

# **Universal Verification Methodology (UVM)**

Umesh

# Introduction

# Why we need Methodology?

- Functional Verification consumes more than 60% of time and effort in a product life cycle.
- Engineers are constantly looking for new and better ways to perform functional verification.
- There are many proven and promising technologies for enhanced verification such as coverage-based verification, assertion-based verification, etc.
- Now the issue is which one to choose and henceforth we require a methodology which tells us what to do at what time.

# What Methodology should provide?

- The methodology should provide guidance on when, where and what techniques to apply in order to achieve maximum efficiency.
- The methodology should provide basic building blocks, coding guidelines and advice on reusing verification components.
- It Should help in achieving better verification results as measured by engineer productivity, schedule predictability and profitability for the chip being verified.

# Overview of Methodologies

2002

**eRM (e-Reuse Methodology)** from **Verisity**.

Provided rules for building **reusable**, **consistent**, **extensible** and Plug and Play verification environments.

**RVM (Reuse Verification Methodology)** from **Synopsis** for **Vera** verification language.

It included base **classes**, **message** capabilities, packing guidelines. Overtime was converted to **VMM** for SV.

2003

2006

**AVM (Advanced Verification Methodology)** from **Mentor**.

It was first open source Verification solution.

Concept of **TLM ports** were adopted from SystemC to communicate between Verification Components.

# Overview of Methodologies

2007

**URM (Universal Reuse Methodology)** from **Cadence**.  
Modified eRM with added features such as **factory**,  
**configuration** and **class automation** for SV.

**OVM (Open Verification Methodology)** from **Cadence**  
and **Mentor**.

Provided multi-vendor verification solution. Integration  
of multi-language testbenches was also added.

2008

2010

**UVM (Universal Verification Methodology)** from **Mentor**,  
**Cadence** and **Synopsis**.

Based on OVM proven library industry wide verification  
methodology was created.

# Universal Verification Components

# UVM Test Bench

- A UVM testbench is composed of reusable UVM-compliant universal verification component (UVC).
- A UVC is an encapsulated, ready to use and configurable verification environment.
- UVM Test Benches are layered and structured test benches, where each structure has a specific role to play.



# Universal Verification Components

- A standard structure of UVC includes following elements:

- Data Items

- ✓ It represents stimulus transaction that are input to the DUT.
- ✓ In a test, many data items are generated and sent in to the DUT.
- ✓ Examples: Network Packets, Instructions.

- Driver (Bus Functional Model)

- ✓ It is an active entity which emulates logic that drives the DUT.
- ✓ A driver repeatedly pulls data items generated by sequencer and

# Universal Verification Components

## ○Sequencer

- ✓ It is an advance stimulus generator that generates and returns data items upon request from the driver.
- ✓ A sequencer has an ability to react to the current state of DUT and generate more useful data items by changing randomization weights.

## ○Monitor

- ✓ It is a passive entity that samples DUT signals but does not drive them.
- ✓ They are useful in collecting coverage and checking performance.
- ✓ It can also be used to predict result based on sampled inputs and verifying them against sampled outputs.

# Universal Verification Components

## ○Agent

- ✓ They are used to encapsulate driver, sequencer and monitor.
- ✓ An UVC can contain more than one agent.
- ✓ Different agents can be used to target different protocols in an SOC based example.
- ✓ Agents can be configured as active and passive.
- ✓ Active agents are one which emulates devices and drives DUT's ports.
- ✓ Passive agents only monitors DUT activity.

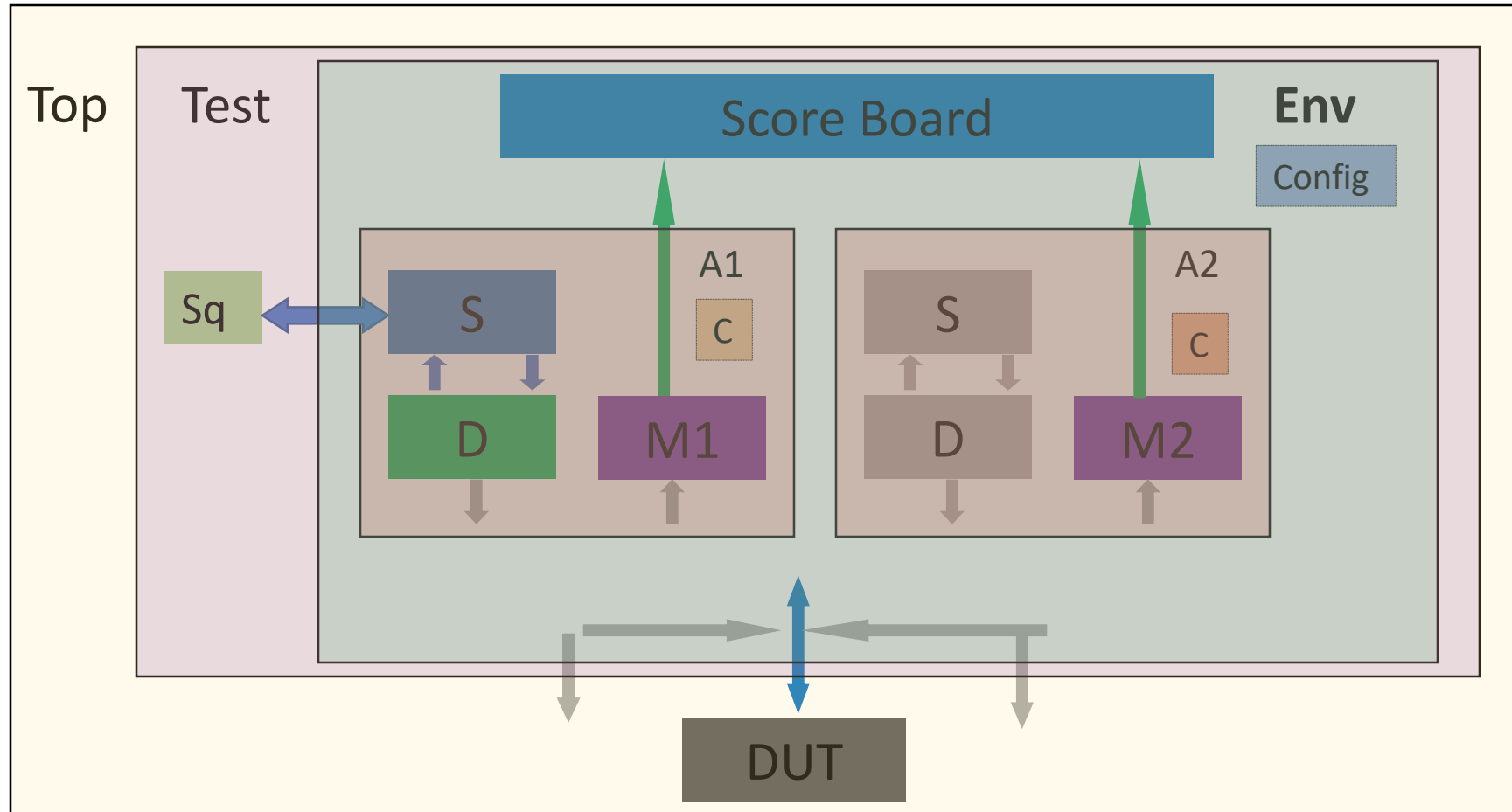
# Universal Verification Components

- Environment

- ✓ It is top-level component of an UVC.
- ✓ It can contain one or more agents.
- ✓ It can also contain Score Boards that are used to perform end-to-end transmission checks or perform checks against reference models.

# Test Bench Hierarchy

# Test Bench Hierarchy



# UVM Class Library

# UVM Library

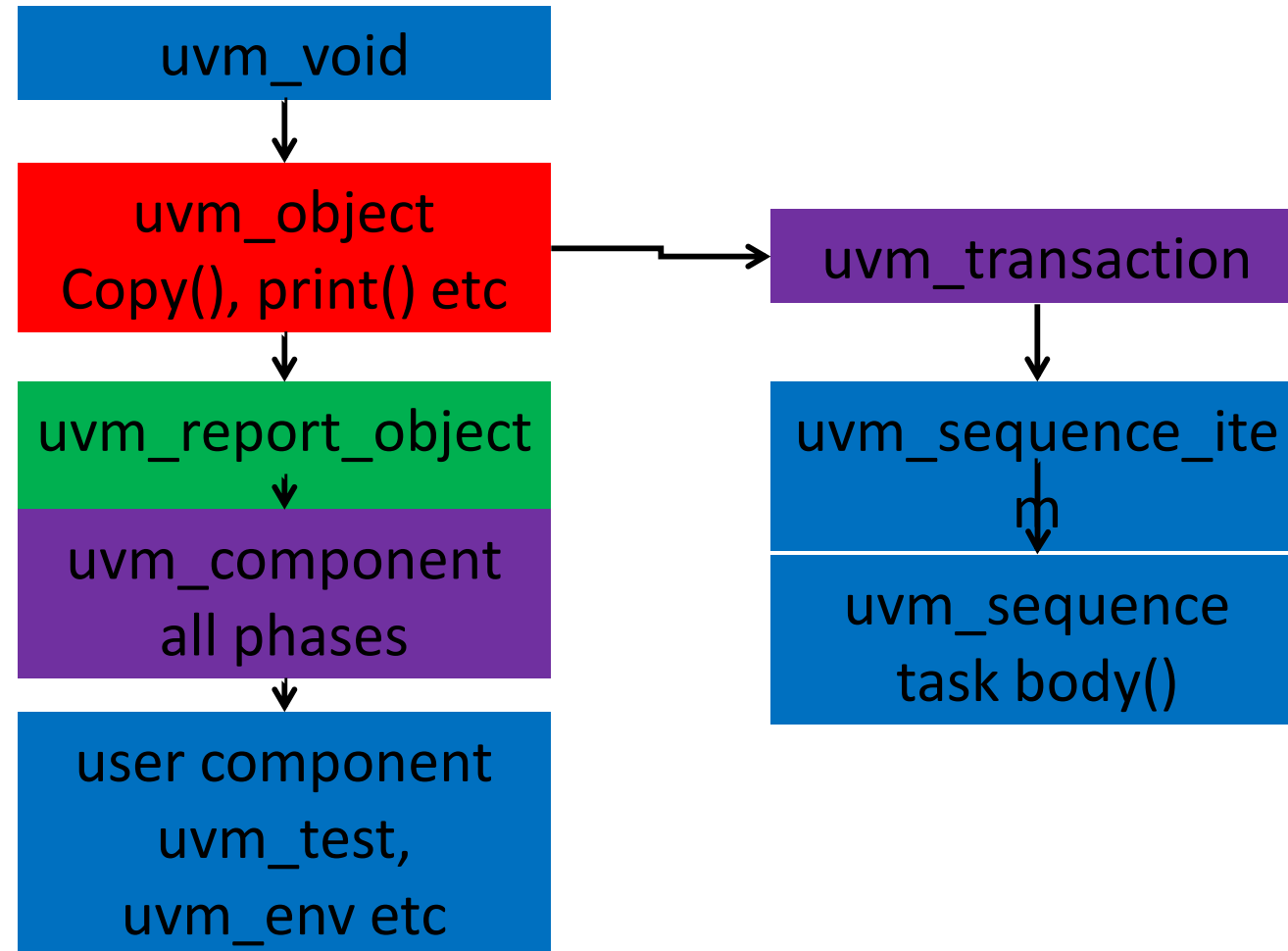
- The UVM library provides all the building blocks that we require to build modular, scalable, reusable, verification environments.
- This library contains base classes, utilities and macros to support the entire verification process.
- In order to use UVM Library, user needs to:
  - Compile `uvm_pkg.sv` file.
  - Import `uvm_pkg` into the desired scope.
  - Include file that contains uvm macros.



# Hello World

```
`include "uvm_pkg.sv"  
//include for compiling if not available by default  
import uvm_pkg :: *;  
//importing definitions inside uvm_pkg  
  
module hello_world;  
initial  
`uvm_info("info1", "Hello World", UVM_LOW);  
endmodule
```

# UVM Library : Base Classes



# Features of Base Classes

- **uvm\_void** class is the base class for all UVM classes. It is an abstract class with no data members or functions.
- **uvm\_object** class is the base class for all UVM data and hierarchical classes.
  - Defines set of methods for common operations such as create, copy, compare, print, etc.

# Features of Base Classes

- **uvm\_transaction** class is base class for all UVM Transactions.
  - It inherits all features of `uvm_object`.
  - Use of this class is depreciated and instead of using this class, it's subtype `uvm_sequence_item` should be used.
- **uvm\_report\_object** class provides an interface to UVM reporting facility.
  - It allows UVM Components to issue various messages during simulation time.

# Features of Base Classes

- **uvm\_component** is the base class for all UVM Components.
- Components are quasi-static objects that exists throughout the simulation.
- This class inherits features from both `uvm_object` and `uvm_report_object`.
- In addition it provides following features:
  - ✓ Hierarchy, searching and traversing component hierarchy.
  - ✓ Phasing, defines test flow of all the components.
  - ✓ Configuration, allows configuring component topology.
  - ✓ Factory, for creating and overriding components.

UVM Object

# uvm\_object methods

- Following are important methods defined inside uvm\_object class.

- **new()** create new uvm\_object for a given instance.

function new (string name=" ")

- **set\_name()** set/override instance name of this object.

virtual function void set\_name (string name)

set\_name() set/override instance name of this object

# uvm\_object methods

- **get\_full\_name()** returns full hierarchal name of this object

virtual function string get\_full\_name()



# uvm\_object methods

- **create()** allocates a new object as same type as this object and returns it via base uvm\_object\_handler.

```
virtual function uvm_object create(string name=" ")
```

```
class my_object extends uvm_object;
```

```
function uvm_object create (string name=" ");
```

```
my_object = new (name);
```

```
return my_object;
```

```
endfunction
```

```
.....
```

# uvm\_object methods

- **copy()** method makes this object a copy of other object.
- ✓ copying is same as deep copy.
- ✓ copy method should not be overridden in derived class.

function void copy (uvm\_object rhs)

- **clone()** method creates and returns the exact copy of this object.

virtual function uvm\_object clone()

# uvm\_object methods

- **print()** method performs deep print of object properties in a format governed by its argument.

# Example1

```
typedef enum bit [1:0] {READ, WRITE, RD_WR, NOP} direction;
```

```
class mem_transaction extends uvm_object;
```

```
    rand bit [7:0] data;    //data
```

```
    rand bit [3:0] addr;    //address
```

```
    rand direction dir;    //direction
```

```
    //Control field (Knob)
```

```
    rand bit [3:0] delay;    //delay between transactions
```

```
    ..... //on Next slide
```

```
endclass
```

# Example1

```
//Registering user class to UVM Factory and including given
//field in implementation of print, copy, etc
`uvm_object_utils_begin(mem_transaction)
    `uvm_field_int(data, UVM_DEFAULT)
    `uvm_field_int(addr, UVM_DEFAULT)
    `uvm_field_enum(direction, dir, UVM_DEFAULT)
    `uvm_field_int(delay, UVM_DEFAULT | UVM_NOCOMPARE)
`uvm_object_utils_end

//Class constructor
function new(string name="mem_transaction");
super.new(name);
endfunction
```

## Example2

```
module top;
import uvm_pkg:: *;           //importing UVM Library
`include "uvm_macros.svh"      //including UVM Macros
`include "mem_transaction.sv"  //including user-defined
class

mem_transaction a1, a2, a3;

initial begin
a1=mem_transaction :: type_id :: create("a1");
//creating object using factory create method, which supports
//overriding, using new will restrict overriding
..... //On Next slide
endmodule
```

## Example2

```
assert(a1.randomize()) else
`uvm_fatal("RFAIL", "Randomization Failed")
a2=mem_transaction :: type_id :: create("a2");
a2.copy(a1);           //copy a1 to a2
$cast(a3, a1.clone()); //create new object of type a1
                        //and then copy to a3
if(!a3.compare(a2))
`uvm_error("CFAIL", "Comparison Failed")
a1.print();
a2.print(default_line_printer);
a3.print(default_line_printer);
end
```

# Result

Table Printer :

```
# -----  
# Name    Type      Size Value  
# -----  
# a1      mem_transaction -  @457  
# data    integral    8  'h50  
# addr    integral    4  'he  
# dir     direction    2  READ  
#delay    integral    4  'hb  
# -----
```



# Result

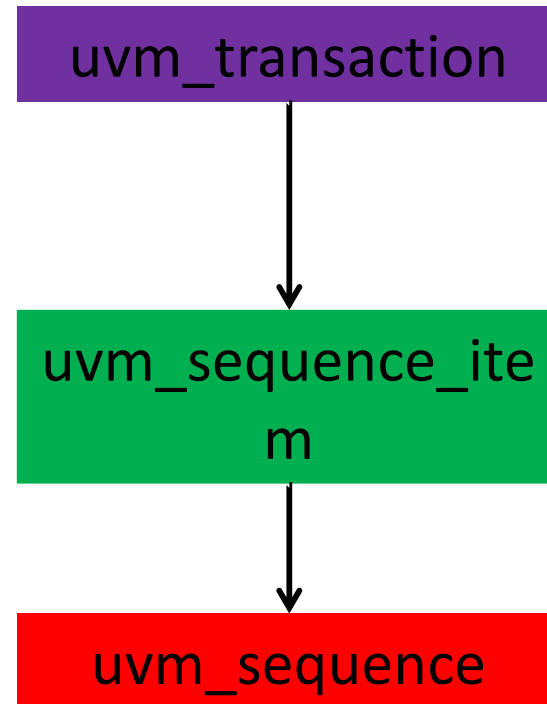
Tree Printer :

```
# a2: (mem_transaction@458) {  
#  data: 'h50  
#  addr: 'he  
#  dir: READ  
#  delay: 'hb  
# }
```

Line Printer :

```
# a1: (mem_transaction@459) { data: 'h50  addr: 'he  dir: READ  
delay: 'hb }
```

# uvm\_transaction



# uvm\_sequence methods

- uvm\_sequence class provides the interfaces to create streams of sequence items or other sequences.
- **start()** executes the given sequence on a sequencer and returns once sequence is over.
- **pre\_body()** is user-defined callback which is called before execution of body only if sequence is started using start.
- **body()** is user-defined task where the main sequence code resides.
- **post\_body()** is user-defined callback which is called after execution of body only if sequence is started using start.

