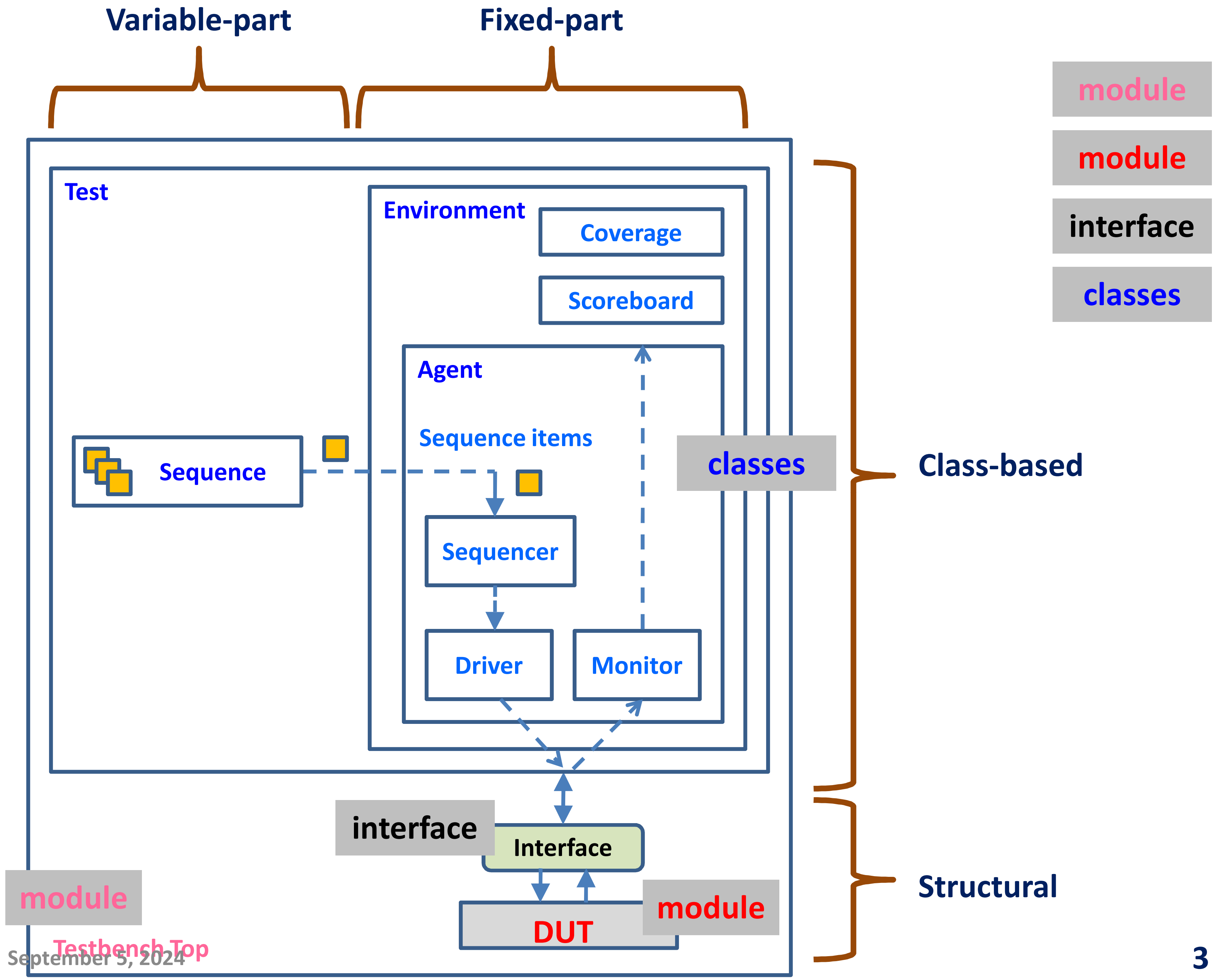
UVM (Universal Verification Methodology)
For whom can know how to program

Tuan Nguyen-viet

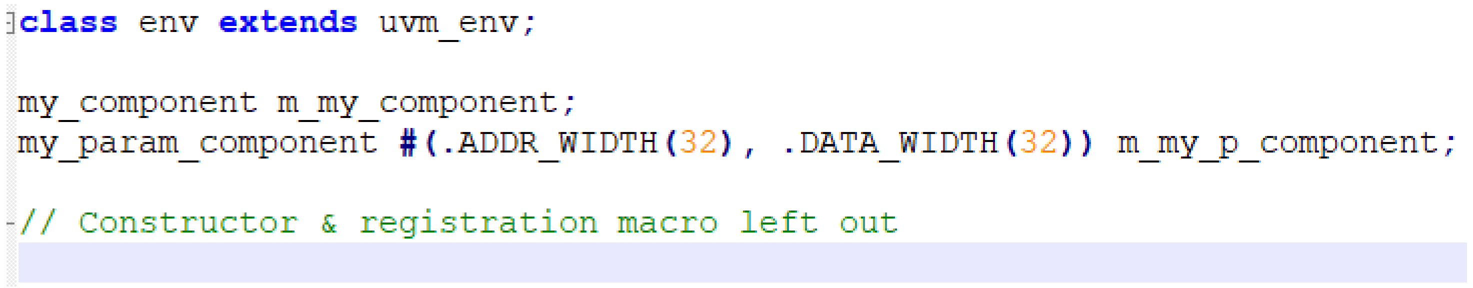
DUT and UVM Testbench – Basic

DUT	Interface	Testbench Top	Test
1	2	3	4
Sequence	Environment	Agent	Sequence Item
5	6	7	8
Sequencer	Driver	Monitor	Scoreboard
9	10	11	12
Configuration	Packages	Coverage	
		13	


Variable-part
Fixed-part
module
module
interface
classes
Test
Environment
Coverage
Scoreboard
Agent
Sequence items
Sequence
Sequencer
Driver
Monitor
classes
Class-based
interface
Interface
module
DUT
module
Testbench Top
Structural

UVM Architecture Implementation

- **1. Interface**
- **2. Testbench Top**
- **3. UVM Components/Objects**
  - 1. Component/object factory registration using Utility macro (see API)
  - 2. Adding factory constructor defaults (see prototype template)
  - 3. Component/object creation (note: **build process** is top-down), e.g.

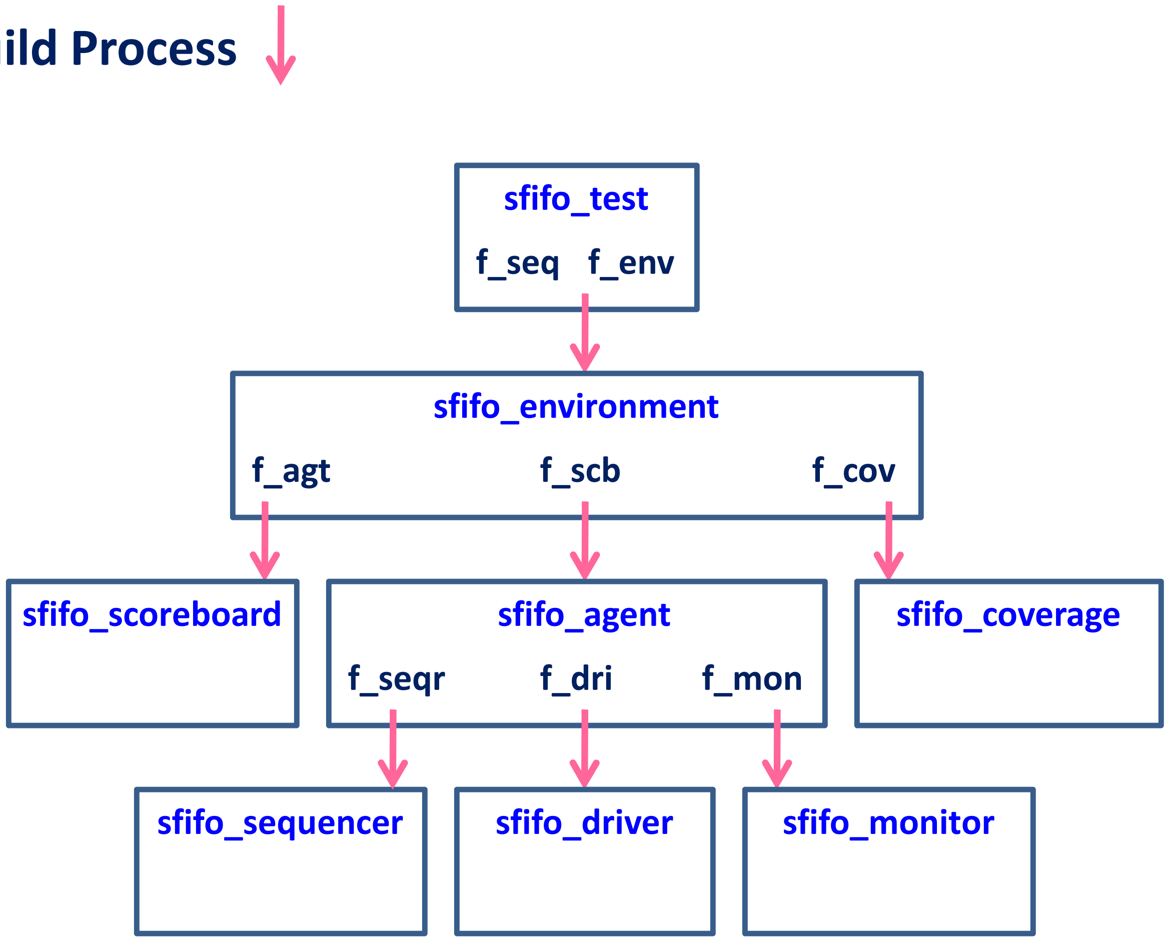


```
class env extends uvm_env;

my_component m_my_component;
my_param_component #(.ADDR_WIDTH(32), .DATA_WIDTH(32)) m_my_p_component;

// Constructor & registration macro left out
```

