A New Epoch is beginning: Are You Getting Ready for Stepping into UVM-1.2?

Roman Wang, +8613482890029, Advanced Micro Devices, Inc., Shanghai, China (roman.wang@amd.com)

Uwe Simm, +49 89 4563 1825, Cadence Design Systems, Munich, Germany (uwes@cadence.com)

Abstract— UVM has experienced great adoption and been a tremendous success throughout the verification industry since the first release of UVM in early 2011. UVM was proved a 486% strong growth in adoption between 2010 and 2012 based on study of the Wilson Research Group. UVM skills are usually a plus or even a must when hiring of verification engineers. Today, it's the defacto standard for any functional verification activity. A key point for UVM is that a lot of focus is being put upon backward compatibility for any change applied to the UVM core library. Lots of enhancements to the library have been deferred in order to avoid migration problems in the past.

In 2014, UVM methodology starts a new epoch and takes another step forward with the next version of UVM labelled 1.2. UVM-1.2 is the first release since the initial release of UVM-1.0 which will have new facilities, enhancements and capabilities which are not fully API compatible with the older UVM versions. In addition, the UVM working group is also working on push the UVM methodology to the IEEE in the future.

This paper will not only share notable changes in UVM-1.2, but also point out why the changes were made and how do the changes affect the way we write UVM test benches. It's to help UVM end users to better understand the differences between UVM-1.2 and UVM-1.0/1.1, and estimate the effort/path to migrate to UVM-1.2. Debug is always the biggest pain, and we will introduce the UVM1.2 generic debug capabilities to address such problem. Migration may introduce big effort and some problems, and for that we will share the migration experience from UVM-1.1 to UVM-1.2 based on our experiment. All verification engineers (from those just starting with UVM to those with years of experience) will gain new knowledge from this paper with the practical patterns.

Keywords— University Verfication Methodology (UVM), Debug, Migration

I. INTRODUCTION

In 2009, The Accellera formed Verification IP technical subcommittee (VIP-TSC), its first job is developing VMM and OVM interoperability and to make them work together. With adoption of UVM over years, it becomes an ecosystem for verification, including methodology, low power, formal, hardware accelerating and mixed signal area, etc. So everyone could contribute UVM to make it move forward from different scope. It becomes outside of the magic box in terms of a methodology. So Accellera changes the name to UVM working group now. It now focuses on developing and releasing UVM, but not necessarily focuses on IP reuse as original intent of committee.
UVM methodology had experienced great adoption and growth throughout the industry for more than three years, and it effectively guides users on how to build a reusable, scalable test bench architecture by components, sequences, TLMs, etc. and how to control the simulation flow by phasing. In 2014, UVM methodology takes another step forward with the next version of UVM labelled 1.2 which will have bug fixes, performance fixes, cleanup, new facilities, enhancements and capabilities NOT API compatible with the older UVM versions. There are about 90 mantis items addressed (60 bugs/clarifications, 30 enhancements) in UVM-1.2. In the release notes, you could find that 50% of items are API changes and 25% break back compatibility. In table below, you could see the number of changes in classes, files and lines between different UVM versions.

	Classes	Files	Code Lines
UVM1.0-P1	288	130	28921
UVM1.1A	322	140	29962
UVM1.1B	317	141	30258
UVM1.1C	317	140	30320
UVM1.1D	316	139	30365
UVM1.2	356	159	34551

Meanwhile, Accellera UVM working group is taking efforts to make UVM methodology as IEEE standard. IEEE usually could take 6 months to 1 year to release it, and end users could study, migrate and review UVM-1.2 during UVM public

review stage for 6 months from design automation conference (DAC, June, 2014). To help UVM end users to better understand the differences between UVM-1.2 and UVM-1.0/1.1, and estimate effort/path to migrate to UVM-1.2. In this paper, we will describe details in notable changes in UVM-1.2, generic debug capability and migration experiments from UVM-1.1x to UVM-1.2

II. NOTABLE CHANGES IN UVM-1.2-RC8

- *Changes in Reporting infrastructure*

The inside core is changed, but the end user need not much changed besides 20 new added macros.

