2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

Easy Testbench Evolution Styling Sequences and Drivers

Rich Edelman, Siemens US
Kento Nishazawa, Siemens Japan

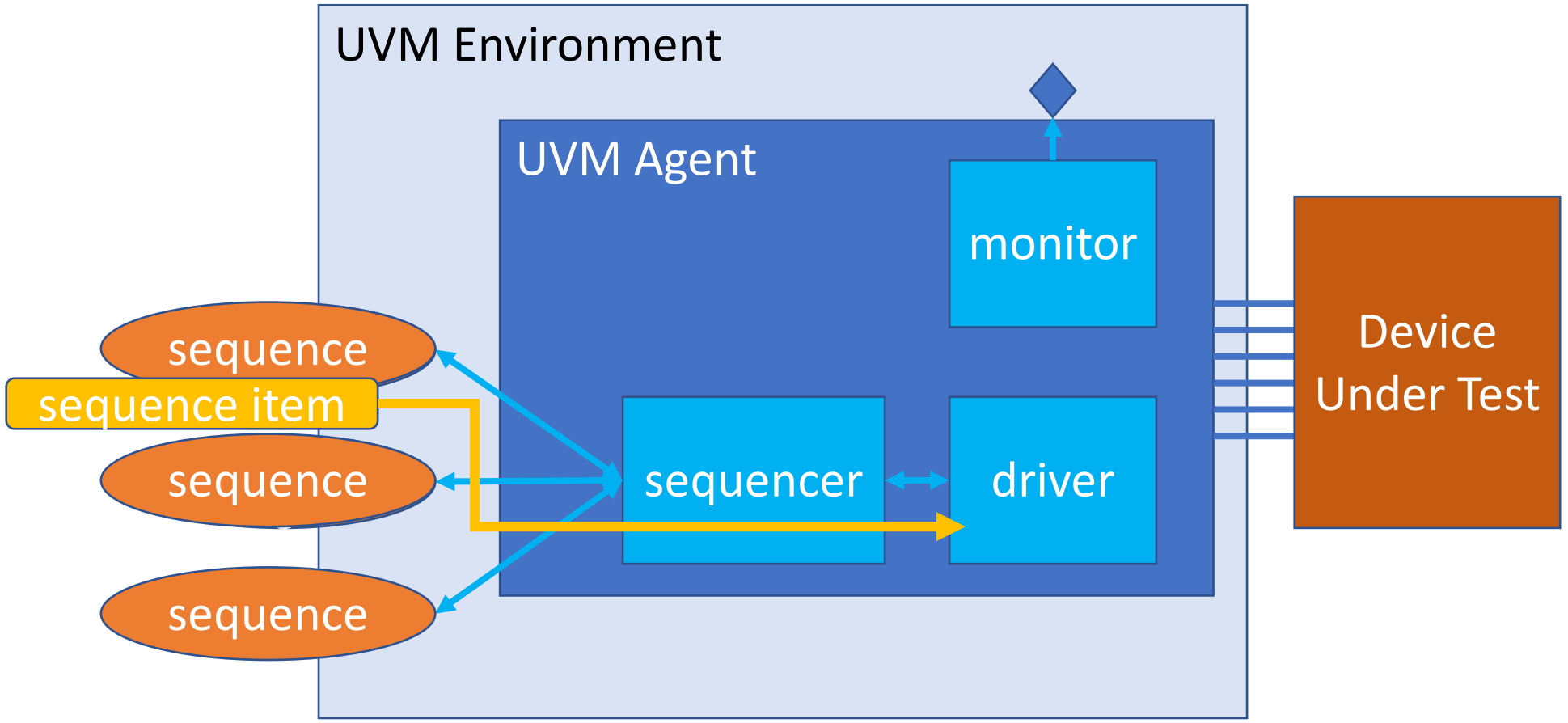
SIEMENS

accellera SYSTEMS INITIATIVE

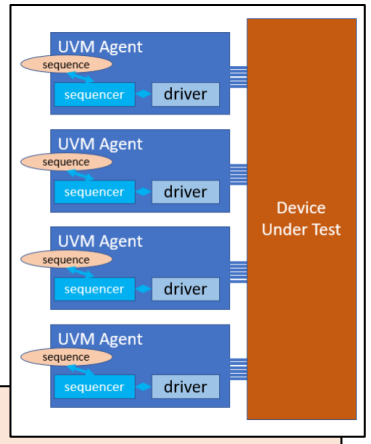
Easy Testbench Evolution

- A typical UVM Testbench
  - Test
  - Environment
  - Agent
    - Driver
    - Sequence
    - Transaction
- UVM Factory
- Overriding class construction
  - By Type
  - By Instance

A Typical UVM Testbench



A Test




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

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

  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
```

The Environment

- The simple wrapper around 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
```

The Agent

- Builds the driver and the sequencer and connects them

```
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
```

The Driver

- Do a get_next_item(t), print the ‘t’, do a delay, call item_done()

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

  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
```

The Sequence

- Generate (construct and randomize) 1000 transactions and send them to the driver.

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

  transaction t;

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

The Transaction

- This the “payload” or the “transaction” that the driver will operate on
- A few ‘rand’ values and constraints

```
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
```

The Transaction (2)

- Transaction continued
- Helper functions defined
  - convert2string()
  - do_record()

```
  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
```

The Package

- To build a test, “import” this package and do ‘run_test()’

```
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
```

The Top

- The top
- Missing are some traditional pieces – since we don’t have a real bus or device-under-test
  - Missing SV ‘interface’
  - Missing uvm_config() settings