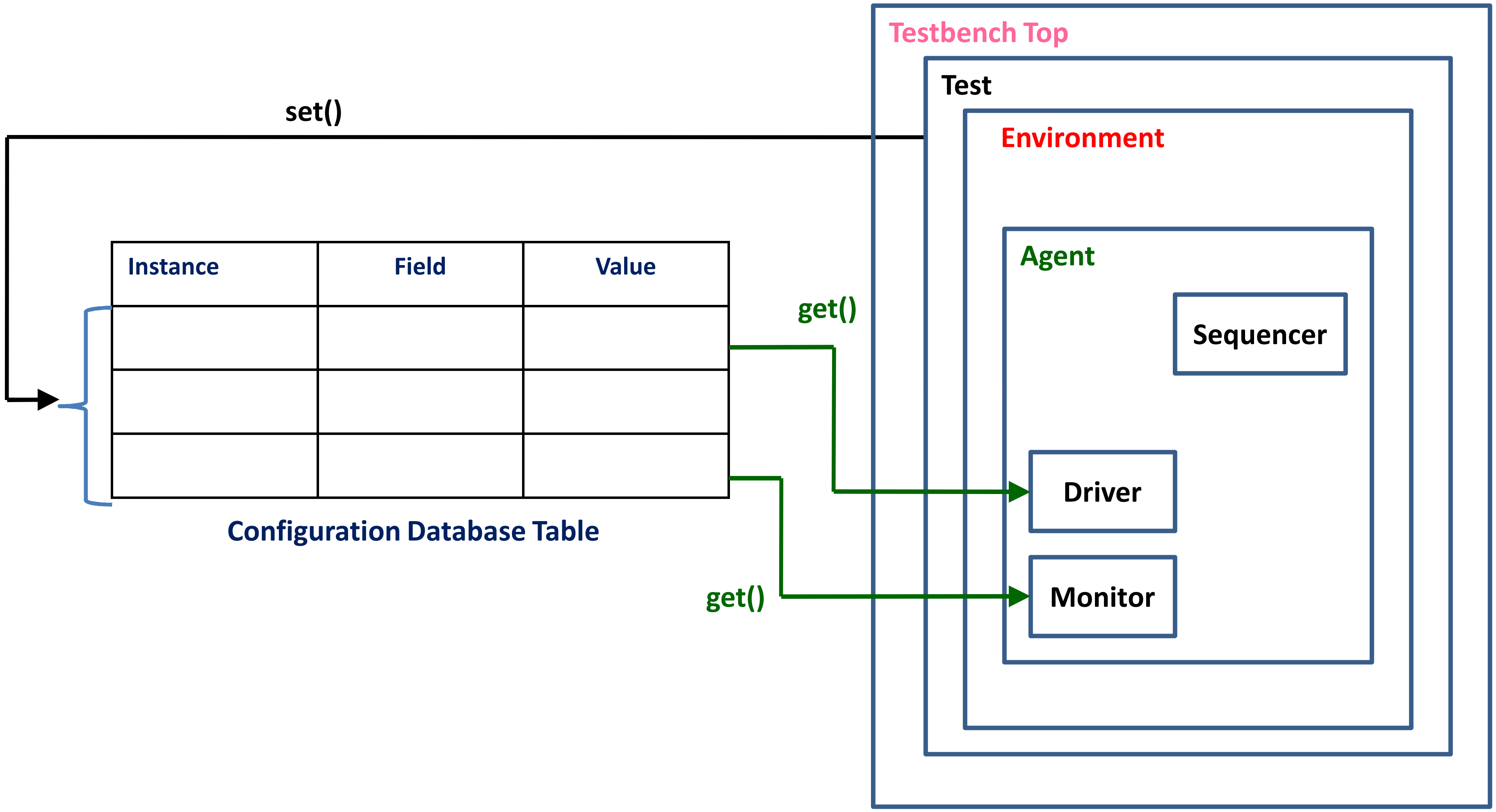
  - 4. Configuration database (see API)

UVM Architecture Implementation (2)

- 4. Build Process



Configuration Database (2)

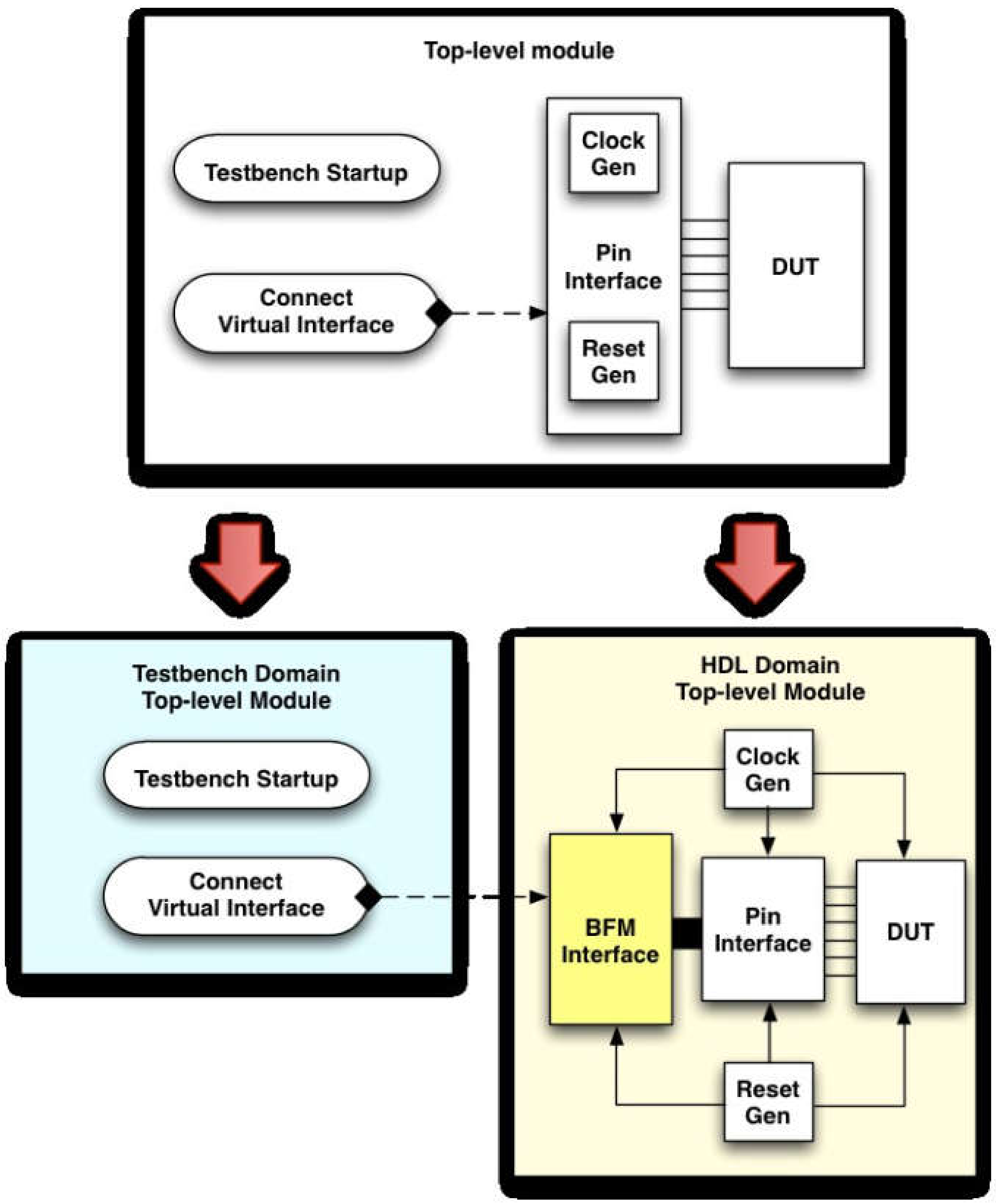


UVM Architecture Implementation (3)

- **5. Connection Process**
  - UVM **connect phase** follows the **build phase**
    - and works back up *from the bottom of the hierarchy to the top*.
  - Its purpose is to
    - make *TLM connections* between components,
    - assign **virtual interface** handles
    - and make any other assignments for resources.
  - Configuration objects are once again at play during the connection process
    - as they may contain references to **virtual interfaces** or other information that guides the connection process.
    - For instance, inside an **agent**,
      - the **virtual interface** assignment to a **driver**
      - and the **TLM connection** between a **driver** and its **sequencer**
        - » are only made if the **agent** is active.

  1. Virtual interface
  2. Sequence - Sequencer
  3. Analysis port

UVM Architecture Implementation (4)



UVM Architecture Implementation (5)

```
module top_tb;

bus_if BUS();
gpio_if GPIO();
bidirect_bus_slave DUT(.bus(BUS), .gpio(GPIO));
```

Interface and DUT Instantiation

```
// Free running clock
initial
  begin
    BUS.clk = 0;
    forever begin
      #10 BUS.clk = ~BUS.clk;
    end
  end
```

Clock Gen

```
// Reset
initial
  begin
    BUS.resetn = 0;
    repeat(3) begin
      @(posedge BUS.clk);
    end
    BUS.resetn = 1;
  end
```

Reset Gen

```
// UVM start up:
initial
  begin
    uvm_config_db #(virtual bus_if)::set(null, "uvm_test_top", "BUS_vif" , BUS);
    run_test("bidirect_bus_test");
  end

endmodule: top_tb
```

