2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

Easy Testbench Evolution

Styling Sequences and Drivers

Rich Edelman Siemens EDA, Fremont, CA US (rich.edelman@siemens.com)

Kento Nishizawa, Siemens, Tokyo, Japan (kento.nishizawa@siemens.com)

***Abstract*— SystemVerilog UVM polymorphism and the factory pattern are explored to increase productivity for UVM testbenches, focusing on transactions, sequences, and drivers.**

Keywords—SystemVerilog; UVM; Polymorphism; Factory Pattern; UVM Sequences; UVM Drivers

I. INTRODUCTION

The SystemVerilog [1] UVM [2] naturally supports object-oriented programming. One of the hardest parts of object-oriented programming for new users is polymorphism and the factory pattern. Yet, polymorphism and the factory pattern combine to improve productivity [3] while with just a minimal amount of work. This productivity leverage will be explored in this paper for sequences, drivers and transactions, with a focus on changing behavior while changing only small amounts of code.

The factory in the UVM supports overriding a type or overriding an instance of a type. Overriding a type causes that type to be replaced with the override. Overriding an instance of a type only replaces that instance of the type.

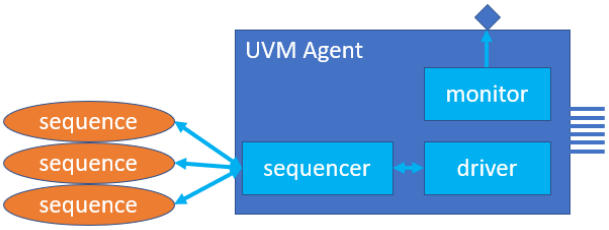
By using the factory with a type override for a driver, when the driver is constructed a different type – the "extended_driver" will be constructed. This means that with no change to the source code of the "regular" UVM testbench, it can be changed or evolved to do new things. The replacement types must derive from the specific base classes. But the power of this technique is shown within this paper. With just a few changes, the simple testbench is processing out-of-order transactions.

There are multiple ways to build productivity with the UVM. Using randomization with different seeds allows many different combinations to be run with a single testbench. Using parameterization, a testbench can be resized easily. This factory-based approach can change the structure and behavior of the testbench significantly. It's a powerful tool to add to randomization and parameterization.

II. BASIC UVM TESTBENCH

The basic UVM Testbench will be outlined and reviewed, including transactions, sequences, drivers, sequencers and tests. A transaction is constructed in a sequence and sent to the sequencer and on to the driver. The driver sends it for processing to the device-under-test via the virtual interface. That's the beginning of most UVM testbenches, and it is the beginning in this paper as well. The testbench is easy to write, concise and readable. It generates transactions and can be used effectively for verification as-is.

The diagram below will help explain these structures and how they interact. Each piece of the UVM Agent will be described, along with virtual interfaces, tests and UVM Environments. This summary will provide a background for both a beginner and experienced UVM verification engineer.



III. THE BASIC TESTBENCH

Running a test in this testbench can be controlled from the command line using the +UVM_TESTNAME argument.

```
vsim -visualizer opt +UVM_TESTNAME=test
```

The above command will run the test named “test”.

A. *The test*

The test creates the needed environments (e1, e2, e3 and e4). It constructs the sequences and starts them – by calling seq.start() on the individual sequencers.

```
class test extends uvm_test;
  `uvm_component_utils(test)

  env e1, e2, e3, e4;
  seq s1, s2, s3, s4;

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

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
    e2 = env::type_id::create("e2", this);
    e3 = env::type_id::create("e3", this);
    e4 = env::type_id::create("e4", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    `uvm_info(get_type_name(), "...running", UVM_MEDIUM)
    pretty_print();
    factory.print();
    s1 = seq::type_id::create("s1");
    s2 = seq::type_id::create("s2");
    s3 = seq::type_id::create("s3");
    s4 = seq::type_id::create("s4");
    fork
      s1.start(e1.a.sqr);
      s2.start(e2.a.sqr);
      s3.start(e3.a.sqr);
      s4.start(e4.a.sqr);
    join
    phase.drop_objection(this);
  endtask
endclass
```

