Solve UVM Debug Problems with the UVM Vault

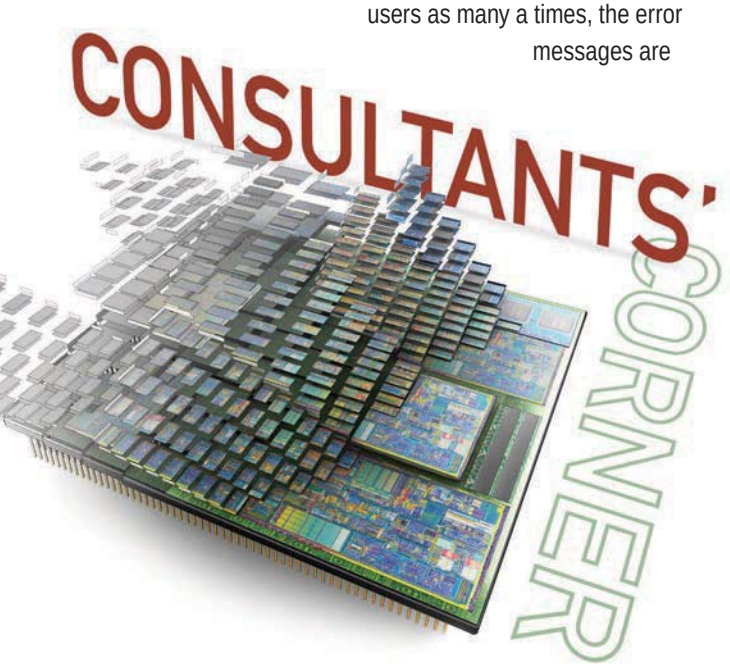
by Srinivasan Venkataramanan and Ajeetha Kumari, VerifWorks

INTRODUCTION

Universal Verification Methodology (UVM) is the industry standard verification methodology for Verification using SystemVerilog (SV). UVM provides means of doing verification in a well-defined and structured way. It is a culmination of well-known ideas, thoughts and best practices.

Given the major adoption of UVM across the globe and across the industry, advanced users are looking for tips and tricks to improve their productivity. UVM does define a structured framework for building complex testbenches. It is built on strong OOP principles and design patterns using underlying SystemVerilog language features. This strong OOP nature presents certain challenges to the end users. Recall that many Design-Verification (DV) engineers come from hardware, electronics backgrounds and not heavy Software backgrounds. Hence at times it gets tricky for users to debug UVM based testbenches when things do not work as expected.

In this article the authors share their long experience of assisting customers with run time debug of common UVM issues and potential solutions to them. During our various training and consulting engagements using UVM we have seen DV engineers struggling to debug relatively simple UVM issues. It will be unfair to blame the users as many a times, the error messages are cryptic and do not point to the actual source code, rather somewhere from the base classes, making the debug difficult. We have captured a series of such common issues and error messages into a collateral form that we call "UVM Vault". As part of our QVP engagement with Mentor Graphics, we are integrating this UVM Vault to Questa® in the near future.



Specifically, we will highlight a few hand-picked features in the UVM factory and provide tips and tricks around them. It is expected that the readers are conversant with the UVM framework to appreciate the tips presented below.

FACTORY

UVM provides a convenient way to control which objects get manufactured at run time via the well-known Factory Design pattern. Wikipedia defines factory pattern as:

In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method rather than by calling a constructor.