Connect Virtual Interface

Testbench Startup

Testbench Domain

HDL Domain

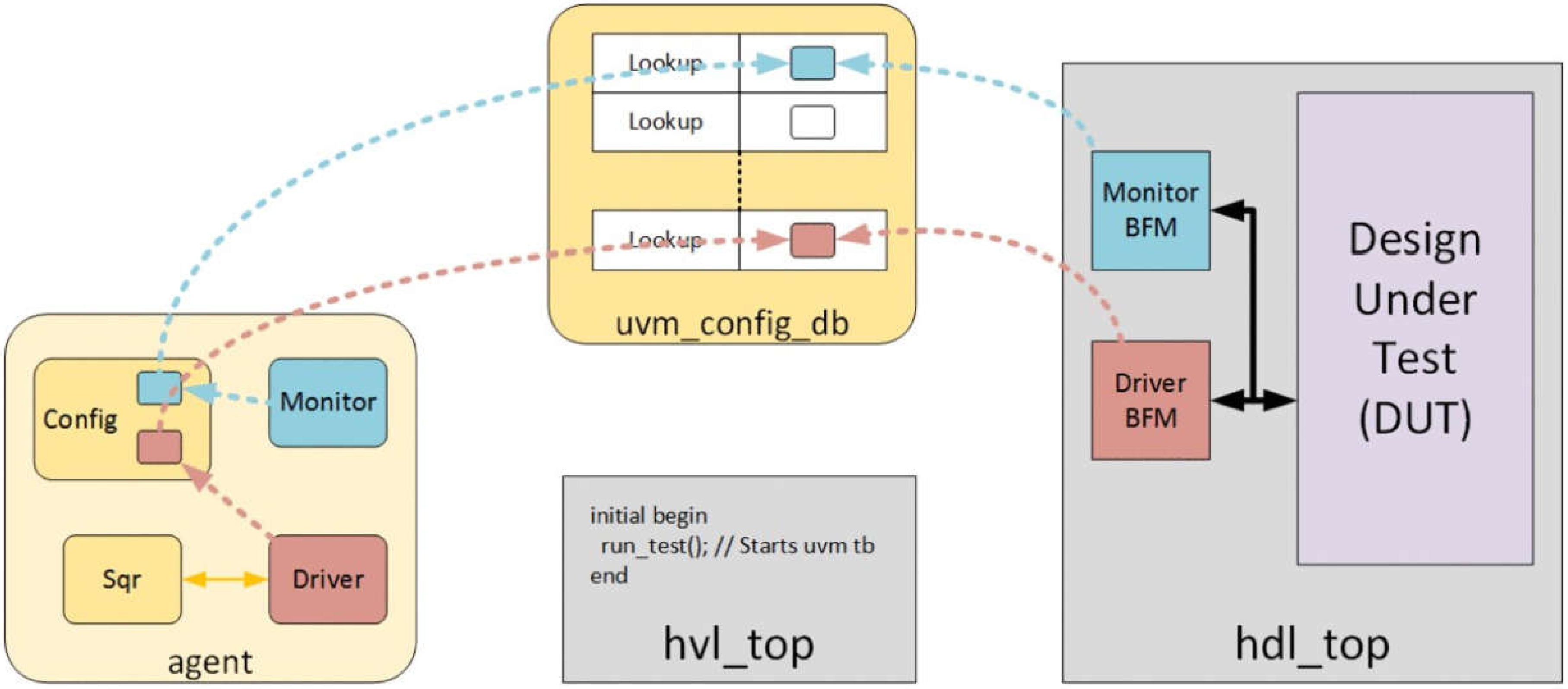
UVM Architecture Implementation (6): tb_top.sv

The **uvm_config_db** is parameterised with the type ***virtual sfifo_interface***

1. The first argument of the **set()** method is **context**, intended to be assigned a UVM component object handle;
   1. in this case since it is in the HDL part of the testbench,
      - a **null** object handle is assigned.
   2. Use "**null**" in the first argument as this code is in a top-level module rather than a uvm_component.
2. The second argument of the **set()** method is a **string** used to identify the UVM component instance name(s) within the UVM testbench component hierarchy that may access the data object.
   – This is “uvm_test_top” here to restrict access to the top level UVM test object.
   ➔ It could have been assigned a *wildcard* such as **“*”**,
      - which means that all components in the UVM testbench could access it,
        – but this may not be helpful, and carrys a potential lookup overhead in the **get()** process.
3. The third argument of the **set()** method is a **string**, intended as the lookup name, i.e. a string that can be used to uniquely identify the ***virtual interface*** from within the **uvm_config_db**.
4. The final argument of the **set()** is the static interface assigned to the ***virtual interface*** handle entry that is created within the **uvm_config_db**.

Connecting the Testbench to the DUT

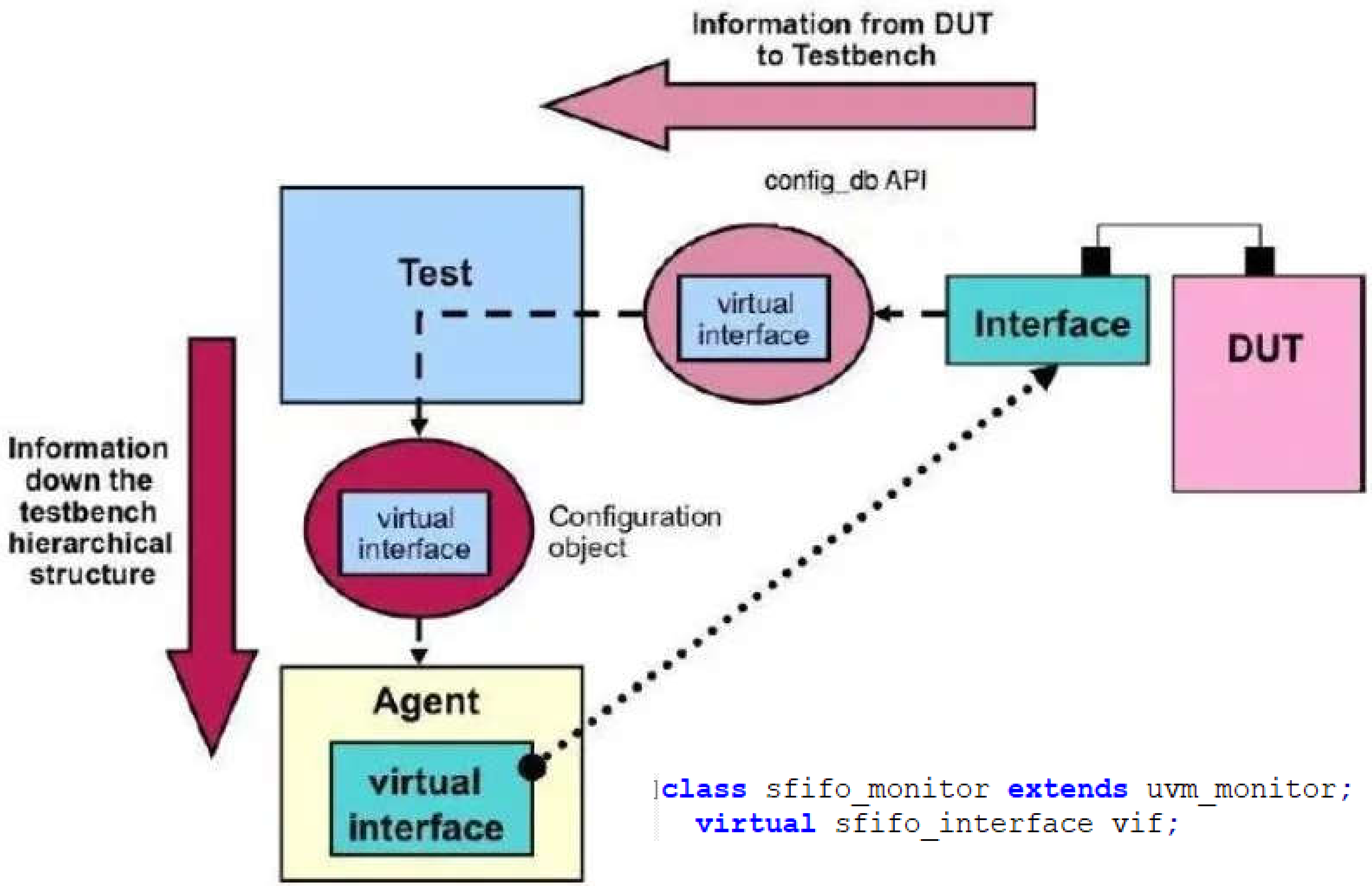
- UVM methodology uses the **uvm_config_db** utility to pass a **virtual interface** handle from a static testbench module to a UVM object class.



```
initial begin
  run_test(); // Starts uvm tb
end
```

Using the uvm_config_db to pass virtual interface handles from hdl_top to an agent

