World Class Verilog, SystemVerilog & OVM/UVM Training

The OVM/UVM Factory & Factory Overrides How They Work - Why They Are Important

Clifford E. Cummings
Sunburst Design, Inc.
www.sunburst-design.com

ABSTRACT

Factory patterns are not new to the software world, and OVM/UVM have incorporated the factory into its primary methodology. But what does the factory really do and why is it important?

This paper will explain fundamental details related to the OVM/UVM factory and explain how it works and how overrides facilitate simple modification to the testbench component and transaction structures on a test by test basis. This paper will further demonstrate that OVM/UVM environments can mostly ignore the factory but will explain why the factory should be used.

Table of Contents

Table of Figures

Table of Examples

1. Introduction

From an OVM/UVM perspective, besides instantiating the Design Under Test (DUT) and the SystemVerilog interface that interacts with the DUT, the top module typically places the virtual interface wrapper into a uvm_object configuration lookup table and executes the **`run_test()`** command.

In the top module source code, the **`run_test()`** command should not include the name of the test to be run, because including the test name would force an engineer to modify the top-level module source code to run a new test, or would require the verification engineer to maintain multiple top-level modules. The **`run_test()`** method call should extract its test name from the command line switch: **`+UVM_TESTNAME=`**<*testname*>

Every test component and transaction type should be registered with the factory, but technically only the test must be registered with the factory (described later) . No other testbench component is required to be registered with the factory (but they should!)

2. The Term "Factory"

The term factory refers to the fact that the recommended OVM/UVM methodology dictates that engineers should never construct components and transactions directly using the **`new()`** class constructor, but should makes calls to a special look-up table to create the requested component and transaction types. The factory is that special look-up table.

When you call the factory to create the requested component or transaction type, the factory itself will create the object by calling the constructor that was defined for that object. The constructors are typically one of the two following templates:

For components, there is a tree-like hierarchy required to build the testbench structure, where each component builds all of the components that are one-level lower in the hierarchy, so each component names (and builds) its children, and passes a pointer to itself (the **`this`** pointer) to each child component, so they know where they are located in the hierarchy (who is the parent device for each constructed component). For components, the typical constructor is shown in Example 1.

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

Example 1 - Standard new() constructor for UVM components

For transactions (data objects), each object is a unit of data with multiple fields, and transactions do not have a parent. For transactions, the typical constructor is shown in Example 2.

```
function new (string name="class_name");
  super.new(name);
endfunction
```

Example 2 - Standard new() constructor for UVM transactions (data objects)

Notice that components typically do not include a default name value, but transactions do. Since a parent builds each component, the parent will name each child component, so any name that you would have given to a component is going to be overridden; hence, there is no good reason to name the components in their user-defined constructor. There are multiple examples on different websites and in different tutorials that include the class name as a default name value, and that set the parent to a default value of **`null`**. This is a complete waste of time and usually causes confusion for engineers that find the examples and try to determine when to add default names and **`null`** and when to omit them. The component defaults are just confusing and should just be omitted.

Transactions are also typically named when constructed, but there are times when transaction handles are declared but not created. For this reason, it is recommended that the user-defined constructor ***should*** include a name that matches the name of the transaction class where the constructor is defined.

3. Transaction Types Terminology

Even though the UVM base classes include a built-in **`uvm_transaction`** type, it is rarely used directly. In general, verification engineers should build transaction types from the **`uvm_sequence_item`** type, which is extended from the **`uvm_transaction`** type, and which is easily executed by **`uvm_sequence`** types on a **`uvm_sequencer`** component.

The term transaction type will be used throughout this paper to represent **`uvm_sequence_item`** type items executed by one or more **`uvm_sequences`**.

4. Tests & Regression Tests

In OVM/UVM, a **`uvm_sequence`** is really a partial or complete test, while the **`uvm_test`** is really a collection of one or more sequences that are started on a **`uvm_sequencer`** and hence what we typically call a test can really be thought of as a single test executing a single sequence, or a group of sequences executed as separate tests within the top-level test.

It is probably easier to think of **`uvm_sequences`** as tests and the top-level test as a regression suite of those tests. Only one test is allowed to run per simulation and that test is called by executing the **`run_test()`** command with the **`+UVM_TESTNAME=`**<*test_name*> command line switch. It is not possible to execute multiple top-level tests, which is why the top-level test should be thought of as the regression suite, while the sequences should be thought of as the individual tests that are run by the top-level regression suite (top-level test).

5. Why is there an OVM/UVM Factory?

If it is technically not necessary to register any components (except the top-level test) or transaction types with the factory, why have a factory?

The recommended method in OVM/UVM for creating testbench components or transaction objects is to use the built-in method **`::type_id::create`** command, whose fuctionality is more fully explained in section 18.

Using the **`::type_id::create`** command makes a call to the factory to extract the requested component or transaction type and then uses the **`new()`** constructor that is included in the class type to build a copy of the class-type object, all of which is done at run time. Whatever class type is stored in the factory look-up table at the requested **`type_id`** location, is extracted and created. The factory makes it possible to allow a compatible type to be stored at the desired location and therefore a compatible substitute can be automatically requested when the **`::type_id::create command`** is executed.

The factory permits a top-level test to make a substitution for one of the component or transaction types in the factory at run-time, before building the entire testbench environment using factory overrides.

For example, it may be desirable to do testing with two different transactions types, **`trans1`** and **`trans2`**, where **`trans2`** is an extension of **`trans1`**, but includes additional randomized data members. Since the sequencer, driver and monitor are all classes that are parameterized to a specific transaction type, the test writer can simply substitute the **`trans2`** type for the **`trans1`** type in the factory at the start of the test. Using a typical scenario, when the test builds the environment, and the environment builds the agent, and the agent builds the **`trans1`**-parameterized sequencer, driver and monitor, the parameterized agent components will be parameterized with the transaction type stored in the factory at the **`trans1`** location, which is now the compatible **`trans2`** type. There was no need to keep two copies of the sequencer, driver and monitor, and therefore no reason to have a second agent type that uses the second copy of the sequencer driver and monitor. It is still possible to run the older tests that used the **`trans1`** type and use the exact same testbench structure to run new tests using the **`trans2`** type.

As a second example, we may want to use the same transaction type, but we would like the data to be sent to the DUT in a serial fashion instead of a parallel fashion. The serial version of the DUT could use the exact same testbench structure as long as the driver sends the transaction as serial data and the monitor samples output transactions as serial data. For this environment, the test could substitute a second version of the driver and monitor into the factory so that when those components are built, the serial versions of those devices will be constructed without requiring the entire testbench environment to be re-coded.

Any components or transactions constructed from calls to the factory will simply look-up the required **`type_id`** and construct that object. Since the components and transactions must be compatible types, the factory ensures the polymorphic type-safety of the required components.

The factory provides a partial replacement for the "when-inheritance" that one might use with Aspect-Oriented languages, only with finer granularity (one can replace specific objects without replacing all objects of the same type) and without the need for the compiler to re-compile all of the classes that make-up the testbench. Aspect Oriented programming has a simpler syntax, but factory substitution is much more compile efficient.

6. Registering Components & Transactions With the Factory

Testbench components should be registered with the factory using the command:
```
`uvm_component_utils(component_name)
```

Testbench transactions should be registered with the factory using the command:
```
`uvm_object_utils(transaction_type)
```

As will be described later, UVM ports are never registered with the factory, and covergroups are also never registered with the factory. In general TLM fifos are also not registered with the factory.

Two other factory registration macros have been deprecated from UVM and should not be used: **`` `uvm_sequencer_utils() ``** and **`` `uvm_sequence_utils(). ``** These macros had the unfortunate side effect of tying a particular sequencer to a particular sequencer_item type and conversely tied a particular sequence_item type to a particular sequencer type, which greatly reduced the flexibility that should have existed in using and swapping different testbench component types and transaction types.

Unfortunately there are a very large number of examples in circulation that use both of these deprecated **`` `ovm_sequence ``**/**`sequencer_utils()`** macros, including in the OVM User Guide[2].

7. Object Constructors

Class based constructors are the built-in or user-defined **`new()`** constructors.

Constructing is typically done by calling the **`new()`** constructor, but OVM/UVM components and **`sequence_items`**/**`sequences`** should be constructed using the **`create()`** command.