In UVM to use factory, there are three important steps:

1. Registration – all classes shall be registered with a global factory singleton table
2. Use *class::type_id::create()* instead of *new()*
3. Use *set_type_override* functions

Steps 1 & 2 above are to be done while a verification environment is being created by the developers. Once such a good framework is ready, verification engineers and VIP users can use step 3 above to tweak the behavior of underlying components and transactions. Below is a sample log with a transaction named *s2p_xactn*.

```
  UVM_INFO ../tb_src/s2p_scoreboard.sv(68) @ 350.00 ns: uvm_test_top.env_0.agent0.scoreboard [SBRD] PASS: Xactn
1 no_of_pass_xactn 1 Expected data 8'hA5 Actual data 8'hA5
#
# UVM_INFO ../tb_src/s2p_driver.sv(74) @ 350.00 ns: uvm_test_top.env_0.agent0.driver [driver]  Next item:
# ----------------------------------------------------------------------------------------------------------
# Name                            Type                    Size  Value
# ----------------------------------------------------------------------------------------------------------
# req                             s2p_xactn  -    @871
#   err_pkt                       integral                1     'h0
#   pkt_len                       integral                8     'h8
#   pkt_pld                       integral                8     'hbb
#   par_data                      integral                8     'hXxx
#   kind                          s2p_pkt_kind_e          32    SEND_PKT
#   begin_time                    time                    64    350.00 ns
#   depth                         int                     32    'd2
```

Figure 1: Sample log with s2p_xactn

Now a simple factory override with a derived transaction (possibly with additional constraints) can be set as shown in Figure 2 below:

```
s2p_rand_seq seq1, seq2, seq3;

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

virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
    `uvm_info("FACTORY","Overriding s2p_xactn with pattern testcase",UVM_LOW)

  factory.set_type_override_by_type(s2p_xactn::get_type(),
                                    s2p_xactn_with_pld_pattern::get_type());
endfunction : build_phase

task main_phase(uvm_phase phase);
```

Figure 2: Simple factory override

A sample run with Questa shows a log as shown below in Figure 3.

```
# UVM_INFO ../tb_src/s2p_driver.sv(74) @ 350.00 ns: uvm_test_top.env_0.agent0.driver [driver]  Next item:
# ----------------------------------------------------------------------------------------------------------
# Name                            Type                              Size  Value
# ----------------------------------------------------------------------------------------------------------
# req                             s2p_xactn_with_pld_pattern  -     @871
#   err_pkt                       integral                  1     'h0
#   pkt_len                       integral                  8     'h8
#   pkt_pld                       integral                  8     'hbb
#   par_data                      integral                  8     'hXxx
#   kind                          s2p_pkt_kind_e            32    SEND_PKT
#   begin_time                    time                      64    350.00 ns
#   depth                         int                       32    'd2
```

Figure 3: Debug tip for factory override

TIP: Look at the *"Type"* column in standard *UVM print (uvm_object::sprint)* to see if your override indeed worked! If not, no need of waveforms to debug, go back to your UVM code!

MULTIPLE OVERRIDES FOR THE SAME OBJECT

There are cases when multiple overrides are set on the same object. There are use cases to *"ignore"* such overrides and then other situations where-in it is desired to *"replace"*. UVM's factory override mechanism supports both requirements. Consider a transaction model derived as shown in Figure 4, there are two similar error transactions derived from a base *s2p_xactn.*

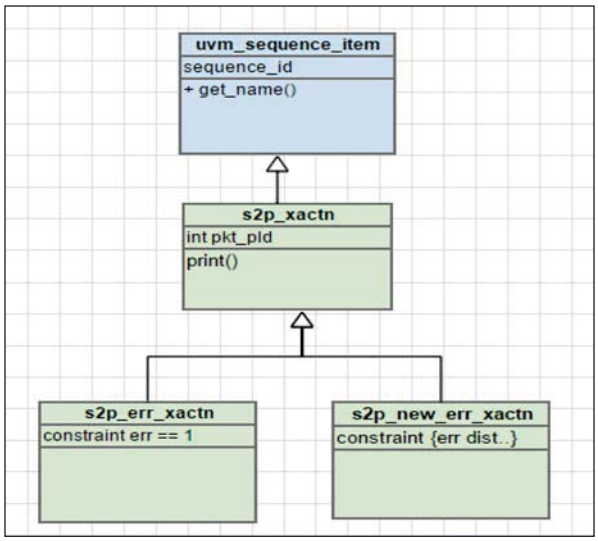


Figure 4: Related transactions

Now if two overrides are specified for the same *s2p_xactn* (assume that's the transaction on which driver, sequencer etc. are parameterized), code snippet in Figure 5 shows a way to "replace" the first override with the second.