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

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

What is the UVM Factory?

- Use the syntax below in your testbench – call ‘create’ from the factory

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

- To override a type in the factory, use this syntax

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

- ‘transaction’ is the existing type in the factory
- ‘extended_transaction_priority’ is the ‘replacement’ type

Building a Transaction override

- Extend the transaction, adding
  - priority
  - item_really_done
- Update
  - convert2string()
  - do_record

```
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
```

Building a Sequence override

- Same loop of 1000
- Create a transaction
- start_item()
- finish_item()
- Tricky!
  - Fork a thread to wait
  - Wait for transaction “completion”

```
class extended_seq_priority extends seq;
  `uvm_object_utils(extended_seq_priority)

  extended_transaction_priority t;

  task body();
    string name;
    for (int i = 0; i < 1000; i++) begin
      name = $sformatf("t%0d", i);
      t =
extended_transaction_priority::type_id::create(name);
      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);
          ...Finish Processing...
        end
      join_none
    end
  endtask
endclass
```

Building a Driver override

- The driver is busy
- Declare 4 queues
- run_phase() pushes the transction onto one of the queues, according to priority
- When 30 items pushed, process the queues

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

  extended_transaction_priority executing_t, pushing_t;

  transaction q0[$], q1[$], q2[$], q3[$];
  int pushed_count;

  task push_command(transaction t);
    $cast(pushing_t, t);
    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);
    forever begin
      seq_item_port.get_next_item(t);
      delay = t.delay;
      #delay;
      push_command(t);
      seq_item_port.item_done();
    end
  endtask
endclass
```

Building a Driver override (2)