# uvm\_sequence methods

```
virtual task start ( uvm_sequencer_base sequencer,  
                   uvm_sequence_base parent_sequence =  
null,  
                   int this_priority = -1,  
                   bit call_pre_post = 1 )
```

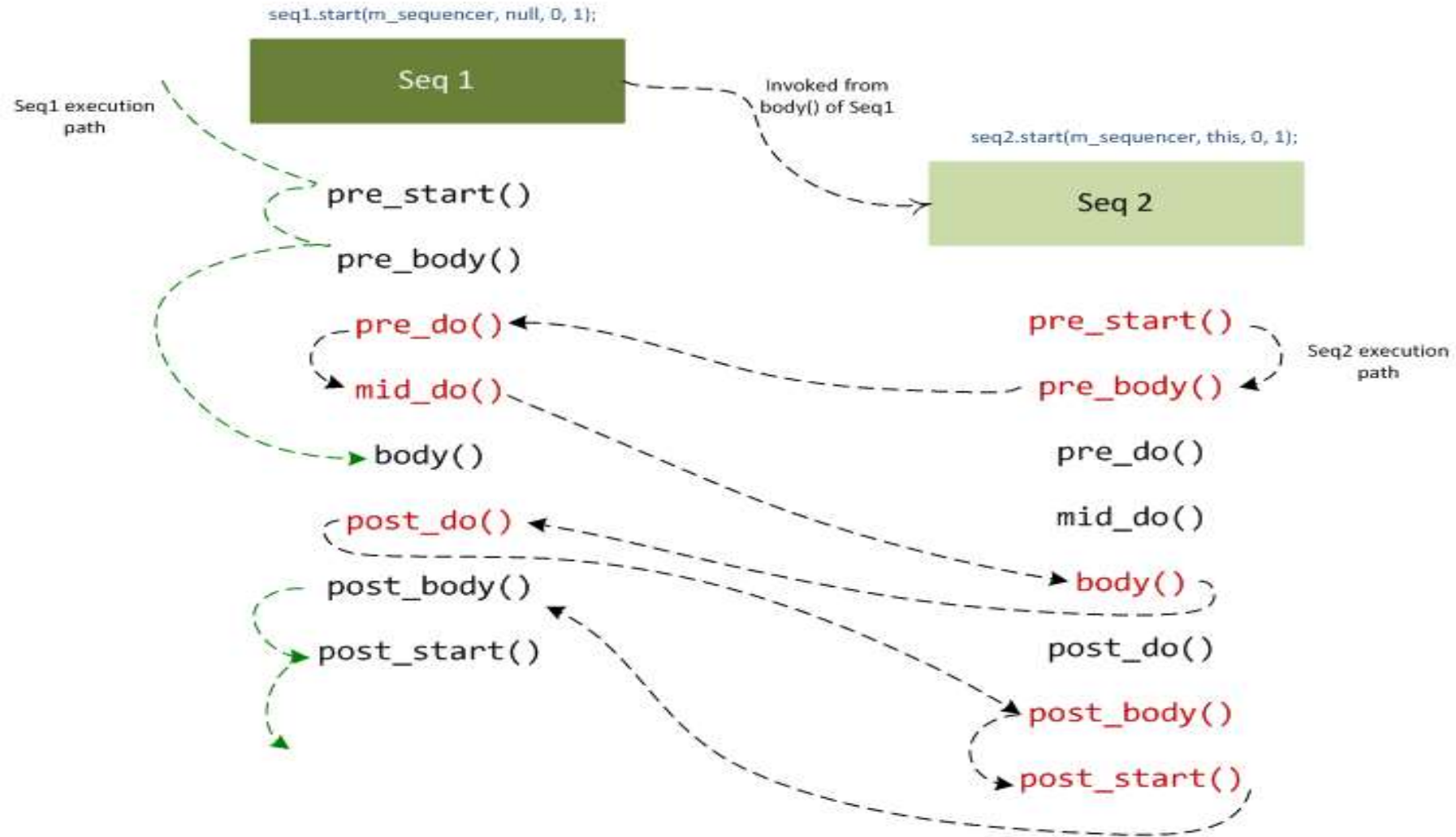
✓sequencer: specifies the sequencer on which the given sequence runs.

# Seq.start

```
seq.randomize (...); // optional
seq.start (m_sequencer, null, , 1);

// The following methods will be called in start()
seq.pre_start();           (task)
seq.pre_body();            (task)  if call_pre_post == 1
    parent_seq.pre_do()    (task)  if parent_seq != null
    parent_seq.mid_do(this) (func)  if parent_seq != null
seq.body()                 (task)  your code
    parent_seq.post_do(this) (func)  if parent_seq != null
seq.post_body()            (task)  if call_pre_post == 1
sub_seq.post_start()       (task)
```

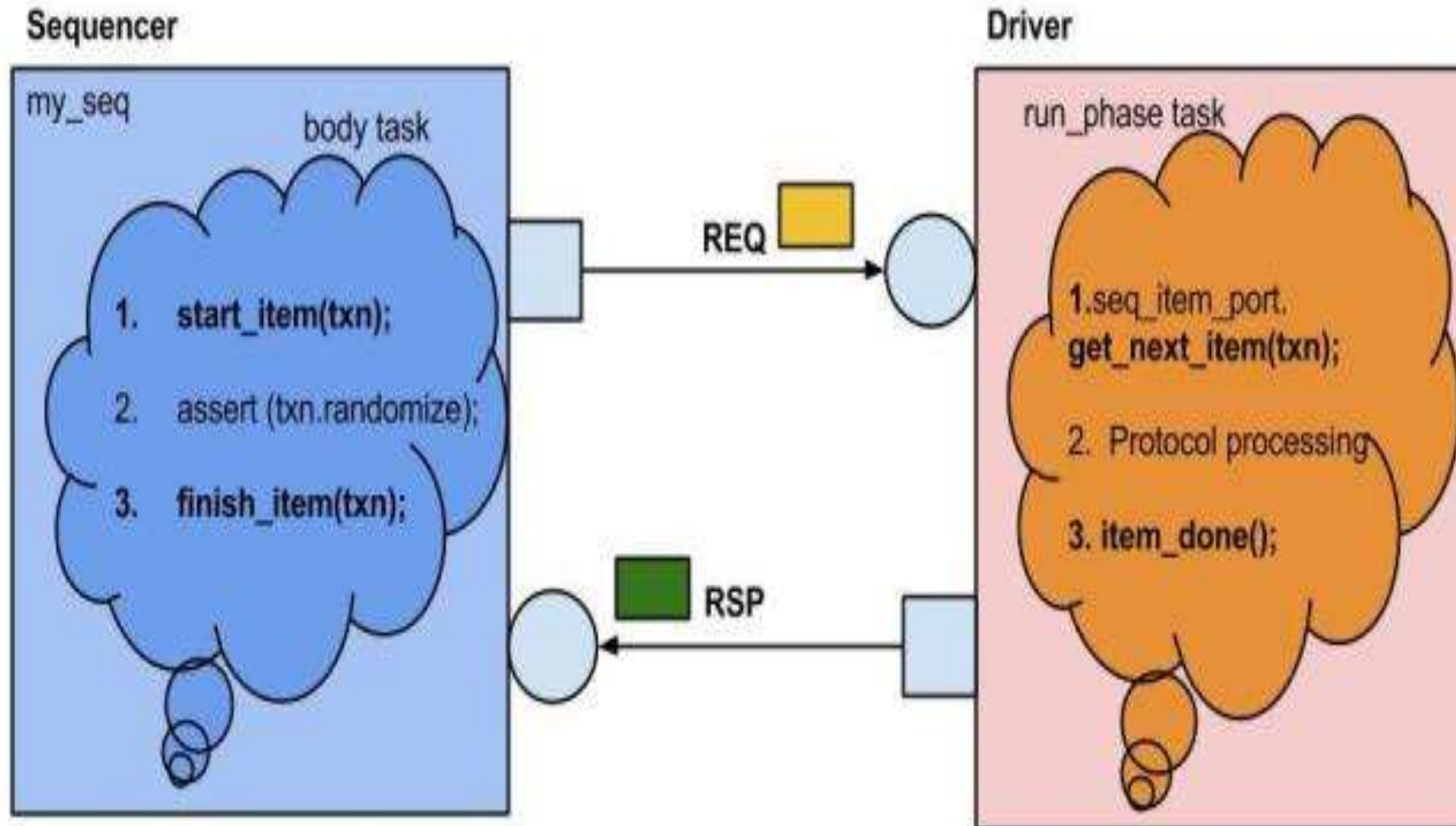
# How a seq starts



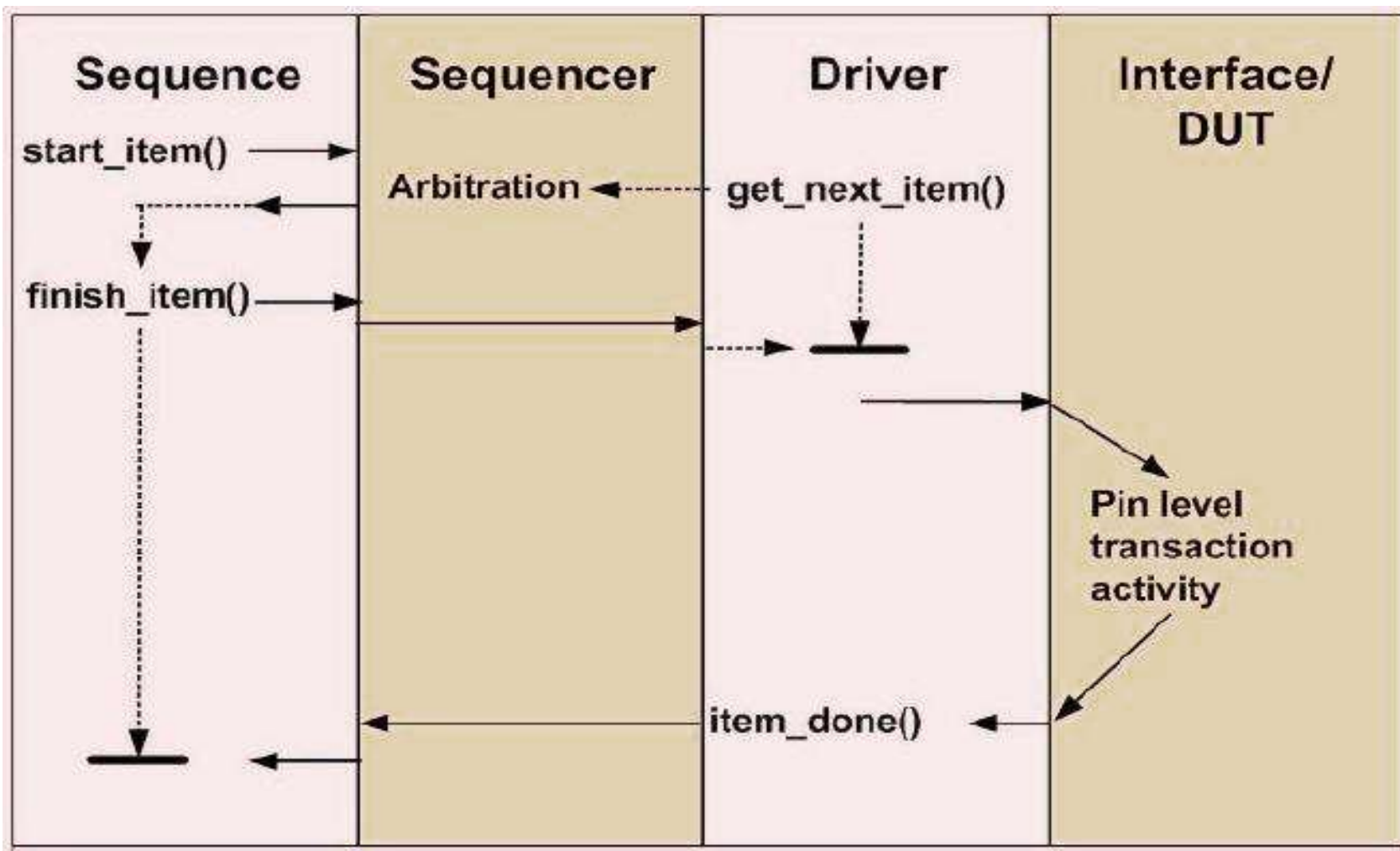
# uvm\_sequence methods

- **create\_item()** will create and initialize a sequence or sequence item using the factory.
- **start\_item()** and **finish\_item()** are used together to initiate operation of a sequence item.
- Body of a sequence contains following in given order:
  - ✓ start\_item()
  - ✓ create sequence item using factory.
  - ✓ Randomize sequence item (useful for late randomization)
  - ✓ finish\_item()

# Sequencer and Driver comm







### Sequencer side operation:

1. Creating the "transaction item" with the declared handle using factory mechanism.
2. Calling "start item(<transaction item handle>)". This call blocks the Sequencer till it grants the Sequence and transaction access to the Driver.
3. Randomizing the transaction OR randomizing the transaction with in-line constraints. Now the transaction is ready to be used by the Driver.
4. Calling "finish item(<transaction item handle>)". This call which is blocking in nature waits till Driver transfer the protocol related transaction data.

### **Driver side operation:**

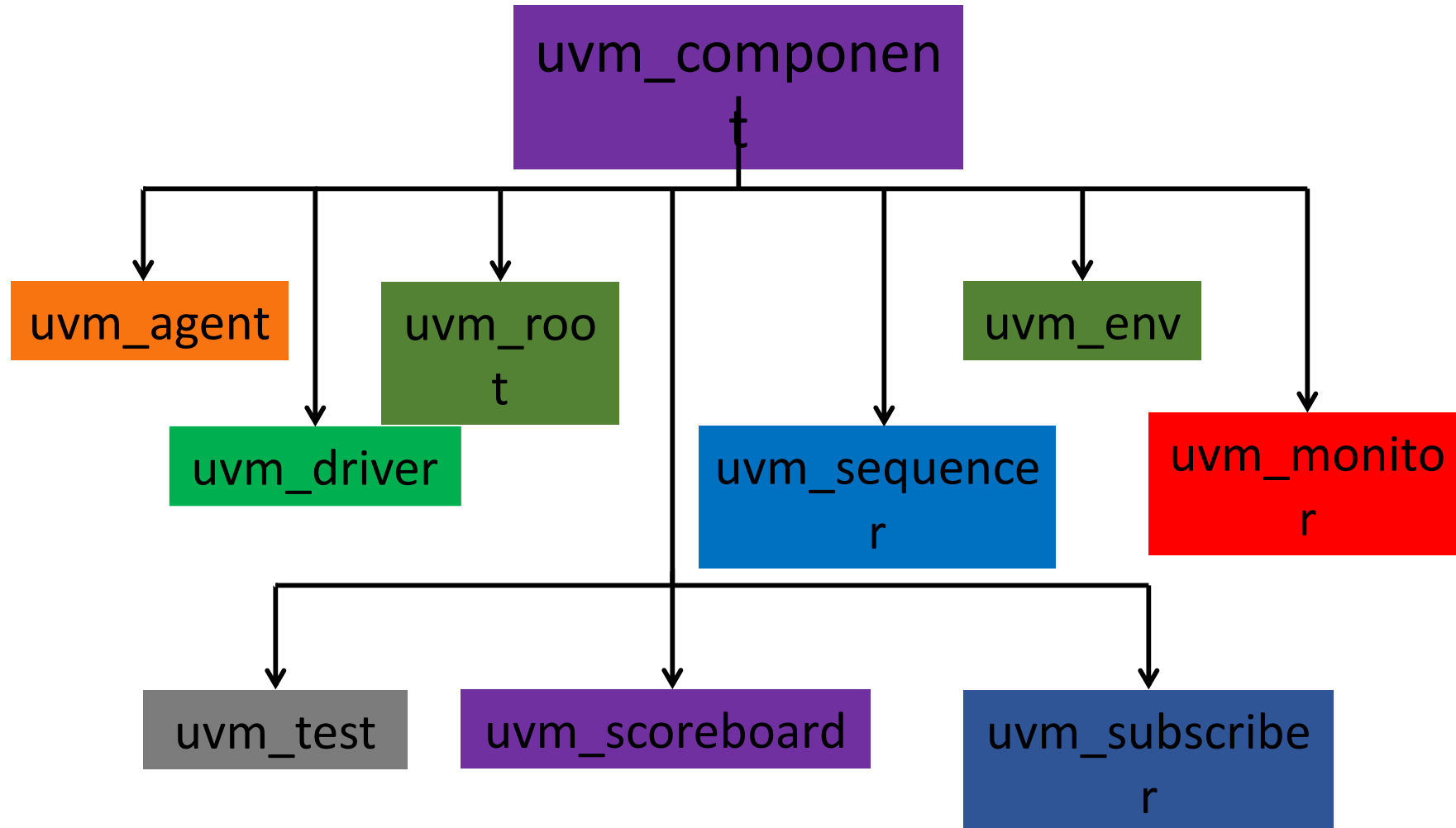
1. Declaring the "*transaction item*" with a handle.
2. Calling the "*get\_next\_item(<transaction item handle>)*". Default transaction\_handle is "*req*". "get\_next\_item()" blocks the processing until the "req" transaction object is available in the sequencer request FIFO & later "get\_next\_item" returns with the pointer of the "req" object.
3. Next, *Driver completes its side protocol transfer* while working with the virtual interface.
4. Calling the "*item\_done()*" OR "*item\_done(rsp)*". It indicates to the sequencer the completion of the process. "item\_done" is a non-blocking call & can be processed with an argument or without an argument. If a response is expected by the Sequencer/Sequence then item\_done(rsp) is called. It results in Sequencer response FIFO is updated with the "rsp" object handle.