A: Reporting is fully object oriented and all UVM core messages now routed through uvm_report_server. It removed all $display calls (mostly for debug output) from base class library (BCL), except report_server. One of the reasons is to address user issue. When user creates their own printer object, and adjust the printer knobs to print to a file rather than STDOUT, $display statement prevents print_topology() to a file. This ensures consistent output and central control.

The change example in print_topology function
UVM-1.1d
$dispaly(printer.emit());
UVM-1.2
`uvm_info("UVMTOP", {"UVM testbench topology:\n", printer.emit()},UVM_NONE)

B: In UVM-1.1, uvm_report_server used a mix of virtual and non-virtual functions what makes it impossible to properly extend the report server in the past. In UVM-1.2, It now becomes a virtual class and can be extended or chained using the delegate pattern.

The changes in uvm_report_server
UVM-1.1d
virtual function void summarize – called in uvm_report_object::report_summarize
virtual function string compose_message
virtual function void process_report
UVM-1.2
[Deprecated] virtual function void summarize
[Deprecated] virtual function string compose_message
[Deprecated] virtual function string process_report
[New] virtual function void report_summarize – called in uvm_report_object::report_summarize
[New] virtual function string compose_report_message
[New] virtual function void process_report_message

C: New added uvm_report_message class which is the basic UVM object message class and provides the fields that are common to all messages. It also has a message element container and provides the APIs necessary to add integral types, strings and uvm_objects to the container.

D: New added message reporting macros, uvm_*_begin/end, uvm_message_add_*, etc.

E: Now object based with ability to add values (int or string)/objects. It can record message to some other storage. It's not transaction recording here!

F: New added uvm_process_report_message function, which is defined in package scope and a convenience function that delegate to the corresponding component method in ~uvm_top~. It can be used in module-based code to use the same reporting mechanism as class-based component.

```
uvm_component level
test_object obj;
uvm_recorder rec;
bit [16:0] test_int = 16'hff55
obj = test_object::type_id::create("my_obj");
uvm_report_message udf_message, message;
//Method 1 ---
udf_message = uvm_report_message::type_id::create("user_defined_message");
udf_message.set_severity(UVM_WARNING);
udf_message.set_id("TEST1");
....
udf_message.add_int("test_int", test_int,16,UVM_HEX);
udf_message.add_string("my_string","string_value");
udf_message.add_object("my_obj",obj);
uvm_config_db#(uvm_recorder)::get(this, "", "rec", rec);
udf_message.record(rec); // record message
uvm_process_report_message(udf_message);
//Method 2 ---
message = uvm_report_message::type_id::create("my_message");
`uvm_info_begin("TEST2", "My info message", UVM_LOW, message)
        `uvm_message_add_tag("my_color", "black") // add string
        `uvm_message_add_int(test_int, UVM_HEX)
        `uvm_message_add_object(obj)
`uvm_info_end
uvm_test level
uvm_text_tr_database db;
function void build_phase(uvm_phase phase);
  db = new("my_db"); // create db
    db.set_file_name("my_db.txt"); // record message to txt file
    db.open_db();
    begin
      uvm_recorder rec = db.open_stream("my_stream").open_recorder("my_recorder");
      uvm_config_db#(uvm_recorder)::set(this, "agent1.comp1", "rec", rec);   end
endfunction
function void final_phase(uvm_phase phase);
     db.close_db();
endfunction
```

- *Changes in Sequences*

A: In UVM-1.1, we adopt "raise the phase's objection prior to executing the sequence and drop the objection after ending the sequence (naturally or via call to <kill>)" to interact with starting phase within a sequence. It's common to have only

parent sequence raise an objection. If the children sequences didn't complete, the parent can't finish, thus only parent needs to raise and lower objections. That's why we implement the raise/drop in base sequence. In UVM-1.2, we simplify it automatically and add new API set_automatic_phase_objection(arg) which automatically performs a raise/drop of the objection before/after the sequence execution. This method can be called any time prior to start() being called. If it's enabled, then the sequence will raise an objection prior to pre_start(), and drop the objection after post_start() or do_kill(). Calling the method after start() would be an error. It's important to keep in mind that NEVER to set the automatic phase objection bit to "1" if your sequence runs with a forever loop inside the body, as the objection will never get drop.

