DOES THE FACTORY SAY OVERRIDE? SUCCESSFULLY USING THE UVM FACTORY

RICH EDELMAN AND MUSTUFA KANCHWALA, MENTOR GRAPHICS

Mentor®
A Siemens Business

WHITEPAPER

FUNCTIONAL VERIFICATION

www.mentor.com

I. INTRODUCTION

The UVM [2] Factory is a SystemVerilog [1] based implementation of aspect oriented programming. It is implemented as a simple global table where overrides are stored for types. When an object is instantiated, first the factory is checked to see if the type has been overridden. If it has been overridden, then the new override type is instantiated instead of the original. This is a powerful technique to change a testbench environment without actually changing the code itself.

Yet the UVM Factory is a mysterious, powerful UVM sub-system that is either overused or underused. By that, we mean it is either used in a way that makes the testbench impossible to decipher, or it is not used at all. Both under use and overuse are a problem.

II. BASIC UVM FACTORY USAGE

Using the factory allows testbench code to remain unchanged while "substitutions" of specific object instances can be used to change the behavior of the code. In the example code below, there is a new 'env' class defined ('env_with_coverage') which contains a coverage collector. Other examples could define substitutions where a sequence is overridden by a new sequence with a different sequence item generation algorithm, or different random distribution. The original code doesn't need to be changed – just a new test or other class which creates the override. It is a very powerful way to get variations on the testbench with minimal changes. It can also be quite hard to debug once many overrides are in place, since the source code text you are looking at may have nothing much in common with the code that is actually running.

A. An example

The code below is a simple UVM example that uses the factory. The code is built to create 'env' classes. Using the factory, two kinds of overrides happen below. The first is a type override for env#(128). It is overridden with a env_with_coverage#(128). Additionally an instance specific override is defined for the instance "uvm_test_top.e1".

```
import uvm_pkg::*;
`include "uvm_macros.svh"

class env #(BITWIDTH = 128) extends uvm_component;
  `uvm_component_param_utils(env#(BITWIDTH))
  ...
endclass

class env_with_coverage #(BITWIDTH = 2048) extends env#(BITWIDTH);
  `uvm_component_param_utils(env_with_coverage#(BITWIDTH))
  ...
endclass

class test extends uvm_test;
  `uvm_component_utils(test)

  env#(128) e0;
  env#(256) e1;
  ...

function void build_phase(uvm_phase phase);
    env#(128)::type_id::set_type_override( env_with_coverage#(128)::get_type());
    env#(256)::type_id::set_inst_override( env_with_coverage#(256)::get_type(),
                                           "uvm_test_top.e1");

    e0 = env#(128)::type_id::create("e0", this);
    e1 = env#(256)::type_id::create("e1", this);
  endfunction

  task run_phase(uvm_phase phase);
    uvm_root::get().print_topology();
    factory.print();
    ...
  endtask
endclass
```

The handle e0 will be assigned the return value from the factory create() call, which will be the override env_with_coverage#128). The handle e1 will be assigned the return value from the factory create(), which will be the override env_with_coverage#(256) obtained from the instance override.

B. Trouble with debug

In the run_phase(), the debug routine factory.print() is called, producing the following unhelpful output:

```
#
#### Factory Configuration (*)
#
# Instance Overrides:
#
#   Requested Type  Override Path    Override Type
#   --------------  ---------------  -------------
#   <unknown>       uvm_test_top.e1  <unknown>
#
# Type Overrides:
#
#   Requested Type  Override Type
#   --------------  ---------------
#   <unknown>       <unknown>
#
```

Furthermore, using the get_type_name() in a print statement produces equally unhelpful output:

```
`uvm_info("OVERRIDE", $sformatf("%s starting", get_type_name()), UVM_MEDIUM)
```

The type name is reported as uvm_component:

```
# UVM_INFO t.sv(24) @ 0: uvm_test_top.e0 [OVERRIDE] uvm_component starting
```

III. PROBLEM

This paper grew out of work debugging a factory override failure. The failure was a technical detail about SystemVerilog type matching as outlined below. But the experience of trying to debug this simple problem highlighted the lack of useful debug facilities for the uvm_factory.