# UVM Component

# uvm\_component

- All the infrastructure components in a UVM verification environment (like monitor, driver, agent, etc.) are derived from uvm\_component.
- This component class is quasi-static in nature and object creation is not allowed after build\_phase.
- Key functionality provided by uvm\_component class are:
  - Phasing and Execution control.
  - Hierarchy information function.
  - Configuration methods.
  - Factory convenience methods.
  - Hierarchical reporting control.

# uvm\_component



# Example1

```
class my_component extends uvm_component;

//Registering user-defined class to UVM Factory
`uvm_component_utils (my_component)

//Constructor for UVM Components
function new (string name, uvm_component parent);
super.new(name, parent);
endfunction

//some additional properties and methods
endclass
```

# uvm\_root and uvm\_top

- The uvm\_root class serves as the implicit top-level and phase controller for all UVM components.
- UVM automatically creates a single instance of uvm\_root called uvm\_top.
- uvm\_top manages phasing for all UVM components.
- uvm\_top can be used to globally configure report verbosity and as a global reporter.



# Features of uvm\_root

- Following are few methods and variables defined inside uvm\_root.
  - **run\_test()** phases all the components through all the registered phases.
  - ✓ Test name can be passed as an argument or can be part of command line argument +UVM\_TESTNAME=TEST\_NAME

virtual task run\_test (string test\_name = "")

- **top\_levels** variable contains the list of all top level components in UVM. It includes uvm\_test\_top variable that is created by run\_test.

# Features of uvm\_root

- **print\_topology()** prints the verification environment's component topology.

```
function void print_topology (uvm_printer printer = null)
```

UVM Factory

# UVM Factory

- `uvm_factory` (UVM Factory) is a class which is used to create UVM objects and components .
- Only one instance of UVM Factory can exist during entire simulation (singleton).
- User defined objects and components can be registered with the factory via typedef or macros invocation.

# Registering with Factory

- Non-Parameterized Objects:

```
class packet extends uvm_object;  
  `uvm_object_utils(packet)  
endclass
```

- Parameterized Objects:

```
class packet #(type T=int, int Width=32) extends  
  uvm_object;  
  `uvm_object_param_utils(packet #(T, Width))  
endclass
```

# Registering with Factory

- Non-Parameterized Components:

```
class comp extends uvm_component;  
  `uvm_component_utils(comp)  
endclass
```

- Parameterized Components:

```
class comp #(type T=int, int Width=32) extends  
  uvm_component;  
  `uvm_component_param_utils(comp #(T, Width))  
endclass
```

# UVM Field Macros

- ``uvm_field` macro includes the given field in implementation of `print()`, `copy()`, `clone()`, `pack()`, `unpack()`, `compare()` and `record()` methods for an object.

``uvm_field_* (field_name, flags)`

- `field_name` must be an existing property identifier of the class.
- `flags` specifies the automation required for a field.
- Flags are numeric values and can be combined using bitwise OR or `+` operator.

# UVM Field Macros

Macros	Implements operations for
<code>`uvm_field_int</code>	any packed integral property
<code>`uvm_field_string</code>	a string property
<code>`uvm_field_enum</code>	an enumerated property
<code>`uvm_field_real</code>	any real property
<code>`uvm_field_object</code>	an uvm_object-based property
<code>`uvm_field_event</code>	an event property



# UVM Field Macros

Macros	Implements operations for
<code>`uvm_field_array_int()</code> <code>`uvm_field_array_object()</code> <code>`uvm_field_array_string()</code>	Dynamic and Static Arrays
<code>`uvm_field_queue_int()</code> <code>`uvm_field_queue_object()</code> <code>`uvm_field_queue_string()</code>	Queues

# Flag Arguments

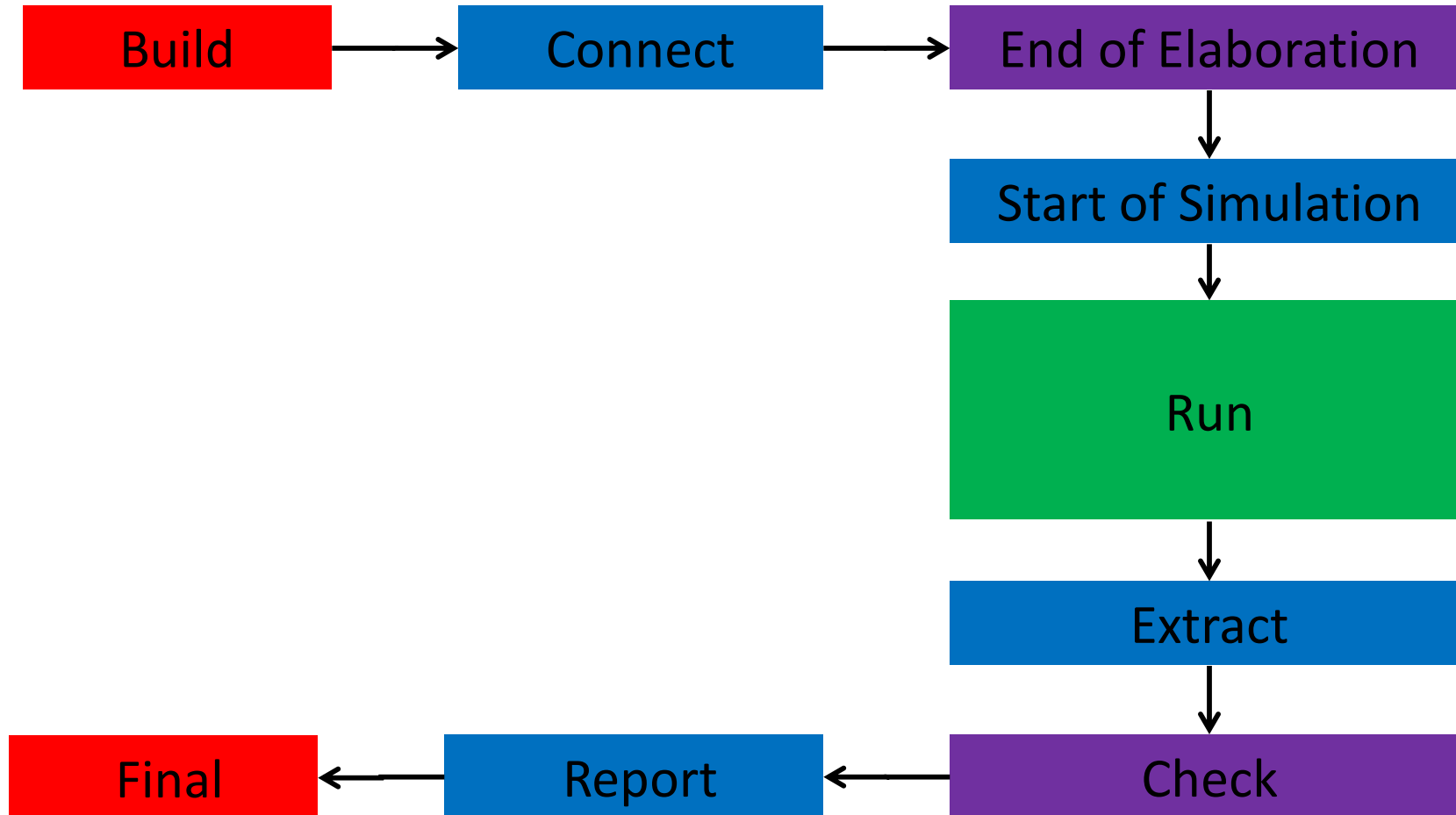
Flags	Description
UVM_ALL_ON	Set all operations on (default)
UVM_DEFAULT	Use the default flag settings
UVM_NOCOPY	Do not copy this field
UVM_NOCOMPARE	Do not compare this field
UVM_NOPRINT	Do not print this field
UVM_NODEFPRINT	Do not print the field if it does not change
UVM_NOPACK	Do not pack or unpack this field

# UVM Phases

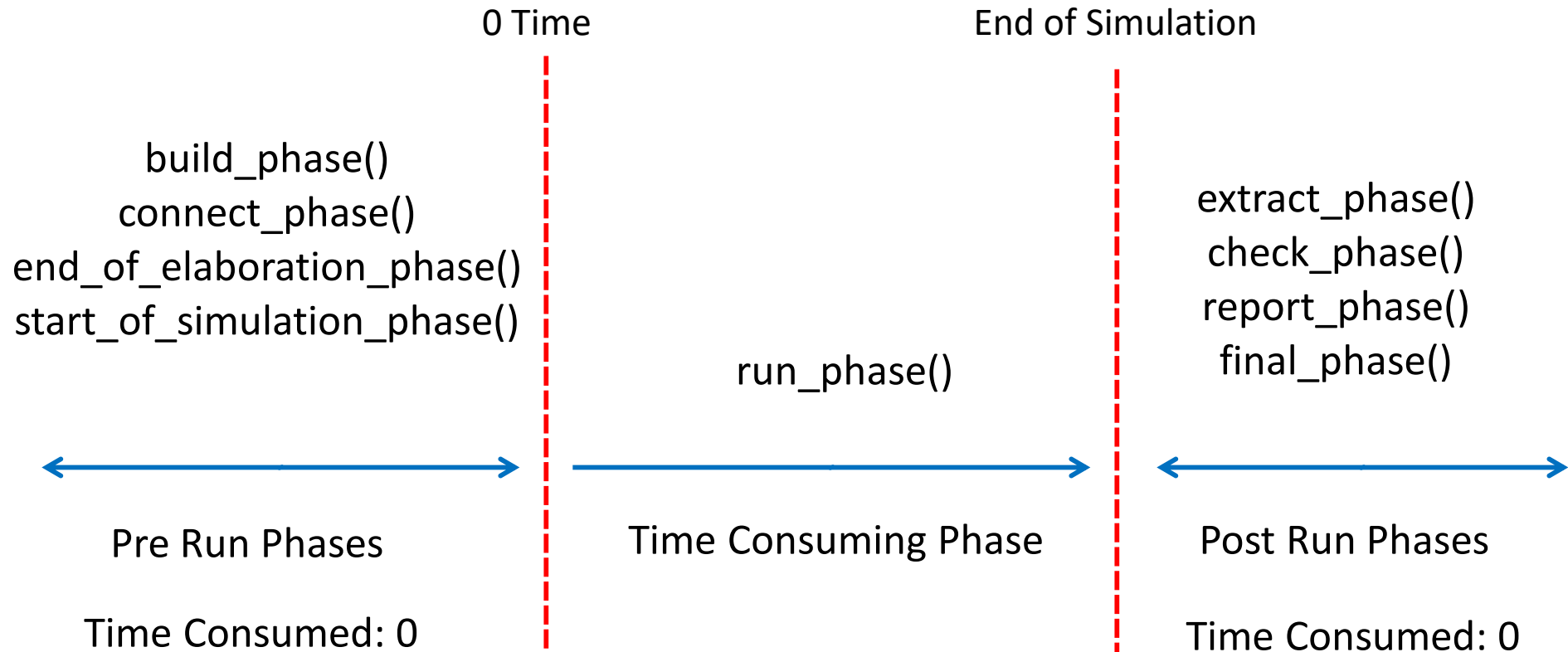
# Why we require Phases

- In Verilog and VHDL, static elaboration of instances occur before simulation starts.
- This ensures that all instances are placed and connected properly before simulation starts.
- In System Verilog classes are instantiated at run-time.
- This leads to questions like what is the good time to start data generation and transfer, assuming all UVC components have been created and connected properly.
- UVM Provides concept of Phases to solve this issue. Each phase has

# Phases in UVM



# Execution of Phases



# build\_phase

- build\_phase is called for all the uvm\_components in a Top-down fashion.
- This method is used to optionally configure and then create child components for a given components.
- If components are created after build phase then UVM will generate error message.
- Configuration settings for descendant components should be set before calling create methods.

# Example

```
class myagent extends uvm_agent;
    mydriver dr;
    ..... //component registration and constructor

    function void build_phase (uvm_phase phase);
        super.build_phase(phase);
        dr=mydriver::type_id::create("driver", this);
    endfunction

endclass
```



# connect\_phase

- Connect phase executes in Bottom-Up fashion.
- This phase is used to specify connection between UVC's.
- Use connect\_phase method to make TLM Connections, assign pointer references, and make virtual interface assignments.
- Components are connected using TLM Ports.

# Example

```
class myagent extends uvm_agent;
    mydriver dr;
    mysequencer sq;
    ..... //component registration and constructor

    function void connect_phase (uvm_phase phase);
        super.connect_phase(phase);
        dr.seq_item_port.connect(sq.seq_item_export);
    endfunction
```

# end\_of\_elaboration\_phase()

- This phase can be used to examine whether all connections and references defined in connect phase are proper and complete.
- This phase executes in Bottom-Up fashion.

```
function void end_of_elaboration_phase(uvm_phase phase);  
  
    super.end_of_elaboration_phase(phase);  
  
    uvm_top.print_topology();  
  
endfunction
```

# start\_of\_simulation\_phase()

- This phase can be used for displaying banners, test bench topology or configuration messages.
- This phase executes in Bottom-Up fashion.

```
function void start_of_simulation_phase(uvm_phase phase);  
    super.start_of_simulation_phase(phase);  
    `uvm_info("info", "start of simulation", UVM_LOW)  
endfunction
```

# run\_phase()

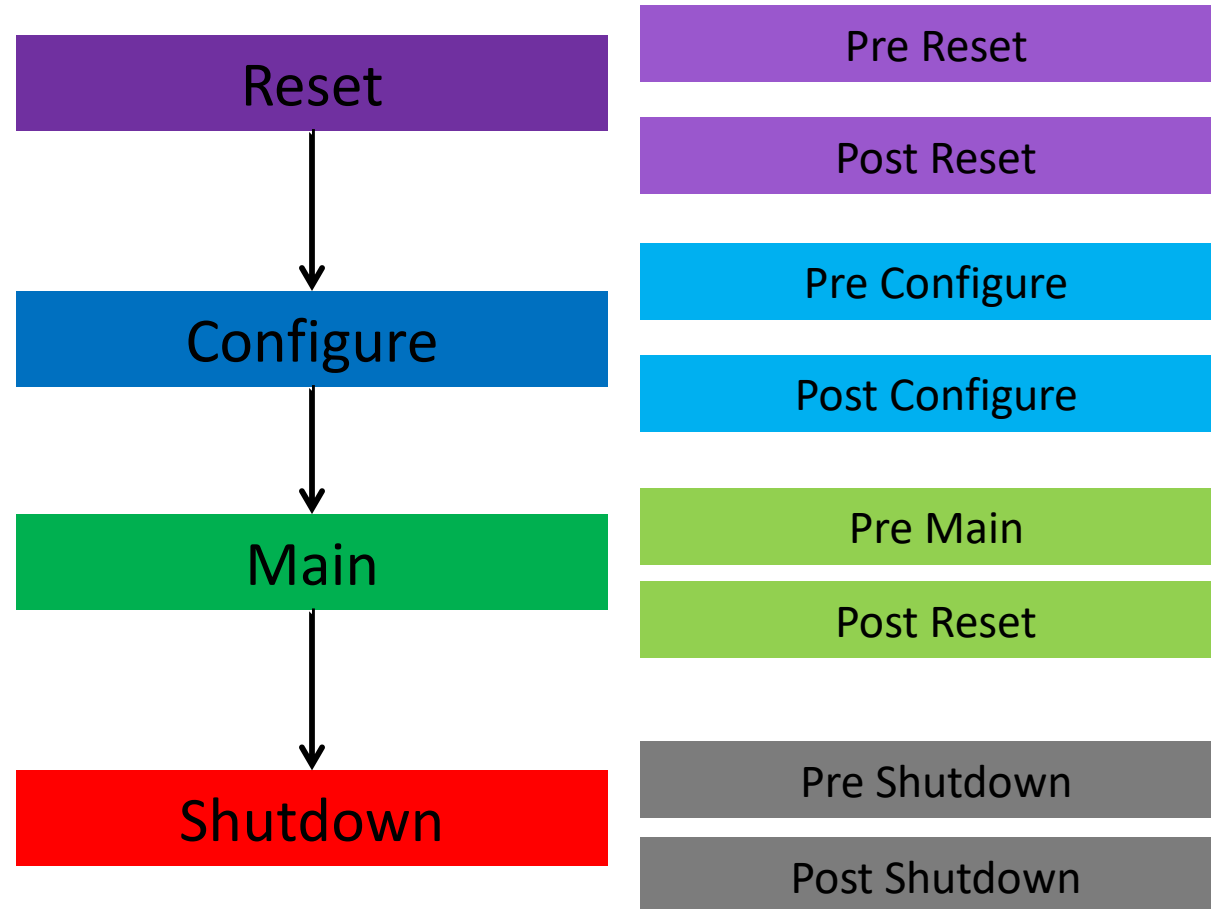
- run\_phase is a time consuming task.
- All run phases are executed in parallel.
- Run phase is used inside driver , monitor and test.
- Run phase is further divided in 12 sub-phases.

