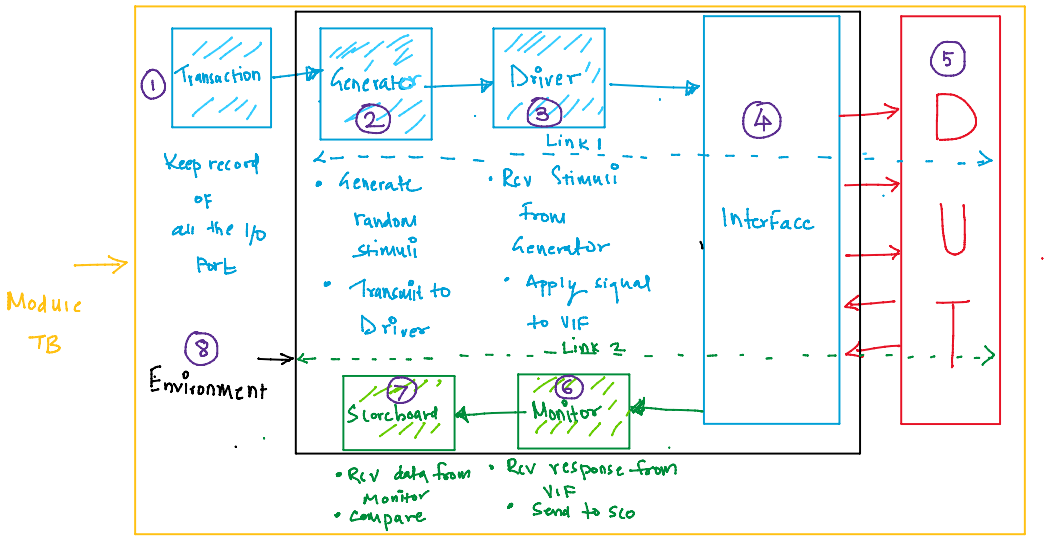
UVM Note Slides

06 July 2021 11:14



☑ Checkpoints

A] Class transaction :

1) Declare all the inputs and outputs with equivalent sizes.
2) Add Modifier (randc, rand) to all inputs
3) Do not add Modifier to output ports

B] Class Generator

1) Generate random stimulus for inputs
2) Send data to Driver with Mailbox
3) Signify to Driver about completion of stimuli generation using Event.

C] Interface

1) Declare all the input and outputs with logic datatypes

D] Class Driver

1) Receives data from Generator through Mailbox
2) Send the data to interface

E] Class Monitor

1) Receives data from Interface
2) Send data to Scoreboard with Mailbox

F] class Scoreboard

1) Receives data from Monitor
2) compare with Golden Ref. data

G] Class Environment

1) Initialize all the Class
2) Connect respective Mailbox

1) Initialize all the Class
2) Connect respective Mailbox
3) Connect respective Event
4) Connect respective Interfaces
5) Schedule execution of different process.

4] Testbench top

1) Instance of Env
2) new Method to Mailbox
3) Connect interface
4) perform connection between Interface and DUT

Verbosity Level

UVM_Debug - 500
UVM_Full 400
UVM_High 300
UVM_Medium 200
UVM_LOW 100
UVM_None 0

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

class display extends uvm_test;
`uvm_component_utils(display)

rand bit [15:0] a;

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


task run();
`uvm_info_begin("DISPLAY", "This is table ", UVM_NONE)
`uvm_message_add_int(a,UVM_DEC)
`uvm_info_end
endtask