The code in question defined two classes – a simple env and an extension that adds coverage:

```
class env #(BIT_WIDTH = 128) extends uvm_component;
  `uvm_component_param_utils(env#(BIT_WIDTH))
  ...
endclass

class env_with_coverage #(BIT_WIDTH = 2048) extends env#(BIT_WIDTH);
  `uvm_component_param_utils(env_with_coverage#(BIT_WIDTH))
   ...
endclass
```

Then the env is instantiated as

```
class wrapper #(UINT32 BIT_WIDTH=256) extends uvm_component;
  `uvm_component_param_utils(wrapper#(BIT_WIDTH))

  env#(BIT_WIDTH) e1;
  ...
  function void build_phase(uvm_phase phase);
     // Factory here. The override is not honored.
     e1 = env#(BIT_WIDTH)::type_id::create("env1", this);
  endfunction
endclass
```

Then the test uses the wrapper

```
class test extends uvm_test;
   `uvm_component_utils(test)
   wrapper #(128) w;

  function new(string name = "test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    env#(128)::type_id::set_type_override( env_with_coverage#(128)::get_type());
    w = wrapper#(128)::type_id::create("w", this);
  endfunction
```

The details of the problem are beyond the scope of this paper, but basically the base 'env' and the extended 'env_with_coverage' get instantiated with an 'int' or an 'unsigned int'. This makes the two classes assignment incompatible, and the factory override will fail.

Interestingly, the effect of the failure is no error message and no notification of failure. The override simply fails. In this case, this means that no coverage is collected. It is silent failure, which in our opinion is a very bad situation. In this case it was easy enough to detect, since no coverage was being collected. In other cases where the override changes a random distribution or seed values, then the failed override might never be detected.

A. *Fix 1 : Remove the UINT32*

The "fix" for this problem is to remove the UINT32. The UINT32 causes the BITWDITH 128 to be a signed number in one case and an unsigned number in another case. Removing the UINT32 allows the BITWIDTH to be a signed number in all cases.

The simple code below demonstrates the issue with simple SystemVerilog using assignment of handles.

```
typedef int unsigned UINT32;

class C #(BITWIDTH);
endclass

class C2 #(UINT32 BITWIDTH) extends C#(BITWIDTH);
endclass

module top();
  C  #(128) c_signed_int_128;
  C2 #(128) c_unsigned_int_128;

  initial begin
    c_unsigned_int_128 = new();
    c_signed_int_128   = c_unsigned_int_128;
  end
endmodule
```

Produces

```
# ** Fatal: Illegal assignment to class work.t_sv_unit::C  #(128)
#                             from class work.t_sv_unit::C2 #(128)
```

If the UINT32 is removed, then no error message is emitted.

B. *Fix 2: Macro replacements*

A different fix is to NOT use the uvm_component_param_util macro, but instead to use the helper functions outlined below (in blue). Using the macro causes no name to be registered ("<unknown>"). Using the helper functions will register a proper name. With the helper functions, the UVM produces a nice error message:

```
  # UVM_FATAL @ 0: reporter [FCTTYP] Factory did not return a component of type 'env#(137)'.
A component of type 'env_with_coverage#(137)' was returned instead. Name=env1
Parent=wrapper#(137) contxt=uvm_test_top.wrapper
```

```
class env #(BIT_WIDTH = 128) extends uvm_component;
  // `uvm_component_param_utils(env#(BIT_WIDTH))
  localparam type_name = $sformatf("env#(%0d)", BIT_WIDTH);
  typedef uvm_component_registry #(env#(BIT_WIDTH), type_name) type_id;

  static  function type_id get_type();
    return type_id::get();
  endfunction

  virtual function uvm_object_wrapper get_object_type();
    return type_id::get();
  endfunction

  virtual function string get_type_name();
    return type_name;
  endfunction
     ...
endclass
```