Guideline: construct OVM/UVM components using the **`create()`** command.
Guideline: construct OVM/UVM sequence_item/sequences using the **`create()`** command.
Guideline: construct OVM/UVM port types using the **`new()`** constructor.
Guideline: construct covergroups using the **`new()`** constructor in the component **`new()`** functions.

Technically, none of the testbench components has to be built using the **`create()`** command. All of the testbench components (except the top-level test) can be built using the **`new()`** constructor, but using the **`new()`** constructor to build any of the testbench components or transaction types should never be done.

Using the **`new()`** constructor hard-codes the exact object type to be built into the testbench component files and severely limits the flexibility that was designed into the OVM/UVM methodologies. Using the factory **`create`** command allows the top-level test to substitute a compatible type into the factory and it is that type that will be created and used at run-time.

By using the factory, the **`create`** command simply calls for a component of the named type to be created, but the top-level test has the option to substitute into the factory a compatible extended type for any registered component type or transaction type.

8. Factory Registering and Creating

Once a component is registered with the factory, the component can be created in the **`build_phase()`**at run-time, during the simulation, without the need to re-compile the new component or transaction types.

The factory provides a simulation-efficient mechanism to provide substitute components or transaction types on a test-by-test basis.

When properly coded, all of the components have been compiled and registered with the factory and it is the top-level test that can make a last-minute substitution at the beginning of the **`build_phase()`**to determine the actual component type that will be created when that component **`type_id`** is requested from another component.

Similarly and again when properly coded, any compatible transaction type that has been compiled and registered with the factory can be substituted by the top-level test before the required transaction type is even requested.

9. Build Phase

The OVM **`build()`** and UVM **`build_phase()`** builds the entire testbench environment top down, and the first component built is the selected test. The test is an ideal place to override any of the testbench component types in the factory before any other component is created.

10. Factory - Keys to Understanding

`build_phase()` execution happens top-down, so the top-level tests (and other components) can change the transaction types and components by doing overrides.

The test first makes run-time substitutions (overrides) into the factory for all components and transactions that do not match the base testbench and transaction structure, then proceeds to build each of the testbench components top-down. Since substitutions for components and transactions happened in the top-level test before any **`::type_id::create`** commands were called, the new components and transactions will be called and constructed out of the factory.

11. Simple Demonstration Model

In this section and in section 12, a simple demonstration model will be coded using the factory and then without the factory. Each example will also be coded using two more tests. In the second test, a second version of the transaction (**`trans2`**) will be added. In the third test, a second version of the **`tb_driver`** will also be added. We will examine the coding efforts to do the additional tests with the modified transaction and **`tb_driver`**.

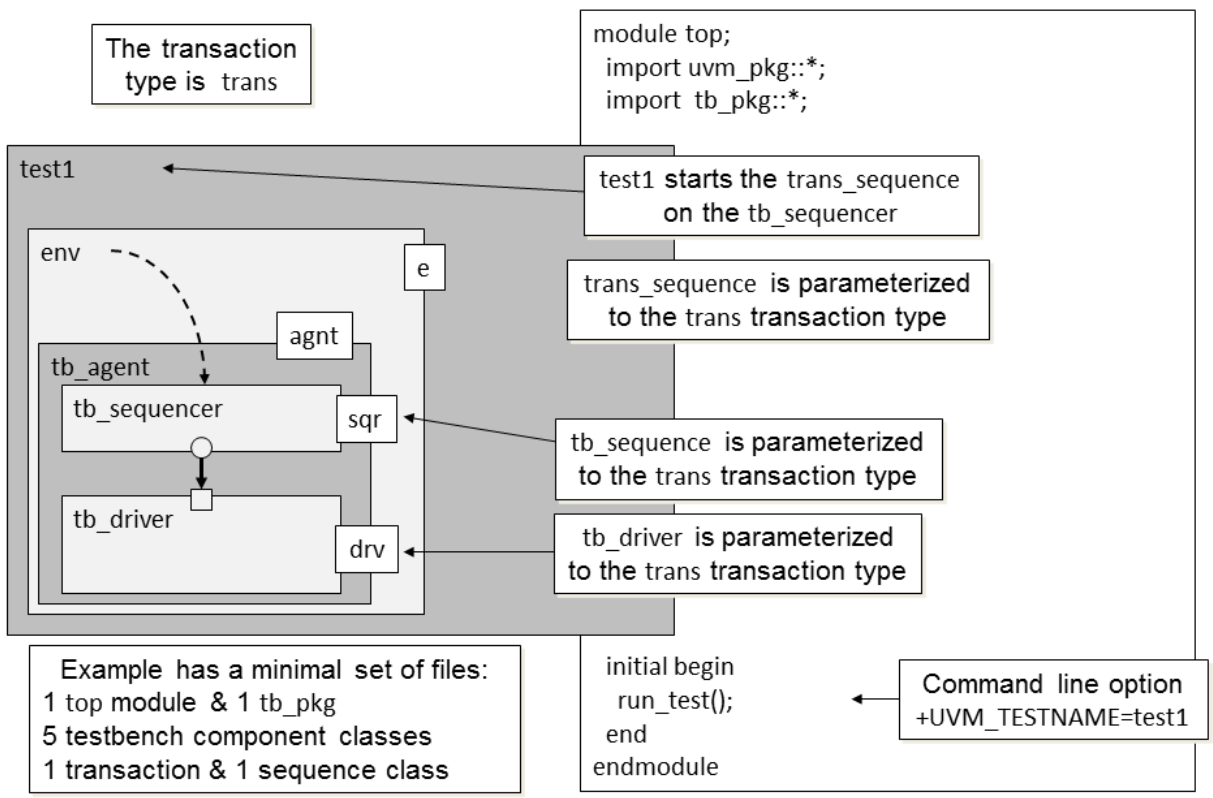

Figure 1 - Simple Demonstration Example Block Diagram

The simple demonstration model uses 1 **`top`**-module, 1 **`tb_pkg`**, 5 component classes, 1 transaction class and 1 sequence class.

Common top module

For both the factory and non-factory versions of the tests, a common **`top`**-module will be used that, for simplicity purposes, will not contain a DUT to test. The top module simply uses the **`run_test()`** command to start the simulations, and each simulation will specify the desired test using the **`+UVM_TESTNAME=`**<*testname*> command line option. The common **`top`**-module is shown in Example 3.

```
module top;
  import uvm_pkg::*;
  import  tb_pkg::*;

  initial begin
    run_test();
  end
endmodule
```

Example 3 - top.sv module used for all tests

12. Testbench Classes Using a Factory

The classes in the **`test1`**-version of the factory-based example are included into the **`tb_pkg`** shown in Example 4. It is important to include the classes in the correct order to ensure that classes that depend upon other class definitions, are included after the required classes are already included.

```
`ifndef TB_PKG
`define TB_PKG

`include "uvm_macros.svh"

package tb_pkg;
  import uvm_pkg::*;

  `include "trans.sv"

  `include "tb_driver.sv"
  `include "tb_sequencer.sv"
  `include "tb_agent.sv"
  `include "env.sv"

  `include "trans_sequence.sv"
  `include "test1.sv"
endpackage

`endif
```

Example 4 - tb_pkg1.sv package file

Factory-based transaction

The transaction class is shown in Example 5, and includes two randomiazable data fields, the standard **`new()`** constructor and the recommended **`convert2string()`** method that can be called to show the current contents of the specified transaction object. The code that registers the transaction with the factory is the command, **`` `uvm_object_utils(trans). ``** A description of the behavior of the **`` `uvm_object_utils() ``** command will be explained in a later section of this paper.

```
class trans extends uvm_sequence_item;
  `uvm_object_utils(trans)
  rand bit  [7:0] data;
  rand bit [15:0] addr;

  function new (string name="trans");
    super.new(name);
  endfunction

  function string convert2string;
    string s;
    $sformat(s, "trans1:  addr = %4h  data = %2h", addr, data);
    return s;
  endfunction
endclass
```

Example 5 - trans.sv sequence_item (factory version)

Factory-based sequence

The demonstration example also uses a simple sequence (shown in Example 6) that generates and randomizes 10 transactions that will be sent to the **`tb_sequencer`**. The code that registers the transaction with the factory is the command, **`` `uvm_object_utils(trans_sequence). ``**

```
class trans_sequence extends uvm_sequence #(trans);
  `uvm_object_utils(trans_sequence)

  function new (string name="trans_sequence");
    super.new(name);
  endfunction

  task body();
    trans tx = trans::type_id::create("tx");
    repeat(10) begin
      start_item(tx);
        assert(tx.randomize());
      finish_item(tx);
    end
  endtask
endclass
```

Example 6 - trans_sequence.sv sequence (factory version)

Factory-based test1

The demonstration example is executed using the **`test1`** class, as shown in Example 7.