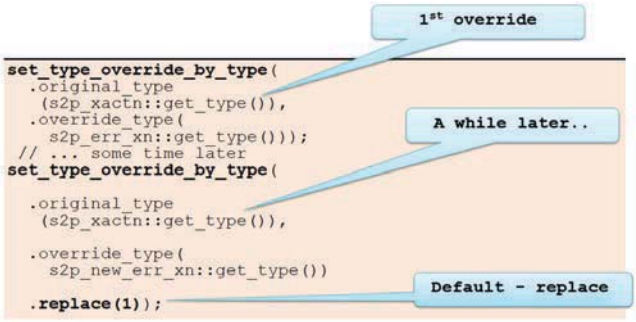


Figure 5: Pseudo-code to replace existing override

This is a handy trick to run a test with a few 100 transactions first with one type and then replace with another derived type and run another few 100s within the same test!

TIP: Look for UVM log ID *"TPREGR"* as shown below to ensure your replacement is guaranteed!

```
[TPREGR] Original object type 's2p_xactn' already
registered to produce 's2p_err_xactn'.
 Replacing with override to produce type
's2p_new_err_xn'.
```

Figure 6: Built-in debug hook in UVM to display factory replacement

Often in a IP to sub-system reuse scenarios the subsystem engineer is not fully aware of IP level sequence functionality. It is possible that a factory override was already set by the IP level sequence. Imagine a scenario where-in a sub-system verification engineer wants to "query" the IP sequence to see if an override exists, if not, set a new override.

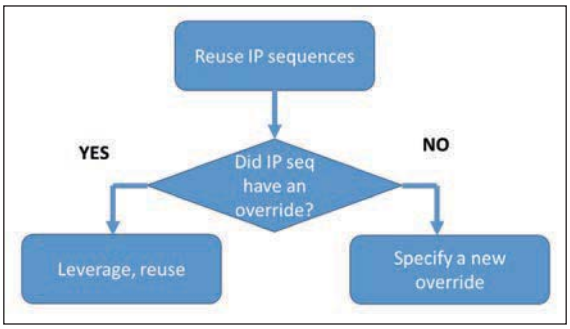


Figure 7: Reuse scenario – verification plan as flowchart

Though SystemVerilog and UVM do not have a full-fledged "reflection API" to "query" such arbitrary questions on the database, UVM factory does support this use case via a "ignore" option to the factory override method. Refer to Figure 8 that shows a similar override as in the previous example, but setting the "replace" argument to 0.

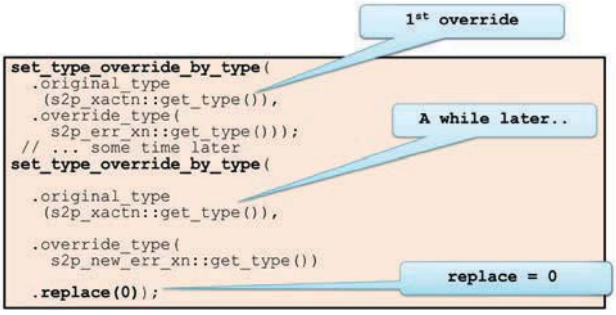


Figure 8: Pseudo-code to query and add a new override

This is very useful for the reuse scenarios as described above for sub-system integrators.

TIP: Look for UVM log ID *"TPREGD"* as shown to ensure your replacement is conditional (if no previous override, then do it!).

```
[TPREGD] Original object type 's2p_xactn' already registered to produce
's2p_err_xactn'.
Set 'replace' argument to replace the existing entry.
```

Figure 9: Built-in debug hook in UVM to display factory override behavior

Sometimes the user wants to "undo" an override – i.e., a base class has been overridden to be replaced with a derived class. A little later, with-in the same simulation one wants to remove/undo the override and revert back to base class. This makes sense only for transactions/sequence items as the override plays a role only when the *create()* gets called AFTER the override setting.