# Example

```
class driver extends uvm_driver;
    ..... //component registration and constructor

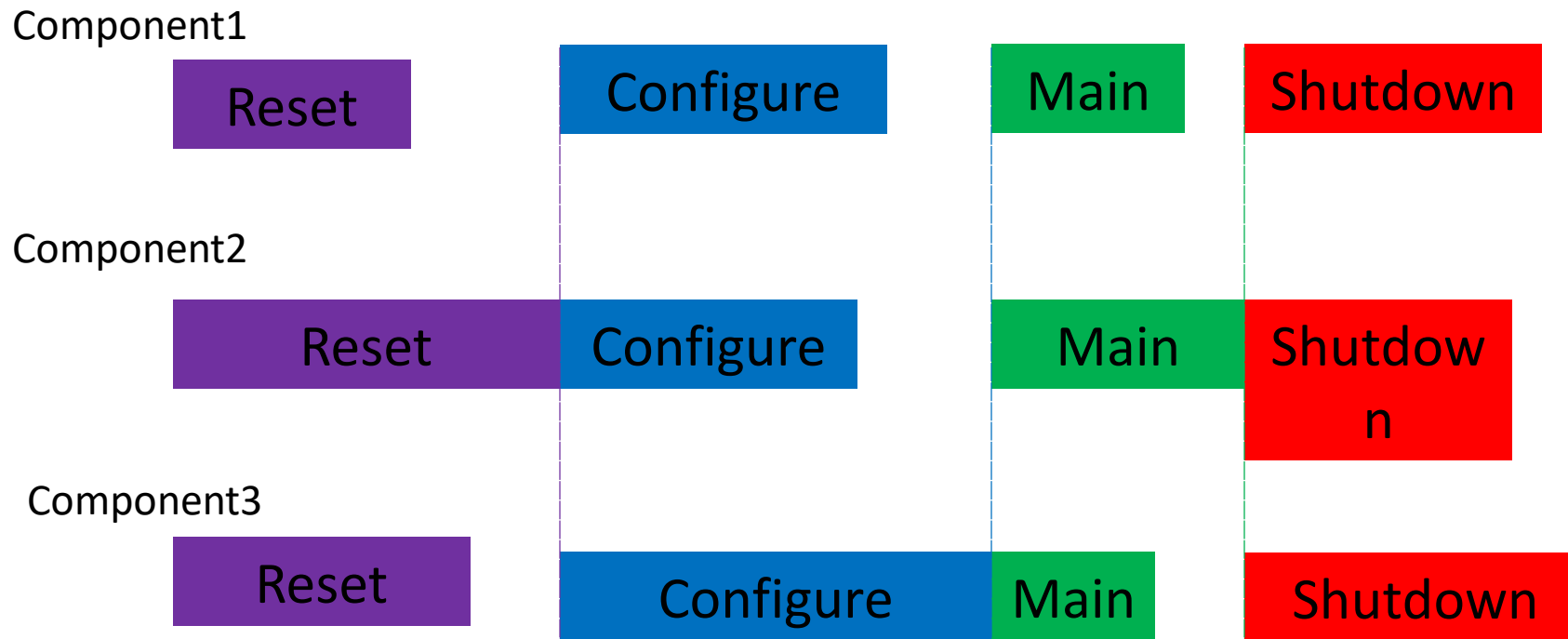
    task run_phase (uvm_phase phase);
        super.run_phase(phase);
    repeat(10) begin
        @(posedge vif.clk) vif.data<=$random;
    end
endtask
```

# run phase – sub phases



# Phase Synchronization

- All components allow other components to complete a phase before they proceed to next phase.





# Post run Phases

- `extract_phase()` is used to retrieve and process information from scoreboards and functional coverage monitors. This phase is executed in Bottom-Up fashion.
- `check_phase()` checks if the DUT behaved correctly and identifies errors that may have occurred during the execution. This phase is executed in Bottom-Up fashion.
- `report_phase()` displays the result of the simulation. This phase is executed in Bottom-Up fashion.
- `final_phase()` completes any other outstanding actions. This phase is executed in Top-Down fashion.

END OF TEST

# Objection

- Components can raise and drop objections.
- A component that raises objection remains in same phase till all the objections are dropped.
- Raising and dropping of objection is done in run phases.
- Usage:
  - `phase.raise_objection(this);` //to raise objection in this object
  - `phase.drop_objection(this);` //to drop objection in this object

# Example

```
class agent extends uvm_agent;
task run_phase(uvm_phase
phase);
phase.raise_objection(this);
#100; //do some work
phase.drop_objection(this);
endtask
endclass
```

```
class driver extends uvm_driver;
task run_phase(uvm_phase
phase);
phase.raise_objection(this);
#10;
phase.drop_objection(this);
endtask
endclass
```



# REPORTING MECHANISM

# Messaging and Reporting

- `uvm_report_object` class provides set of methods that can be used to report messages in order to facilitate debugging.

- Severity of messages could be

- Info

- Warning

- Error

- Fatal

- User can filter out messages depending upon their importance.

# Reporting Methods

```
virtual function void uvm_report_info ( string id,  
                                     string message,  
  
int verbosity=UVM_MEDIUM,  
  
                                     string filename="",  
  
                                     int line=0);
```

- id is user defined string id given to message.
- message is user defined message.

# Reporting Methods

```
virtual function void uvm_report_warning ( string id,  
                                          string message,  
                                          int verbosity=UVM_MEDIUM,  
                                          string filename="",  
                                          int line=0);
```

```
virtual function void uvm_report_error ( string id,
```



# Reporting Methods

```
virtual function void uvm_report_fatal ( string id,  
  
                                         string message,  
  
int verbosity=UVM_NONE,  
  
                                         string filename="",  
  
                                         int line=0);
```

- `\_\_FILE\_\_ and `\_\_LINE\_\_ are predefined macros to get file name and line number.

# UVM Macros

``uvm_info (id, message, verbosity)`

``uvm_warning (id, message)`

``uvm_error (id, message)`

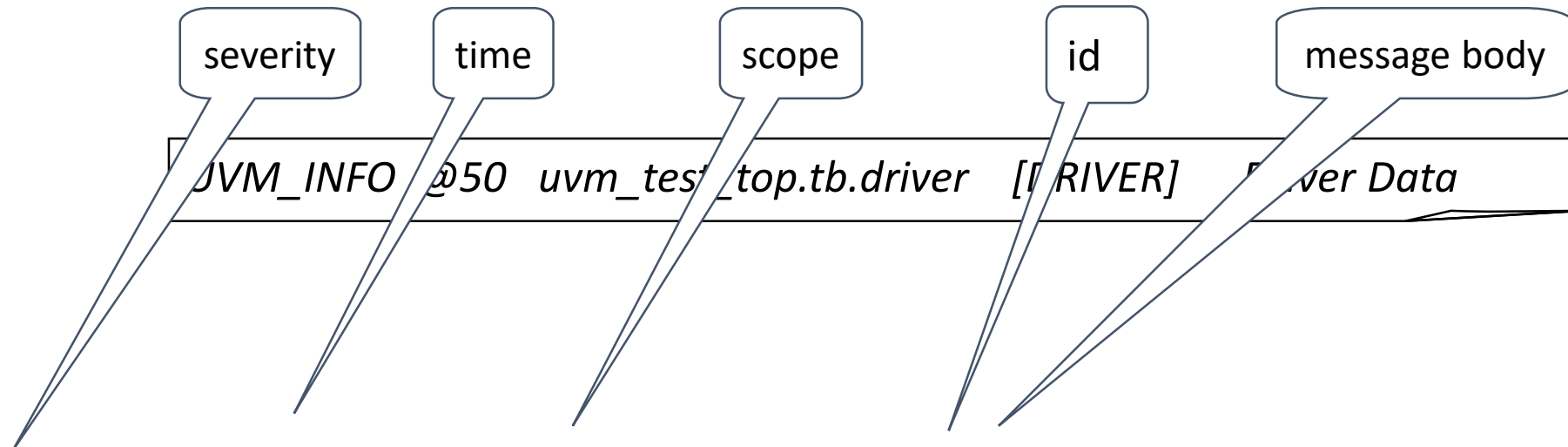
``uvm_fatal (id, message)`

- UVM\_NONE is verbosity for warning, error and fatal.
- UVM recommends usage of UVM Macros over UVM reporting methods.

# Example

```
`uvm_info ( "DRIVER", "Driver Data", UVM_MEDIUM);
```

Output:



# Verbosity Settings

- The verbosity level is an integral value to indicate the relative importance of the message.
- If verbosity value is smaller or equal to current verbosity level, then report is issued.
- For example if current verbosity level is set to 100 then messages with verbosity 101 or higher won't be printed.
- An enumerated type verbosity provides some standard verbosity levels UVM\_NONE=0, UVM\_LOW=100, UVM\_MEDIUM=200, UVM\_HIGH=300, UVM\_FULL=400, UVM\_DEBUG=500.

# Example

```
`uvm_info ( "1", "Verbosity MEDIUM", UVM_MEDIUM);  
`uvm_info ( "2", "Verbosity LOW", UVM_LOW);  
`uvm_info ( "3", "Verbosity NONE", UVM_NONE);
```

```
myobject a;  
initial begin  
  a=new("a");  
  a.set_report_verbosity_level (UVM_LOW); //set verbosity level  
end
```

Output:




```
UVM_INFO @0 : [2] Verbosity LOW  
UVM_INFO @0 : [3] Verbosity NONE
```

# TLM Ports

# TLM

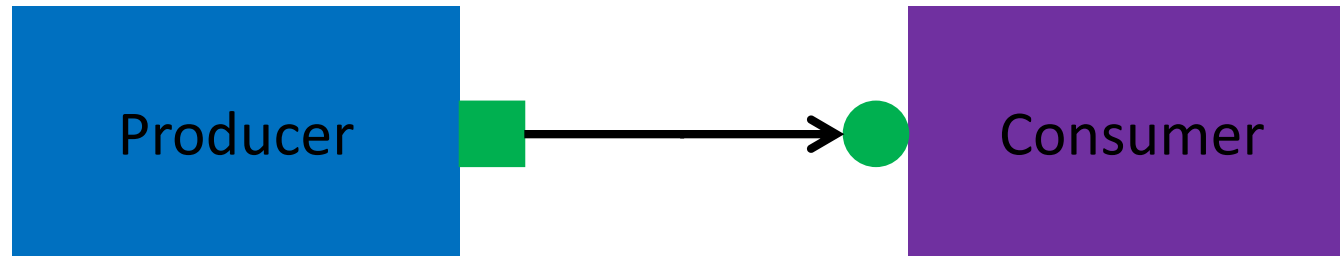
- TLM stands for Transaction Level Modeling.
- In UVM transaction is a class that extends from `uvm_sequence_item`.
- A transaction typically contains sufficient data fields to enable driver or transactor to create the actual signal-level activity.
- A transaction may contain additional data fields to control how randomization occurs.

# TLM

- TLM API provides set of methods that are used to communicate transaction between components.
- There are three basic types of TLM Objects:
  - Port: A port object specifies set of methods that can be called like get(), put(), peek(), etc. (Symbol : )
  - Export: Export port are used to forward implementation of a method. (Symbol : )
  - Imp: Imp port provides implementation of these methods. (Symbol : )
- Ports and Exports/Imp are connected together via connect() method where verification environment are constructed.



# Blocking PUT port



- Producer calls the put method and sends transaction (object) to TLM port.
- Consumers implements put method, receives transaction using TLM imp.
- Component that initiates transaction uses TLM port.

# Example - Transaction

```
class packet extends uvm_sequence_item;
rand bit [3:0] src_addr, dst_addr;
rand bit [7:0] data;
constraint addr_constraint {src_addr != dst_addr;}
`uvm_object_utils_begin(packet)
    `uvm_field_int(src_addr, UVM_DEFAULT)
    `uvm_field_int(dst_addr, UVM_DEFAULT)
    `uvm_field_int(data, UVM_DEFAULT)
`uvm_object_utils_end
//constructor not mentioned here
endclass
```

# Example – Producer (Put)

```
class producer extends uvm_component;
  uvm_blocking_put_port #(packet) put_port; //defining put port

  packet p;

  `uvm_component_utils(producer)

  function new (string name, uvm_component parent);
    super.new(name, parent);
    put_port= new("put_port", this);
  endfunction
```

## Example – Producer (Put)

```
task run_phase (uvm_phase phase);  
  repeat(10)  
  begin  
    p=new();  
    p.randomize;  
    put_port.put(p);  
  end  
endtask  
  
endclass
```

## Example – Consumer (Put)

```
class consumer extends uvm_component;
  uvm_blocking_put_imp #(packet, consumer) put_export;
  //defining imp port

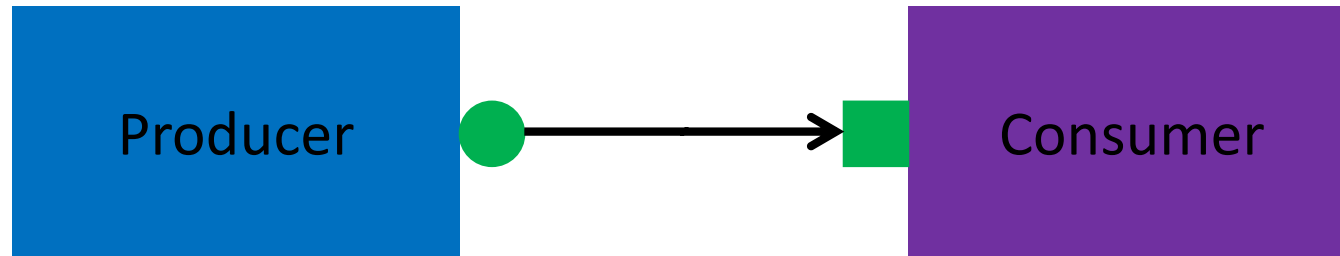
  `uvm_component_utils(consumer)

  function new (string name, uvm_component parent);
    super.new(name, parent);
    put_export= new("put_export", this);
  endfunction
```

## Example – Consumer (Put)

```
task put (packet p);  
  `uvm_info("packet", "packet received", UVM_LOW)  
  p.print();  
endtask  
  
endclass
```

# Blocking GET port



- Consumer calls the get method and receives transaction using TLM port.
- Producer implements get method and sends generated transaction using TLM imp.
- Component that initiates transaction uses TLM port.