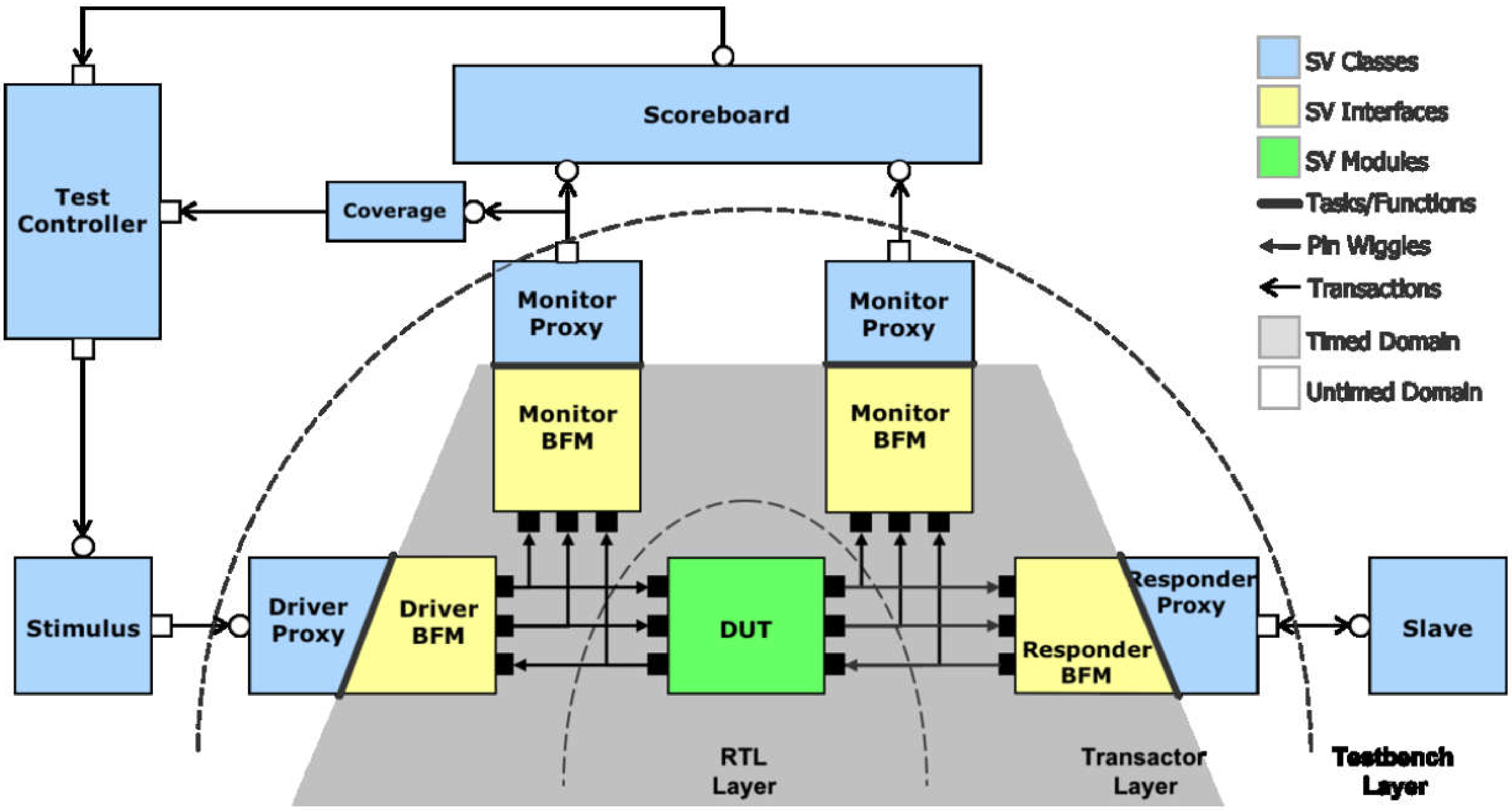
```
initial begin
  uvm_config_db#(virtual sfifo_interface)::set(null, "*", "vif", tif);
```



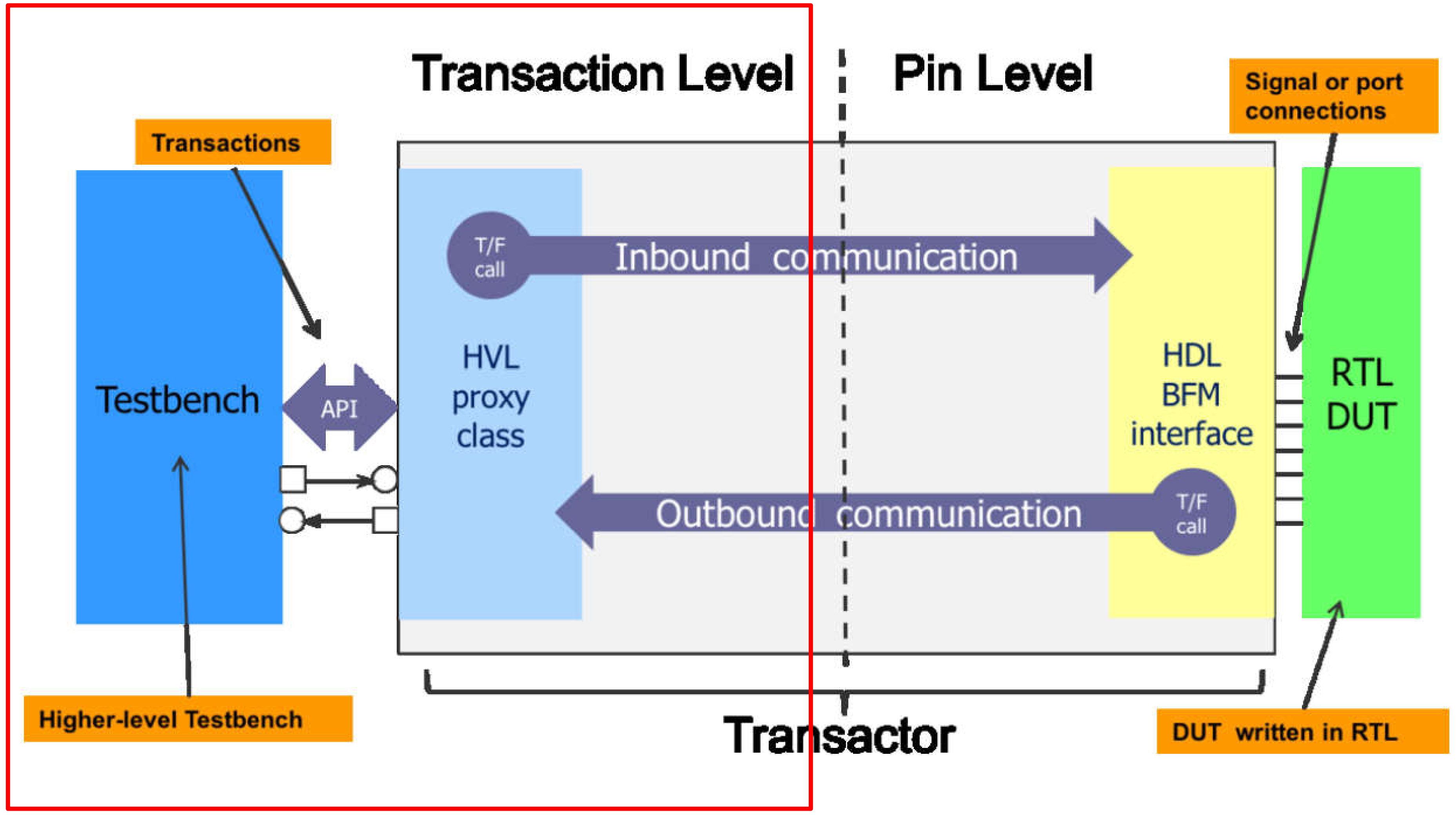
```
class sfifo_monitor extends uvm_monitor;
  virtual sfifo_interface vif;
```

Diagram 1: Graphical View of DUT-TB Connection (Source: Cookbook)

UVM Testbench Architecture (Cookbook)



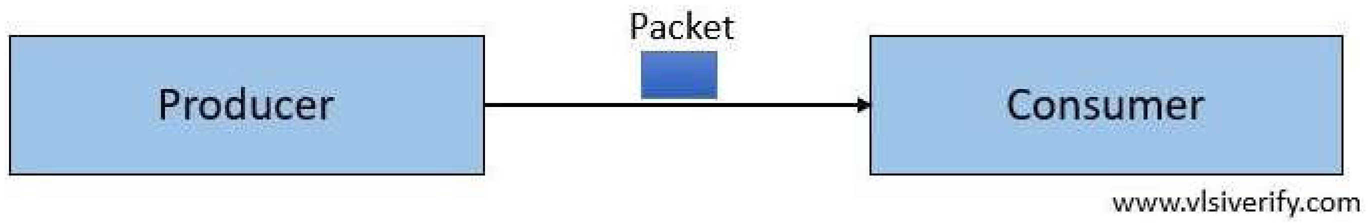
Transactor (Cookbook)



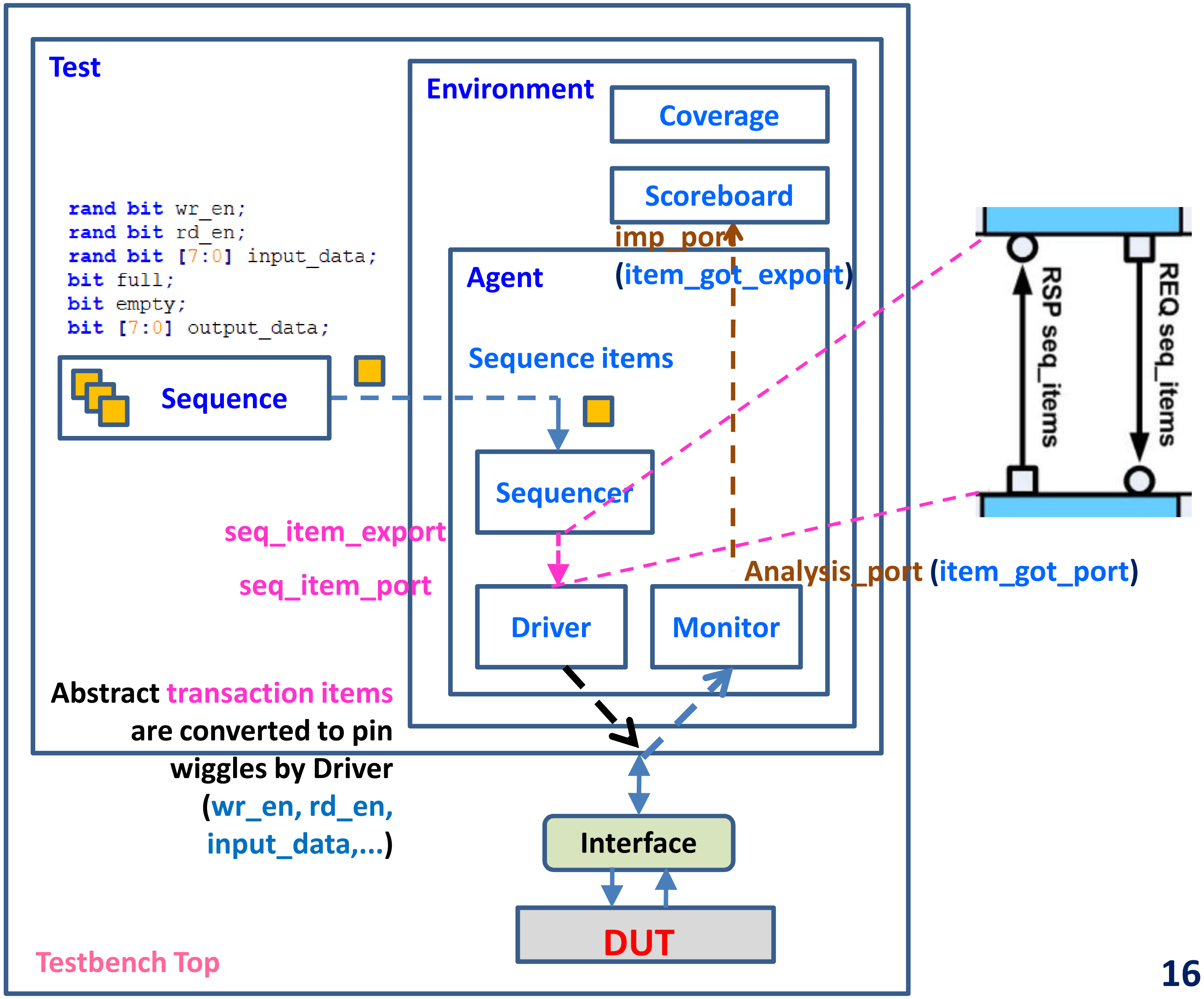
From Next Slide on

Transaction / Sequence Item

- In Transaction Level Modeling (TLM),
  - **data** is represented as **transactions** that flow between components via *TLM interfaces*.
- *These interfaces* provide a way to connect and transfer **data packets** between components,
  - enabling efficient communication within a chip design verification process.
- TLM establishes a connection between producer and consumer components through which **transactions** are sent.
  - A **transaction** is nothing but a class object containing specific information.