endclass
```

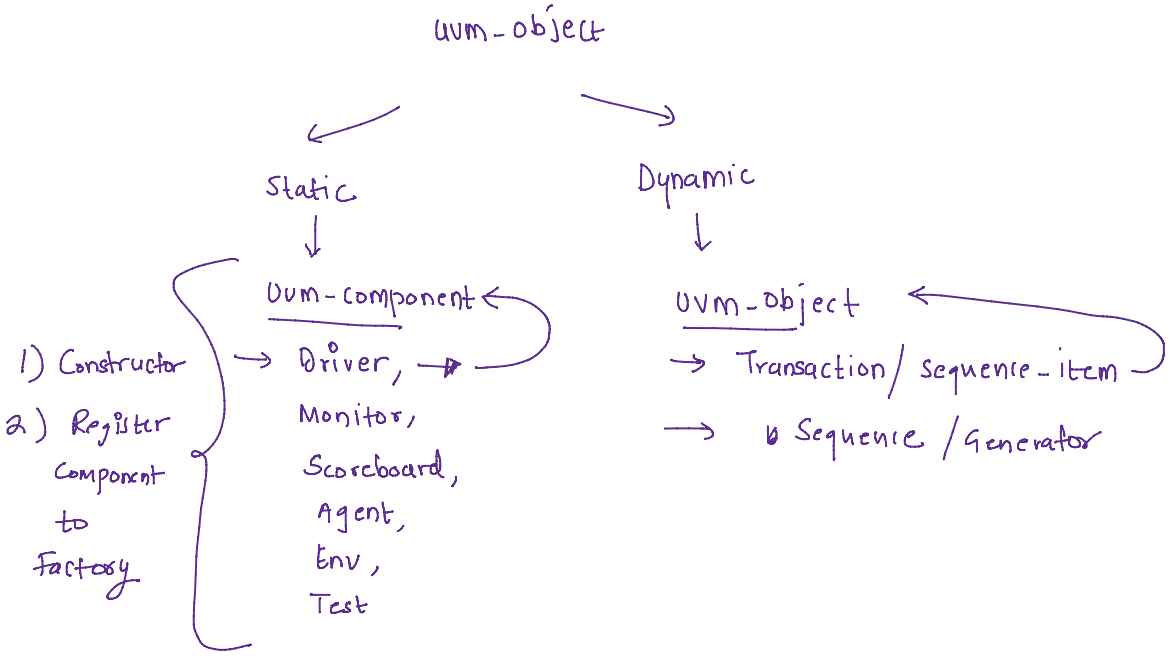
UVM_test → (UVM_component)

Instance

null

```
module tb;
display d1;  <-- Handler

initial begin
d1 = new("DISPLAY", null);
d1.randomize();
d1.run();
end
endmodule
```



1 Transaction sequence item
(uvm_sequence_item or uvm_transaction)
Container for all the I/O ports

2) Sequence → (uvm_sequence)
Generate multiple randomized transactions

3) Sequencer (uvm_sequencer)
Send generated transactions to the Driver

4) Driver (uvm_driver)
Apply received transaction to DUT with help of interface

5) Monitor (uvm_monitor)
receives response from DUT and convert into data which could be understood by scoreboard

6) Agent (uvm_agent)
Connect Driver, Sequence and Monitor together

7) Scoreboard (uvm_scoreboard)

Connect Driver, Sequence and Monitor together

7) Scoreboard (uvm_scoreboard)

Compare response with Golden data

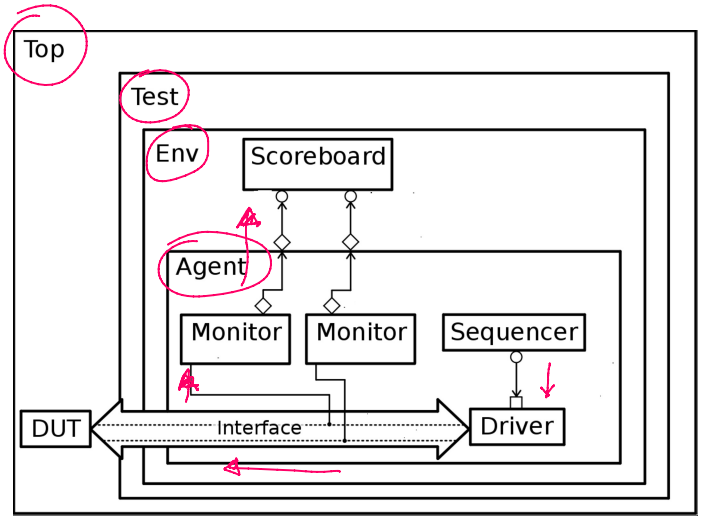
8) Environment (uvm_env)

Connect Scoreboard and agent together

9) Test (uvm_test)

connect Sequence and Sequencer together

Instance of env



UVM_object

→ UVM_object_utils (class_name)

→ Macro utils

→ function new (input string name);
super.new(name);
endfunction

→ Macro → object.print();

→ object_utils → do_print();

UVM_component

→ UVM_component_utils (class_name)

→ function new (input string name,
UVM_component c);
super.new(name, c);
endfunction

UVM - phases

- Prior Simulation → Build phase
- Simulation → run phase
- Post simulation → Cleanup phase

All the phases are virtual methods, so we need to explicitly call super.____(phase) to use them.

Build phase

- build-phase
  - Construct testbench component
  - Create object for TLM ports and exports
- Connect-phase
  - Connect TLM ports and exports
- end-of-elaboration
  - understand Topology
  - check all the components are properly build

Run phase

- start-of-simulation
  - Set initial configuration (runtime) values
  - Set debug break points
- run-phase
  - apply stimulus to DUT and capture response
  - perform simulation

Cleanup phase

- Extract phase
  - extract information from scoreboard
- Check-phase
  - verify DUT behavior
- report-phase
  - Display the result of simulation
  - write result to the file
- Final-phase
  - close the files
  - ready to exit simulator

TLM Blocking port

Sender

1) Declare uvm_blocking_put_port #(↑) user-defined-name;

Type of data object