B. The environment

The environment class is trivial in this example. It constructs the agent.

```
class env extends uvm_component;
  `uvm_component_utils(env)
  agent a;
  function new(string name = "env", uvm_component parent = null);
    super.new(name, parent);
  endfunction
  function void build_phase(uvm_phase phase);
    a = agent::type_id::create("a", this);
  endfunction
endclass
```

C. The agent

The agent builds the driver and sequencer. Its also connects the driver and sequencer. This connection is how the sequence item gets from the sequence to the driver.

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

  driver d;
  sequencer sqr;

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

  function void build_phase(uvm_phase phase);
    d = driver::type_id::create("d", this);
    sqr = sequencer::type_id::create("sqr", this);
  endfunction

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

D. The driver

The driver has a transaction and a delay variable as class member variables. In this paper, the driver does nothing useful. It calls get_next_item() to get the next transaction and then it prints a message and does a #delay to emulate some time passing interacting with a virtual interface and a device-under-test. In this paper, there is no virtual interface and there is no device-under-test. Finally, the driver signals that it has processed the transaction by calling item_done().

```
class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

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

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

  transaction t;
  int delay;

  task run_phase(uvm_phase phase);
    `uvm_info(get_type_name(), "...starting", UVM_MEDIUM)
    forever begin
      seq_item_port.get_next_item(t);
      `uvm_info(get_type_name(), {"Executing: ", t.convert2string()}, UVM_MEDIUM)
      delay = t.delay;
      #delay;
      seq_item_port.item_done();
    end
  endtask
endclass
```

E. The sequence

The basic sequence in this example has a transaction ‘t’ as a class member variable. In the body() task, a loop repeats 1000 times. Each time through the loop a transaction is constructed in the variable ‘t’. It does a start_item(t), t.randomize() and finish_item(t).

```
class seq extends uvm_sequence#(transaction);
  `uvm_object_utils(seq)

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

  transaction t;

  task body();
    string name;
    `uvm_info(get_type_name(), "...running", UVM_MEDIUM)

    for (int i = 0; i < 1000; i++) begin
      name = $sformatf("t%0d", i);
      t = transaction::type_id::create(name);
      start_item(t);
      `uvm_info(t.get_type_name(), "...started transaction", UVM_MEDIUM)
      if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize Failed")
      finish_item(t);
      `uvm_info(t.get_type_name(), "...finished transaction", UVM_MEDIUM)
    end
  endtask
endclass
```

F. The sequence item or transaction

The sequence item (transaction) for this example has a 3-bit id field and a 32-bit serial number. It uses the 'delay' member to emulate the behavior of a DUT – it can be thought of as the time or "cost" of the operation. The transaction has a read/write field with an address and data field. Convert2string() and do_record() are implemented to help with debug and checking.

```
class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

  bit  [2:0] id; // 0 to 7
  bit [31:0] serial_number;

  rand int delay;
  rand RW_T rw;
  rand bit [31:0] addr;
  rand bit [31:0] data;

  constraint values {
    addr > 0; addr < 100;
    data >= 0; data < 8;
    delay > 3; delay < 10;
  }

  function new(string name = "transaction");
    super.new(name);
    id = gid++;
    serial_number = gserial_number++;
  endfunction

  function string convert2string();
    return $sformatf("id: %0d %s(%0d, %0d) #%0d",
                     id, rw.name(), addr, data, serial_number);
  endfunction

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("id", id);
    `uvm_record_field("serial_number", serial_number);
    `uvm_record_field("rw", rw.name());
    `uvm_record_field("addr", addr);
    `uvm_record_field("data", data);
    `uvm_record_field("delay", delay);
  endfunction
endclass
```

G. The package

Finally, the files for this IP and test are combined in a package.

```
package ip_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"
  `include "types.svh"

  `include "transaction.svh"
  `include "driver.svh"
  `include "sequencer.svh"
  `include "sequence.svh"
  `include "agent.svh"
  `include "env.svh"
  `include "test.svh"
endpackage
```

H. The top

The top level is quite simple – there's no virtual interface, nor config database – just the package imports and the call to run_test(). In a real testbench with a device-under-test, these things would exist.