Another scenario when "undo" operation is desired is when a IP is being reused at the next level and the integrator wants to nullify the override set by the IP.

An intuitive way to do this would be to override with the "same" class when desired. Consider an example of pseudo-code below:

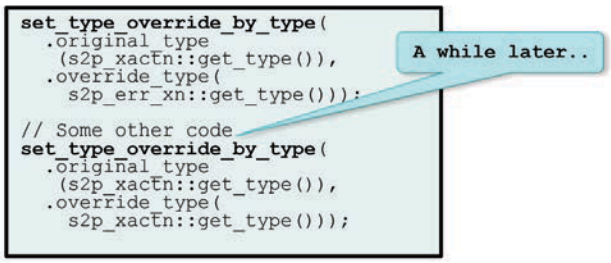


Figure 10: Undo an override

While the above code will work fine with the latest UVM 1.2 release, in case of UVM 1.1d (which is more prevalent at many customer designs as of today), the above code will lead to an error as shown below in Figure 11.

```
[TYPDUP] Original and override type arguments are identical:
s2p_xactn
```

Figure 11: Questa® simulation log for factory undo in UVM 1.1d

A work-around for achieving the same in UVM 1.1d version could be to derive a dummy, new class from the base class and use it. The new class that is derived should contain no extra code and is intended to just fool the compiler to treat it as a derived type, but functionally same as the base type. Figure 12 shows a possible s2p_undo_xactn UML diagram to explain this.

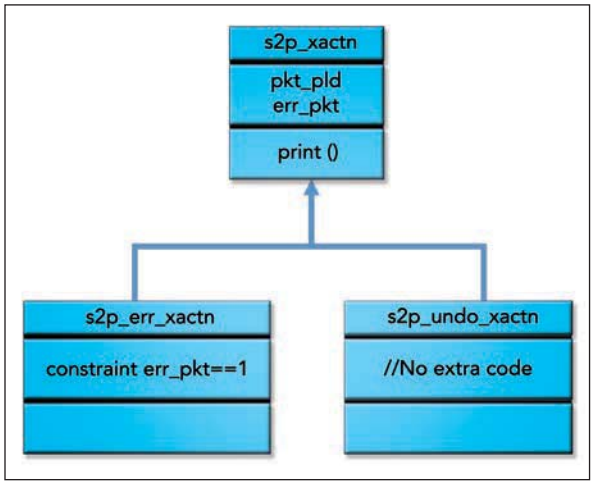


Figure 12: Deriving an extra, dummy transaction to implement factory UNDO in UVM 1.1d

TIP: Factory undo works out-of-the-box in UVM 1.2

TIP: Factory undo requires a small workaround in UVM 1.1d

DEBUGGING FACTORY OVERRIDES

Given that UVM supports global and instance based overrides (with instance name being a string, computed at run time), it is possible for the users to get it wrong the first time – i.e., user expects an override to occur, but it did not occur. A less popular API in the factory (and not so highly recommended to use) uses names to specify the original and override objects. Since all SystemVerilog strings are computed at run time and typically users tend to use "regexp" on these string names, there are definite possibilities to get these paths, names wrong. Good thing however is UVM factory already has auditing capabilities to help in such cases. There exists a *print* routine in *uvm_factory* that can display things like:

- All the registered classes
- All the overrides seen so far
  (at the time of calling *print* routine)
- Specific instance where the overrides shall take place

A code snippet of internal implementation of UVM factory is shown in Figure 13 below.

```
// print
// -----

function void uvm_factory::print (int all_types=1);

  string key;
  uvm_factory_queue_class sorted_override_queues[string];

  string tmp;
  int id=0;
  uvm_object_wrapper obj;
```