For more complex types, the technique is extensible:

```
// Complex type names
class C #(type E1, type E2) extends
uvm_component;

  localparam type_name = $sformatf("C#(%s,%s)", E1::type_name, E2::type_name);
  typedef uvm_component_registry #( C#(E1, E2), type_name) type_id;

  static  function type_id get_type();
    return type_id::get();
  endfunction

  virtual function uvm_object_wrapper get_object_type();
    return type_id::get();
  endfunction

  virtual function string get_type_name();
    return type_name;
  endfunction
  ...
endclass
```

There were previously many reasons to not use the uvm_component_param_utils macro. This is an additional one. Due to the coding structure in the UVM Factory, when the override name is "<unknown>", then a quiet failure will occur. When the override name is defined (as when using the helper functions above), then the failure is detected and printed, generating a FATAL error and immediately stopping simulation.

IV. UVM FACTORY API

The UVM Factory has a simple interface and simple use model. An override is registered, and later a type is looked up to find any registered override. The registration of types takes two forms. The first form is an override for type A to type B that applies to any request for type A. The second form is a refinement of the first form. It overrides a type for a specific instance. This registered instance override will be applied for this instance instead of any of the first form type overrides. These two forms are "override_by_type" and "instance_override". An unnecessary complication is the use of wildcards. Wildcards can be used with the instance overrides. Using wildcards may make the code slower, and will cause unnecessary complexity.

The UVM Factory has a variety of lookup routines, but is normally used directly through the "create()" interface, to either create a UVM component or a UVM object.

The UVM Factory has many other ways to register overrides or perform lookups. As a rule, we advise against using them both from a simplicity and transparency point of view, but also from a bug-free point of view. Using the recommended APIs below has proven to be simple, transparent, and relatively bug-free.

A. Setting *an override by type, or by instance:*

```
static function void set_type_override(uvm_object_wrapper override_type, bit replace=1);
static function void set_inst_override(uvm_object_wrapper override_type, string inst_path,
                                       uvm_component parent=null);
```

To create a type override, call set_type_override() with the replacement type as the argument.

```
env#(128)::type_id::set_type_override( env_with_coverage#(128)::get_type());
```

To create an instance override, call set_inst_override() with the replacement type and the instance name as the arguments.

```
env#(128)::type_id::set_inst_override( env_with_coverage2#(128)::get_type(),
                                       "uvm_test_top.w.e0");
```

B. Creating a new object by type (either UVM object or UVM component):

```
static function T create (string name="", uvm_component parent=null, string contxt="");
```

To create an object using the UVM Factory, declare class handles using the base class (the original type). Then use the create() call to access the factory, checking for an override behind the scenes.

```
env#(BITWIDTH) e0;
env#(BITWIDTH) e1;

function void build_phase(uvm_phase phase);
  e0 = env#(BITWIDTH)::type_id::create("e0", this);
  e1 = env#(BITWIDTH)::type_id::create("e1", this);
endfunction
```

V. Previous Work

The references below outline details about best practices and experience with the UVM factory. They each have a contribution for the eager factory user. The Cummings paper [4] is a general overview of the UVM factory that should not be missed. For Specman fans, there is a discussion on the factory in terms of 'e' [5]. The final factory reference, starting on slide 48, has a good discussion of the UVM Factory in tutorial form with lots of detail and background [6]. There are many other internet references and guides for the UVM factory.

VI. UVM 1.2 Changes

The UVM 1.2 updates have changed the factory in a few significant, non-backward-compatible ways. None of the changes are fatal. First the global 'factory' variable has been removed. Second the uvm_factory class is now an abstract class, and you must either define your own implementation or use the uvm_default_factory. Third, related to the abstract base class, many of the function calls in the factory are now 'pure virtual', meaning if you do create your own factory, you'll need to implement each of the calls yourself. These changes to add 'virtual-ness" to the UVM Factory seem out of place. These techniques are known-good ways to create "better object oriented software". But in the case of the UVM, the uvm_factory was already released and working as a non-optimal implementation. The changes did not fix any bugs nor did they improve the debug-ability. We have not heard users complain about the OOP-ness of the factory. In our opinion the effort to make these UVM 1.2 changes should have been applied elsewhere, or at least to improving debugability of the factory.

