. The following statement

```
class my_trans extends ovm_transaction;
```

should be modified as follows:

```
class my_trans extends uvm_sequence_item;
```

Some advanced users had been accessing non-public API's of the OVM library. The area that seemed most common was the usage of `` `OVM_FIELD_DATA `` within data objects. This macro is deprecated from UVM however with some reviews of the macro usage it is normally possible to use `` `M_UVM_FIELD_DATA `` instead

5) Sequence, Sequencer Automation and Relationship

As part of the review and improvement process when creating UVM the sequence utility macros were enhanced for a more flexible and complete usage model. Fristly, the sequence and sequencer automation macros now align with their internal base-class types of `uvm_object` and `uvm_component` respectively. Secondly, the separation of the sequencer registration and the sequence data automation gives the sequence the freedom to not be tightly coupled with a single sequencer. In OVM for the sequence one would have written the syntax:

```
`ovm_sequence_utils(arg1, arg2)
```

Now with UVM-1.X the syntax becomes:

```
`uvm_object_utils()
`uvm_declare_p_sequencer(arg2)
```

Similarly, for the sequencer the OVM syntax was:

```
`ovm_sequencer_utils
```

This in UVM becomes:

```
`uvm_component_utils
```

6) *Default Sequence, counting and the sequence library*

OVM had a `default_sequence` which was set to `ovm_random_sequence` as default. This allowed users to use the internal count variable to control how many `sequence_items` were generated. UVM has enhanced the usage of `default_sequence` to be per runtime phase aware and it no longer defaults to `ovm_random_sequence`. Therefore, there is no longer a requirement to have the `count` variable within the library. If users require a `count` variable it would belong in their sequences outside of the library code. In summary code such as `set_config_int(<sqcnr>, "count", 0)` is no longer valid and should be removed.

To configure the default_sequence per phase there are a variety of techniques available within UVM. If a user in the OVM code had the following:

```
set_config_string("<hier_to_sqncr>",
default_sequence, "<seq_type>");
```

They could use the UVM configuration database:

```
uvm_config_db#(uvm_object_wrapper)::set(this,
"<hier_to_sqncr>.run_phase",
"default_sequence,
<seq_type>::type_id::get());
```

Alternatively use the factory to find the type:

```
uvm_config_db#(uvm_object_wrapper)::set(this,
"<hier_to_sqncr>.run_phase",
"default_sequence,
factory.find("<seq_type>"));
```

Or supply path of a sequence rather than factory:

```
myseq_type  my_seq=new("my_seq");
uvm_config_db#(uvm_sequence_base)::set(this,
"<hier_to_sqncr>.run_phase",
"default_sequence", "my_seq").
```

Occasionally, it is better reuse and readability to have `seq.start()` in each of your testcases to start different sequences per test:

```
myseq_type my_seq= new("my_seq");
my_seq.start(my_sqcnr, null);
```

This leads directly onto how to categorize sequences into a library for reuse. In OVM there were many caveats and tight coupling between objects preventing arbitrary reuse of sequences and sequence libraries across arbitrary sequencer. The OVM sequencer based sequence library is deprecated and replaced with a more reusable implementation. As shown above sequences can be spawned by simply calling `start()`. Also, see the methodology section on using UVM-1.x sequence library. The main code that affects users is the removal of the sequence library macros in OVM, which means the code must be removed to work with UVM

```
`ovm_update_sequence_lib() and
`ovm_update_sequence_lib_and_item(<arg>)
```

F. *TLM API inclusion and usage*

The TLM directories and files were an area where OVM did not use `ovm_` or `OVM_` nomenclature. This was addressed by UVM therefore OVM syntax `tlm_*` or `TLM_*` has been fixed in UVM to become `uvm_tlm_*` or `UVM_TLM_*`. User including, importing or using the TLM files and features of OVM would need to modify their code to use the corrected UVM prefix nomenclature with TLM.

The UVM library also addressed the OVM `export_connections*` and `import_connections*` by deprecating these and replacing them by having calls within `connect()`.

IV. CONCLUSION

Knowledge of what, how and where modification are required de-risks the migration process from OVM to UVM. At AMD and within Synopsys there are technologies and resources that make the OVM to UVM transition painless and can even be done unbeknown to the user. For AMD the UVM upgrade is a long term strategy that needs to coincide with project specific needs and requirements. The process has been proven and well understood thereby enabling users to take advantage of the OVM bugs addressed by UVM and the new features UVM brings for advanced verification.

V. ACKNOWLEDGEMENTS

The authors would like to acknowledge the contributions of John Fowler, who architected and implemented much of the AMD transition kit. Also, Janick Bergeron provided significant material for the definitive guide. Numerous AMD, Synopsys colleagues and various companies aided by qualifying the scripts developed preceding this paper. The scripts were coupled with the OVM to UVM upgrade guide and with the UVMKit, to all of those users involved we owe much appreciation as they drove the requirement for creating this paper. Also a huge thank you must go out to all Accellera members whom are continuing to strive for a better UVM that can fulfill many users verification needs.

VI. REFERENCES

1. UVM-1.1.c User guide by Accellera
2. UVM-1.1.c Accellera source code
3. OVM-2.1.2 Release notes by OVM team
4. OVM-2.1.2 VCS source
5. AMD internal TWiki by John Fowler & Justin Refice
6. OVM to UVM Upgrade guide by Adiel Khan
7. http://eda.org/svdb UVM Mantis by Accellera