The code that registers **`test1`** with the factory is the command, **`` `uvm_component_utils(test1). ``** Although the transaction and sequence were both registered with **`` `uvm_object_utils(), ``** the test is a testbench component, so it uses **`` `uvm_component_utils(). ``**

The test is the first component built after executing the **`run_test()`** command in the **`top`** module. The test also references the factory to build the **`env`** and the command that accesses the **`env`** type from the factory is the command:

```
e = env::type_id::create("e", this);
```

The **`::type_id::create()`** command will be contrasted to the **`new()`** constructor used in the non-factory version of this testbench.

```
class test1 extends uvm_test;
  `uvm_component_utils(test1)
  env e;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    e = env::type_id::create("e", this);
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    this.print();
    factory.print();
  endfunction

  task run_phase(uvm_phase phase);
    trans_sequence seq;
    phase.raise_objection(this);
    seq = trans_sequence::type_id::create("seq");
    seq.start(e.agnt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Example 7 - test1.sv test (factory version)

Factory-based enviornment

The environment class for the demonstration example is shown in Example 8.

The code that registers the **`env`** with the factory is the command**,** **`` `uvm_component_utils(env). ``** This will be the top-level environment that will be bulit by all of the tests in this example.

The test also references the factory to build the **`tb_agent`** and the command that accesses the **`tb_agent`** type from the factory is the command:

```
agnt = tb_agent::type_id::create("agnt", this);
```

```
class env extends uvm_env;
  `uvm_component_utils(env)
  tb_agent agnt;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agnt = tb_agent::type_id::create("agnt", this);
  endfunction
endclass
```

Example 8 - env.sv environment (factory version)

Factory-based agent

The agent class for the demonstration example is shown in Example 9.

The code that registers the **`tb_agent`** with the factory is the command, **`` `uvm_component_utils(tb_agent). ``** This agent will build the sequencer-driver pair and frequently also builds a monitor. In this example, the monitor has been omitted to simplify the example.

The agent references the factory to build the **`tb_driver`** and **`tb_sequencer`**, using the commands:

```
drv =    tb_driver::type_id::create("drv", this);
sqr = tb_sequencer::type_id::create("sqr", this);
```

```
class tb_agent extends uvm_agent;
  `uvm_component_utils(tb_agent)
  tb_driver    drv;
  tb_sequencer sqr;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    drv =    tb_driver::type_id::create("drv", this);
    sqr = tb_sequencer::type_id::create("sqr", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    drv.seq_item_port.connect(sqr.seq_item_export);
  endfunction
endclass
```

Example 9 - tb_agent.sv agent (factory version)

Factory-based sequencer

The sequencer class for the demonstration example is shown in Example 10.

The code that registers the **`tb_sequencer`** with the factory is the command, **`` `uvm_component_utils(tb_sequencer). ``** Sequencers and drivers are classes that must be parameterized to the transaction type. The transaction type for this sequencer is **`trans`**, which is extended from the uvm_**`sequence_item`** type. UVM will check to make sure that the parameterized driver and sequencer are using compatible transaction types. When we change the transaction type for the demonstration example, we will again talk about what happens to the parameterized sequencer and driver.

Sequencers are typically pretty simple blocks of standard code and sequencers do not build any other sub-components.

```
class tb_sequencer extends uvm_sequencer #(trans);
  `uvm_component_utils(tb_sequencer)

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

Example 10 - tb_sequencer.sv sequencer (factory version)

Factory-based driver

The driver class for the demonstration example is shown in Example 11.

The code that registers the **`tb_driver`** with the factory is the command, **`` `uvm_component_utils(tb_driver). ``** As stated in the sequencer section, sequencers and drivers are classes that must be parameterized to the transaction type. The transaction type for this driver is **`trans`**, which is extended from the **`uvm_sequence_item`** type. UVM will check to make sure that the driver and sequencer are using compatible transaction parameters.

Drivers do not typically build any other sub-components.

```
class tb_driver extends uvm_driver #(trans);
  `uvm_component_utils(tb_driver)

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction

  task run_phase(uvm_phase phase);
    trans tx;
    forever begin
      seq_item_port.get_next_item(tx);
      #10 `uvm_info("tb_driver", tx.convert2string(), UVM_MEDIUM)
      seq_item_port.item_done();
    end
  endtask
endclass
```

Example 11 - tb_driver.sv driver (factory version)

Debugging the UVM structure and factory contents

After assembling a rather complex UVM testbench environment, it is often useful to print out the structure of the testbench in tabular form and to query the types that were registered with the factory. A great technique to view the structural composition of the testbench classes and the factory setup is to call the **`this.print()`** and **`factory.print()`** methods in the **`end_of_elaboration_phase()`** (as shown in Example 12) from the top-level testbench. By the time the **`end_of_elaboration_phase()`** executes, the entire environment has already been built and connected, so these **`print()`** methods show what had been built in the testbench and the types that were registered with the factory.

```
  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    this.print();
    factory.print();
  endfunction
```

Example 12 - Debug printing for structure and factory registration

The structural printout for **`test1`** is shown in Figure 3. In this table, we can see the 5 testbench components: **`test1`**, **`env (e), tb_agent (agnt), tb_driver (drv)`** and **`tb_sequencer (sqr).`**

```
UVM_INFO /home/uvm/src/base/uvm_root.svh(355) @ 0: reporter [NO_DPI_TSTNAME]
UVM_NO_DPI defined--getting UVM_TESTNAME directly, without DPI
UVM_INFO @ 0: reporter [RNTST] Running test test1...
-------------------------------------------------------------
Name                          Type                    Size  Value
-------------------------------------------------------------
uvm_test_top                  test1                   -     @489
  e                           env                     -     @499
    agnt                      tb_agent                -     @508
      drv                     tb_driver               -     @518
        rsp_port              uvm_analysis_port       -     @535
        sqr_pull_port         uvm_seq_item_pull_port  -     @526
      sqr                     tb_sequencer            -     @544
        rsp_export            uvm_analysis_export     -     @552
        seq_item_export       uvm_seq_item_pull_imp   -     @658
        arbitration_queue     array                   0     -
        lock_queue            array                   0     -
        num_last_reqs         integral                32    'd1
        num_last_rsps         integral                32    'd1
-------------------------------------------------------------
```

Figure 2 - test1 structure printout using this.print (factory version)

The types registered in the factory for **`test1`** are shown in Figure 3. In this list, we can see that there is currently no instance or type overrides in the factory, but the 5 testbench component types: **`test1`**, **`env`**, **`tb_agent`**, **`tb_driver`** and **`tb_sequencer`**, along with the transaction type (**`trans`**) and sequence type (**`trans_sequence`**) have been registered with the factory.

```
## Factory Configuration (*)
  No instance or type overrides are registered with this factory

All types registered with the factory: 44 total
(types without type names will not be printed)

  Type Name
  ---------
  env
  tb_agent
  tb_driver
  tb_sequencer
  test1
  trans
  trans_sequence
(*) Types with no associated type name will be printed as <unknown>
##
```