Transaction / Sequence Item (2)



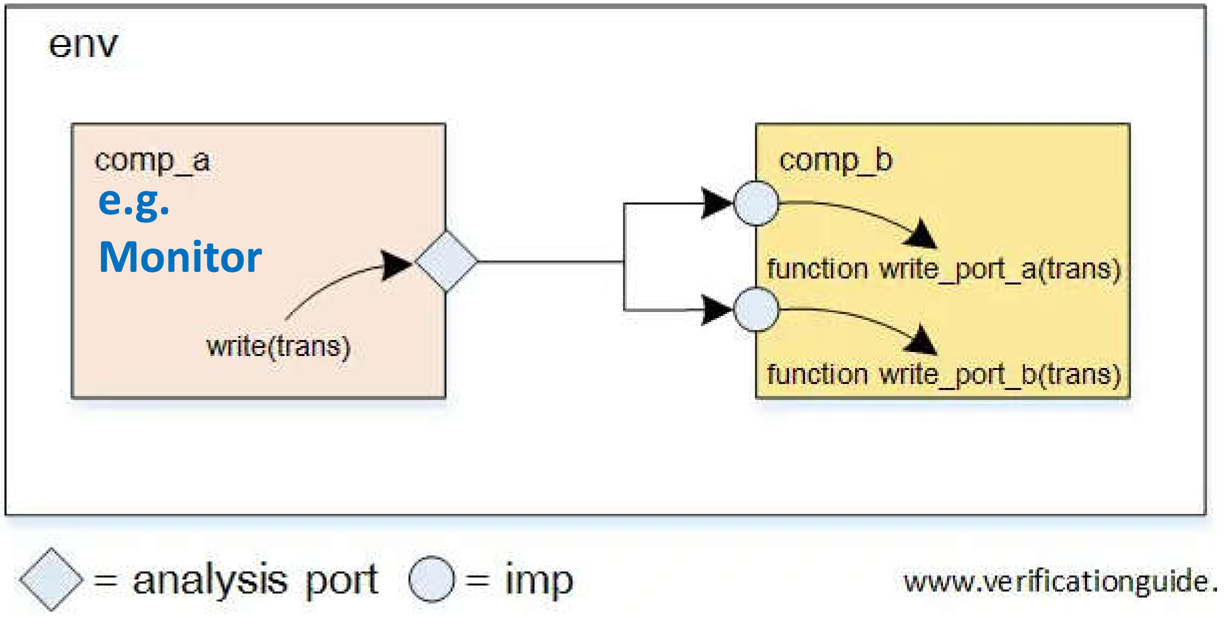
```
rand bit wr_en;
rand bit rd_en;
rand bit [7:0] input_data;
bit full;
bit empty;
bit [7:0] output_data;
```

Abstract transaction items are converted to pin wiggles by Driver (wr_en, rd_en, input_data,...)

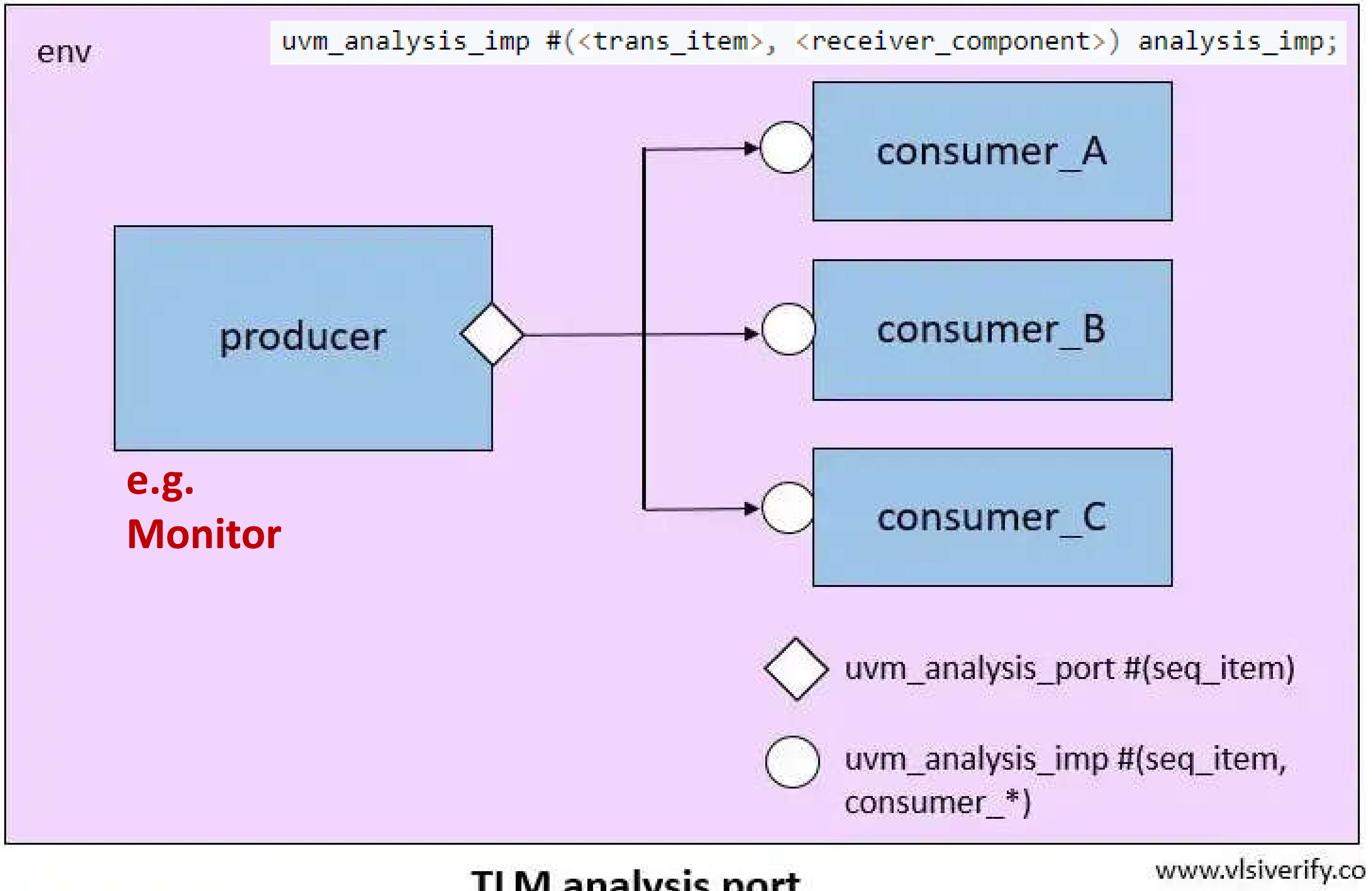
TLM Analysis port / Multi Analysis imp port

- Connecting multiple ports to a single **Analysis port**:
  - The uvm_analysis_port is a TLM-based class that provides a **write** method for communication.
  - TLM **Analysis port** broadcasts ***transactions*** to one or multiple components.

```
uvm_analysis_imp #(<trans_item>, <receiver_component>) analysis_imp;
```