B: In UVM-1.1, the starting_phase member is only set automatically if the sequence is started as the default sequence (if you have) for a particular phase. In UVM-1.2, the starting_phase variable is now data access protected within uvm_sequence_base, and the end user must use the "get_starting_phase" and "set_starting_phase" functions. The uvm_sequence::starting_phase is deprecated.

UVM-1.1d

In the base sequence:

```
virtual task pre_body();
  if(starting_phase != null)
    starting_phase.raise_objection(this);
endtask
virtual task post_body();
  if(starting_phase != null)
    starting_phase.drop_objection(this);
endtask
```

In the test layer:

```
task run_phase (uvm_phase phase);
 seq.starting_phase = phase;
 seq.start(foo_agent.sequencer); // blocking
endtask
```

UVM-1.2

In the base sequence:

```
function new(string name="my_base_seq");
  super.new(name);
  set_automatic_phase_objection(1);
endfunction
```

In the test layer:

```
task run_phase (uvm_phase phase);
 seq.set_starting_phase(phase);
 seq.start(foo_agent.sequencer);
endtask
```

Other usecase:

This functionality can also be enabled in sequences which were not written with UVM Run-Time Phasing.

```
my_legacy_seq_type seq = new("seq");
seq.set_automatic_phase_objection(1);
```

```
seq.start(my_sequencer);
```

C: In UVM-1.2, new added: +uvm_set_default-sequence=<sqr>, <phase>, <type> allows you to start a sequence from the command-line

- *Changes in Registers*

A: In UVM-1.2, the uvm_hdl.c is updated to allow vendor tools to perform backdoor access to VHDL.

B: In UVM-1.1, the transaction order is unclear and can't be changed when bus and register size are different. For example: the design under test(DUT) registers are 32bits wide and big endian access, and the bus interface universal verification component (UVC) is 16 bits wide. For each register access, we should preform two transactions on the bus. The access & register address should look like as below on the bus.

Transaction 1 --→ Lower address + First word [31:16]

Transaction 2 --→ High address + Second word [15:0]

But using UVM_REG, we are seeing two transactions as below:

Transaction 1 --→ High address + Second word [15:0]

Transaction 2 --→ Lower address + First word [31:16]

The address issued by UVM_REG should be starting from lower address and then higher address.

In UVM-1.2, it has the ability to control transaction order when register access result in multiple bus transactions in case register size and bus size mismatch. New added virtual class "uvm_reg_transaction_order_policy" could address this challenge. The pure virtual function "uvm_reg_transaction_order_policy::order(ref uvm_reg_bus_op q[$])" may reorder the sequence of bus transactions produced by a single uvm_reg transaction (read/write). The first item (0) of the queue will be the first bus transaction (the last($)) will be the final transaction. End user should implement the order in derived subclass.

UVM-1.2

```
class high_first extends uvm_reg_transaction_order_policy;
    virtual function void order(ref uvm_reg_bus_op q[$]);
        `uvm_info("TEST_high1",$sformatf("%p",q),UVM_NONE)

      q.sort with (item.addr);
        `uvm_info("TEST_high2",$sformatf("%p",q),UVM_NONE)
    endfunction
    function new(string name = "dut");
      super.new(name);
    endfunction
  endclass
// the blk is a uvm_reg_block object (wide is 32bits)
// the bus16 is a uvm_reg_map object (wide is 16bits with UVM_BIG_ENDIAN attribute).
high_first up=new("high-first");
blk.bus16.set_transaction_order_policy(up);
blk.r0.write(status, 'hdeadbeef); // r0 is a 32bits wide register in the uvm_reg_block.
```

- *Changes in Objects*

A: In UVM-1.1, if you don't define the `UVM_OBJECT_MUST_HAVE_CONSTRUCTOR symbol, the new() is always called without arguments and the name is explicitly set later. It makes it impossible to use the name to perform actions that

must be done in constructors (such as initiating coverage groups) and will also cause a difference in behavior between using the factory and calling new() directly.