Figure 3 - test1 factory.print (factory version)

test1 simulation output - factory version

When the **`test1`** test is executed, we see that indeed 10 **`trans`** transactions were executed as printed in the simulation output shown in Figure 4.

```
UVM_INFO tb_driver.sv(27) @ 10: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = 74e9  data = 71
UVM_INFO tb_driver.sv(27) @ 20: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = 15e4  data = c8
UVM_INFO tb_driver.sv(27) @ 30: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = 0929  data = 05
UVM_INFO tb_driver.sv(27) @ 40: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = 6a6a  data = 56
UVM_INFO tb_driver.sv(27) @ 50: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = 003e  data = 33
UVM_INFO tb_driver.sv(27) @ 60: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = 8249  data = e4
UVM_INFO tb_driver.sv(27) @ 70: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = 4fe4  data = 7b
UVM_INFO tb_driver.sv(27) @ 80: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = 6e25  data = b6
UVM_INFO tb_driver.sv(27) @ 90: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = b8be  data = d6
UVM_INFO tb_driver.sv(27) @ 100: uvm_test_top.e.agnt.drv [tb_driver] trans1:  addr = 5a20  data = 2b
UVM_INFO /home/uvm/src/base/uvm_objection.svh(1120) @ 100: reporter [TEST_DONE] 'run' phase is ready to
proceed to the 'extract' phase

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   13
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[NO_DPI_TSTNAME]     1
[RNTST]     1
[TEST_DONE]     1
[tb_driver]    10
** Note: $finish    : /home/uvm/src/base/uvm_root.svh(408)
   Time: 100 ns  Iteration: 60  Instance: /top
```

Figure 4 - test1 UVM simulation output (factory version)

13. test2 With Modified Transaction

Now it is time to see what the factory can do to facilitate the creation of additional tests with minimal changes to the testbench structure and minimal modifications to the testbench source code.

Factory-based trans2 transaction

For **`test2`**, we are going to use a modified transaction type called **`trans2`**. The only modifications to the **`trans2`** type are the addition of a randomizable **`valid`** bit, and overriding the **`convert2string()`** method to report that this is a **`trans2`** transaction and to add to the printout the contents of the **`valid`** bit. Using class extension, **`trans2`** will be extended from **`trans`**, so the **`addr`** and **`data`** fields will be inherited from the original **`trans`** class. The **`trans2`** code is shown in Example 13.

Just like the **`trans`** base class, the code that registers the transaction with the factory is the command, **`` `uvm_object_utils(trans2). ``**

```
class trans2 extends trans;
  `uvm_object_utils(trans2)
  rand bit        valid;

  function new (string name="trans2");
    super.new(name);
  endfunction

  function string convert2string;
    string s;
$sformat(s, "trans2:  addr = %4h  data = %2h  valid=%b",addr,data,valid);
    return s;
  endfunction
endclass
```

Example 13 - trans2.sv sequence_item (factory version)

Factory-based test2

The code for the second test is shown in Example 14 and the code that registers **`test2`** with the factory is the command, **`` `uvm_component_utils(test2). ``**

Just like **`test1`**, when **`test2`** is executed, it will be the first component built after executing the **`run_test()`** command in the **`top`** module. All of the tests in this example will reference the factory to build the **`env`** and the command that accesses the **`env`** type from the factory is the command:

```
e = env::type_id::create("e", this);
```

`test2` also includes a command to change the transaction type that is returned whenever the **`trans`** type is requested. The **`test2`** code instructs the factory to override the return type of the transaction from **`trans`** to **`trans2`**. The UVM command to perform this action is:

```
set_type_override_by_type(trans::get_type(), trans2::get_type());
```

All components and sequences built in this test will now create a **`trans2`** type whenever **`trans`** is requested. This is a powerful technique that allows engfineers to reuse testbench sequences and components without modifying the original source files. This is why UVM uses a factory!

```
class test2 extends uvm_test;
  `uvm_component_utils(test2)
  env e;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    set_type_override_by_type(trans::get_type(), trans2::get_type());
    e = env::type_id::create("e", this);
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    this.print();
    factory.print();
  endfunction

  task run_phase(uvm_phase phase);
    trans_sequence seq;
    phase.raise_objection(this);
    seq = trans_sequence::type_id::create("seq");
    seq.start(e.agnt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Example 14 - test2.sv test (factory version)

Factory-based package - test2

In order to run the second test with the override transaction type, the **`tb_pkg`** must also be modified to include the **`trans2.sv`** file and the **`test2.sv`** file, as shown in Example 15.

```
`ifndef TB_PKG
`define TB_PKG

`include "uvm_macros.svh"

package tb_pkg;
  import uvm_pkg::*;

  `include "trans.sv"
  `include "trans2.sv"    // *NEW*

  `include "tb_driver.sv"
  `include "tb_sequencer.sv"
  `include "tb_agent.sv"
  `include "env.sv"

  `include "trans_sequence.sv"
  `include "test1.sv"
  `include "test2.sv"     // *NEW*
endpackage

`endif
```

Example 15 - tb_pkg2.sv package file

test2 structure and factory types reports

The **`test2`** code included the same **`end_of_elaboration_phase()`** request to execute the **`this.print()`** and **`factory.print()`** methods as were used in the **`test1`** code. The corresponding testbench struction is shown in Figure 5 and the factory types registered for **`test2`** now include the **`trans2`** and **`test2`** types, as shown in Figure 6.

```
UVM_INFO /home/uvm/src/base/uvm_root.svh(355) @ 0: reporter [NO_DPI_TSTNAME]
UVM_NO_DPI defined--getting UVM_TESTNAME directly, without DPI
UVM_INFO @ 0: reporter [RNTST] Running test test2...
--------------------------------------------------------------
Name                       Type                     Size  Value
--------------------------------------------------------------
uvm_test_top               test2                    -     @491
  e                        env                      -     @501
    agnt                   tb_agent                 -     @510
      drv                  tb_driver                -     @520
        rsp_port           uvm_analysis_port        -     @537
        sqr_pull_port      uvm_seq_item_pull_port   -     @528
      sqr                  tb_sequencer             -     @546
        rsp_export         uvm_analysis_export      -     @554
        seq_item_export    uvm_seq_item_pull_imp    -     @660
        arbitration_queue  array                    0     -
        lock_queue         array                    0     -
        num_last_reqs      integral                 32    'd1
        num_last_rsps      integral                 32    'd1
--------------------------------------------------------------
```

Figure 5 - test2 structure printout using this.print (non-factory version)

It can also be seen in the **`factory.print()`** output of Figure 6 that whenever trans is requested from the factory, **`trans2`** will be the override type (the return type).

```
## Factory Configuration (*)
No instance overrides are registered with this factory

Type Overrides:

  Requested Type  Override Type
  --------------  -------------
  trans           trans2

All types registered with the factory: 46 total
(types without type names will not be printed)

  Type Name
  ---------
  env
  tb_agent
  tb_driver
  tb_sequencer
  test1
  test2
  trans
  trans2
  trans_sequence
(*) Types with no associated type name will be printed as <unknown>
##
```

Figure 6 - test2 - set_type_override_by_type factory.print (factory version)

test2 simulation and factory substitution

In order to run **`test2`**, we can now use all of the other component and sequence types that were used by **`test1`** without modification. Each component or sequence that does that executes the **`trans::type_id::create("trans")`** command will actually cause the factory to create the **`trans2`** type.

When the **`test2`** test is run, we see that indeed 10 **`trans2`** transactions were executed and printed in the simulation output shown in Figure 7.

```
UVM_INFO tb_driver.sv(27) @ 10: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = 5847  data = a3  valid=0
UVM_INFO tb_driver.sv(27) @ 20: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = 68b7  data = ef  valid=0
UVM_INFO tb_driver.sv(27) @ 30: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = 0524  data = 90  valid=0
UVM_INFO tb_driver.sv(27) @ 40: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = 2262  data = f6  valid=0
UVM_INFO tb_driver.sv(27) @ 50: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = 382f  data = ca  valid=0
UVM_INFO tb_driver.sv(27) @ 60: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = 2e7d  data = 38  valid=1
UVM_INFO tb_driver.sv(27) @ 70: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = bc44  data = 44  valid=1
UVM_INFO tb_driver.sv(27) @ 80: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = 6eb0  data = 7c  valid=1
UVM_INFO tb_driver.sv(27) @ 90: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = bc51  data = ad  valid=1
UVM_INFO tb_driver.sv(27) @ 100: uvm_test_top.e.agnt.drv [tb_driver] trans2:  addr = a28c  data = 97  valid=1
UVM_INFO /home/uvm/src/base/uvm_objection.svh(1120) @ 100: reporter [TEST_DONE] 'run' phase is ready to proceed to
the 'extract' phase