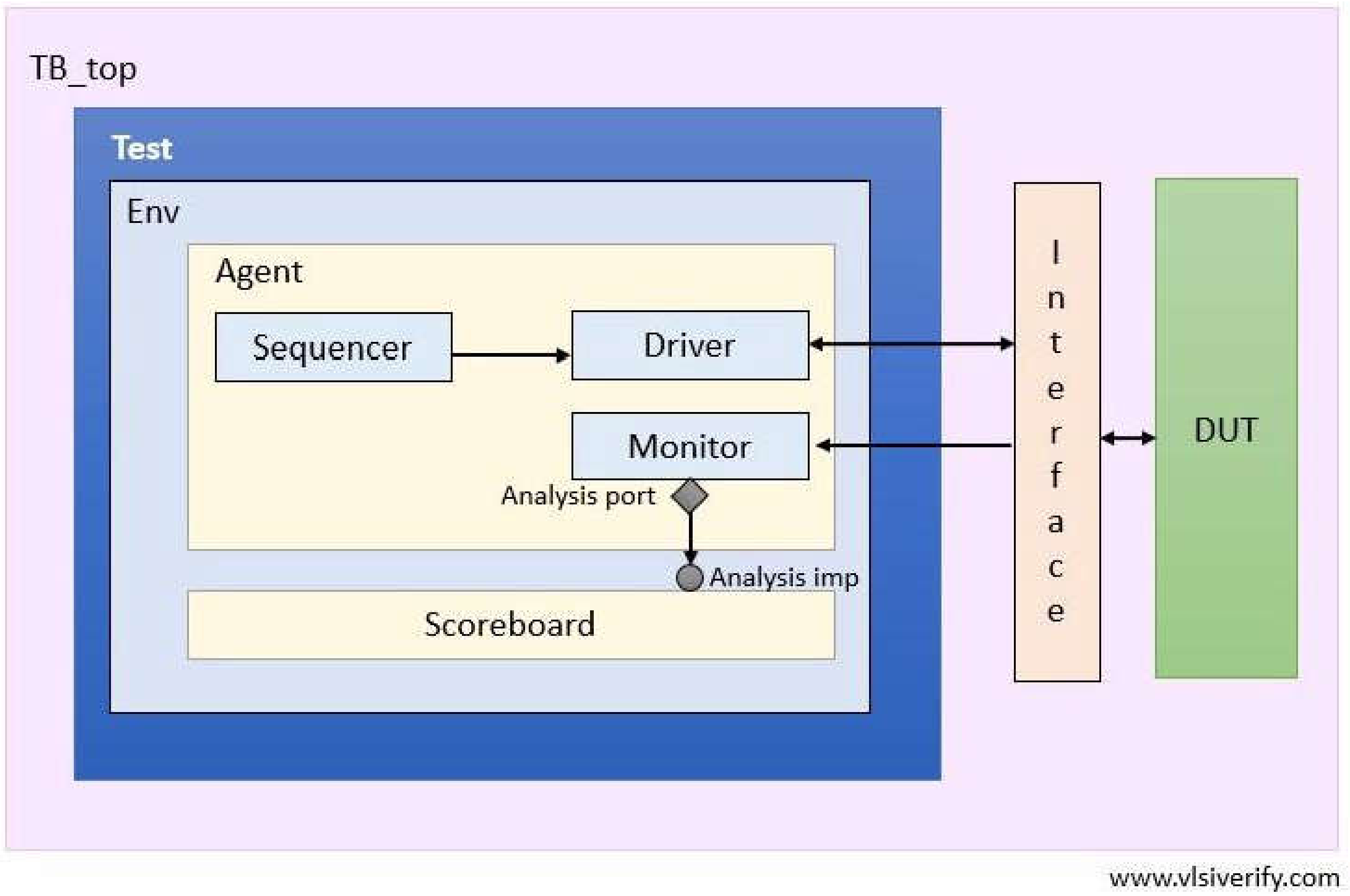
TLM Analysis port / Multi Analysis imp port (2)


```
uvm_analysis_imp #(<trans_item>, <receiver_component>) analysis_imp;
```
env
consumer_A
producer
consumer_B
e.g.
Monitor
consumer_C
uvm_analysis_port #(seq_item)
uvm_analysis_imp #(seq_item, consumer_*)
www.vlsiverify.co

TLM analysis port

TLM Analysis port / Multi Analysis imp port (3)

- An example of connecting to a single Analysis port:



TLM Analysis port / Multi Analysis imp port (4)

at sfifo_monitor

```
sfifo_seq_item item_got;
//Step-1. Declaring analysis port
uvm_analysis_port#(sfifo_seq_item) item_got_port;

//Step-2. Creating analysis port
item_got_port = new("item_got_port", this);

//Step-3. Calling write method
item_got_port.write(item_got);
```

at sfifo_scoreboard

Syntax: uvm_analysis_imp<> #(t,T) port_name;

```
uvm_analysis_imp #(<trans_item>, <receiver_component>) analysis_imp;
```

```
class sfifo_scoreboard extends uvm_scoreboard;
  // 1. Declare the analysis port
  uvm_analysis_imp#(sfifo_seq_item, sfifo_scoreboard) item_got_export;
```

Syntax:

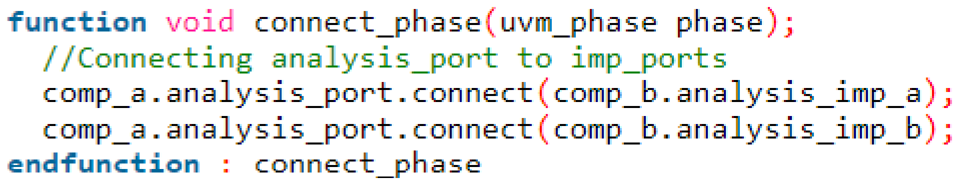
```
virtual function void write_port_a(transaction trans);
```

```
// 2. Implement the write method for the analysis port
//---------------------------------------
// Analysis port write method
//---------------------------------------
function void write(input sfifo_seq_item item_got);
```

TLM Analysis port / Multi Analysis imp port (5)

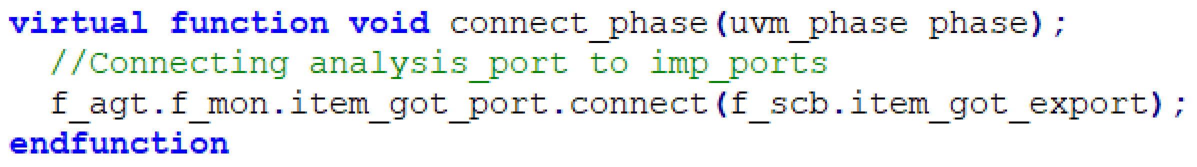
- Connecting **Analysis port** with the **imp_port** in environment:

Syntax:



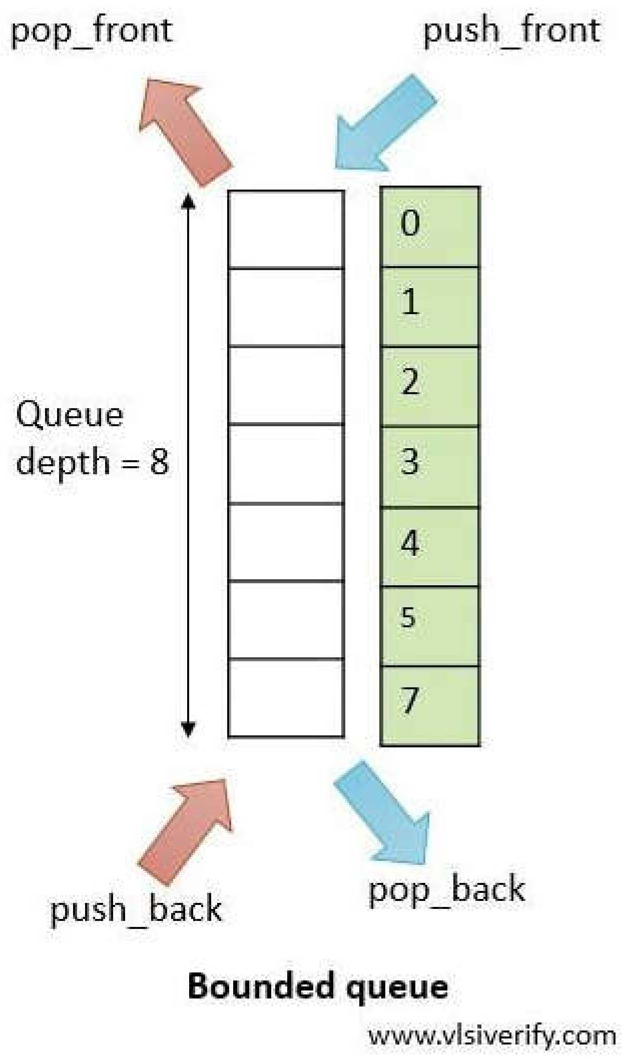
```
function void connect_phase(uvm_phase phase);
  //Connecting analysis_port to imp_ports
  comp_a.analysis_port.connect(comp_b.analysis_imp_a);
  comp_a.analysis_port.connect(comp_b.analysis_imp_b);
endfunction : connect_phase
```

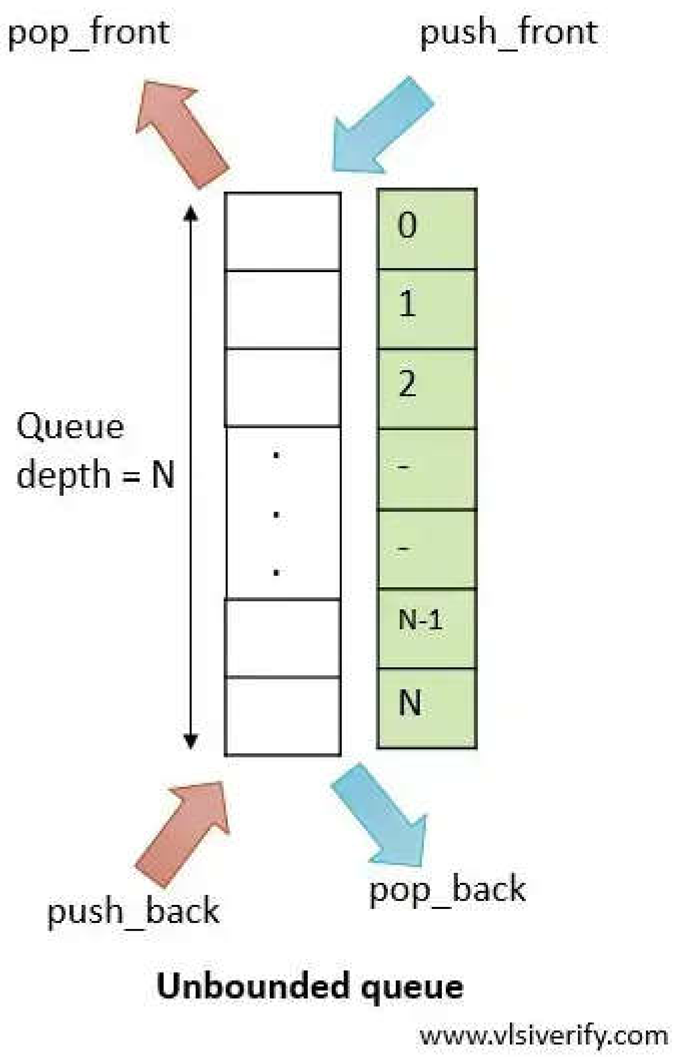
at sfifo_environment



```
virtual function void connect_phase(uvm_phase phase);
  //Connecting analysis_port to imp_ports
  f_agt.f_mon.item_got_port.connect(f_scb.item_got_export);
endfunction
```