```
uvm_blocking_put_port #(int) send;

function new(input string inst = "PROD", uvm_component c);
super.new(inst, c);
send  = new("PUT", this);
endfunction

task run();
```

Sender Side

Type of data object

2) Instantiation of port with new method is Mandatory

3) Packets are send by calling put method

```
super.new(inst, c);
send  = new("PUT", this);
endfunction

task run();
`uvm_info("PROD", $sformatf("Sending data to Consumer %0d",a), UVM_NONE);
send.put(a);
endtask
```

Receiver Side

1) Declare uvm_blocking_put_imp #( ) user-defined-name;

2) Instantiate new method

3) Implementation of put method happen in receiver class

```
uvm_blocking_put_imp #(int,consumer)  recv;

function new(input string inst = "CONS", uvm_component c);
super.new(inst, c);
recv  = new("RECV", this);
endfunction

virtual task put(input int t);
`uvm_info("CONS", $sformatf("Data received : %0d",t), UVM_NONE);
endtask
```

* Connection between port and its implementation happens in higher class → (Env, Test using connect phase)

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

class transaction extends uvm_sequence_item;
rand bit [3:0] a;
rand bit [7:0] b;
rand bit [7:0] c;

function new (input string inst = "TRANS");
super.new(inst);
endfunction

`uvm_object_utils_begin(transaction)
`uvm_field_int(a, UVM_DEFAULT);
`uvm_field_int(b, UVM_DEFAULT);
`uvm_field_int(c, UVM_DEFAULT);
`uvm_object_utils_end

endclass
```

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

transaction t;

function new(input string inst = "GEN");
super.new(inst);
endfunction

virtual task body();
t= transaction::type_id::create("TRANS");
start_item(t);
t.randomize();
t.print();
finish_item(t);
`uvm_info("GEN", "Data send to Driver", UVM_NONE);
endtask


endclass
```

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

UVM-Sequence_item ⟶ UVM-object

1) Keep track of all inputs and outputs in RTL
2) Use Modifier for data inputs / control inputs ⟶ rand, randC
3) Do not use Modifier for o/p ports
4) Register data members to Factory using macros (object-utils)

```
class transaction extends uvm_sequence_item;
rand bit [3:0] a;
rand bit [3:0] b;
bit [4:0] c;

function new (input string inst = "TRANS");
super.new(inst);
endfunction

`uvm_object_utils_begin(transaction)
`uvm_field_int(a, UVM_DEFAULT);
`uvm_field_int(b, UVM_DEFAULT);
`uvm_field_int(c, UVM_DEFAULT);
`uvm_object_utils_end

endclass
```

(1)

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

transaction t;
integer i;

function new(input string inst = "GEN");
super.new(inst);
endfunction

virtual task body();
t= transaction::type_id::create("TRANS");

for(i =0; i<10; i++) begin
start_item(t);
`uvm_info("GEN", "Data send to Driver", UVM_NONE);
t.randomize();
  t.print(uvm_default_line_printer);
#10;
finish_item(t);
end
endtask
endclass
```