We're hoping UVM 1.2a will get some of the changes to make parameterized classes first class citizens in the factory, and will plug holes and fix the bugs that exist.

VII. What to do when the Factory goes wrong

A. Better tracking and tracing – called FILE and LINE

The UVM Factory is a way to override type definitions. The source of the override – file and line number need to be tracked. This would require additional arguments in the major API calls, along with a simple addition of file and line number origin for each override.

B. Each decision gets explained

A debug mode should be added, so that each override that either matches or doesn't match is printed, along with the extra information. In this way a user can follow the execution flow to understand what decisions are being made.

C. Current state – simulate interactively

The factory has many lines (about 2000 lines in uvm_factory.svh and uvm_registry.svh) of intricate code and data structures. But basically it is a simple lookup. We have a type and an optional instance path. We want to know if there is an override for the type based on the instance path or if there is a general override for any instance of this type. If there is an override, then we'll use that.

Focusing on the find_override_by_type() routine (83 lines of code and functionally unchanged in UVM 1.2), after removing the instance path lookup and some loop checking:

```
function uvm_object_wrapper uvm_factory::find_override_by_type(
  uvm_object_wrapper requested_type, string full_inst_path);
  ...Loop check
  ...Instance path type overrides
  // type override - exact match
  foreach (m_type_overrides[index]) begin
    if (m_type_overrides[index].orig_type == requested_type ||
        (m_type_overrides[index].orig_type_name != "<unknown>" &&
         m_type_overrides[index].orig_type_name != "" &&
         requested_type != null &&
         m_type_overrides[index].orig_type_name == requested_type.get_type_name())) begin
      // Match!
      m_override_info.push_back(m_type_overrides[index]);
      ...Match related work
    end
  end
  ...Recursive check on further overrides
  return requested_type;
endfunction
```

This is straightforward code. For each override that exists, check to see if the types match, or if the type names match, if the type name is NOT "<unknown>", if the type name is NOT empty ("") and if the requested type is not NULL. Although it is simple code, it is still making complex decisions. For better debug, these decisions need to be instrumented or annotated, so that mismatches and false matches can be understood.

VIII. Best Practices

If using parameterized classes with the factory, avoid the uvm_component_params_util macro. Use the helper functions outlined above.

Use a small subset of the UVM Factory API (set_type_override(), set_inst_override() and create()).

```
sum_seq::type_id::set_type_override( sum_plus_one_seq::get_type());
sum_seq::type_id::set_inst_override( sum_plus_two_seq::get_type(),
         "uvm_test_top.virtual_sequencer.sum_virtual_sequence.sum2");
```

Don't use wildcards in instance path overrides.

If you are going to use factory overrides, use them sparingly.

Don't override an override. It makes things complicated.

In any overridden object, print out a message identifying that it is running. That way a logfile check can prove that it was created and executed.

```
task run_phase(uvm_phase phase);
  `uvm_info("OVERRIDE", $sformatf("%s starting", get_type_name()), UVM_MEDIUM)
…
endtask
```

```
# UVM_INFO t.sv(58) @ 0: uvm_test_top.w.env1 [OVERRIDE] env_with_coverage#(137) starting
# UVM_INFO t.sv(37) @ 0: uvm_test_top.e1     [OVERRIDE] env#(137) starting
# UVM_INFO t.sv(37) @ 0: uvm_test_top.e0     [OVERRIDE] env#(137) starting
# UVM_INFO t.sv(58) @ 0: uvm_test_top.c.env2 [OVERRIDE] env_with_coverage#(256) starting
# UVM_INFO t.sv(58) @ 0: uvm_test_top.c.env1 [OVERRIDE] env_with_coverage#(137) starting
```

IX. New Factory

The UVM Factory is complex code, but generally not the place where most testbench problems are located. Nonetheless, the code could be made simpler, more transparent and thus have fewer bugs and be more debuggable.