Figure 13: Built-in debug hook in UVM factory

When things do not work as expected, the user can add the following code snippet inside a test:

```
virtual function void end_of_elaboration_phase (uvm_phase phase);
  uvm_factory f;

  f = uvm_factory::get();
  f.print();
  f.print(.all_types(0));
```

Figure 14: User code to display factory contents at run time

A sample print from Questa® simulation with the above debug code is shown below in Figure 15. Note that we used *all_types (0)* to see only relevant information.

```
#### Factory Configuration (*)
:
: Instance Overrides:
:
:  Requested Type   Override Path                          Override Type
:  --------------   ----------------------------------    ----------------------------
:  s2p_xactn        uvm_test_top.env_0.agent0.sequencer.* s2p_xactn_with_pld_pattern
:  s2p_xactn        uvm_test_top.env_0.agent0.sequencer.* s2p_xactn_extra
:
: No type overrides are registered with this factory
:
```

Figure 15: Questa sample log for factory debug

TIP: Use *uvm_factory::print(.all_types(0))* – it is built to show the overrides alone.

IMPACT OF "NAME" IN FACTORY OVERRIDES

In UVM components are built hierarchically and hooked up at the top level. The hook-up happens through 2 key fields – *name* and *parent*. Every UVM component in user code implements *new()* as shown below in Figure 16.

```
class s2p_driver extends uvm_driver #(s2p_xactn);

  function new(string  name,
               uvm_component parent);
    super.new(name, parent);
  endfunction : new
```

Figure 16: Typical constructor in a UVM component

For the first-timer the above code looks a bit strange, but since the base class (*uvm_component* to be precise) requires these 2 arguments, it becomes important to stick to this style (though technically speaking there are other crude ways with default values, etc.). The *parent* argument in the component's constructor is expected to be connected to the "testbench" *parent* component (and not necessarily a standard OOP *parent* class). A typical hierarchy/topology of a UVM testbench looks as shown in Figure 17. The standard UVM *print_topology* prints in a tree format as shown below and every "indentation" level indicates a layer in the testbench. For instance, *driver* is indented inside *agent*. It indicates that *driver's parent* argument is connected to the *agent* object.

```
  UVM_INFO ../tb_src/s2p_agent.sv(40) @ 0.00 ns: uvm_test_top.env_0.agent0 [agent
  UVM_INFO @ 0.00 ns: reporter [UVMTOP] UVM testbench topology:
# ------------------------------------------------------------------------------
# Name                     Type                        Size  Value
# ------------------------------------------------------------------------------
# uvm_test_top             Factory_test                -     @463
#   env_0                  s2p_env                     -     @471
#     agent0               s2p_agent                   -     @479
#       driver             s2p_driver                  -     @670
#         rsp_port         uvm_analysis_port           -     @687
#         seq_item_port    uvm_seq_item_pull_port      -     @678
#       par_monitor        s2p_par_mon                 -     @513
#         par_port         uvm_analysis_port           -     @521
#       s2p_fcov_h         s2p_fcov                    -     @644
#         analysis_export  uvm_analysis_imp            -     @661
#       scoreboard         s2p_scoreboard              -     @530
#         par_afifo        uvm_tlm_analysis_fifo #(T)  -     @591
#           analysis_export uvm_analysis_imp           -     @635
#         ser_afifo        uvm_tlm_analysis_fifo #(T)  -     @538
#           analysis_export uvm_analysis_imp           -     @582
#       sequencer          s2p_sequencer               -     @696
#         rsp_export       uvm_analysis_export         -     @704
#         seq_item_export  uvm_seq_item_pull_imp       -     @810
#       ser_monitor        s2p_ser_mon                 -     @496
#         ser_aport        uvm_analysis_port           -     @504
#       is_active          uvm_active_passive_enum     1     UVM_ACTIVE
# ------------------------------------------------------------------------------
```

Figure 17: Typical testbench topology in UVM

The other argument *name* is a string value that represents the name of the object in the given hierarchy. In UVM, all object names must be unique at a given level of hierarchy. In general, it is recommended to keep the value that is passed to the *name* argument the same as the *handle* name. There are cases such as AXI fabric environments where-in the same agent is instantiated many times using a dynamic array. Hence the *handle* name is an array variable, where-as the *name* argument must be created to be unique for each element in that array. While doing this, care must be taken to:

- Keep each object's *name* to be unique
- Keep sensible names – will see why below.