In UVM-1.2, Classes extended from uvm_object now require an explicit constructor with a string-type name argument and this functionality is now the default. it could make it obsolete by define ` UVM_OBJECT_DO_NOT_NEED_CONSTRUCTOR, but we don't recommend to do that.

UVM-1.1d

```
virtual function uvm_object create_object(string name="");
    T obj;
    `ifdef UVM_OBJECT_MUST_HAVE_CONSTRUCTOR
      if (name=="")  obj = new();
      else obj = new(name);
    `else
      obj = new();
      if (name!="")  obj.set_name(name);
    `endif
      return obj;
```

UVM-1.2

```
virtual function uvm_object create_object(string name="");
    T obj;
    `ifdef UVM_OBJECT_DO_NOT_NEED_CONSTRUCTOR
      obj = new();
      if (name!="")  obj.set_name(name);
    `else
      if (name=="")  obj = new();
      else obj = new(name);
    `endif
      return obj;
```

B: Component names are being checked for compliance. This avoids bad names such as "…", "☺" or "a.b.c.d"

C: In UVM-1.2, in order to improve memory performance of "bitstream [4K bits]" interfaces for report/record/compare/pack etc. The implementation now could support a less memory-expensive uvm_integral_t which is sized as a 64bits packed logic vector. This type is used in the "*_field_int()" methods for reporting/recording/comparing and packing/unpacking. In the uvm_printer, it newly adds the print_field_int function to adopt the uvm_integral_t type comparing with print_field function (which uses the uvm_bitstream_t arguement).

UVM-1.1d

```
typedef logic signed [UVM_STREAMBITS-1:0] uvm_bitstream_t;
```

UVM-1.2

```
typedef logic signed [UVM_STREAMBITS-1:0] uvm_bitstream_t;
typedef logic signed [63:0] uvm_integral_t; -- New
```

- *Changes in Phasing*

A: In UVM-1.2, it removed objections from non-task-imps (because objections do not make sense with function phases).

B: In UVM-1.1, once a phase has been placed into a schedule, the only way to get a reference to that phase is by using find function. But it has two limitations:

1. It requires that you know either the name (or IMP type) of the phase in advance, so you can pass it to the find function.
2. Find function will return after finding the first matching phase, if there are multiple phases with the same name (or IMP type), then there is no way to locate them.

In UVM-1.2, it adds simple schedule introspection to uvm_phase to let end user programmatically traverse the entire phase graph via get_adjacent_predecessor_nodes and get_adjacent_successor_nodes functions. They could provide an array of nodes which are predecessors/successors to this phase node.

UVM-1.2

```
function void main_phase(uvm_phase phase);
  uvm_phase phase_nodes[];
  phase.get_adjacent_predecessor_nodes(phase_nodes);
  phase.get_adjacent_successor_nodes(phase_nodes);
// you could display it by phase[i].get_name();
```

C: New added uvm_phase.get_objection_count() can be used to retrieve pending objections for the phase.

D: New added phase transitions callbacks.

UVM-1.2

```
typedef uvm_callbacks#(uvm_phase, uvm_phase_cb) uvm_phase_cb_pool;
```

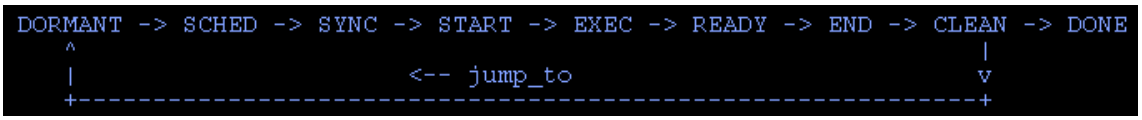
E: New added functions for phase jumping: set_jump_phase and end_prematurely.

set_jump_phase function specifies a phase to transaction to when phase is complete.

end_prematurely function sets a flag to cause the phase to end prematurely.