Queue and Its Methods at Scoreboard





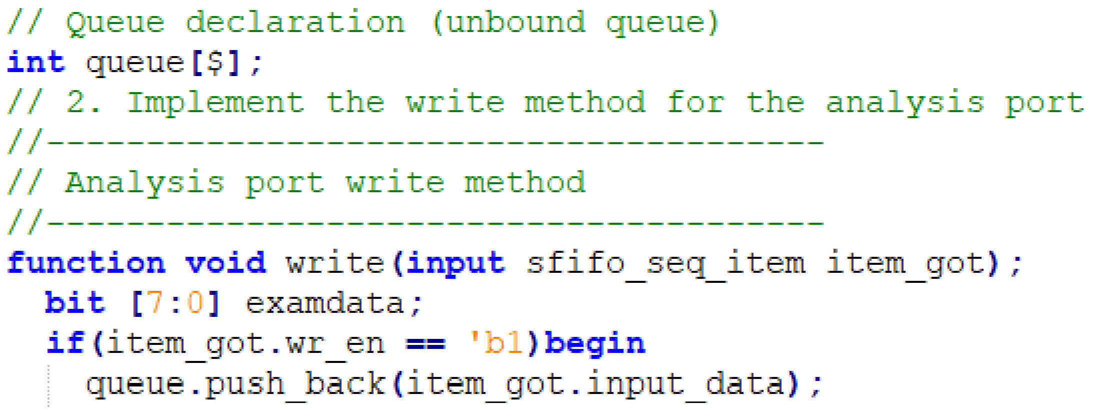
Queue and Its Methods at Scoreboard (2)

- Queue declaration:
  - **data_type queue_name[$];**
    - data_type – data type of the queue elements.
    - queue_name – name of the queue.
  - E.g.
    - **int queue[$];** // queue of int, (unbound queue)

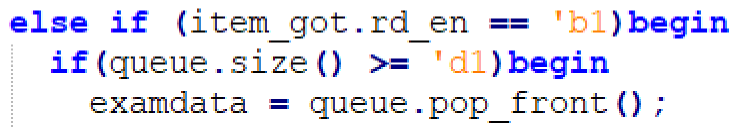
- Typical methods:
  - **push_back()**
    - inserts the given element at the end of the queue
  - **pop_front()**
    - removes and returns the first element of the queue

Queue and Its Methods at Scoreboard (3)

at sfifo_scoreboard

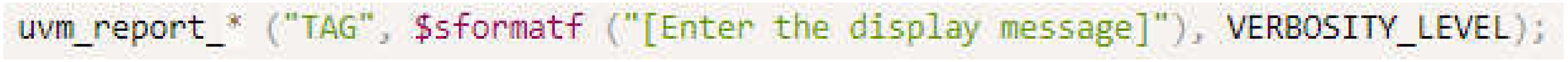


```
// Queue declaration (unbound queue)
int queue[$];
// 2. Implement the write method for the analysis port
//-----------------------------------------
// Analysis port write method
//-----------------------------------------
function void write(input sfifo_seq_item item_got);
  bit [7:0] examdata;
  if(item_got.wr_en == 'b1)begin
    queue.push_back(item_got.input_data);
```



```
  else if (item_got.rd_en == 'b1)begin
    if(queue.size() >= 'd1)begin
      examdata = queue.pop_front();
```

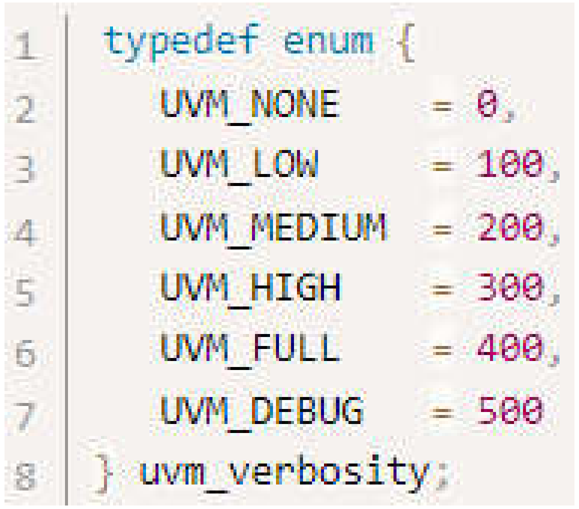
User Communication



```
uvm_report_* ("TAG", $sformatf ("[Enter the display message]"), VERBOSITY_LEVEL);
```

where * can be either **info**, **error**, **warning**, **fatal**.

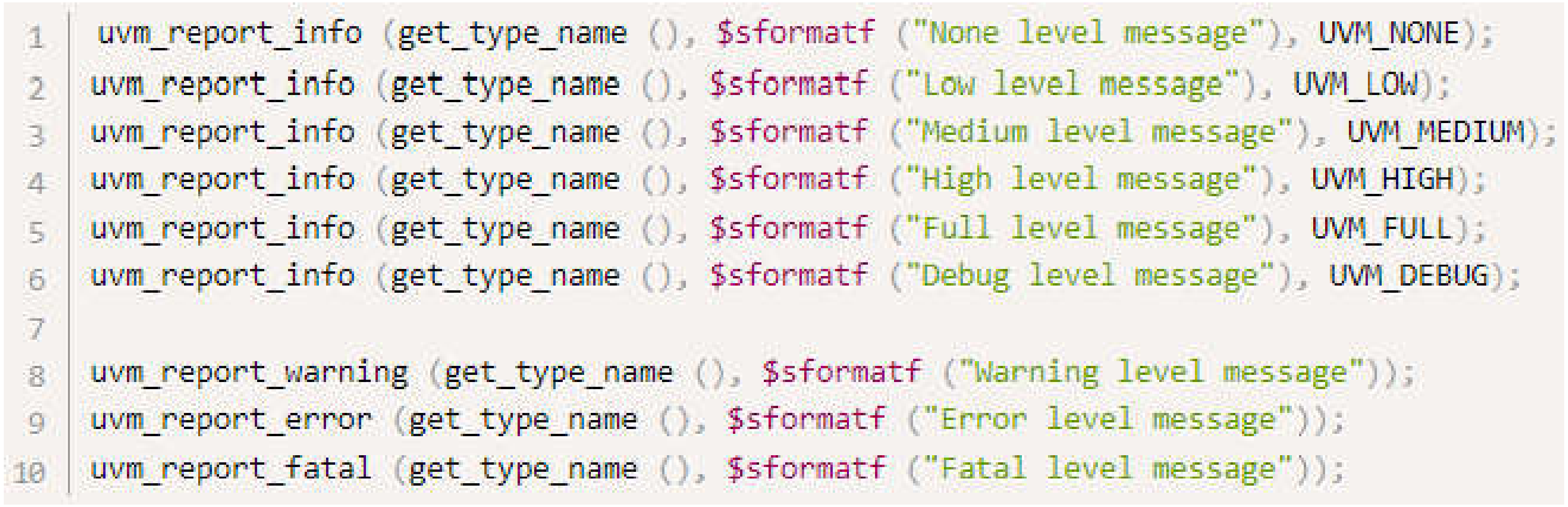
- UVM has **six** levels of verbosity with each one represented by an integer.



```
typedef enum {
   UVM_NONE     = 0,
   UVM_LOW      = 100,
   UVM_MEDIUM   = 200,
   UVM_HIGH     = 300,
   UVM_FULL     = 400,
   UVM_DEBUG    = 500
} uvm_verbosity;
```

User Communication (2)

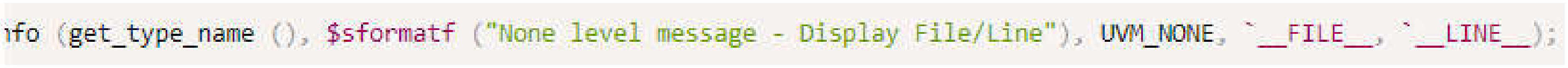
- Note that the VERBOSITY_LEVEL is only required for **uvm_report_info**.
- Usage of **uvm_report_fatal** will exit the simulation.



```
uvm_report_info (get_type_name (), $sformatf ("None level message"), UVM_NONE);
uvm_report_info (get_type_name (), $sformatf ("Low level message"), UVM_LOW);
uvm_report_info (get_type_name (), $sformatf ("Medium level message"), UVM_MEDIUM);
uvm_report_info (get_type_name (), $sformatf ("High level message"), UVM_HIGH);
uvm_report_info (get_type_name (), $sformatf ("Full level message"), UVM_FULL);
uvm_report_info (get_type_name (), $sformatf ("Debug level message"), UVM_DEBUG);

uvm_report_warning (get_type_name (), $sformatf ("Warning level message"));
uvm_report_error (get_type_name (), $sformatf ("Error level message"));
uvm_report_fatal (get_type_name (), $sformatf ("Fatal level message"));
```

User Communication (3)

- We can also display the filename and line number of the display message
  - by using **`__FILE__** and **`__LINE__**,
    - which will be useful for debug purposes.