(2)

uvm-sequence #(type of data object) ⟶ uvm-object

1) Generate random transactions for DUT
2) start-item() and finish-item() for sending data to Driver with sequencer

a) start-item() ⟶ we have valid sequence
b) Generate sequence
c) finish-item() ⟶ sequence send to driver

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

transaction t;
virtual top_if vif;

function new(input string inst = "DRV", uvm_component c);
super.new(inst,c);
endfunction

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
```

uvm-driver #(type of data object) ⟶ uvm-component

1) Get stimulus from sequence and apply to DUT with Interface

```
super.new(inst,c);
endfunction

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
t= transaction::type_id::create("TRANS");
if(!uvm_config_db#(virtual top_if)::get(this,"","vif",vif))
`uvm_info("DRV", "Unable to access the config db", UVM_NONE);
endfunction

virtual task run_phase(uvm_phase phase);
forever begin
seq_item_port.get_next_item(t);
vif.a = t.a;
vif.b = t.b;
`uvm_info("DRV", "Rcvd data from seq", UVM_NONE);
t.print(uvm_default_line_printer);
seq_item_port.item_done();
  end
endtask
endclass
```

1) Get stimulus from sequence and apply to DUT with interface
2) Access interface with config-db
3) get_next_item → receives next item from uvm_sequence
4) item_done → request completed ready to receive new request
5) apply data to DUT with virtual interface

```
class monitor extends uvm_monitor;
`uvm_component_utils(monitor)

uvm_analysis_port #(transaction) send;

function new(input string inst = "MON", uvm_component c);
super.new(inst,c);
send = new("WRITE",this);
endfunction

transaction t;
virtual top_if vif;

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
t = transaction::type_id::create("TRANS", this);

 if(!uvm_config_db#(virtual top_if)::get(this,"","vif",vif))
      `uvm_info("MON", "Unable to access the config db", UVM_NONE);
endfunction

virtual task run_phase(uvm_phase phase);
forever begin
#10;
  t.a = vif.a;
  t.b = vif.b;
  t.c = vif.c;
  `uvm_info("MON", "Data send to Scoreboard", UVM_NONE);
  t.print(uvm_default_line_printer);
  send.write(t);
end
endtask

endclass
```

uvm_monitor → uvm component

1) Receives data from DUT with help of interface and apply it to Scoreboard
2) Update data container with value receive from virtual interface.
3) use analysis port to send data to Scoreboa

```
class scoreboard extends uvm_scoreboard;
`uvm_component_utils(scoreboard)

uvm_analysis_imp #(transaction,scoreboard) recv;

transaction data;


function new(input string inst = "SCO", uvm_component c);
super.new(inst,c);
recv = new("READ",this);
endfunction


virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
data = transaction::type_id::create("TRANS", this);
endfunction
```

uvm_scoreboard → uvm_component

1) receives transaction from Monitor and perform comparison with Golden data
2) implementation of analysis port

```
virtual function void write(input transaction t);
data = t;
`uvm_info("SCO", "Data rcvd from Monitor", UVM_NONE);
 data.print(uvm_default_line_printer);
 if(data.c == data.a + data.b)
`uvm_info("SCO", "Test Passed", UVM_NONE)
else
`uvm_info("SCO", "Test Failed", UVM_NONE)
endfunction


endclass


class agent extends uvm_agent;
 `uvm_component_utils(agent)
 driver d;
 monitor m;
 uvm_sequencer #(transaction) sequencer;

 function new(input string inst = "AGENT", uvm_component c);
 super.new(inst,c);
 endfunction

 virtual function void build_phase(uvm_phase phase);
 super.build_phase(phase);
 d = driver::type_id::create("DRV",this);
 sequencer = uvm_sequencer #(transaction)::type_id::create("SEQ",this);
   m = monitor::type_id::create("MON",this);
 endfunction

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

uvm_agent → uvm_component

1) Create instances of driver, Sequencer and Monitor

2) connect driver and sequencer

driver → initiator

Sequencer → Target

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

agent a;
scoreboard s;

function new(input string inst = "ENV", uvm_component c);
super.new(inst,c);
endfunction