E: New added UVM_PHASE_UNINITIALIZED into uvm_phase_state. The state is uninitialized. This is the default state for phases, and for nodes which have not yet been added to a schedule.

The state transitions occur as follows in UVM-1.1:



The state transitions occur as follows in UVM-1.2:

- *Changes in Configure Database*

A: In UVM-1.2, Meta characters/ regex in field names disabled due to performance and semantic issues.

```
uvm_config_int::set(this,"","/z?mycomplexint/",4);
uvm_config_int::set(this,"","/my_complex.* /",3);
uvm_config_string::set(this,"","/my_complexint/","xxxx");
```

B: In UVM-1.2, set_config_*, get_config_* now deprecated. It should be careful when covering *_config_object with clone semantic during migration.

- *Changes in Factory*

A: In UVM-1.2, uvm_pkg::factory has been removed. You can retrieve the factory via uvm_factory::get() instead. It should be aware of migration issues.

B: In UVM-1.1, once a factory is overridden, you could not undo it. In UVM-1.2, it adds the ability to undo a factory override now.

UVM-1.2

factory.set_type_override_by_type(comp::get_type(), my_comp::get_type());
 	
factory.set_type_override_by_type(my_comp::get_type(), my_comp::get_type());

C: In UVM-1.2, it can replace factory in order to trace or log factory calls or build a dynamic factory. The example will be in debug chapter.

- *Changes in Objections*

A: In UVM-1.2, it allows for hierarchical propagation of uvm_objections to be disabled via set_propagate_mode. It could be used to avoid rippling of objections through hierarchy. Better performance gain in high-frequent raise/drop use-case, but less visibility of debug. End user should balance the right thing at right time. Since the propagation mode changes the behavior of the objection, it can only be safely changed if there are no objections ~raised~ or ~draining~. Any attempts to change the mode while objections are ~raised~ or ~draining~ will result in an error.

UVM-1.2

obj= phase.get_objection();
 	
obj.set_propagate_mode(0);
 	
---------- 	
 	
Any objections raised by “child” would get propagated down to parent and then to uvm_test_top.
 	
//| | count | total |
 	
//| uvm_top.parent.child | 1 | 1 |
 	
//| uvm_top.parent | 0 | 1 |
 	
//| uvm_top | 0 | 1 |
 	
When propagation mode is set to “0”
 	
//| | count | total |
 	
//| uvm_top.parent.child | 1 | 1 |
 	
//| uvm_top.parent | 0 | 0 |
 	
//| uvm_top | 0 | 1 |

B: In UVM-1.1, it will throw decrement-below-zero error if the objection count is 1.

In UVM-1.2, uvm_objection::drop_objection now works even when total objection count is 0.

```
function void m_drop (uvm_object obj, ..)
// Ignore drops if the count is 0
if (count == 0)  return;
```

- *Changes in UVM Event*

In UVM-1.2, it provided parameterized uvm_event, previously, uvm_event only worked with uvm_object.

```
//The optional parameter ~T~ allows the user to define a data type which can be passed during an event trigger.

class uvm_event#(type T=uvm_object) extends uvm_event_base;
```

- *Changes in Transaction Recording*

1. In UVM-1.1, by default, the transaction recording is performed automatically when get_next_item() and item_dome() are called in driver. However, it works only for simple, in-order, blocking transaction execution. For pipelined and out-of-order transaction execution, the driver must turn off this automatic recording and call uvm_transaction::accept_tr, uvm_transaction::begin_tr and uvm_transation::end_tr explicitly at appropriate points in time. Once it’s disabled, automatic recording can’t be re-enabled. It defines the

`UVM_DISABLE_AUTO_ITEM_RECORDING to disable the auto item recording in start/finish_item. In UVM-1.2, to support for run-time disabling of auto item recording, it adds new disable_auto_item_recording() function.

2. In UVM-1.2, to make recording system object based, it introduces uvm_tr_database and uvm_tr_stream classes. They are more vendor specific staffs, so we don't talk more here.