## Example – Producer (Get)

```
class producer extends uvm_component;
  uvm_blocking_get_imp #(packet, producer) get_export;
  //defining get imp

  `uvm_component_utils(producer)

  function new (string name, uvm_component parent);
    super.new(name, parent);
    get_export=new("get_export", this);
  endfunction
```



# Example – Producer (Get)

```
task get (output packet p);  
packet temp;  
temp=new();  
temp.randomize;  
p=temp;  
endtask  
  
endclass
```

## Example – Consumer (Get)

```
class consumer extends uvm_component;
  uvm_blocking_get_port #(packet) get_port; //defining get port

`uvm_component_utils(consumer)

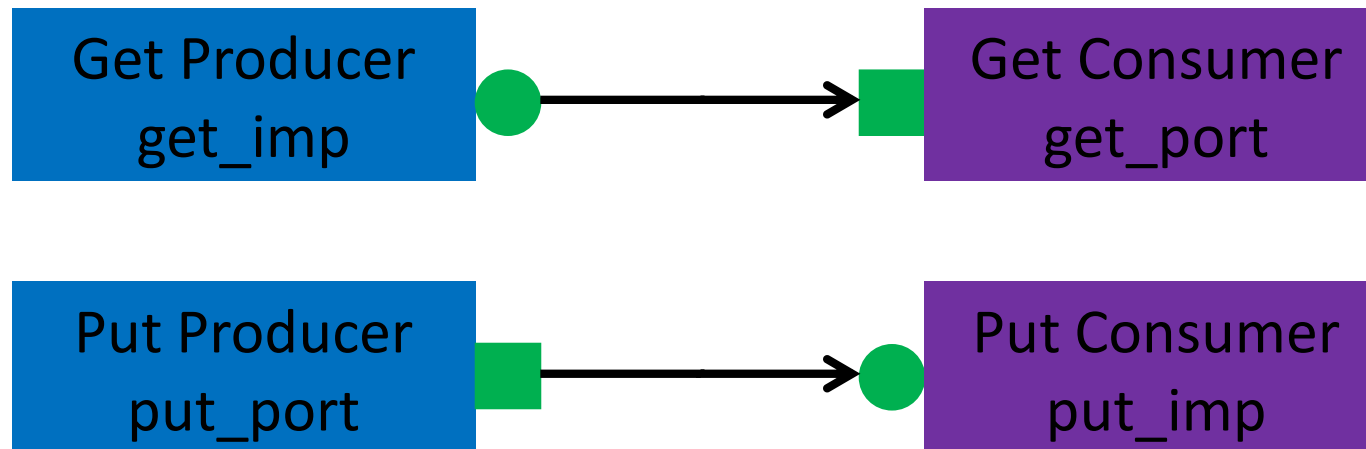
function new (string name, uvm_component parent);
  super.new(name, parent);
  get_port=new("get_port", this);
endfunction
```

## Example – Consumer (Get)

```
task run_phase (uvm_phase phase);  
  repeat(10)  
  begin  
    packet p;  
    get_port.get(p);  
    p.print;  
  end  
endtask  
  
endclass
```

# Connecting TLM Ports

- When we are connecting components that lie in same level of hierarchy, then ports are always connected to exports.
- All connection call between components is done in parent's connect phase.



# Example

```
class agent extends uvm_agent;
  get_producer get_prod;    //Get Components
  get_consumer get_cons;

  put_producer put_prod;    //Put Components
  put_consumer put_cons;

  function void connect_phase (uvm_phase phase);
    get_cons.get_port.connect(get_prod.get_imp);
    put_prod.put_port.connect(put_cons.put_imp);
  endfunction
endclass
```

# Blocking vs. Non-Blocking

- uvm\_blocking\_get\_port/uvm\_blocking\_put\_port provides set of blocking methods that blocks the process if target is not ready:

- put()

- get()

- peek()

- uvm\_non\_blocking\_get\_port/uvm\_non\_blocking\_put\_port provides set of methods that returns immediately even if target is not ready:

- try\_put()

- try\_get()

- try\_peek()

# TLM FIFO

- In certain situations we want to connect a get consumer to a put producer.
- In such cases we use TLM FIFO. These are verification component independent of implementation.
- They contain all TLM interface methods like put, get, peek etc.



- Producer puts transaction in `uvm_tlm_fifo` where as consumer gets transaction from the fifo.

# Example

```
class fifo_example extends uvm_component;
  get_consumer get_cons;    //Consumer of get type
  put_producer put_prod;    //Producer of put type

  uvm_tlm_fifo #(packet) fifo_inst; //TLM FIFO declaration

  function new (string name, uvm_component parent);
    super.new(name, parent);
    put_prod=new("producer", this); //using create is recommended
    get_cons=new("consumer", this);
    fifo_inst=new("fifo_inst", this, 16); //set fifo depth to 16
  endfunction
```



# Example

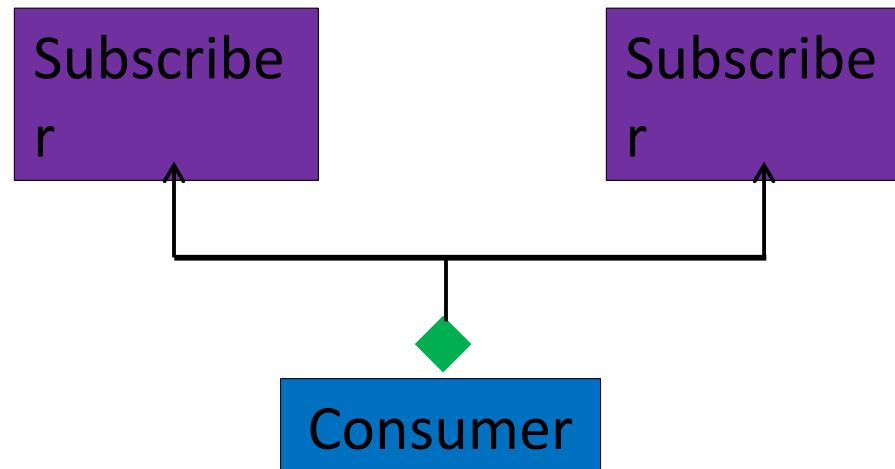
```
function void connect_phase(uvm_phase phase);  
  put_prod.put_port.connect(fifo_inst.put_export);  
  get_cons.get_port.connect(fifo_inst.get_export);  
endfunction  
  
endclass
```

# Analysis port and export

- While using put and get method, it is mandatory to connect them to exactly one export/imp before simulation starts.
- UVM will report errors if ports are left unconnected.
- In certain cases (like in monitors) we require a port that can either be left unconnected or connected to one or more components.
- In UVM, Analysis ports serve this purpose. Symbol (◆) .
- Analysis port has a single non-blocking write function that can be called with single transaction argument.

# Analysis port and export

- When write method of analysis port is called, analysis port calls write method of every connected analysis exports.
- `uvm_subscriber` class contains inbuilt analysis export and pure virtual function called as `write`, Hence it become compulsory for derived class to provide implementation for `write`.



# Analysis port

```
class monitor extends uvm_component;
uvm_analysis_port #(packet) analysis_port;

function new (string name, uvm_component parent);
super.new(name, parent);
analysis_port=new("analysis_port", this);
endfunction

virtual task run ();
packet p=new;
.....//collect packet from lower level
analysis_port.write(p); //write collected transaction
endtask
endclass
```

# Analysis export

```
class checker extends uvm_component;  
  uvm_analysis_imp #(packet, checker) analysis_export;
```

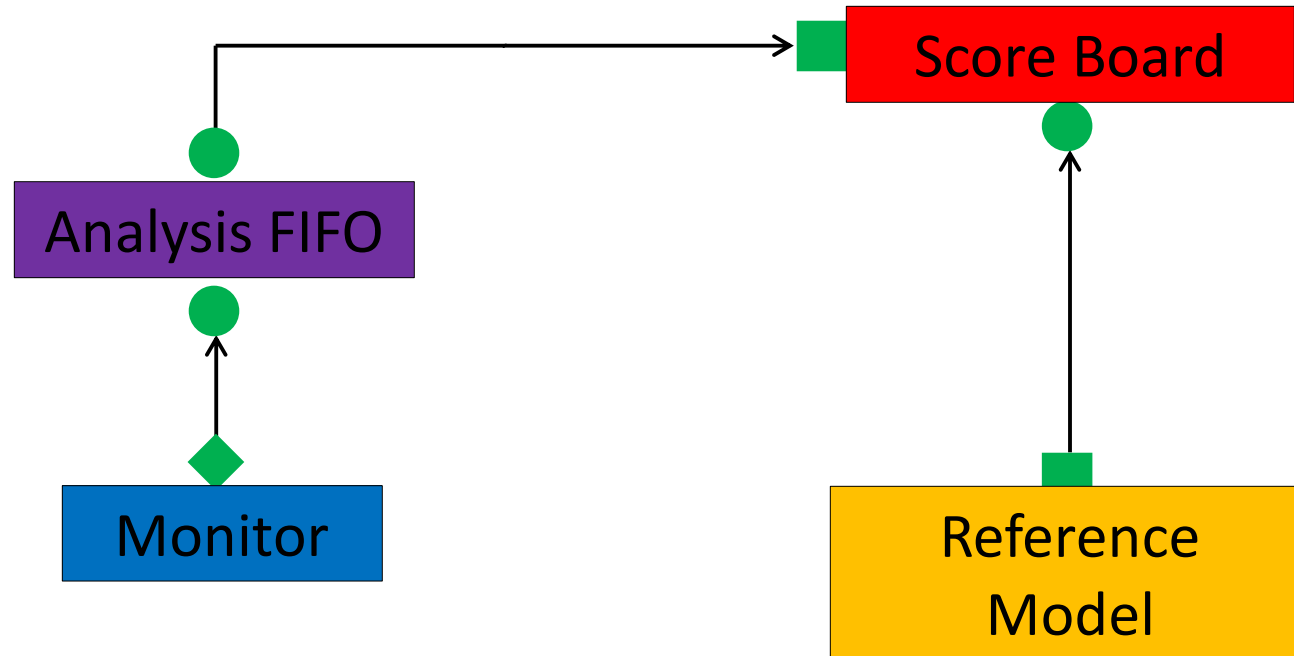
```
function new (string name, uvm_component parent);  
  super.new(name, parent);  
  analysis_export=new("analysis_export", this);  
endfunction
```

```
function void write (packet p);  
  //perform some check on packet p  
endfunction  
endclass
```

# Analysis FIFO

- Sometimes transactions that are passed through analysis port cannot be processed immediately.
- In such case the transaction needs to be stored before they are consumed.
- Example scoreboard needs to compare actual packet coming from DUT with expected packet coming from a reference model.
- `uvm_tlm_analysis_fifo` is used to address this requirement. It has an `analysis_export` that can be directly connected to analysis ports.
- Analysis fifo has unbound size so that write always succeeds.

# Example Usage



# UVM CONFIGURATION



# Factory Overriding

- UVM factory has the ability to substitute a child class with another class of a derived type when it is constructed.
- This helps in changing the behavior of test bench by substituting one class for another without editing or re-compiling the code.
- There are two types of overriding :
  - `set_type_override_by_type` used for overriding type i.e. global override.
  - `set_inst_override_by_type` used for overriding an instance i.e. instance override.

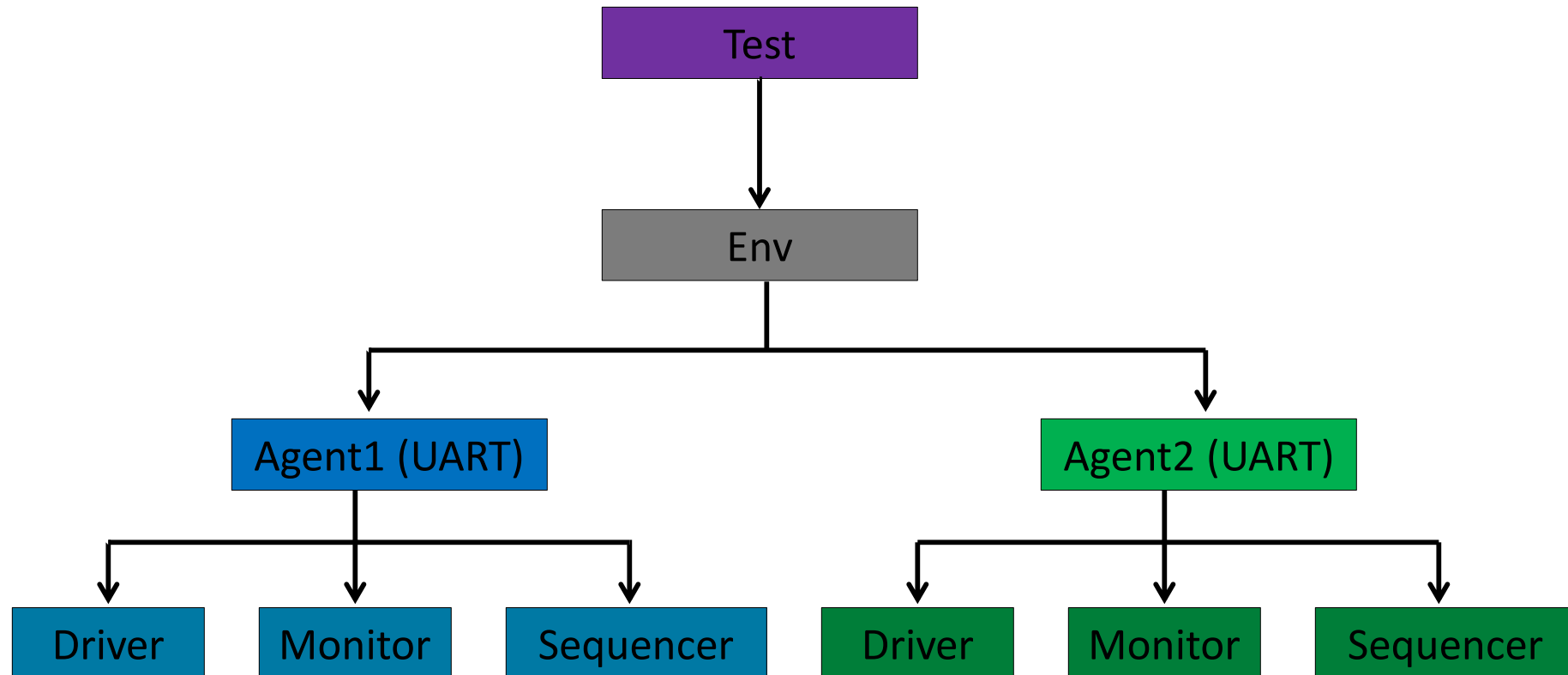
# Usage

```
set_type_override_by_type ( original_type :: get_type(),  
                             substitute_type :: get_type(),  
                             replace = 1);
```

```
set_inst_override_by_type ( "path",  
                             original_type :: get_type(),  
                             substitute_type :: get_type(),  
                             replace = 1);
```

- original\_type is name of the component that has to be replaced.
- substitute\_type is name of the component by which original has to be replaced.
- path refers to the hierarchal position of the component.
- replace specifies whether to replace definition in factory or not.

# Example

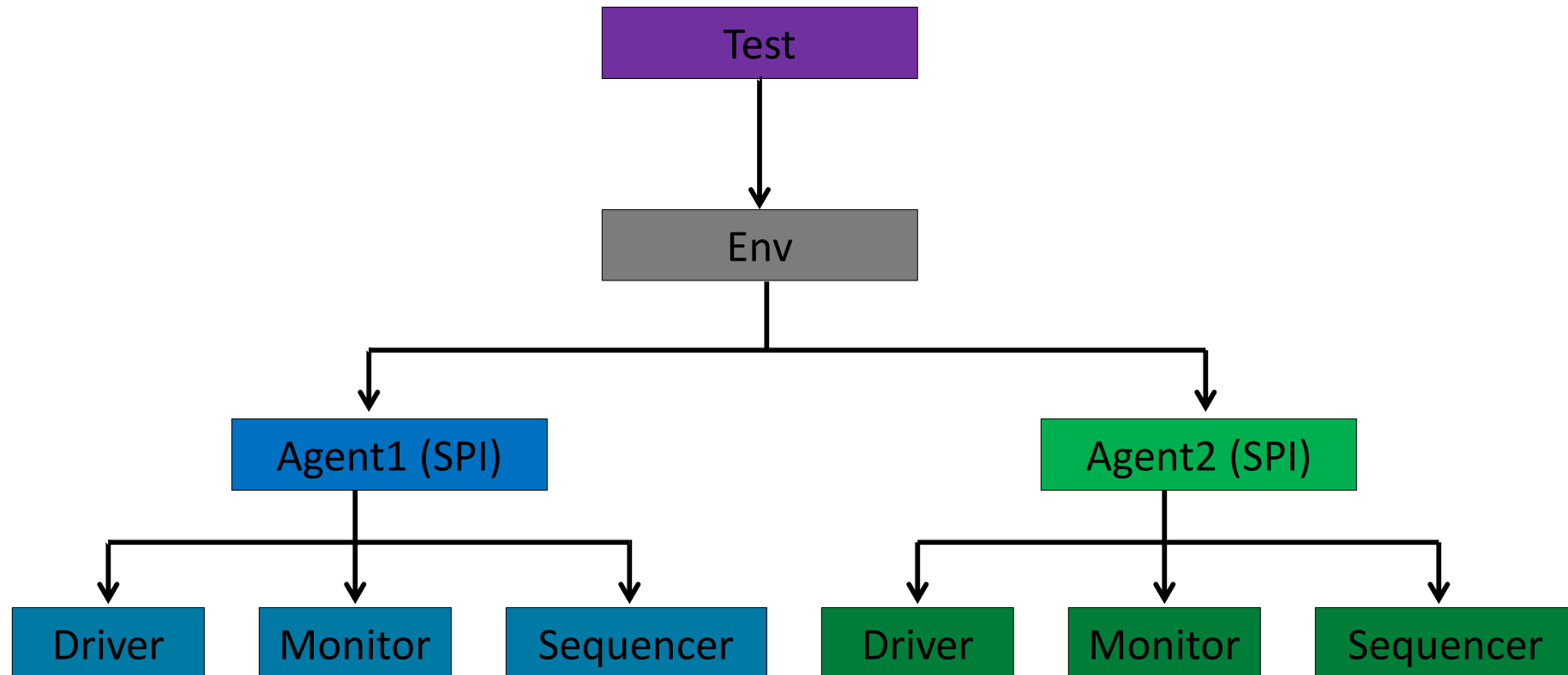


# Global Override

```
class env extends uvm_env;

function void build_phase (uvm_phase phase);
super.build_phase (phase);
set_type_override_by_type ( UART :: get_type(),
                           SPI :: get_type());
//Both UART and SPI classes are registered with UVM Factory
.....
endfunction
endclass
```

# Result



# uvm\_config\_db

- set\_config methods discussed earlier are deprecated method for setting and getting configuration fields.
- UVM provides Configuration Database class to set integers, strings, virtual interfaces, objects etc resources with hierarchal context.
- All functions in uvm\_config\_db #(type T) are static in nature and can be accessed with help of scope resolution operator(::).