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

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

IV. THE BASIC UVM FACTORY

The UVM factory is easy to use with the uvm_component_utils() and uvm_object_utils() macros. The macros create the necessary infrastructure for each type. Once each type has the right infrastructure set up, then an object can be constructed via the factory by calling the 'create()' routine.

```
t = transaction::type_id::create(name);
```

In the create() call above, the sequence is asking for a 'transaction' type to be constructed. But if we have overridden the type 'transaction' in the factory, then instead of a transaction type being constructed, the override type will be constructed. For example, the following type override will cause the call above to construct an object of type 'extended_transaction_priority'.

```
transaction::type_id::set_type_override(extended_transaction_priority::get_type(), 1);
```

That's a lot of characters, but it is the simplest and safest way to override types. Using the other methods relies on strings, which are fraught.

The main attribute of using a factory and polymorphic, extended classes is simple. Any "replacement class" must inherit from the base class (or a class extended from the base class).

V. BUILDING TRANSACTION OVERRIDES

The extended_transaction_priority inherits from the base class and re-implements convert2string() and do_record(), adding two new properties – 'pri' and 'item_really_done'.

```
class extended_transaction_priority extends transaction;
  `uvm_object_utils(extended_transaction_priority)

  rand bit [1:0] pri; // 0, 1, 2, 3
  bit item_really_done;

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

  function string convert2string();
    return $sformatf("id: %0d %s(%0d, %0d) <pri:%0d> #%0d",
      id, rw.name(), addr, data, pri, serial_number);
  endfunction

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("pri", pri);
  endfunction
endclass
```

VI. BUILDING SEQUENCE OVERRIDES

The extended sequence re-implements body() with only a small actual change. This sequence is going to send many transactions to the driver before they are "acted" on. Once the driver has accumulated a certain quantity, then the driver "executes" the transactions – reads and writes in a real testbench. Once each of the executed transactions is "really done", it sets the bit and the thread that is waiting can complete processing.

```
class extended_seq_priority extends seq;
  `uvm_object_utils(extended_seq_priority)
  function new(string name = "extended_seq_priority");
    super.new(name);
  endfunction

  extended_transaction_priority t;

  task body();
    for (int i = 0; i < 1000; i++) begin
      t = extended_transaction_priority::type_id::create($sformatf("t%0d", i));
      start_item(t);
      if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize Failed")
      finish_item(t);
      fork
        begin
          extended_transaction_priority my_t = t;
          wait(my_t.item_really_done);
          `uvm_info(my_t.get_type_name(), "...item_really_done transaction",
              UVM_MEDIUM)
        end
      join_none
    end
  endtask
endclass
```

VII. BUILD DRIVER OVERRIDES

The extended driver class has more going on compared to the other extended classes. It is designed to accept large groups of transactions from the sequences. As it gets a transaction it pushes that transaction into one of four queues, based on the priority level of the transaction. Once it has received 30 transactions it empties the queues according to priority. Once the queues are empty, the extended driver resumes accepting transactions from the sequences.