- *Changes in Misc (UVM-1.2)*

1. It introduces Data access policy (DAP) objects which provide controlled access to embedded objections.

   i) uvm_set_before_get_dap

   The "set before get" data access policy enforces that the value must be written at least once before it is read. This DAP can be used to pass shared information to multiple components during standard configuration, even if that information hasn't yet been determined. Such DAP objects can be useful for passing a 'placeholder' reference, before the information is actually available.

   ii) uvm_get_to_lock_dap

   The "get to lock" data access policy allows for any number of 'sets', until the value is retrieved via a 'get'. Once 'get' has been called, it's illegal to 'set' a new value. UVM uses this policy to protect the {starting phase} and {automatic objection} values in uvm_sequence_base.

   iii) uvm_simple_lock_dap

   The "simple lock" data access policy allows for any number of 'sets', so long as the vaule is not 'locked'. The value can be retrived using 'get' at any time. UVM uses this policy to protect the {file name} value in the <uvm_text_record_database>.

2. It introduces uvm_coreservice_t common container for package scope variables with set/get accessors. The singleton instance of uvm_coreservice_t provides a common point for all central uvm services such as uvm_factory, uvm_report_server, etc. The service class provides a static ::get which returns an instance adhering to uvm_coreservice_t. The reset of the set_<facility> and get_<facility> pairs provide access to the internal uvm services.

```
uvm_factory f = uvm_factory::get(); // the same as below
uvm_coreservice_t cs = uvm_coreservice_t::get();
uvm_factory f = cs.get_factory();
```

```
class uvm_delegate_factory extends uvm_factory;
  uvm_factory delegate;
```

```
class user_factory extends uvm_delegate_factory;
user_factory f1= new(); // create a new factory
f1.delegate = f; // set the delegate
cs.set_factory(f); // enable new factory.
```

3. In the UVM infrastructure, we often need to traverse all or parts of the component hierarchy. In UVM-1.2, it introduces visitor pattern infrastructure which added (uvm_visitor, uvm_structue_proxy, uvm_visitor_adapter) to do this. For general information regarding the visitor pattern, please see http://en.wikipedia.org/wiki/Visitor_pattern.

4. It introduces uvm_enum_wrapper#(T) class, and functionality to set enumerations by string name. It provides a <from_name> function which attempts to covert a string <name> to an enumerate value. It allows for enum fields to be configured using uvm_config_db#(string)::set, as well as from the command line using +uvm_set_config_string.

```
typedef enum {YES,NO } op_e;
op_e inst;
uvm_enum_wrapper#(op_e)::from_name("YES", inst);
```

5. Messages in DPI-C now routed back to UVM message facilities.

6. The confusing uvm_severity_type(int) was deprecated and replaced internal using uvm_severity(enum)

7. Separation of classes into abstract API and "_default_" implementation for uvm_factory, uvm_report_server

```
Class my_server extends uvm_default_report_server
```

8. Cleanup of package scope variables (factory, missing UVM_ prefex Ex. "UVM_"SEQ_ARB_RANDOM)
9. uvm_sequence_library is new documented

In UVM-1.1, the protocol layering proposal in section 6.5.2.3.1 is not reusable nor scalable. In User guide 1.2, it adds new protocol layering idea (layering driver + pass through sequence) proposed by Janick.

III. GENERIC DEBUG CAPABILITIES IN UVM-1.2

UVM debug becomes a big challenge to every verification engineer. In this chapter, we will discuss some generic debug capability in UVM-1.2.