# uvm\_config\_db : set method

```
uvm_config_db #(type T) :: set ( cntxt,  
                                "inst_name",  
                                "field_name",  
                                value);
```

- type is used to specify type of field.
- cntxt is a prefix added to instance name, if null then instance name provides complete scope of setting.
- inst\_name is the place from which setting will be affected.
- field\_name is the name of parameter to be updated.
- value is the updated value for the given field\_name.

# uvm\_config\_db : get method

```
assert ( uvm_config_db #(type T) :: get ( cntxt, "inst_name",  
                                           "field_name", value) );
```

- get method returns the status for availability of field.
- cntxt is used to provides starting of search point.
- inst\_name is name of instance where the field is available, can be null if cntxt is the instance name.

- Example

```
uvm_config_db #(bit) :: set(this, "*.ag[1].*", "is_active", 1);  
void' ( uvm_config_db #(bit) :: get(this, " ", "is_active", is_active));
```



# Example - Env

```
class env extends uvm_env;
agent ag [2];
.....
function void build_phase (uvm_phase phase);
super.build_phase(phase);
uvm_config_db #(bit) :: set(null, "*.ag[0].*", "is_active", 1);
uvm_config_db #(bit) :: set(null, "*.ag[1].*", "is_active", 0);
ag[0]=agent :: type_id :: create("ag[0]", this);
ag[1]=agent :: type_id :: create("ag[1]", this);
endfunction
endclass
```

# Example - Agent

```
class agent extends uvm_agent;
```

```
bit is_active;
```

```
monitor mon;
```

```
driver dvr;
```

```
sequencer sqr;
```

```
`uvm_component_utils(agent)
```

```
.....//On Next Slide
```

```
endclass
```

# Example - Agent

```
function void build_phase (uvm_phase phase);  
super.build_phase (phase);  
void' ( uvm_config_db #(bit) :: get (this, " ", "is_active", is_active));  
if(is_active==1)  
begin  
dvr= driver :: type_id :: create("dr", this);  
sqr= sequencer :: type_id :: create("sqr", this);  
end  
mon= monitor :: type_id :: create("mon", this);  
endfunction
```

## Example – Virtual Interface (Top)

```
module top;
mem_inf i0 (clk);    //Memory interface
mem_dut u0 (i0);    // DUT instance

initial begin
uvm_config_db #(virtual mem_inf) :: set (null, "*", "mem_vif", i0);
//registering virtual interface with all components(*) using
    mem_vif
//as the field name
run_test();          //calling uvm_top.run_test();
end
endmodule
```

# Example – Virtual Interface (Driver)

```
class mem_driver extends uvm_driver;
virtual mem_if vif;

function void build_phase(uvm_phase phase);
super.build_phase(phase);
if(! uvm_config_db #(virtual mem_inf) :: get(this, "", "mem_vif",
    vif))
`uvm_fatal("Vif Error", "cannot locate mem_vif in config
    database");
endfunction

task run_phase(uvm_phase phase); .....
endtask
```

# uvm\_resource\_db

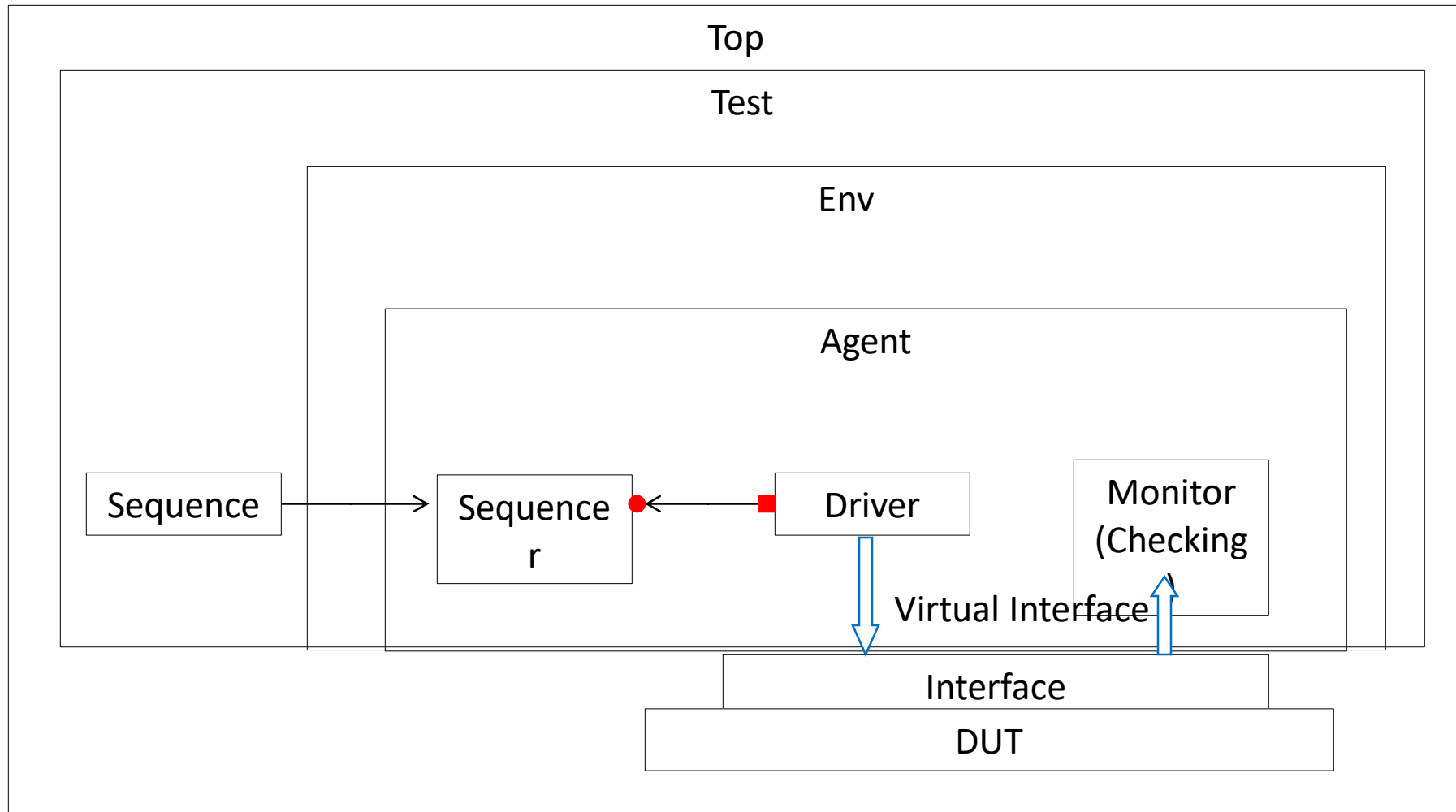
- uvm\_resource\_db is a depreciated class that offer functionality similar to that of uvm\_config\_db.

```
uvm_resource_db#(type T) :: set ( "scope", "field_name", value);  
uvm_resource_db#(type T) :: get_by_name( "scope",  
                                           "field_name", value));
```

- scope is the lookup parameter
  - field\_name is the parameter to be updated
  - value is the value or object to update the field\_name
- If same thing is written at multiple places in the hierarchy then last write wins. UVM recommends usage of uvm\_config\_db.

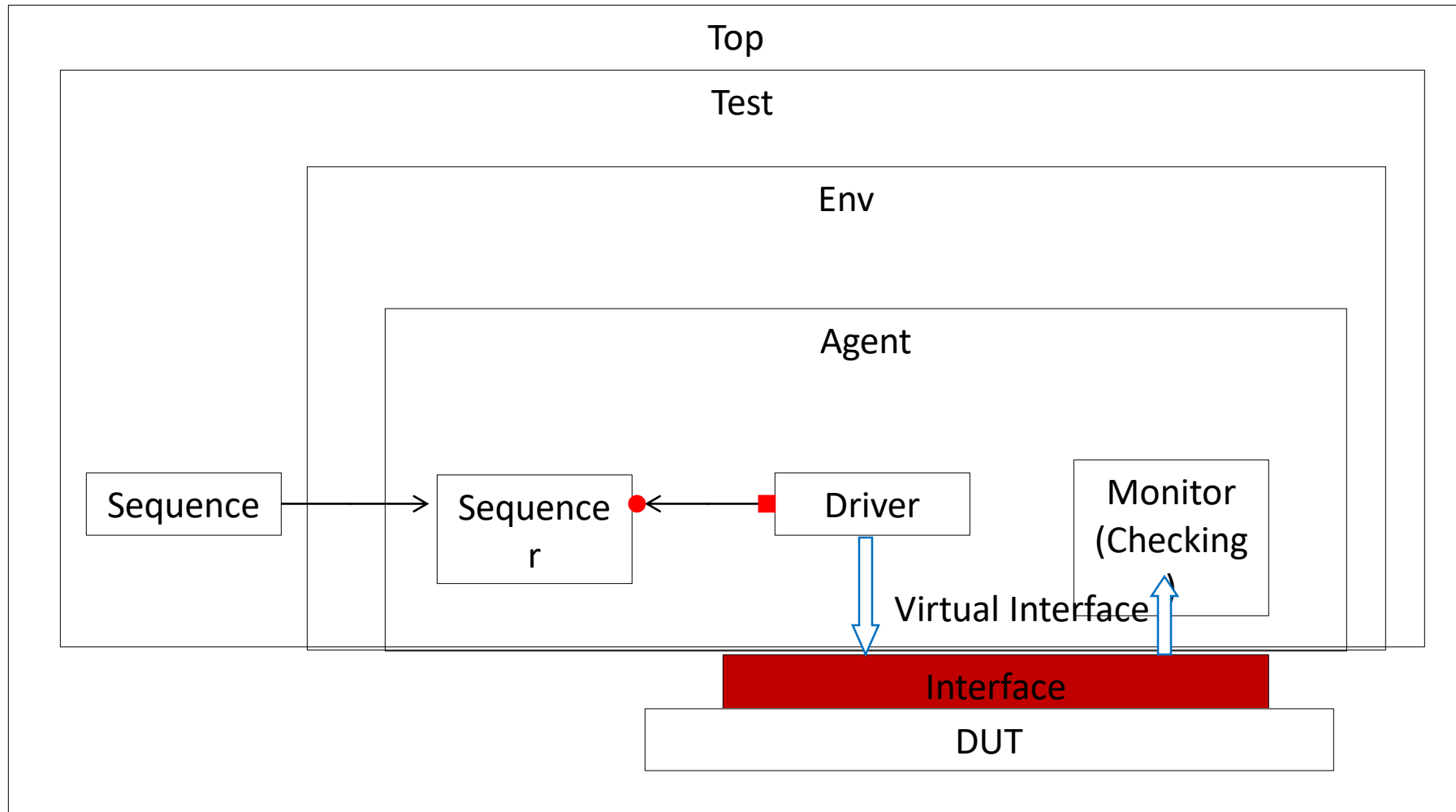
# Creating Components (Adder)

# Test Bench Hierarchy





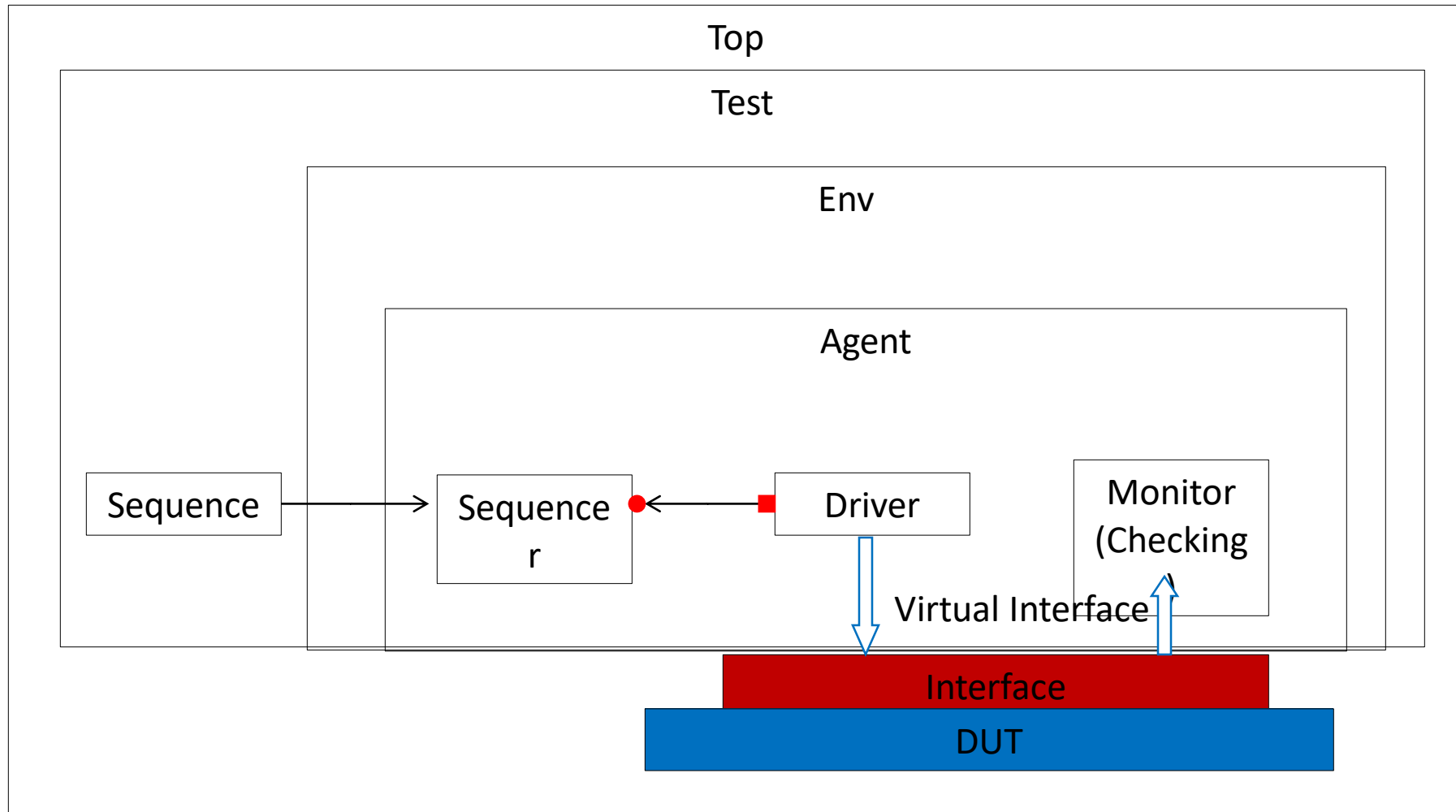
# Test Bench - Interface



# Adder- Interface

```
interface adder_inf (input bit clk);  
logic [3:0] a, b;      //inputs a and b  
logic c;               //input c  
logic [4:0] sum;       //output sum  
  
modport DUT (output sum, input a, b, c, clk);  
//Defining direction of signals  
  
endinterface
```