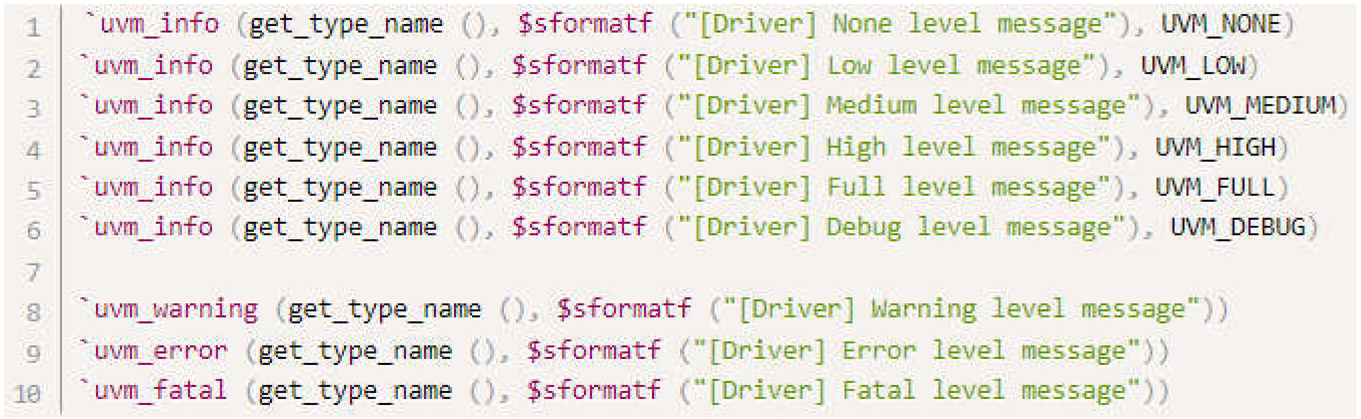


```
nfo (get_type_name (), $sformatf ("None level message - Display File/Line"), UVM_NONE, `__FILE__, `__LINE__);
```

- This can be disabled from **command-line** by defining **+UVM_REPORT_DISABLE_FILE_LINE**

User Communication (4)

- UVM reporting macros will automatically display the file and line information
  - without explicitly mentioning the **`__FILE__** and **`__LINE__** arguments.



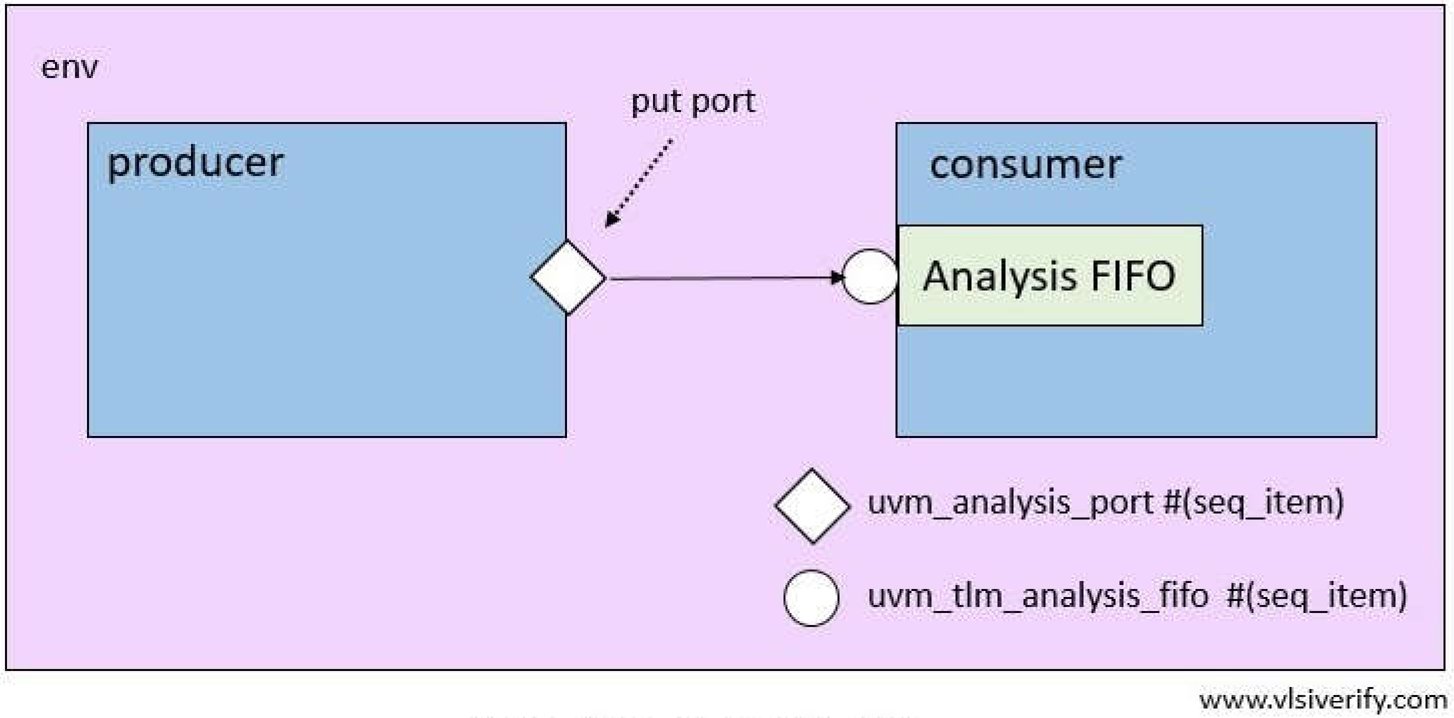
```
`uvm_info (get_type_name (), $sformatf ("[Driver] None level message"), UVM_NONE)
`uvm_info (get_type_name (), $sformatf ("[Driver] Low level message"), UVM_LOW)
`uvm_info (get_type_name (), $sformatf ("[Driver] Medium level message"), UVM_MEDIUM)
`uvm_info (get_type_name (), $sformatf ("[Driver] High level message"), UVM_HIGH)
`uvm_info (get_type_name (), $sformatf ("[Driver] Full level message"), UVM_FULL)
`uvm_info (get_type_name (), $sformatf ("[Driver] Debug level message"), UVM_DEBUG)

`uvm_warning (get_type_name (), $sformatf ("[Driver] Warning level message"))
`uvm_error (get_type_name (), $sformatf ("[Driver] Error level message"))
`uvm_fatal (get_type_name (), $sformatf ("[Driver] Fatal level message"))
```

User Communication (5)

- Reporting functions:
  - Reference
    - https://www.chipverify.com/uvm/report-functions

Other Notes

TLM Analysis FIFO


env
put port
producer
consumer
Analysis FIFO
uvm_analysis_port #(seq_item)
uvm_tlm_analysis_fifo #(seq_item)
www.vlsiverify.com

uvm_tlm_analysis_fifo

Organizing package files into a directory

- **All files** *included in* ***a given package*** should be put together in a **single directory**.
  - This is particularly important for **Agents**
    - where the **Agent** directory structure needs to be **a complete stand-alone package**.

- A **single include directory** for **package files**
  - facilitates compilation flow set-up,
  - and also aids reuse
    - since **all the files** for **a package** can be gathered together easily.

Agent Note

1. **Active Agent**
   - Active **Agents** generate stimulus and drive to **DUT**
   - An active **Agent** shall consists of all the three components **Driver**, **Sequencer**, and **Monitor**.
2. **Passive Agent**
   - Passive **Agents** sample **DUT** signals but do not drive them
   - A passive **Agent** consists of only the **Monitor**.

3. An **Agent** can be configured as ACTIVE/PASSIVE by using a set config method,
   - the default **Agent** will be ACTIVE.

4. **get_is_active() Method**
   - **get_is_active()** returns
     - UVM_ACTIVE
       - if the **Agent** is acting as an active **Agent**
     - and UVM_PASSIVE
       - if the **Agent** is acting as a passive **Agent**.

Starting a sequence in UVM testbench

The Test is The Starting Point for The Build Process

➔ There are 2 ways of starting a sequence in UVM testbench.

1. Starting a sequence with **default_sequence** (implicit)

```
// build phase of uvm test
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = my_env::type_id::create("env", this);
    // starting a sequence with default_sequence
    uvm_config_db#(uvm_object_wrapper)::set(this,"env.agent.sequencer.run_phase", "default_sequence", my_sequence::type_id::get());
endfunction: build_phase
```

```
// build phase of uvm test
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  env = my_env::type_id::create("env", this);
  // starting a sequence with default_sequence
  uvm_config_db#(uvm_object_wrapper)::set(this, "env.agent.sequencer.run_phase", "default_sequence", my_sequence::type_id::get());
endfunction: build_phase
```

Starting a sequence in UVM testbench (2)

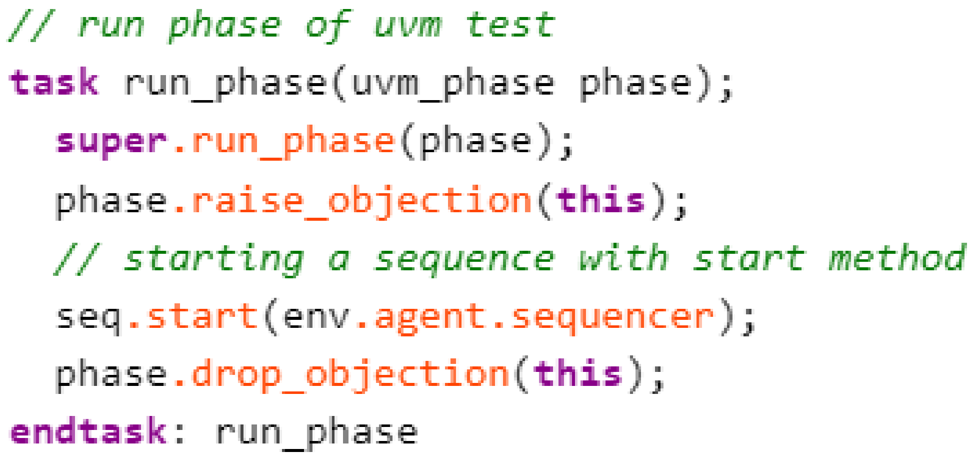
➔ There are 2 ways of starting a sequence in UVM testbench (cont'd).

2. Starting a sequence with ***start*** method (explicit)

```
// run phase of uvm test
task run_phase(uvm_phase phase);
    super.run_phase(phase);
    phase.raise_objection(this);
    // starting a sequence with start method
    seq.start(env.agent.sequencer);
    phase.drop_objection(this);
endtask: run_phase
```

Starting a sequence in UVM testbench (3)

- Many people recommend using the ***start*** method to start a sequence



```
// run phase of uvm test
task run_phase(uvm_phase phase);
  super.run_phase(phase);
  phase.raise_objection(this);
  // starting a sequence with start method
  seq.start(env.agent.sequencer);
  phase.drop_objection(this);
endtask: run_phase
```

Thank You