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   13
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[NO_DPI_TSTNAME]     1
[RNTST]     1
[TEST_DONE]     1
[tb_driver]    10
** Note: $finish    : /home/uvm/src/base/uvm_root.svh(408)
   Time: 100 ns  Iteration: 60  Instance: /top
```

Figure 7 - test2 UVM simulation output (factory version)

14. test3 With Modified Driver

Whenever the transaction type is modified, it typically requires that the driver and monitor are also modified to use the new fields in the transaction. There is no monitor in the simple demonstration example, but there is a driver. A new **`tb_driver2`** type will be extended from the **`tb_driver`** and added to the testbench for **`test3`**.

Modified tb_driver2

For **`test3`**, a new **`tb_driver2`** class will be extended from the **`tb_driver`** class and the **`run_phase()`** will be modified to print a "*NEW DRIVER*" message. Typically the new **`run_phase()`** would be using new fields from the new transaction class, but this simple example is not really driving any signals to a DUT. The **`tb_driver2`** code is shown in Example 16.

Just like the **`tb_driver`** class, the code that registers the driver with the factory is the command, **`` `uvm_component_utils(tb_driver2).``**

```
class tb_driver2 extends tb_driver;
  `uvm_component_utils(tb_driver2)

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction

  task run_phase(uvm_phase phase);
    trans tx;
    forever begin
      seq_item_port.get_next_item(tx);
      `uvm_info("tb_driver2", "*NEW DRIVER*", UVM_MEDIUM)
      #10 `uvm_info("tb_driver2", tx.convert2string(), UVM_MEDIUM)
      seq_item_port.item_done();
    end
  endtask
endclass
```

Example 16 - tb_driver2.sv driver (factory version)

Factory-based test3

The code for the third test is shown in Example 17 and the code that registers **`test3`** with the factory is the command, **`` `uvm_component_utils(test3).``**

Note that in addition to the previous command to override the **`trans`** type with **`trans2`**, there is now a command to override just one instance of the **`tb_driver`** with the new **`tb_driver2`**. The instance to be overridden is located in the testbench structure at **`e.agnt.drv`** and the command to perform this action is:

```
set_inst_override_by_type("e.agnt.drv", tb_driver::get_type(),
                                        tb_driver2::get_type());
```

```
class test3 extends uvm_test;
  `uvm_component_utils(test3)
  env e;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    set_type_override_by_type(trans::get_type(), trans2::get_type());
    set_inst_override_by_type("e.agnt.drv", tb_driver::get_type(),
                                            tb_driver2::get_type());
    e = env::type_id::create("e", this);
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    this.print();
    factory.print();
  endfunction

  task run_phase(uvm_phase phase);
    trans_sequence seq;
    phase.raise_objection(this);
    seq = trans_sequence::type_id::create("seq");
    seq.start(e.agnt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Example 17 - test3.sv test (factory version)

Factory-based package - test3

In order to run the third test with the override driver type, the **`tb_pkg`** must also be modified to include the **`tb_driver2.sv`** file and the **`test3.sv`** file, as shown in Example 18.

```
`ifndef TB_PKG
`define TB_PKG

`include "uvm_macros.svh"

package tb_pkg;
  import uvm_pkg::*;

  `include "trans.sv"
  `include "trans2.sv"

  `include "tb_driver.sv"
  `include "tb_driver2.sv" // *NEW*
  `include "tb_sequencer.sv"
  `include "tb_agent.sv"
  `include "env.sv"

  `include "trans_sequence.sv"
  `include "test1.sv"
  `include "test2.sv"
  `include "test3.sv"     // *NEW*
endpackage

`endif
```

Example 18 - tb_pkg3.sv package file

test3 structure and factory types reports

The **`test3`** code also includes the same **`end_of_elaboration_phase()`** request to execute the **`this.print()`** and **`factory.print()`** methods as were used in the **`test1`** and **`test2`** code. The corresponding testbench struction is shown in Figure 8 and the factory types registered for **`test3`** now include the **`tb_driver2`** and **`test3`** types, as shown in Figure 9.

```
UVM_INFO /home/uvm/src/base/uvm_root.svh(355) @ 0: reporter [NO_DPI_TSTNAME]
UVM_NO_DPI defined--getting UVM_TESTNAME directly, without DPI
UVM_INFO @ 0: reporter [RNTST] Running test test3...
--------------------------------------------------------------
Name                        Type                     Size  Value
--------------------------------------------------------------
uvm_test_top                test3                    -     @493
  e                         env                      -     @503
    agnt                    tb_agent                 -     @512
      drv                   tb_driver2               -     @522
        rsp_port            uvm_analysis_port        -     @539
        sqr_pull_port       uvm_seq_item_pull_port   -     @530
      sqr                   tb_sequencer             -     @548
        rsp_export          uvm_analysis_export      -     @556
        seq_item_export     uvm_seq_item_pull_imp    -     @662
        arbitration_queue   array                    0     -
        lock_queue          array                    0     -
        num_last_reqs       integral                 32    'd1
        num_last_rsps       integral                 32    'd1
--------------------------------------------------------------
```

Figure 8 - test3 structure printout using this.print (factory version)

It can also be seen in the **`factory.print()`** output of Figure 10 that whenever **`trans`** is requested from the factory, **`trans2`** will be the override type (the return type). It can also be seen that whenever the **`uvm_test_top.e.agnt.drv`** instance of the **`tb_driver`** is requested from the factory, **`tb_driver2`** will be used.

```
## Factory Configuration (*)
Instance Overrides:

  Requested Type  Override Path            Override Type
  --------------  -----------------------  -------------
  tb_driver       uvm_test_top.e.agnt.drv  tb_driver2

Type Overrides:

  Requested Type  Override Type
  --------------  -----------------------
  trans           trans2

All types registered with the factory: 48 total
(types without type names will not be printed)

  Type Name
  ---------
  env
  tb_agent
  tb_driver
  tb_driver2
  tb_sequencer
  test1
  test2
  test3
  trans
  trans2
  trans_sequence
(*) Types with no associated type name will be printed as <unknown>
##
```

Figure 9 - test3 - set_type_override_by_type factory.print (factory version)

test3 simulation and factory substitution

In order to run **`test3`**, we can now use all of the other component and sequence types that were used by **`test2`** without modification. In this test, the request for **`tb_driver`** will actually use the **`tb_driver2`**.

When the **`test3`** test is run, we see that indeed 10 *NEW DRIVER* messages are printed along with 10 **`trans2`** messages, as shown in Figure 10.

```
UVM_INFO tb_driver2.sv(27) @ 0: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 10: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = 5847  data = a3  valid=0
UVM_INFO tb_driver2.sv(27) @ 10: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 20: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = 68b7  data = ef  valid=0
UVM_INFO tb_driver2.sv(27) @ 20: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 30: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = 0524  data = 90  valid=0
UVM_INFO tb_driver2.sv(27) @ 30: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 40: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = 2262  data = f6  valid=0
UVM_INFO tb_driver2.sv(27) @ 40: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 50: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = 382f  data = ca  valid=0
UVM_INFO tb_driver2.sv(27) @ 50: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 60: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = 2e7d  data = 38  valid=1
UVM_INFO tb_driver2.sv(27) @ 60: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 70: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = bc44  data = 44  valid=1
UVM_INFO tb_driver2.sv(27) @ 70: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 80: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = 6eb0  data = 7c  valid=1
UVM_INFO tb_driver2.sv(27) @ 80: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 90: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = bc51  data = ad  valid=1
UVM_INFO tb_driver2.sv(27) @ 90: uvm_test_top.e.agnt.drv [tb_driver2] *NEW DRIVER*
UVM_INFO tb_driver2.sv(28) @ 100: uvm_test_top.e.agnt.drv [tb_driver2] trans2:  addr = a28c  data = 97  valid=1
UVM_INFO /home/uvm/src/base/uvm_objection.svh(1120) @ 100: reporter [TEST_DONE] 'run' phase is ready to proceed to
the 'extract' phase

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   23
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[NO_DPI_TSTNAME]     1
[RNTST]     1
[TEST_DONE]     1
[tb_driver2]    20
** Note: $finish    : /home/uvm/src/base/uvm_root.svh(408)
   Time: 100 ns  Iteration: 60  Instance: /top
```

Figure 10 - test3 UVM simulation output (factory version)

15. Testbench Classes Without Using the Factory

Now let's re-code the entire **test1**-**test3** examples without any factory code, and compare the required coding efforts to those when the factory was used. We will still use the exact same **top** module show in Example 3, and the first **tb_pkg.sv** file shown in Example 4.