For simplicity, let's consider a case where-in we change the *name* to be different than the *handle*. Figure 18 shows an AXI fabric UVM setup. The environment is instantiated with a *handle* named *axi_fabric_env_0*.

```
virtual class axi_fabric_base_test extends uvm_test;

  axi_fabric_env axi_fabric_env_0;

  `uvm_component_utils(axi_fabric_base_test)

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

  extern virtual function void build_phase(uvm_phase

endclass : axi_fabric_base_test

function void axi_fabric_base_test::build_phase(uvm_phase
  super.build_phase(phase);
  axi_fabric_env_0 = axi_fabric_env::type_id::create(.name("CRAZY_ENV"),
                                                     .parent(this));
endfunction : build_phase
```

Crazy??

Figure 18: Using different name for the object than the handle in UVM

In the *build_phase* the environment is constructed using factory's *create()* – but for experimentation sake the *name* has been changed to *CRAZY_ENV*.

This directly impacts the hierarchical name of this component and all underneath (as the path has a different name). Fast-forward, consider a test writer setting an instance based factory override for the sequence item within this env-agent-sequencer. Refer to Figure 19 with an attempt to override via instance specific API.

```
function void axi_fabric_factory_test::build_phase(uvm_phase phase);
  uvm_factory f;
  super.build_phase (phase);
      set_inst_override_by_type(
      .relative_inst_path("axi_fabric_env_0.*agent*0*.*sequencer.*"),
      .original_type(axi_master_xactn::get_type()),
      .override_type(axi_hword_xactn::get_type()));
```

Figure 19: Instance specific override for AXI transaction

While the above code shall compile, run, it will not produce the intended overrides! The reason being that the *name* of the environment with the handle *axi_fabric_env_0* has been set to *CRAZY_ENV* during the build phase (Refer to Figure 18).

Figure 20 shows the correct way to get the instance specific override working in this case. Please note that this is an experimental example to demonstrate the impact of *name* argument in UVM and the authors do NOT recommend this unless it is really necessary.

```
function void axi_fabric_factory_test::build_phase(uvm_phase phase);
  uvm_factory f;
  super.build_phase (phase);
      set_inst_override_by_type(
      .relative_inst_path("CRAZY_ENV.*agent*2.axi_master_sequencer.*"),
      .original_type(axi_master_xactn::get_type()),
      .override_type(axi_byte_xactn::get_type()));

    f = uvm_factory::get();
    f.print(.all_types(0));

endfunction : build_phase
```