This driver "executes" a transaction by printing it and waiting for the delay period in the transaction. This emulates the DUT behavior which may take some time depending on the packet type or operation. When the transaction is executed, the field 'item_really_done' is set.

```
class extended_driver_priority extends driver;
  `uvm_component_utils(extended_driver_priority)

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

  extended_transaction_priority executing_t;
  extended_transaction_priority pushing_t;

  transaction q0[$];
  transaction q1[$];
  transaction q2[$];
  transaction q3[$];

  task execute_command(transaction t);
    int delay;
    $cast(executing_t, t);
    `uvm_info(get_type_name(), {"Executing: ", executing_t.convert2string()}, UVM_MEDIUM)
    delay = executing_t.delay;
    #delay;
    executing_t.item_really_done = 1;
  endtask

  transaction t_we_are_processing;

  task process_a_queue(ref transaction q_we_are_processing[$]);
    while (q_we_are_processing.size() > 0) begin
      t_we_are_processing = q_we_are_processing.pop_back();
      execute_command(t_we_are_processing);
    end
  endtask

  int pushed_count;

  task process_queues();
    process_a_queue(q0);
    q0.delete();
    process_a_queue(q1);
    q1.delete();
    process_a_queue(q2);
    q2.delete();
    process_a_queue(q3);
    q3.delete();
  endtask
```

```
    task push_command(transaction t);
      $cast(pushing_t, t);
      `uvm_info(get_type_name(), {"Pushing: ", pushing_t.convert2string()}, UVM_MEDIUM)
      case (pushing_t.pri)
      0: q0.push_front(pushing_t);
      1: q1.push_front(pushing_t);
      2: q2.push_front(pushing_t);
      3: q3.push_front(pushing_t);
      endcase
      if (pushed_count++ > 30) begin
        process_queues();
        pushed_count = 0;
      end
    endtask

    task run_phase(uvm_phase phase);
      `uvm_info(get_type_name(), "...starting", UVM_MEDIUM)
      forever begin
        seq_item_port.get_next_item(t);
        delay = t.delay;
        #delay;
        push_command(t);
        seq_item_port.item_done();
      end
    endtask
  endclass
```

The drivers are perhaps the most overlooked area for specialized behavior. They are normally thought of as static components of a testbench. But, using the factory and by running many tests – many simulations – each one can have a different specialized driver. This way many different scenarios can be explored with the same basic testbench.

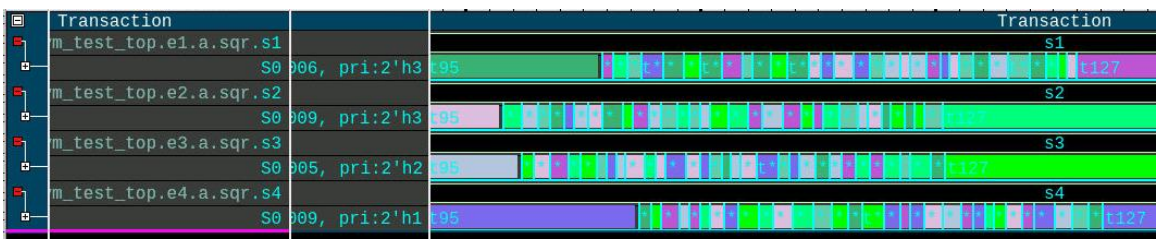
VIII. The Extended Test

The extended test itself is quite simple. It simply re-uses all the functionality of the base class and sets three type overrides in the factory – this is what evolves the testbench.

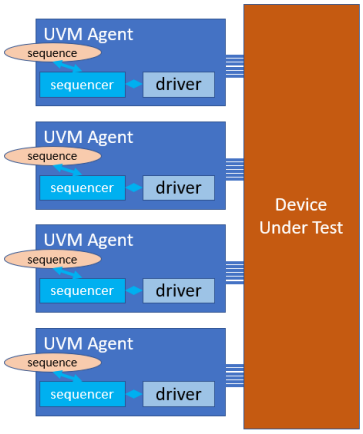
```
  class extended_test_priority extends test;
    `uvm_component_utils(extended_test_priority)

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

          driver::type_id::set_type_override(     extended_driver_priority::get_type(), 1);
             seq::type_id::set_type_override(        extended_seq_priority::get_type(), 1);
     transaction::type_id::set_type_override(extended_transaction_priority::get_type(), 1);
    endfunction
  endclass
```

The invocation of the extended, evolved testbench is easy. Just use the new testname on the command line.

```
vsim -visualizer opt +UVM_TESTNAME=extended_test_priority
```



The extended test builds the 4 environments and replaces the driver with extended_driver_priority, replaces seq with extended_seq_priority and replaces transaction with extended_transaction_priority.