Using UVM, we will still need to register all of the tests with the factory, because UVM checks to make sure the requested top-level test was registered with the factory, but these will be the only components that are required to use **the `uvm_component_utils()** macros.

These classes have the exact same names at the factory-version of the classes, but for testing purposes, this set of files was kept in a separate directory.

Non-Factory-based transaction

The transaction class shown in Example 19 includes all of the same code as the factory-version of this class, except for the omission of the command, **`uvm_object_utils(trans).** This transaction is not registered with the factory.

```
class trans extends uvm_sequence_item;
  rand bit  [7:0] data;
  rand bit [15:0] addr;

  function new (string name="trans");
    super.new(name);
  endfunction

  function string convert2string;
    string s;
    $sformat(s, "trans1:  addr = %4h  data = %2h", addr, data);
    return s;
  endfunction
endclass
```

Example 19 - trans.sv sequence_item (non-factory version)

Non-Factory-based sequence

The seqeunce class shown in Example 20 includes all of the same code as the factory-version of this class, except for the omission of the command, **`uvm_object_utils(trans_sequence)** and the **body()** task creates a transaction using the standard **new()** class constructor. This sequence is not registered with the factory and generates a **trans** object directly.

```
class trans_sequence extends uvm_sequence #(trans);

  function new (string name="trans_sequence");
    super.new(name);
  endfunction

  task body();
    trans tx = new("tx");
    repeat(10) begin
      start_item(tx);
        assert(tx.randomize());
      finish_item(tx);
    end
  endtask
endclass
```

Example 20 - trans_sequence.sv sequence (non-factory version)

Non-Factory-based test1

The **test1** class shown in Example 21 includes all of the same code as the factory-version of this class, except for the omission of the command, **`uvm_component_utils(test1)** and the **build_phase()** creates the environment using the standard **new()** class constructor. This test is registered with the factory but still generates an envronment object directly.

```
class test1 extends uvm_test;
  `uvm_component_utils(test1)
  env e;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    e = new("e", this);
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    this.print();
    factory.print();
  endfunction

  task run_phase(uvm_phase phase);
    trans_sequence seq;
    phase.raise_objection(this);
    seq = new("seq");
    seq.start(e.agnt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Example 21 - test1.sv test (non-factory version)

Non-Factory-based enviornment

The **`env`** class shown in Example 22 includes all of the same code as the factory-version of this class, except for the omission of the command, **`` `uvm_component_utils(env) ``** and the **`build_phase()`** creates the agent using the standard **`new()`** class constructor. This **`env`** is not registered with the factory and generates an agent object directly.

```
class env extends uvm_env;
  tb_agent agnt;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agnt = new("agnt", this);
  endfunction
endclass
```

Example 22 - env.sv environment (non-factory version)

Non-Factory-based agent

The **`tb_agent`** class shown in Example 23 includes all of the same code as the factory-version of this class, except for the omission of the command, **`` `uvm_component_utils(tb_agent) ``** and the **`build_phase()`** creates the agent using the standard **`new()`** class constructor. This agent is not registered with the factory and generates both the driver and sequencer objects directly.

```
class tb_agent extends uvm_agent;
  tb_driver    drv;
  tb_sequencer sqr;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    drv = new("drv", this);
    sqr = new("sqr", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    drv.seq_item_port.connect(sqr.seq_item_export);
  endfunction
endclass
```

Example 23 - tb_agent.sv agent (non-factory version)

Non-Factory-based sequencer

The **`tb_sequencer`** class shown in Example 24 includes all of the same code as the factory-version of this class, except for the omission of the command, **`` `uvm_component_utils(tb_sequencer). ``** This sequencer is not registered with the factory.

```
class tb_sequencer extends uvm_sequencer #(trans);

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

Example 24 - tb_sequencer.sv sequencer (non-factory version)

Non-Factory-based driver

The **`tb_driver`** class shown in Example 25 includes all of the same code as the factory-version of this class, except for the omission of the command, **`` `uvm_component_utils(tb_driver). ``** This driver is not registered with the factory.

```
class tb_driver extends uvm_driver #(trans);

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction

  task run_phase(uvm_phase phase);
    trans tx;
    forever begin
      seq_item_port.get_next_item(tx);
      #10 `uvm_info("tb_driver", tx.convert2string(), UVM_MEDIUM)
      seq_item_port.item_done();
    end
  endtask
endclass
```

Example 25 - tb_driver.sv driver (non-factory version)

test1 simulation output - non-factory version

The structural printout for the non-factory version of **`test1`** is shown in Figure 11. In this table, we can see all of the testbench components, but unlike the factory version of thie printout, we only see the base class names for the non-registered components. Remember that the **`test1`** component had to be registered with the factory, so that name is visible in the report.

```
-------------------------------------------------------------
Name                          Type                    Size  Value
-------------------------------------------------------------
uvm_test_top                  test1                   -     @484
  e                           uvm_env                 -     @494
    agnt                      uvm_agent               -     @503
      drv                     uvm_driver #(REQ,RSP)   -     @513
        rsp_port              uvm_analysis_port       -     @530
        sqr_pull_port         uvm_seq_item_pull_port  -     @521
      sqr                     uvm_sequencer           -     @539
        rsp_export            uvm_analysis_export     -     @547
        seq_item_export       uvm_seq_item_pull_imp   -     @653
        arbitration_queue     array                   0     -
        lock_queue            array                   0     -
        num_last_reqs         integral                32    'd1
        num_last_rsps         integral                32    'd1
-------------------------------------------------------------
```

Figure 11 - test1 structure printout using this.print (non-factory version)

As might be expected, we only see that the **`test1`** type is registered with the factory.

```
## Factory Configuration (*)
  No instance or type overrides are registered with this factory

All types registered with the factory: 39 total
(types without type names will not be printed)

  Type Name
  ---------
  test1
 (*) Types with no associated type name will be printed as <unknown>
##
```

Figure 12 - test1 factory.print (non-factory version)

And we do see 10 **`trans`** type transactions were created during the simulation.

```
UVM_INFO tb_driver.sv(25) @ 10: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = 65cf  data = 5b
UVM_INFO tb_driver.sv(25) @ 20: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = ce3c  data = 58
UVM_INFO tb_driver.sv(25) @ 30: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = da4b  data = 03
UVM_INFO tb_driver.sv(25) @ 40: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = 6af2  data = 79
UVM_INFO tb_driver.sv(25) @ 50: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = dc6a  data = 6f
UVM_INFO tb_driver.sv(25) @ 60: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = fd5d  data = f5
UVM_INFO tb_driver.sv(25) @ 70: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = 9697  data = fa
UVM_INFO tb_driver.sv(25) @ 80: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = 0720  data = ac
UVM_INFO tb_driver.sv(25) @ 90: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = ffc5  data = 49
UVM_INFO tb_driver.sv(25) @ 100: uvm_test_top.e.agnt.drv [tb_driver] trans:  addr = 41d9  data = 23
UVM_INFO /home/uvm/src/base/uvm_objection.svh(1120) @ 100: reporter [TEST_DONE] 'run' phase
is ready to proceed to the 'extract' phase

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   13
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[NO_DPI_TSTNAME]     1
[RNTST]     1
[TEST_DONE]     1
[tb_driver]    10
** Note: $finish    : /home/uvm/src/base/uvm_root.svh(408)
   Time: 100 ns  Iteration: 60  Instance: /top [RNTST]     1
```

Figure 13 - test1 UVM simulation output (non-factory version)

16. test2 With Modified Transaction

Now it is time to see what is required to run a second version of the test with the **`trans2`** type, when we do not have access to the components or transactions from the factory.