# Test Bench - DUT



# Adder- DUT

```
module adder (adder_inf.DUT inf);
```

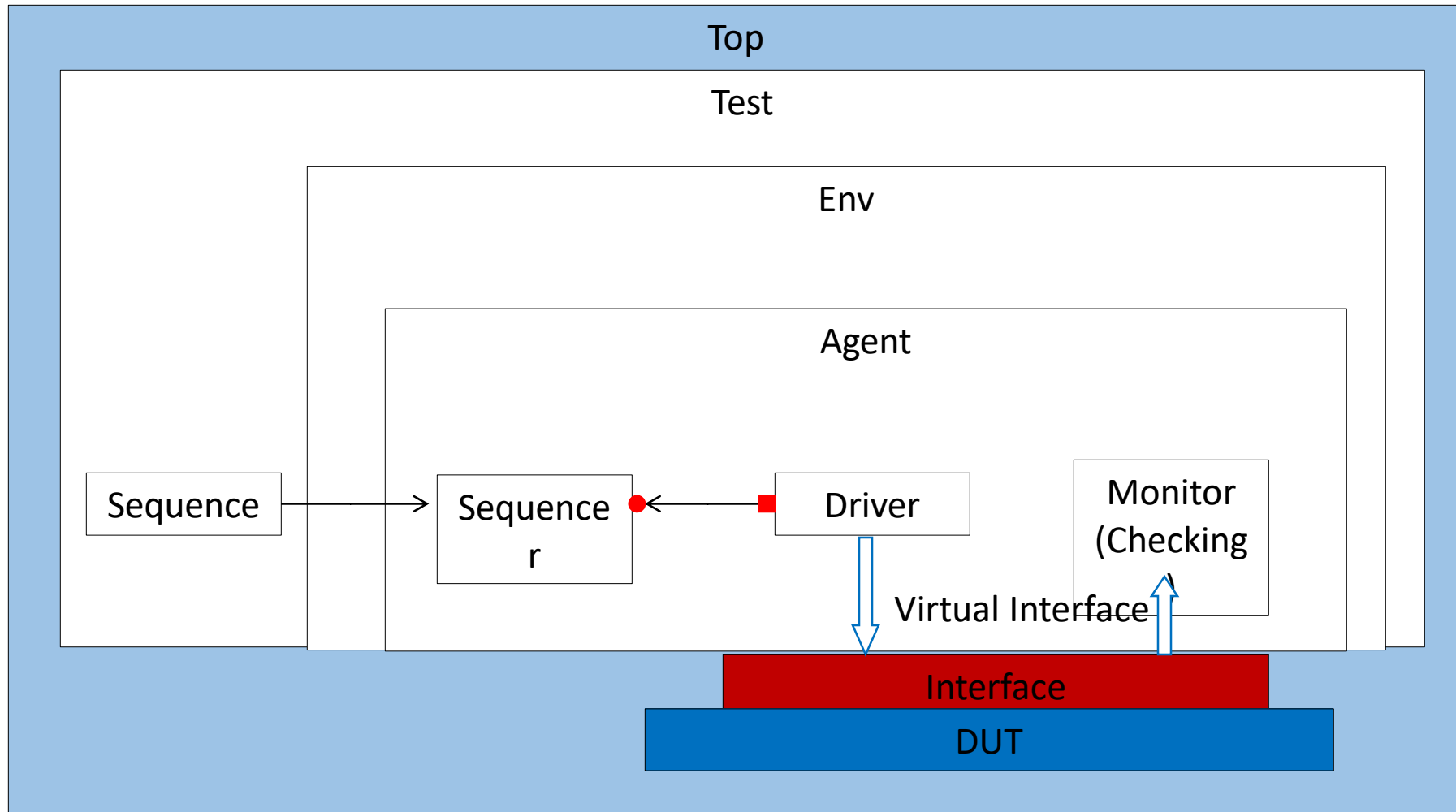
```
    always @(posedge inf.clk)
```

```
    inf.sum<= inf.a + inf.b + inf.c;
```

```
    //Definition of adder
```

```
endmodule
```

# Test Bench – TOP



# Adder - Top

```
module top;
import uvm_pkg :: *;          //import UVM package
`include "uvm_macros.svh"     //include UVM macros
`include "myfiles.sv"         //include user defined file

bit clk;

adder_inf i0 ();    //Adder interface
adder u0 (i0);      // Adder instance

.....              //On Next Slide

endmodule
```

# Adder - Top

```
initial
begin
  uvm_config_db #(virtual adder_inf) :: set (null, "*", "add_vif", i0);
  //registering virtual interface to configuration database
  run_test();
  //calling uvm_top.run_test();
  //Test name can be specified as an argument
end

always #5 clk<=~clk;
```

# Adder - transaction

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

  `uvm_object_utils_begin(transaction)
    `uvm_field_int(a, UVM_DEFAULT)
    `uvm_field_int(b, UVM_DEFAULT)
    `uvm_field_int(c, UVM_DEFAULT)
    `uvm_field_int(sum, UVM_DEFAULT)
  `uvm_object_utils_end
  .....//On Next Slide

endclass
```

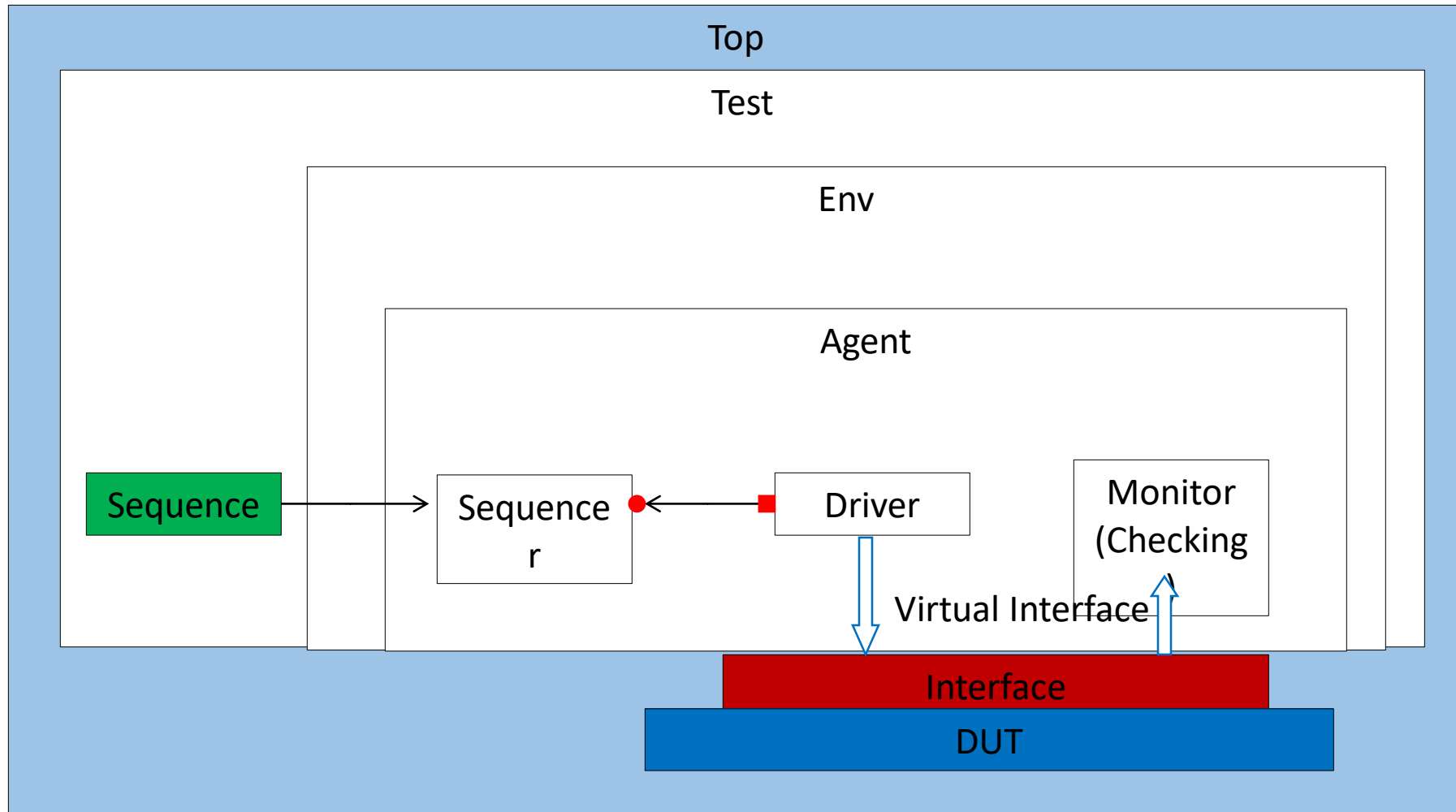


# Adder - transaction

//Defining Constructor is not compulsory

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

# Test Bench – Sequence



# Adder – sequence1

```
class sequence1 extends uvm_sequence#(transaction);  
  
transaction pkt;  
  
`uvm_object_utils(sequence1)  
  
function new (string name="sequence1");  
super.new(name);  
endfunction  
  
..... //On Next Slide  
  
endclass
```

# Adder – sequence1

```
task body();  
repeat(1000)  
begin  
  pkt=transaction :: type_id :: create("pkt");  
  start_item(pkt);  
  void'(pkt.randomize);  
  finish_item(pkt);  
end  
endtask
```

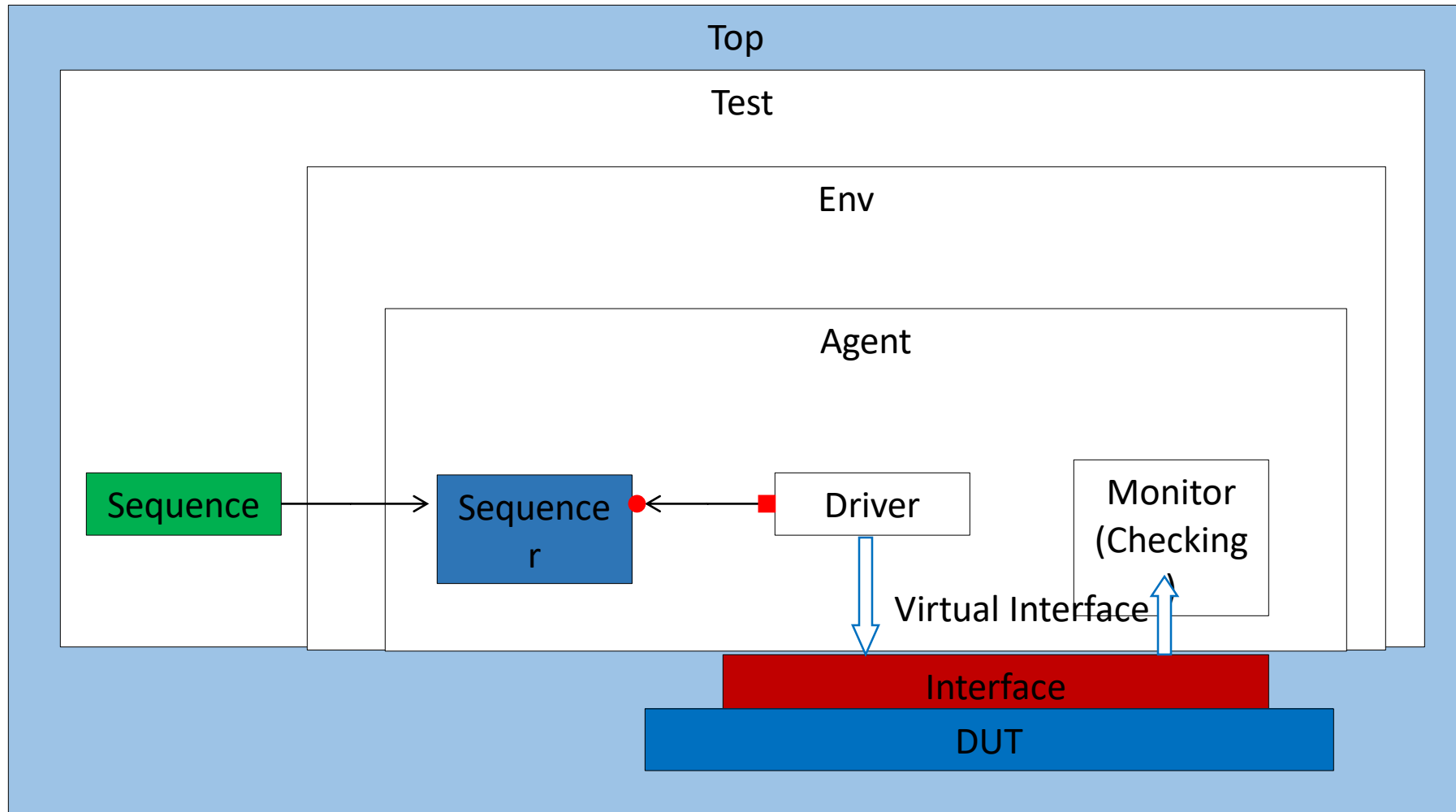
# Adder – sequence2

```
class sequence2 extends uvm_sequence#(transaction);  
  
transaction pkt;  
  
`uvm_object_utils(sequence2)  
  
function new (string name="sequence2");  
super.new(name);  
endfunction  
  
..... //On Next Slide  
  
endclass
```

# Adder – sequence2

```
task body();  
repeat(1000)  
begin  
  pkt=transaction :: type_id :: create("pkt");  
  start_item(pkt);  
  pkt.randomize with {a<16; b<16;};  
  //inline constraints Randomization  
  finish_item(pkt);  
end  
endtask
```

# Test Bench – Sequencer



# sequencer

- Sequencer routes stimulus data from sequence and passes it to a driver.
- Extend user define sequencer from uvm\_sequencer base class. This class is parameterized by request and response item types.

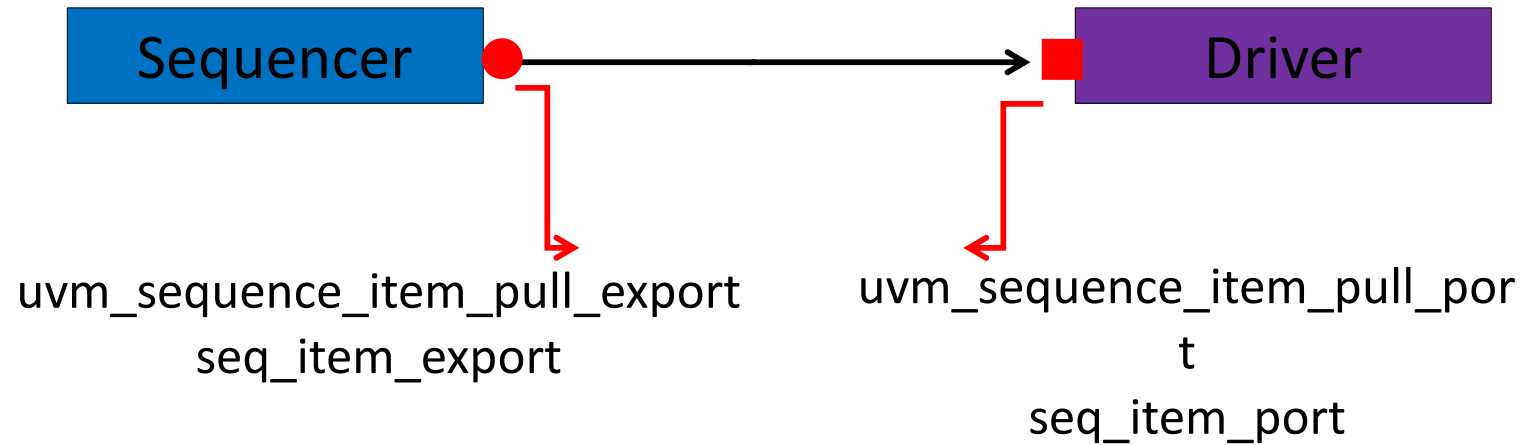
Syntax: `typedef uvm_sequencer #(REQ, RSP) sequencer;`

- RSP(Response) is same type as that of REQ(Request) if it is not specified separately.