A factory model is listed in Appendix I. It is not supplied as a drop in replacement for the existing UVM Factory, but simply as an example of a much simpler model with most (if not all) the desired functionality. It requires the use of $cast(), but with a bit more integration this could be eliminated. There was no attempt at a more complete integration with the existing UVM Factory and UVM Registry aside from using the registry to manage the proxies.

Using the new factory is simple – call the create_component_by_type() in the new factory.

```
$cast(a1, new_factory::create_component_by_type(agent1::get_type(), "", "a1", this));
```

This is equivalent to

```
a1 = new("a1", this);
```

Or

```
a1 = agent1::type_id::create("a1", this);
```

Creating an instance or type override is simple – call register_instance_path_override() or register_type_override().

Instance "a.b.c" will be overridden by type 'agent5'.

```
new_factory::register_instance_path_override("a.b.c", agent5::get_type());
```

The type 'agent1' will be overridden by type 'agent2'.

```
new_factory::register_type_override(agent1::get_type(), agent2::get_type());
```

X. SUMMARY

This paper outlined a simple UVM Factory example and provided sample code. A user problem with type matching and the poor messages from the UVM Factory were discussed, along with solutions to the underlying problem, changes to the user code and changes to the UVM Factory that might improve results.

A new factory is listed as a starting point for thinking about simplifying the UVM Factory, improving transparency and improving debuggability. The source code for all examples is available from the authors.

REFERENCES

[1] IEEE 1800-2012 SystemVerilog LRM, https://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[2] Universal Verification Methodology, UVM 1.1d, http://workspace.accellera.org/downloads/standards/uvm/uvm-1.1d.tar.gz

[3] "Please! Can someone make UVM easier to use?", Rich Edelman, Raghu Ardeishar, DVCON India 2014, https://dvcon-india.org/sites/dvcon-india.org/files/archive/2014/proceedings/dv-papers/Can_Someone_Make_UVM_Easy_Paper.pdf

[4] "The OVM/UVM Factory & Factory Overrides How They Work - Why They Are Important", Cliff Cummings, http://www.sunburst-design.com/papers/CummingsSNUG2012SV_UVM_Factories.pdf

[5] "Rambling About UVM Factory Overrides - Per Instance", Verification Gentleman, http://blog.verificationgentleman.com/2015/07/rambling-about-uvm-factory-overrides-per-instance.html

[6] "Behind the Scenes of the UVM Factory", Mark Litterick, Verilab, DVCON 2014 Tutorial, http://www.verilab.com/files/verilab_dvcon_tutorial_a.pdf