- Replace factory in order to trace or log factory

UVM-1.2

```
class uvm_to_factory extends uvm_delegate_factory;

 virtual function void set_inst_override_by_type (uvm_object_wrapper original_type, uvm_object_wrapper override_type, string full_inst_path);

  `uvm_info( " F A C T O R Y " , "…", U V M _ N O N E )

  delegate.set_inst_override_by_type (original_type,override_type,full_inst_path);

 endfunction

endclass

uvm_coreservice_t cs= uvm_coreservice_t::get();

uvm_to_factory f = new();

f.delegate=uvm_factory::get();

cs.set_factory(f);
```

- Using new added message macros, the ability to add values/objects to aid debug via UVM reporting. The example had mentioned above chapter.
- Using phase transition callbacks to debug phase.

```
Class my_cb extends uvm_phase_cb;
  Virtual function void phase_state_change (uvm_phase phase, uvm_phase_state_change change);
     uvm_phase_state state = change.get_state();
     `uvm_info("CALLBACK", $sformat("detected phase state change %s for phase %s", state.name(),
phase.get_name()),UVM_LOW);
  endfunction
uvm_callback#(uvm_phase, uvm_phase_cb)::add (phaseinst, my_cb_inst);
```

IV. MIGRATION EXPERIENCE FROM UVM-1.X TO UVM-1.2

When you do environment migration from UVM-1.1 to UVM-1.2, most of things could be backward compatible, but it's not 100%. It could not mix the UVM-1.1 and UVM-1.2 library in the same environment. That means you have to use the migration script (provides in the UVM-1.2 tarball) to change the incompatible changes as below.

```
./bin/add_uvm_object_new.pl  // add uvm_object constructor if missing
./bin/uvm11-to-uvm12.pl // it may help to do the simple changes around starting_phase, set/get_config, reporting
./bin/ovm2uvm.pl // the old OVM->UVM10 script
```

The script may not do all of migration changes, you have to manual change them which don't be included in the script.

uvm_global_report_server global_server = new(); report_server = global_server.get_server();	uvm_report_server report_server = uvm_report_server::get_server();
uvm_report_server report_server; report_server.<any_member_moved_to_default> ⇨	uvm_default_report_server report_server;
Factory.any_method();	Uvm_factory factory = uvm_factory::get();

	Factory.any_method();
UVM states FINISHED, BODY, STOPPED, etc.	UVM_FINISHED, UVM_BODY, etc.

After the changing by script, you may have to do some manual changes to the script's output.

Comp.uvm_config_int::get(this,…)	Uvm_config_int::get(comp,…);
Uvm_pkg::uvm_config_object::set(null,"*", "ABC_if",abc_vif ,0);	Uvm_pkg::uvm_config_object::set(null,"*", "ABC_if",abc_vif);
uvm_config_object::set(this, "*", "ab", this, 0)	uvm_config_object::set(this, "*", "ab", this)
uvm_pkg::uvm_config_object::set(this, "*", name, this, 0)	uvm_pkg::uvm_config_object::set(this, "*", name, this)

If your environment used the 3rd part VIP which is encrypted based on old UVM version, it's impossible to run the migration script to change. You have to get the upgraded 3rd part VIP based on UVM-1.2.
The release-notes do have a list of addressed mantis items with marker for backward compatibility. You could check them.

V. CONCLUSION

UVM-1.2 is coming, it introduces more benefits to users and we should get ready for that to move forward. This paper could bridge users to the gap and be a UVM-1.2 primer to all verification engineers (from those just starting with UVM to those with years of experience) and they will gain new knowledge for sure.

ACKNOWLEDGEMENTS

First, we would like to thank for continued support to my wife (Liangliang Li) and AMD managers (Davis.Wan& Leo.zhang). The authors also wish to acknowledge Tom Fitzpatrick's seminar at mentor verification academy.

REFERENCES

[1] http://www.accellera.org/apps/org/workgroup/uvm/

[2] http://www.eda.org/svdb

[3] UVM-1.1d User guide and UVM-1.2 reference manual

[4] Tom Fitzpatrick, "UVM 1.2 is coming, so be prepared", Verification Academy, 2014

[5] Uwe Simm, "UVM 1.2 introduction", DVCon 2014

[6] Uwe Simm, "UVM – what's now and what's next", DVClub Shanghai Q1 2014

[7] Roman Wang, Uwe Simm, "Making UVM Verification Life Easier: UVM Debug Capabilities", CDNLive China and Boston 2013.