IX. The Overrides Creating More Tests

A. *The base test class*

The basic test – the base class written to build "factory friendly" environments and start the sequences.

```
class test extends uvm_test;
  `uvm_component_utils(test)

  env e1, e2, e3, e4;
  seq s1, s2, s3, s4;

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

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
    e2 = env::type_id::create("e2", this);
    e3 = env::type_id::create("e3", this);
    e4 = env::type_id::create("e4", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    pretty_print();
    factory.print();
    s1 = seq::type_id::create("s1");
    s2 = seq::type_id::create("s2");
    s3 = seq::type_id::create("s3");
    s4 = seq::type_id::create("s4");
    fork
      s1.start(e1.a.sqr);
      s2.start(e2.a.sqr);
      s3.start(e3.a.sqr);
      s4.start(e4.a.sqr);
    join
    phase.drop_objection(this);
  endtask
endclass
```

```
/*
# UVM_INFO @ 0: reporter [RNTST] Running test test...
# UVM_INFO test.svh(27) @ 0: uvm_test_top [test] ...running
# uvm_test_top(test)
#   e1(env)
#     a(agent)
#       d(driver)
#       sqr(sequencer)
#   e2(env)
#     a(agent)
#       d(driver)
#       sqr(sequencer)
#   e3(env)
#     a(agent)
#       d(driver)
#       sqr(sequencer)
#   e4(env)
#     a(agent)
#       d(driver)
#       sqr(sequencer)
#
#### Factory Configuration (*)
#
#   No instance or type overrides are registered with this factory
#
*/
```

B. A simple extended class

This first override is the driver and sequence. Pretty printing the UVM component tree helps make sure the overrides were implemented correctly.

```
class extended_test extends test;
  `uvm_component_utils(extended_test)

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

    driver::type_id::set_type_override(extended_driver::get_type(), 1);
       seq::type_id::set_type_override(extended_seq::get_type(),    1);
  endfunction
endclass

/*
# UVM_INFO @ 0: reporter [RNTST] Running test extended_test...
# UVM_INFO test.svh(27) @ 0: uvm_test_top [extended_test] ...running
# uvm_test_top(extended_test)
#   e1(env)
#     a(agent)
#       d(extended_driver)
#       sqr(sequencer)
#   e2(env)
#     a(agent)
#       d(extended_driver)
#       sqr(sequencer)
#   e3(env)
#     a(agent)
#       d(extended_driver)
#       sqr(sequencer)
#   e4(env)
#     a(agent)
#       d(extended_driver)
#       sqr(sequencer)
#

#### Factory Configuration (*)
#
# No instance overrides are registered with this factory
#
# Type Overrides:
#
#   Requested Type  Override Type
#   --------------  ---------------
#   driver          extended_driver
#   seq             extended_seq
#
*/
```

C. An instance override

This second override is an instance override of two of the drivers. One in ‘e1’ and one in ‘e2’.

```
class extended_test_inst_override extends test;
  `uvm_component_utils(extended_test_inst_override)

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    driver::type_id::set_inst_override(extended_driver::get_type(), "a.d",   e1);
    driver::type_id::set_inst_override(extended_driver::get_type(), "a.d",   e2);
  endfunction
endclass

/*
# UVM_INFO @ 0: reporter [RNTST] Running test extended_test_inst_override...
# UVM_INFO test.svh(27) @ 0: uvm_test_top [extended_test_inst_override] ...running
# uvm_test_top(extended_test_inst_override)
#   e1(env)
#     a(agent)
#       d(extended_driver)
#       sqr(sequencer)
#   e2(env)
#     a(agent)
#       d(extended_driver)
#       sqr(sequencer)
#   e3(env)
#     a(agent)
#       d(driver)
#       sqr(sequencer)
#   e4(env)
#     a(agent)
#       d(driver)
#       sqr(sequencer)

#### Factory Configuration (*)
#
# Instance Overrides:
#
#   Requested Type  Override Path        Override Type
#   --------------  -------------------  ---------------
#   driver          uvm_test_top.e1.a.d  extended_driver
#   driver          uvm_test_top.e2.a.d  extended_driver
#
# No type overrides are registered with this factory
*/
```