APPENDIX I

An example factory model which is smaller and simpler than the UVM Factory. This model could be a starting point for improvements to the UVM Factory in a future release. There is limited error checking, and no attempt to duplicate all the functionality of the UVM Factory was made.

```
new_factory.svh:
class new_factory;
  // Lookup by type: old_type -> new_type
  static uvm_object_wrapper override_by_type                [uvm_object_wrapper];
  static int                override_by_type_count          [uvm_object_wrapper]; //Stats

  // Lookup by instance path name: instance_path_name -> new_type
  static uvm_object_wrapper override_by_instance_path       [string            ];
  static int                override_by_instance_path_count [string            ]; //Stats

  // Registration
  static function void register_instance_path_override(string instance_path,
                                                       uvm_object_wrapper obj);
    override_by_instance_path[instance_path] = obj;
    override_by_instance_path_count[instance_path] = 0; //Stats
  endfunction

  static function void register_type_override(uvm_object_wrapper old_type_obj,
                                              uvm_object_wrapper new_type_obj);
    override_by_type[old_type_obj] = new_type_obj;
    override_by_type_count[old_type_obj] = 0; //Stats
  endfunction

  // Lookup
  static function uvm_object_wrapper get_override(uvm_object_wrapper obj,
                                                  string instance_path = "");
    if (instance_path != "") // Get an instance override first.
      if (override_by_instance_path.exists(instance_path))
        if (override_by_instance_path[instance_path] == obj) // Loop. Probably an error.
          return obj; // Return original type.
        else begin
          override_by_instance_path_count[instance_path]++; //Stats
          return get_override(override_by_instance_path[instance_path], instance_path);
        end

    if (override_by_type.exists(obj)) // Get a global type override second.
      if (override_by_type[obj] == obj) // Loop. Probably an error.
        return obj; // Return original type.
      else begin
         override_by_type_count[obj]++; //Stats
         return get_override(override_by_type[obj], instance_path);
      end
    else
      return obj; // Return original type.
  endfunction

  // Construction
  static function uvm_object create_object_by_type(uvm_object_wrapper requested_type,
                                                   string instance_path = "",
                                                   string name = "");
    // Get the "new" type.
    requested_type = get_override(requested_type, instance_path);
    // Create one.
    return requested_type.create_object(name);
  endfunction
```

```
  static function uvm_object create_component_by_type(uvm_object_wrapper requested_type,
                                                      string instance_path = "",
                                                      string name = "",
                                                      uvm_component parent = null);
    // Get the "new" type.
    requested_type = get_override(requested_type, instance_path);
    // Create one.
    return requested_type.create_component(name, parent);
  endfunction

  // Debug
  static function void print();
    foreach (override_by_type[obj])
      `uvm_info("FACTORY", $sformatf("         Type '%-6s' maps to '%-6s' used %0d times",
        obj.get_type_name(),
        override_by_type[obj].get_type_name(),
        override_by_type_count[obj]
          ), UVM_MEDIUM)

    foreach (override_by_instance_path [instance_path])
      `uvm_info("FACTORY", $sformatf("Instance Path '%-6s' maps to '%-6s' used %0d times",
        instance_path,
        override_by_instance_path[instance_path].get_type_name(),
        override_by_instance_path_count[instance_path]
          ), UVM_MEDIUM)
  endfunction
endclass

t.sv:
import uvm_pkg::*;
`include "uvm_macros.svh"

`include "new_factory.svh"

class agent extends uvm_agent;
  `uvm_component_utils(agent);
  int N;

  function new(string name = "agent", uvm_component parent = null);
    super.new(name, parent);
    N = -1;
  endfunction

  task run_phase(uvm_phase phase);
    `uvm_info(get_type_name(), "Starting...", UVM_MEDIUM)
    for(int i = 0; i < 3; i++)
      #10 `uvm_info(get_type_name(), $sformatf("...%0d...%0d", N, i), UVM_MEDIUM)
  endtask
endclass

class agent0 extends agent;
  `uvm_component_utils(agent0);

  function new(string name = "agent0", uvm_component parent = null);
    super.new(name, parent);
    N = 0;
  endfunction
endclass

class agent1 extends agent;
  `uvm_component_utils(agent1);

  function new(string name = "agent1", uvm_component parent = null);