Figure 20: Using correct name in the instance path for a factory override

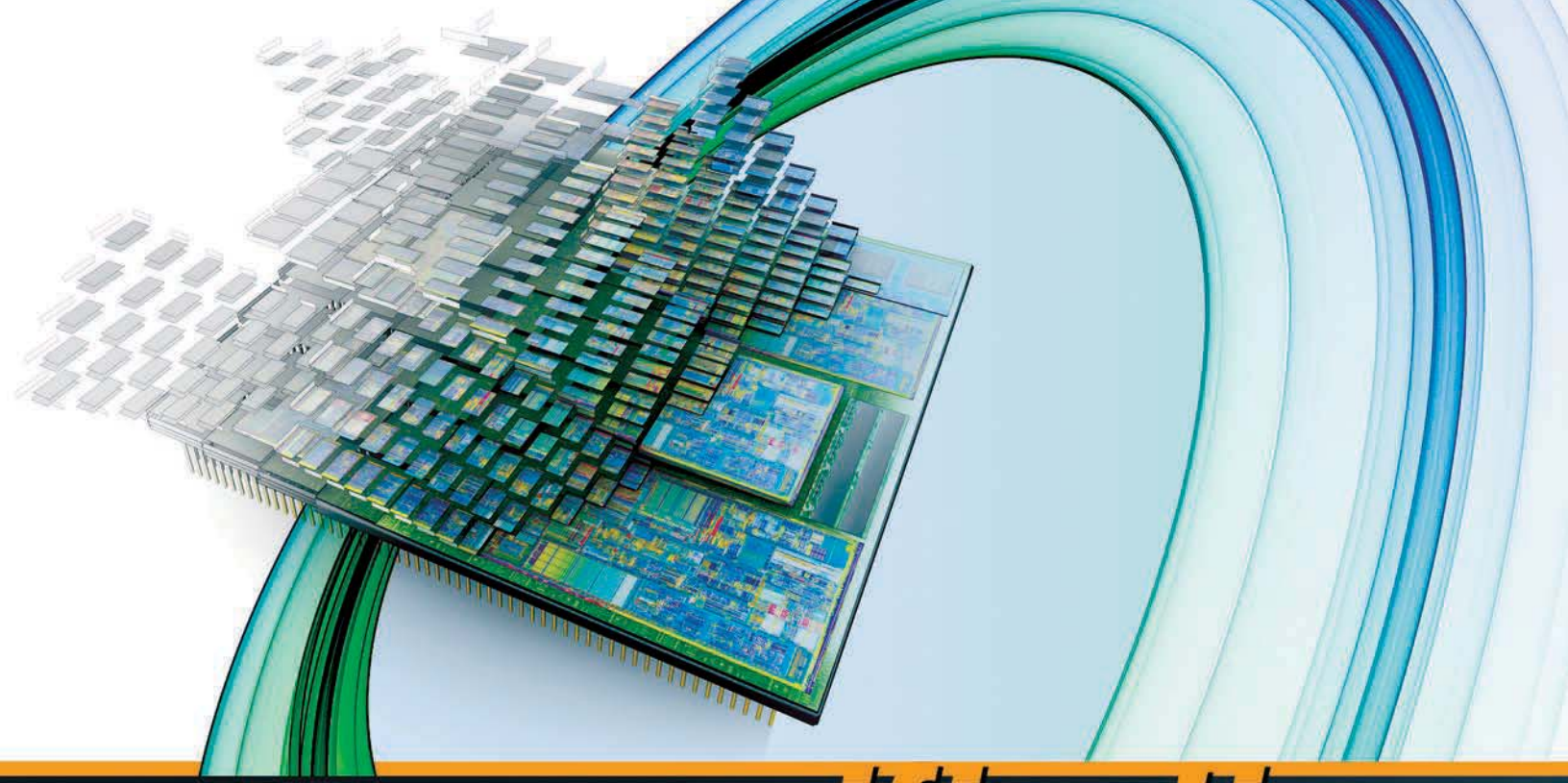
TIP: In case of instance specific overrides, the "instance path" is important and recall that it uses the "*name*" of the object than the name of the *handle*!
TIP: Keep the *name* simple and straight forward and match it to the *handle* name as much as possible
TIP: Use *get_full_name* in case the instance path is not clear.

SUMMARY

UVM is a very powerful methodology. Since it is well structured, generation of standard UVM framework as a starting point is quite common practice. Such simple automation brings productivity to the teams. Debugging UVM issues can become tricky and painful if users are not fully aware of many of the built-in debug hooks in the base class library of UVM. In this article the authors shared some of the time tested tips with respect to UVM factory. All the tips mentioned are tool and vendor independent and comes for free with UVM. Readers are encouraged to see the references section to learn more such tips and tricks.

REFERENCES

- Accellera UVM standard: http://accellera.org/activities/working-groups/uvm/
- DVCon US 2016 UVM tutorial, available on request via http://www.verifnews.org



verification HORIZONS

Editor: Tom Fitzpatrick
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777
Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

VERIFICATION HORIZONS 10 CELEBRATING TEN YEARS

Mentor Graphics®

www.mentor.com