Factory-based trans2 transaction

The transaction class shown in Example 26 includes all of the same code as the factory-version of this class, except for the omission of the command, **`` `uvm_object_utils(trans2). ``** This transaction is not registered with the factory, but since it is extended from the **`trans`** base class, it is assignment compatible, which is why it is not necessary to make new copies of the **`tb_driver`** and **`tb_sequencer`**, which are both parameterized to the **`trans`** transaction type, are only passed between the **tb_driver** and **tb_sequencer** and are not created in either of these components.

```
class trans2 extends trans;
  rand bit        valid;

  function new (string name="trans2");
    super.new(name);
  endfunction

  function string convert2string;
    string s;
  $sformat(s, "trans2:  addr = %4h  data = %2h  valid=%b",addr,data,valid);
    return s;
  endfunction
endclass
```

Example 26 - trans2.sv sequence_item (non-factory version)

Non-Factory-based test2

The **test2** class shown in Example 27 includes all of the same code as the factory-version of this class, except for the omission of the command, **`uvm_component_utils(test2)** and the **build_phase()** creates the environment using the standard **new()** class constructor. This test is registered with the factory but still generates an envronment object directly.

```
class test2 extends uvm_test;
  `uvm_component_utils(test2)
  env e;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    e = new("e", this);
  endfunction

  function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    this.print();
    factory.print();
  endfunction

  task run_phase(uvm_phase phase);
    trans_sequence2 seq;
    phase.raise_objection(this);
    seq = new("seq");
    seq.start(e.agnt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Example 27 - test2.sv test (non-factory version)

Non-Factory-based sequence

The **`trans_sequence2`** class shown in Example 28 includes all of the same code as the factory-version of this class, except for the omission of the command, **`` `uvm_component_utils(trans_sequence2) ``** and the **`build_phase()`** creates the **`trans2`** transaction using the standard **`new()`** class constructor. Because the transaction is **`new()`**-constructed, we could not use the **`trans_sequence`** class.

Without access to factory versions of the transactoin types, all sequences will need to be duplicated in order to directly generate transactions of the **`trans2`** type. This shows why the factory is so important to efficient testbench management.

```
class trans_sequence2 extends uvm_sequence #(trans2);

  function new (string name="trans_sequence2");
    super.new(name);
  endfunction

  task body();
    trans2 tx = new("tx");
    repeat(10) begin
      start_item(tx);
        assert(tx.randomize());
      finish_item(tx);
    end
  endtask
endclass
```

Example 28 - trans_sequence2.sv sequence (non-factory version)

Factory-based package - test2

In order to run the second test with the override transaction type, the **`tb_pkg`** must also be modified to include the **`trans2.sv`** file, the **`test2.sv`** file and the **`trans_sequence2.sv`** file, as shown in Example 29.

```
`ifndef TB_PKG
`define TB_PKG

`include "uvm_macros.svh"

package tb_pkg;
  import uvm_pkg::*;

  `include "trans.sv"
  `include "trans2.sv"           // *NEW*

  `include "tb_driver.sv"
  `include "tb_sequencer.sv"
  `include "tb_agent.sv"
  `include "env.sv"

  `include "trans_sequence.sv"
  `include "trans_sequence2.sv"  // *NEW*
  `include "test1.sv"
  `include "test2.sv"            // *NEW*
endpackage

`endif
```

Example 29 - tb_pkg2.sv package file

test2 simulation - non-factory version

The **`test2`** version of this testbench simulates as expected and the **`factory.print()`** report shows that now both tests, **`test1`** and **`test2`**, are registered with the factory. The reports were not included in the paper.

17. test3 simulation - non-factory version

The code for a third test using a second version of the **`tb_driver`** command without using the factory would introduce significant extra effort. Since the **`tb_driver2`** cannot be automatically created from the agent, a second version of the agent would be necessary. And since the **`tb_agent2`** cannot be automatically created from the environment, a second version of the environment would also be necessary.

This demonstration example is a small subset of an actual UVM testbench, so the problem would only get worse when more components and more sequences are needed to run a different version of the tests with a second transaction type and a second **`tb_driver`** (and **`tb_monitor`**).

It does not take too much imagination to realize that the factory overrides greatly reduce the coding effort of UVM verification environments when just a few transactions or components need to be replaced.

18. Where Does The ::type_id::create Command Come From?

If you use UVM, you do not need to fully understand how each command is defined and where the definitions exist inside of the UVM source code. If you efficiently use UVM to create powerful testbenches and if you really do not care about how certain commands work, skip this section! If you want to understand where the **`::type_id::create`** command comes from, read this section.

Trying to understand the inner workings of UVM by examining the source code can be a difficult task. The myriad of intertwined macro definitions coupled with frequent polymorphic replacement and indirection of base classes with extended classes, and repeated use of similar or the same method names in different classes makes it difficult to understand how some of the UVM features actually work. There is no greater example of this indirection and confusion than the **`::type_id::create`** command.

To understand this command, first recognize that the use of the **`::`** operators in this command indicate that you are probably using one or more static function calls, which is indeed the case.

To better understand this command, consider the following lines of code from the creation of the **`tb_agent`** in the simple example used in this paper:

```
tb_driver drv;  // declaration of a tb_driver handle
…

function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  drv =    tb_driver::type_id::create("drv", this);
  ...
endfunction
```

This command is going to call the **`::type_id::create`** command from the **`tb_driver`**, which happens to be code largely inherited from other macros and classes. The source of this command can be traced to the following:

(1) **`tb_driver`** is an extension of **`uvm_driver`**, which is an extension of **`uvm_component`**, which is a derivative of **`uvm_object`**, which defines a virtual method called **`create()`** with a single input argument. So how can the **`tb_driver::type_id::create("drv", this)`** command pass two arguments to this virtual method? The **`create()`** method defined in the **`uvm_object`** base class and passed down to the **`tb_driver`** class IS NOT THE **`create()`** COMMAND USED IN THIS FACTORY CONSTRUCTOR! (This is a point of confusion!)

(2) At the top of the **`tb_driver`** class definition is the macro-invocation:

```
`uvm_component_utils(tb_driver)
```

(3) The **`` `uvm_component_utils ``** macro is defined in the *<uvm_src_dir>***`/src/macros/uvm_object_defines.svh`** file

(4) In this file, **`` `uvm_component_utils(T) ``** is defined to be the macros:

```
`define uvm_component_utils(T)             \
   `m_uvm_component_registry_internal(T,T) \
   `m_uvm_get_type_name_func(T)
```

(5) In this same file, **`` `m_uvm_component_registry_internal(T,S) ``** is defined to be:

```
`define m_uvm_component_registry_internal(T,S)            \
   typedef uvm_component_registry #(T,`"S`") 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
```

(6) And in this same file, **`` `m_uvm_get_type_name_func(T) ``** is defined to be:

```
`define m_uvm_get_type_name_func(T)          \
   const static string type_name = `"T`";    \
   virtual function string get_type_name (); \
     return type_name;                       \
   endfunction
```

Doing the macro expansion, the top of the **tb_driver** class now includes the code:

```
class tb_driver extends uvm_driver #(trans);

//`uvm_component_utils(tb_driver)
//`define uvm_component_utils(T)                  \
//   `m_uvm_component_registry_internal(T,T)      \
//   `m_uvm_get_type_name_func(T)

//`define m_uvm_component_registry_internal(T,S) \

   typedef uvm_component_registry #(tb_driver,"tb_driver") 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

//`define m_uvm_get_type_name_func(T) \

   const static string type_name = "tb_driver";

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

This macro added the **type_id** type definition, **get_type()** method, **get_object_type()** method, static **type_name** string, and **get_type_name()** method, to the **tb_driver** class code.