    super.new(name, parent);
    N = 1;
  endfunction
endclass

class agent2 extends agent;
  `uvm_component_utils(agent2);

  function new(string name = "agent2", uvm_component parent = null);
    super.new(name, parent);
    N = 2;
  endfunction
endclass
```

```
class agent3 extends agent;
  `uvm_component_utils(agent3);

  function new(string name = "agent3", uvm_component parent = null);
    super.new(name, parent);
    N = 3;
  endfunction
endclass

class agent4 extends agent;
  `uvm_component_utils(agent4);

  function new(string name = "agent4", uvm_component parent = null);
    super.new(name, parent);
    N = 4;
  endfunction
endclass

class agent5 extends agent; // Only used for instance_path overrides
  `uvm_component_utils(agent5);

  function new(string name = "agent5", uvm_component parent = null);
    super.new(name, parent);
    N = 5;
  endfunction
endclass

class env extends uvm_env;
  `uvm_component_utils(env);

  agent a1, a2, a3, a4, ai;
  agent am[10]; // Many

  function new(string name = "env", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
   //d = agent::type_id::create("agent", this);
   $cast(a1, new_factory::create_component_by_type(agent1::get_type(), "", "a1", this));
   $cast(a2, new_factory::create_component_by_type(agent2::get_type(), "", "a2", this));
   $cast(a3, new_factory::create_component_by_type(agent3::get_type(), "", "a3", this));
   $cast(a4, new_factory::create_component_by_type(agent4::get_type(), "", "a4", this));
   $cast(ai, new_factory::create_component_by_type(agent4::get_type(), "a.b.c", "ai",
                                                                              this));
    foreach (am[i])
      $cast(am[i], new_factory::create_component_by_type(
        agent4::get_type(), "", $sformatf("am[%0d]", i), this));
  endfunction
endclass

class test_base extends uvm_test;
  `uvm_component_utils(test_base)

  env e;

  function new(string name = "test_base", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    new_factory::register_instance_path_override("a.b.c", agent5::get_type());
    new_factory::register_instance_path_override("x.y.z", agent0::get_type());
    //e = env::type_id::create("e", this);
    $cast(e, new_factory::create_component_by_type(env::get_type(), "", "e", this));
  endfunction

  task run_phase(uvm_phase phase);
    `uvm_info(get_type_name(), "Starting...", UVM_MEDIUM)
    phase.raise_objection(this);
    #100;
    phase.drop_objection(this);
    new_factory::print(); // Print at the end, so that usage is up-to-date
  endtask
endclass
```

```
class test_no_override extends test_base;
  `uvm_component_utils(test_no_override)

  function new(string name = "test_no_override", uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass

class test_even extends test_base;
  `uvm_component_utils(test_even)

  function new(string name = "test_even", uvm_component parent = null);
    super.new(name, parent);
    new_factory::register_type_override(agent1::get_type(), agent2::get_type());
    new_factory::register_type_override(agent3::get_type(), agent4::get_type());
  endfunction
endclass

class test_odd extends test_base;
  `uvm_component_utils(test_odd)

  function new(string name = "test_odd", uvm_component parent = null);
    super.new(name, parent);
    new_factory::register_type_override(agent2::get_type(), agent1::get_type());
    new_factory::register_type_override(agent4::get_type(), agent3::get_type());
  endfunction
endclass

class test_all_2 extends test_base;
  `uvm_component_utils(test_all_2)

  function new(string name = "test_all_2", uvm_component parent = null);
    super.new(name, parent);
    new_factory::register_type_override(agent1::get_type(), agent2::get_type());
    // Chain. 3 -> 4 -> 2
    new_factory::register_type_override(agent3::get_type(), agent4::get_type());
    new_factory::register_type_override(agent4::get_type(), agent2::get_type());
  endfunction
endclass

module top();
  initial begin
    run_test();
  end
endmodule
```

This paper was originally presented at DVCon India 2016, Bangalore.

Corporate Headquarters
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204

Sales and Product Information
Phone: 800.547.3000
sales_info@mentor.com

Silicon Valley
Mentor Graphics Corporation
46871 Bayside Parkway
Fremont, CA 94538 USA
Phone: 510.354.7400
Fax: 510.354.7467

North American Support Center
Phone: 800.547.4303

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics (Taiwan)
11F, No. 120, Section 2,
Gongdao 5th Road
HsinChu City 300,
Taiwan, ROC
Phone: 886.3.513.1000
Fax: 886.3.573.4734

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Trust Tower
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140-0001
Japan
Phone: +81.3.5488.3033
Fax: +81.3.5488.3004

MSB 11-17 TECH16430-w