- Driver continued
- When process_queues() is called, the queues are emptied in turn
- Calling execute_command on each transaction
- execute_command(t) does a delay, emulating DUT timing and then marks ‘item_really_done’

```
task execute_command(transaction t);
  int delay;
  $cast(executing_t, t);
  delay = executing_t.delay;
  #delay;
  executing_t.item_really_done = 1;
endtask

transaction my_t;

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

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

Building a Test override

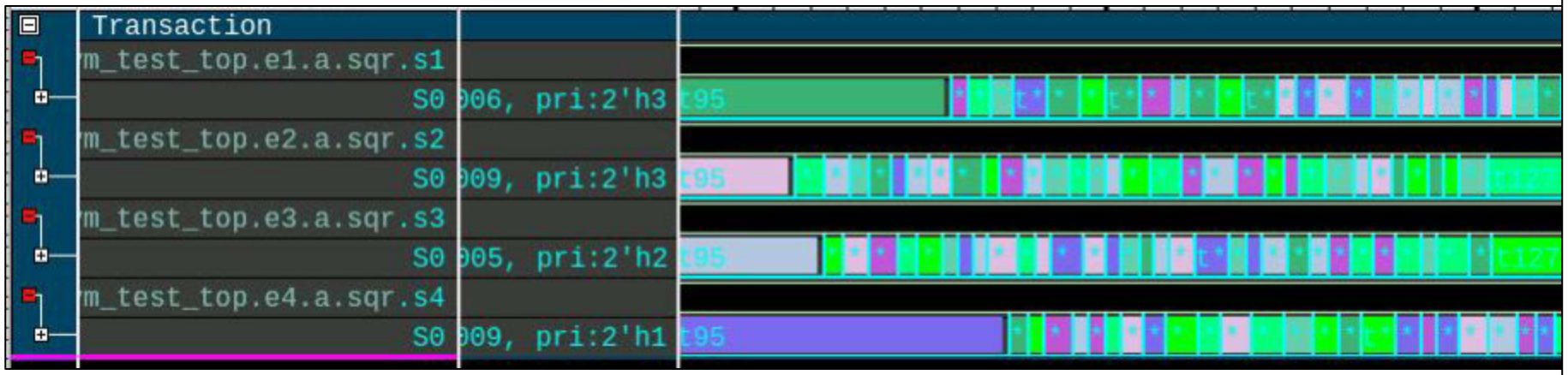
- Extend the test
- Define the overrides

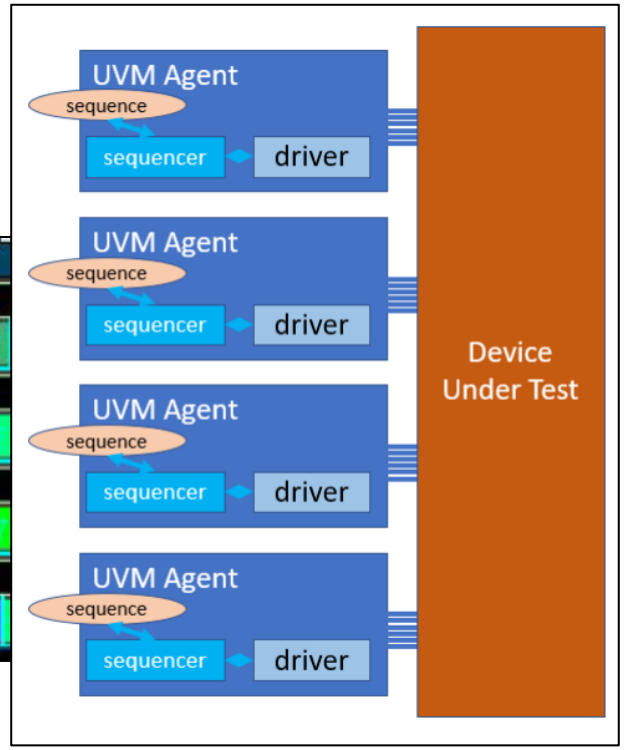
```
class extended_test_priority extends test;
  `uvm_component_utils(extended_test_priority)

  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
```

Running the override simulation

- Change the UVM_TESTNAME to change which test is running

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





Simple Override

```
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_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
```

Instance Override

```
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_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
```

Override for transaction, sequence and driver

```
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_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
```

Conclusion

- UVM helps increase productivity in verification with various techniques

Technique	More or different tests run many times …
Randomization	Run the same test with different random seeds
Parameterization	Run the same test with different parameters (change the BUS WIDTH easily)
Polymorphism	Change the test behavior with “substitute class definitions”. Run different tests

- Polymorphism and the factory are easy to use

Questions?

- For source code or questions, please contact the authors
  - Rich Edelman – rich.edelman@siemens.com
  - Kento Nishizawa – kento.nishizawa@siemens.com

Extra

- Pretty Print the UVM Component, Skipping any UVM library classes

```
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
```