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
a = agent::type_id::create("AGENT",this);
s = scoreboard::type_id::create("SCO",this);
endfunction

virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
a.m.send.connect(s.recv);
endfunction
endclass
```

uvm_env → uvm_component

1) Create instance of agent and scoreboard

2) Connect analysis port of Monitor and Scoreboard

Monitor → Initiator

Scoreboard → Target

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

function new(input string inst = "TEST", uvm_component c);
super.new(inst,c);
endfunction

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

virtual task run_phase(uvm_phase phase);
phase.raise_objection(phase);
g.start(e.a.sequencer);
#10;
phase.drop_objection(phase);
endtask
endclass
```

uvm_test ⟶ uvm_component

1) create instance of environment

2) connect generator and Sequencer

Choose specific sequence to send to driver

Generator ⟶ Sequencer

```
module tb;
test t;
top_if vif();

top dut (.a(vif.a), .b(vif.b), .c(vif.c));

initial begin
  $dumpvars;
  $dumpfile("dump.vcd");
t = new("TEST",null);
uvm_config_db #(virtual top_if)::set(null, "*","vif",vif);
run_test();
end

endmodule
```

1) Create test instance

2) Create interface instance

3) Connect DUT to interface

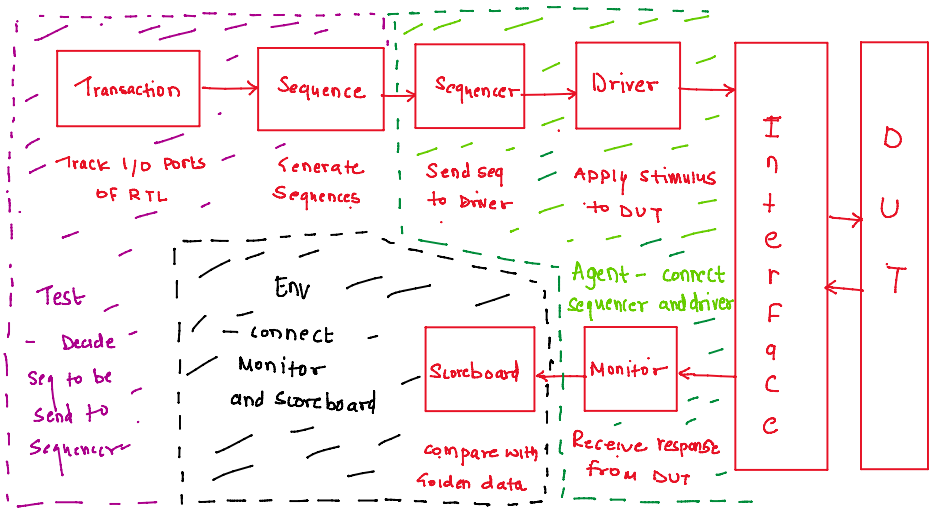
4) Set u

```
module top (
input [3:0] a,b,
output [4:0] c
);

assign c =  a + b;

endmodule
```

```
interface top_if();
logic [3:0] a;
logic [3:0] b;
logic [4:0] c;
endinterface
```



Agent → Monitor + Driver + Sequencer

Env → Agent + Scoreboard

Test → Gen + Env

1) Transaction → UVM-Sequence-item → UVM-object (Constructor → Instance name)
   - Register data Members to Factory (Macro)
   - Input ports → rand, randc

2) Sequence / Generator → UVM-sequence #(Type) → UVM-object
   - Generate Randomize sequence (start-item() → Finish-item())

3) Driver → UVM-driver #(Type) → UVM-Component
   - Receive data from the TLM port (seq-item-port.get-next-item() seq-item-port.item-done())
   - Get an access to Interface (UVM-config-db)
   - Drive the Inputs

4) Monitor → UVM-Monitor → UVM-Component
   - Receive the response from DUT
   - Send transactions to Scoreboard with analysis port.

5) Scoreboard → UVM-Scoreboard → UVM-component
   - Receive transactions from Monitor
   - Compare with Golden data.

UVM agent → UVM-component

- Compare with Golden data.

6) Agent → uvm-agent → uvm-component

M + D + Seq

- connect Driver and sequence (connect-phase)

7) Env → uvm-env → uvm-component

A + S

- connect analysis port of M and S.

8) Test → uvm-test → uvm-component

G + E

- Send specific sequence to the Driver