D. Overriding driver, sequence and transaction classes to support out-of-order completion

This last test is an override of the drivers, sequences and transactions – it changes the underlying behavior significantly.

```
class extended_test_priority extends test;
  `uvm_component_utils(extended_test_priority)

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

        driver::type_id::set_type_override(     extended_driver_priority::get_type(), 1);
           seq::type_id::set_type_override(        extended_seq_priority::get_type(), 1);
   transaction::type_id::set_type_override(extended_transaction_priority::get_type(), 1);
  endfunction
endclass

/*
# UVM_INFO @ 0: reporter [RNTST] Running test extended_test_priority...
# UVM_INFO test.svh(27) @ 0: uvm_test_top [extended_test_priority] ...running
# uvm_test_top(extended_test_priority)
#   e1(env)
#     a(agent)
#       d(extended_driver_priority)
#       sqr(sequencer)
#   e2(env)
#     a(agent)
#       d(extended_driver_priority)
#       sqr(sequencer)
#   e3(env)
#     a(agent)
#       d(extended_driver_priority)
#       sqr(sequencer)
#   e4(env)
#     a(agent)
#       d(extended_driver_priority)
#       sqr(sequencer)

#### Factory Configuration (*)
#
# No instance overrides are registered with this factory
#
# Type Overrides:
#   Requested Type  Override Type
#   --------------  -----------------------------
#   driver          extended_driver_priority
#   seq             extended_seq_priority
#   transaction     extended_transaction_priority
*/
```

X. CONCLUSION

Using polymorphism and the factory pattern can improve productivity and thereby quality with little effort. The example code is clear and concise, showing real SystemVerilog UVM transactions, sequences and drivers and their specialized cases. These examples are appropriate for the reader to copy/paste and include or update in their own testbenches. They are easy to expand and control from the command line providing another efficient way to use polymorphism to increase productivity.

This source code is available for any reader by request from the authors.

XI. ACKNOWLEDGMENT

The authors thank the reviewers for their encouragement and consideration.

XII. REFERENCES

[1] SystemVerilog LRM, "1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language", https://ieeexplore.ieee.org/document/8299595

[2] UVM 1.1d - https://www.accellera.org/downloads/standards/uvm

[3] "What Does The Sequence Say? Powering Productivity with Polymorphism", DVCON US 2022, https://dvcon-proceedings.org/wp-content/uploads/What-Does-the-Sequence-Say-Powering-Productivity-with-Polymorphism-2.pdf

XIII. APPENDIX

The package for the basic test with pretty_print()

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

  `include "types.svh"

  `include "transaction.svh"
  `include "driver.svh"
  `include "sequencer.svh"
  `include "sequence.svh"
  `include "agent.svh"
  `include "env.svh"
  `include "test.svh"

  function void pretty_print();
    _pretty_print(uvm_top, "");
  endfunction

  function automatic void _pretty_print(uvm_component c, string spaces);
    uvm_component children[$];
    uvm_component child;
    string type_name;

    c.get_children(children);
    foreach (children[name]) begin
      child = children[name];
      type_name = child.get_type_name();
      if (type_name.substr(0,3).compare("uvm_") != 0) begin
        // This is not a UVM library component
        $display("%s%s(%s)", spaces, child.get_name(), type_name);
        _pretty_print(child, {spaces, "  "});
      end
    end
  endfunction
endpackage
```