The **type_id** type definition is part of the **::type_id::create** command, and **type_id** is just a type definition for the class type:

```
uvm_component_registry #(tb_driver,"tb_driver")
```

(7) The **uvm_component_registry** parameterized class is defined in the *<uvm_src_dir>***/src/base/uvm_registry.svh** file
(8) In this file, is the definition for the **uvm_component_registry** class. An abbreviated section of this class definition is shown below:

```
 1 class uvm_component_registry #(type T=uvm_component, string Tname="")
 2                              extends uvm_object_wrapper;
 3
 4 typedef uvm_component_registry #(T,Tname) this_type;
 5
 6 virtual function uvm_component
 7       create_component (string name, uvm_component parent);
 8   T obj;
 9   obj = new(name, parent);
10   return obj;
11 endfunction
12
13 const static string type_name = Tname;
14
15 virtual function string get_type_name();
16   return type_name;
17 endfunction
18
19 local static this_type me = get();
20
21 static function this_type get();
22   if (me == null) begin
23     uvm_factory f = uvm_factory::get();
24     me = new;
25     f.register(me);
26   end
27   return me;
28 endfunction
29
30 static function T create(string name, uvm_component parent, ...);
31   uvm_object obj;
32   uvm_factory f = uvm_factory::get();
33   ...
34   obj = f.create_component_by_type(get(),contxt,name,parent);
35   if (!$cast(create, obj)) begin
36     string msg;
37     msg = {<"... error message ...">};
38     uvm_report_fatal("FCTTYP", msg, UVM_NONE);
39   end
40 endfunction
41  ...
42 endclass
```

On line 4, this_type is set to
uvm_component_registry #(tb_driver,"tb_driver")

On line 21 is the static **get()** function that, if the **this_type (tb_driver registry class)** is **null**, will call the **uvm_factory** static **get()** method to create a handle for this **tb_driver** registry and copy that handle to the **this_type** handle, register the **tb_driver** registry with the factory and then return the handle to the caller of the **get()** function. The same **get()** function just returns the handle if it already exists.

On line 19 is the static handle declaration for the **local static this_type me** declaration, which calls the static **get()** function (described in the preceding paragraph) to register this **tb_driver** registry class with the factory and assign the corresponding handle. Since this handle and the **get()** function are both static, they will both happen automatically when the testbench is compiled without any required user invocation.

In this way, when the **`uvm_component_utils()** macro is called from each component, it literally registers the corresponding registry class with the factory, which makes it possible to **::type_id::create** any registered component from anywhere in the testbench class components.

Line 30 - When the **uvm_component_registry #(tb_driver,"tb_driver")** is compiled, the static **create()** command is also made statically available. The most confusing piece of the static **create()** command code is on line 35. It took me 2 hours to figure out this command happens to be a rather simple command, once you understand the code.

Most of the function methods in the UVM base classes use the SystemVerilog return command to return the correct value from the function, but line 35 is using the old Verilog way to return a function value. On line 34, the factory is asked to create the requested component by type and return the component handle to a **uvm_object** handle (**obj** - declared on line 31). This handle is then cast to the correct component type, which happens to be the type of the **create()** function. By casting back to the function name, the cast is actually assigned back to type of the function, and the caller of this function is then given a handle to the created component. Assigning (casting) to the function name is the old Verilog way to return a function value. The new SystemVerilog way to return a value would have been to declare a handle of the function **type (T),** cast the **uvm_object** handle to declared **T**-type function handle, and then call **return** to give back the **T**-type function handle. Hopefully this description just saved you 2 hours!

In any **uvm_*_registry** class, is the static **create()** function. This is the last piece of the **::type_id::create()** command. This explains where this command comes from. For components, the *<component_type>***::type_id::create(***<component_handle>***, this)** command comes from the **uvm_component_registry** class parameterized to the component type. Similarly, for transactions, the *<object_type>***::type_id::create(***<object_handle>***, this)** command comes from the **uvm_object_registry** class parameterized to the object type.

uvm_object_wrappers are the proxy (substitute, place holder) types that are actually stored in the factory. When you create a component, you have actually created a **uvm_component_registry** class that is an extension of the **uvm_object_wrapper**. When you create a transaction, you have actually created a **uvm_object_registry class** that is an extension of the **`uvm_object_wrapper`**. It is handles of these component and object extensions of the wrapper class that are actually stored in the type-based factory.

19. Factory Overrides - Debugging

If the factory override does not appear to be working, what are some of the common errors that can keep the override from working?

The most common problem is probably the inadvertent use of a **`new()`** constructor in place of the **`::type_id::create`** command. I personally have experienced a frustrating failure to override the transaction type in a design where I accidentally used the **`new()`** command in one of the test sequences. The problem with the inadvertent **`new()`** command is that it is a perfectly legal way to construct components and sequences, so if it sneaks into the component-structure or test transactions inadvertently, it will still allow the testbench to compile and simulate, but will then fail to create a component or transaction at some point during the simulation. Short summary: the testbench compiles and simulates but fails to create all of the necessary components or transactions from the factory.

How does one even notice these problems? In my example, I had a base transaction and an extended transaction, and during simulation, it appeared that the extended transaction was not being used. To verify the diagnosis, I included "**`trans1`**" as part of the **`convert2string()`** method of the base transaction, and included "**`trans2`**" as part of the **`convert2string()`** method of the extended transaction. During simulation, I called the **`convert2string()`** methods when the transactions were used and indeed verified that in the override-version of my test, I was still generating **`trans1`** versions of my transaction.

So when the override-version of the simulation fails, how can an engineer find these types of problems? In the example above, I suspected that I had forgotten to use the **`::type_id::create`** command, so I did a "grep new <all sv-files>" and then searched through the output until I found the use of a **`new()`** constructor where I should have used the **`::type_id::create()`** command.

After replacing the inadvertent **`new()`** commands with the proper **`::type_id::create`** commands, the override-version of the simulation worked as expected.

20. Using the new() Constructor

What happens if you use the **`new()`** constructor in an OVM/UVM verificatoin environment instead of the factory?

Advantage:

- It is a very simple syntax. This is the only advantage to using the **`new()`** constructor.

Disadvantages:

- The **`new()`** constructor will only create a transaction or component of the specified type.
- The **`new()`** consturtor fixes the type during constructon. No run-time changes will be possible.
- Changing components and transactions frequently requires significant source code changes.
- If you never intend to change the code, using the **`new()`** constructor is fine. But it is highly unlikely that you will build a large verificatoin enviornment without some required changes.
- If you want to use two transaction types, you will need extra components and sequences to be coded.

21. Using the Factory

What happens if you use the OVM/UVM factory to create constructor in an OVM/UVM verificatoin environment?

Disadvantage:

- Ugly syntax - **`::type_id::create()`** - but it is a very common syntax to create components and transactions.

Advantages:

- The **`create()`** command creates an object of the **`type_id`** stored in the factory.
- Tests (and some components) can make **`type_id`** substitutions using instance and type override commands.
- Tests can insert modified components and transactions into factory before building the rest of the test enviornment.
- Even if you never are required to change the code, the factory will work just fine.
- If you want to use two transaction types, you can more easily reuse components and sequences.

22. Conclusions

Use factories in your OVM/UVM verification environments. It is already built into the methodology and requires very little extra work on the part of the verification engineer; indeed, it will likely save a great deal of effort in the future as the test components and sequences evolve.

Use the instance and type overrides to take full advantage of the benefits ofered by the OVM/UVM factory.

23. Acknowledgements

I am grateful to my colleague and good friend Janick Bergeron for suggested improvements to the UVM examples in this paper. Janick was the one who informed me that it was not necessary to replace the sequencer-driver pair with versions that referenced a new transaction type because the extended transaction was compatible with the base transaction. That is an important point that I had missed while coding the original examples.

24. References

[1] Mark Glasser, "Open Verification Methodology Cookbook", Springer, www.springeronline.com, 1st Edition., 2009. ISBN: 978-1-4419-0967-1 Free downloadable PDF Version at: http://verification-academy.mentor.com/content/open-verification-methodology-advanced-ovm-uvm-module

[2] OVM User Guide, March 2010, Available for download from: https://verificationacademy.com/topics/verification-methodology

[3] Universal Verification Methodology (UVM) 1.1 Class Reference, May 2011, Accellera, Napa, CA. www.accellera.org/home

[4] UVM source code (it is sometimes easier to grep the UVM source files than to search the UVM Reference Guide)

25. AUTHOR & CONTACT INFORMATION

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 30 years of ASIC, FPGA and system design experience and 21 years of SystemVerilog, synthesis and methodology training experience.

Mr Cummings has presented more than 100 SystemVerilog seminars and training classes in the past nine years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr Cummings has participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee, and has presented more than 40 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

Mr Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Sunburst Design, Inc. offers World Class Verilog, SystemVerilog & OVM/UVM training courses. For more information, visit the www.sunburst-design.com web site.
Email address: cliffc@sunburst-design.com

Last Updated: January 14, 2013