```
typedef uvm_sequencer #(transaction) sequencer;  
//user defined sequencer
```

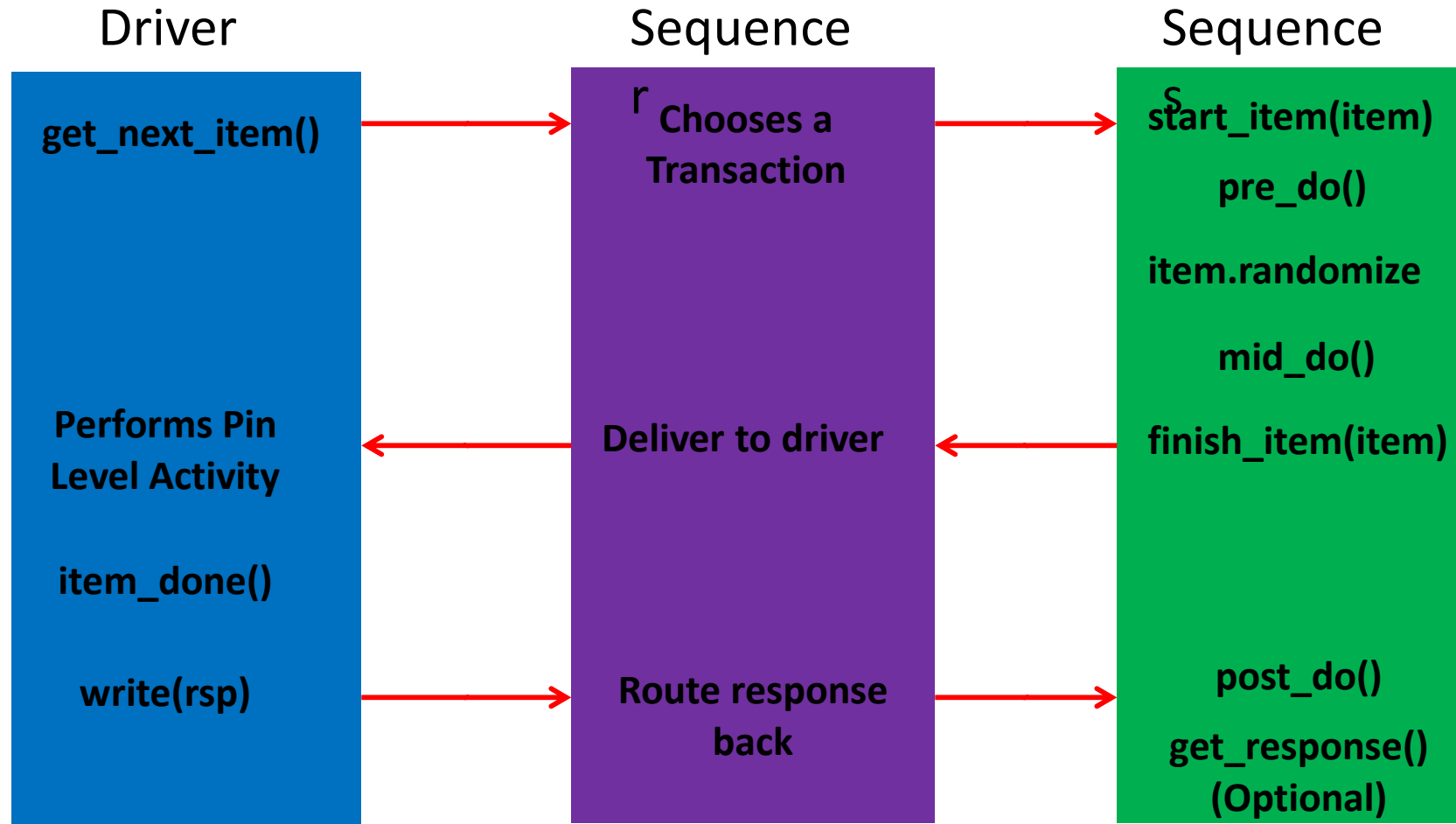


# Sequencer - Driver Connection

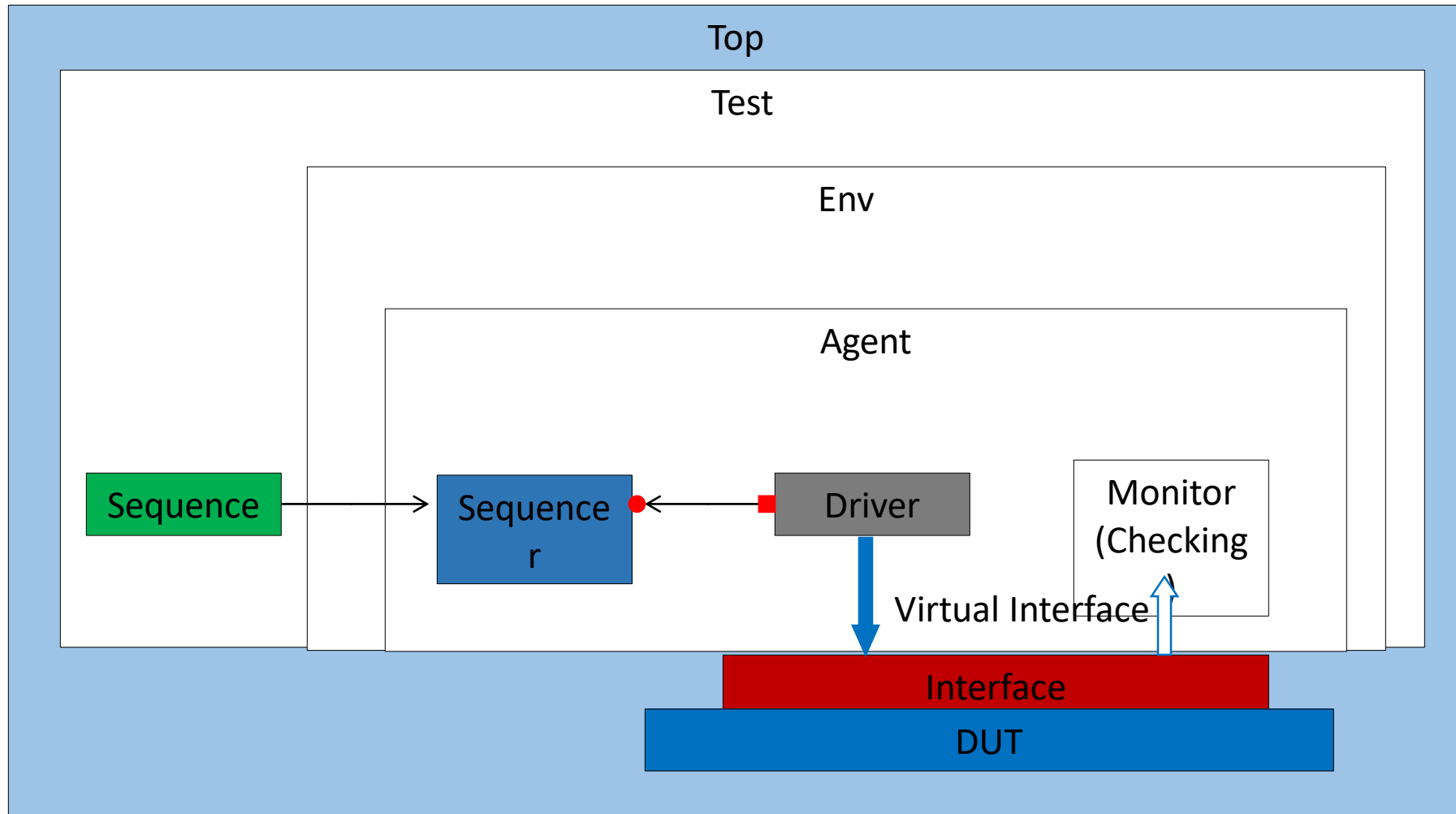


- Driver gets the data through seq\_item\_port and optionally provides response
- Sequencer provides the data through seq\_item\_export.
- Connection between seq\_item\_port and seq\_item\_export is done in agent.

# Transaction Flow



# Test Bench – Driver



# driver

- A driver will drive DUT signals with help of virtual interface.
- `uvm_sequence_pull_port` class provides two methods to fetch data from a sequencer.
  - `get_next_item()` : Blocking Method
  - `try_next_item()` : Non Blocking Method
- Driver needs to return the processed response back to sequencer, `uvm_sequence_pull_port` provides following methods returning response
  - `item_done()` : Non Blocking Method
  - `put_response()` : Blocking Method, sequence must do `get_response()`
  - `rsp_port.write(rsp)` : built-in Analysis port in driver (Optional)

# Adder – driver

```
class driver extends uvm_driver #(transaction);  
  
virtual adder_inf vif;  
transaction pkt;  
  
`uvm_component_utils(driver)  
  
function new (string name, uvm_component parent);  
super.new(name, parent);  
endfunction  
  
function void build_phase (uvm_phase phase);  
if(! uvm_config_db #(virtual adder_inf)::get(this, "", "add_vif", vif))  
`uvm_error("Driver", "Unable to get virtual interface");  
..... // On Next Slide
```

# Adder – driver

```
super.build_phase(phase);  
endfunction
```

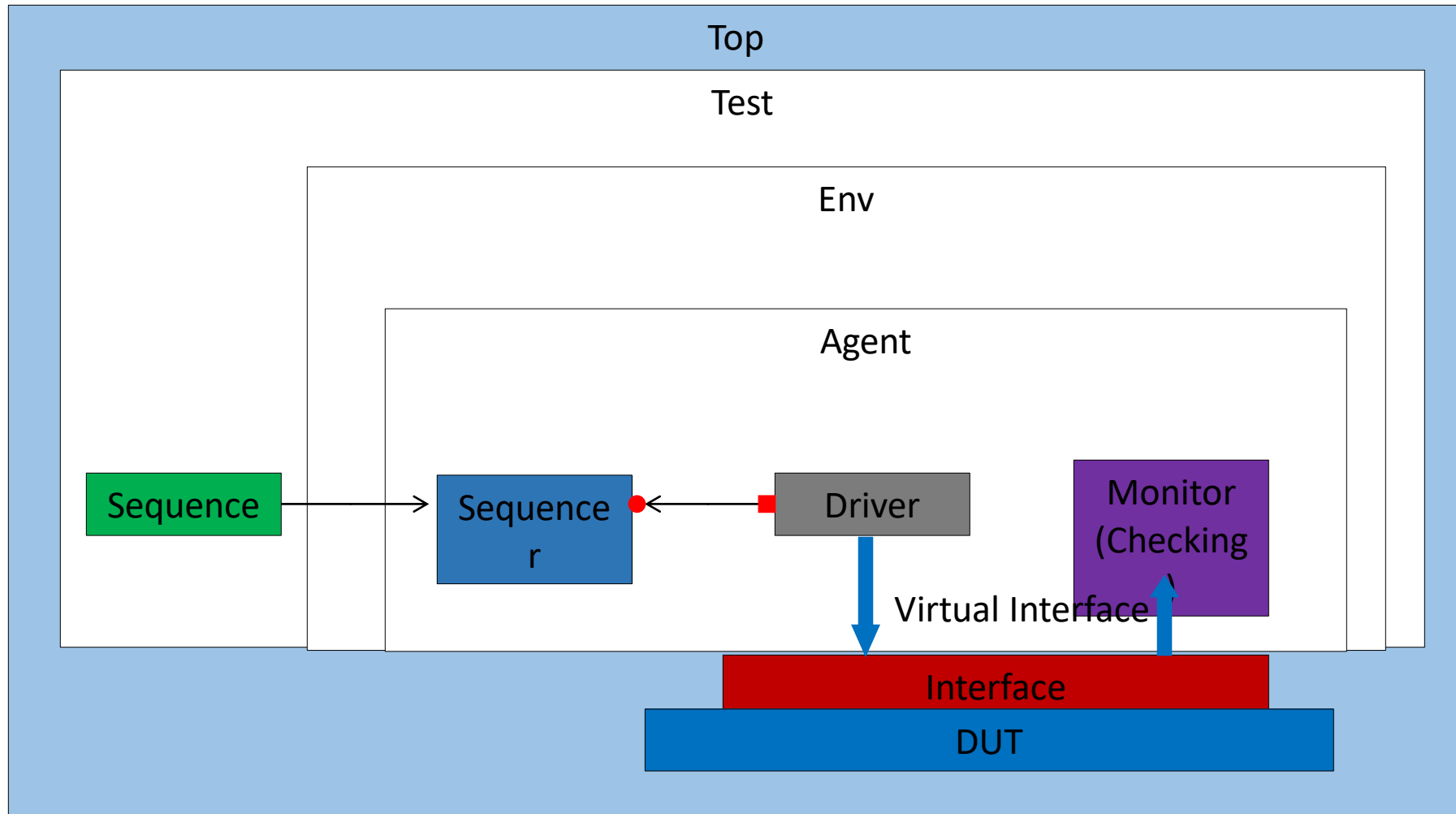
```
task run_phase(uvm_phase phase);  
  forever begin  
    seq_item_port.get_next_item(pkt);  
    drive_item(pkt);  
    seq_item_port.item_done();  
  end  
endtask
```

```
..... //On Next Slide
```

# Adder – driver

```
task drive_item (transaction pkt);  
  @(posedge vif.clk)  
  vif.a<=pkt.a;  
  vif.b<=pkt.b;  
  vif.c<=pkt.c;  
endtask  
  
endclass
```

# Test Bench – Monitor





# Adder – monitor

```
class monitor extends uvm_monitor;

virtual adder_inf vif;
bit [4:0] sum;

`uvm_component_utils_begin(monitor)
  `uvm_field_int(sum, UVM_DEFAULT)
`uvm_component_utils_end

covergroup cg;
a : coverpoint vif.a;
b : coverpoint vif.b;
c : coverpoint vif.c;
cross a, b, c;
endgroup .....//On Next Slide
```

# Adder – monitor

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

```
function void build_phase (uvm_phase phase);  
if(! uvm_config_db #(virtual adder_inf)::get(this," ", "add_vif",vif))  
`uvm_error("Monitor", "Unable to get virtual interface")  
super.build_phase(phase);  
endfunction
```

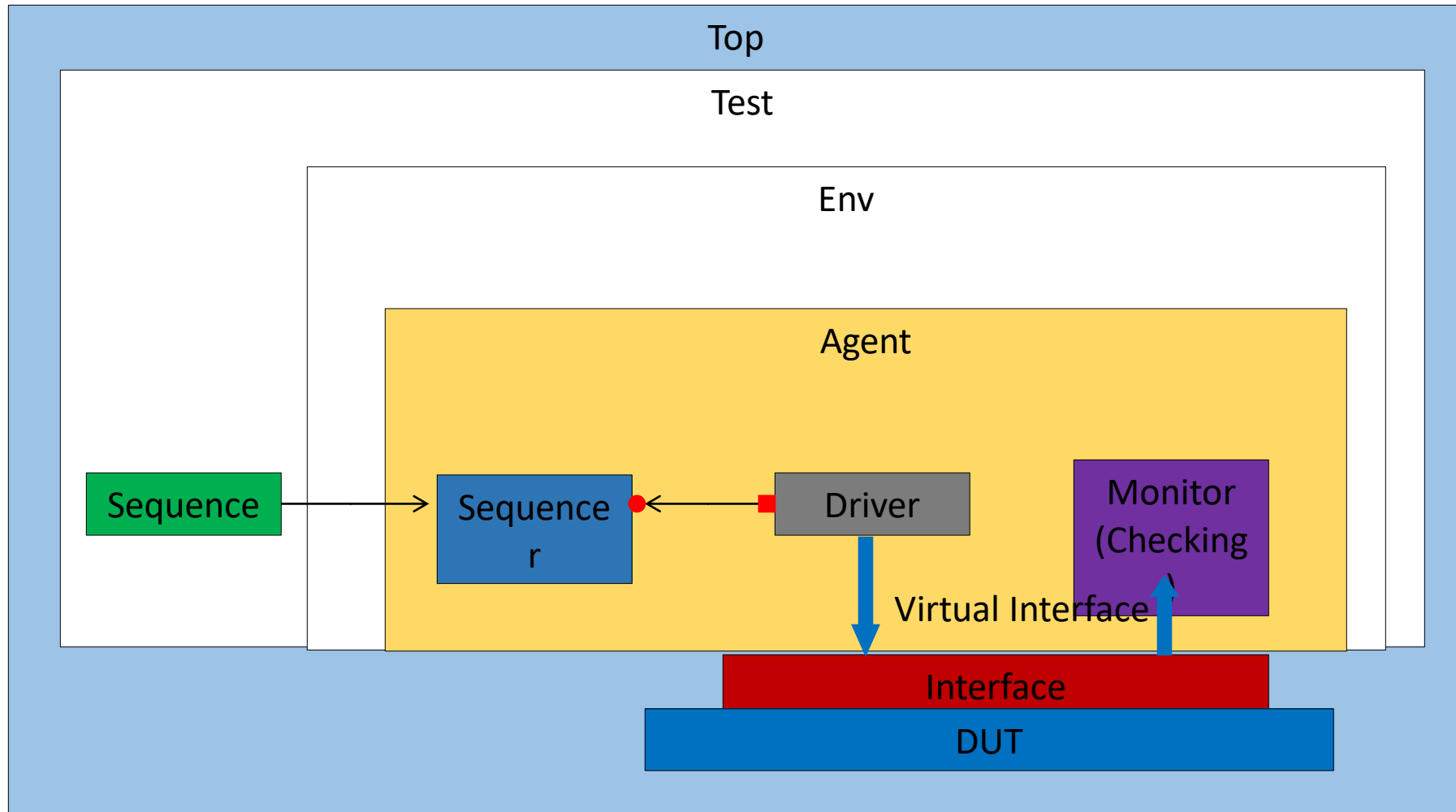
.....//On Next Slide

# Adder – monitor

```
task run_phase (uvm_phase phase);  
  forever predict_and_compare();  
endtask
```

```
task predict_and_compare();  
  @(posedge vif.clk);  
  cg.sample;  
  sum=vif.a + vif.b + vif.c;  
  @(posedge vif.clk);  
  if(sum!=vif.sum)  
    `uvm_error("Monitor", "Result Mismatch")  
endtask  
endclass
```

# Test Bench – Agent



# Adder – agent

```
class agent extends uvm_agent;

bit is_active=1;

`uvm_component_utils(agent)

monitor mon;
driver dvr;
sequencer sqr;

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

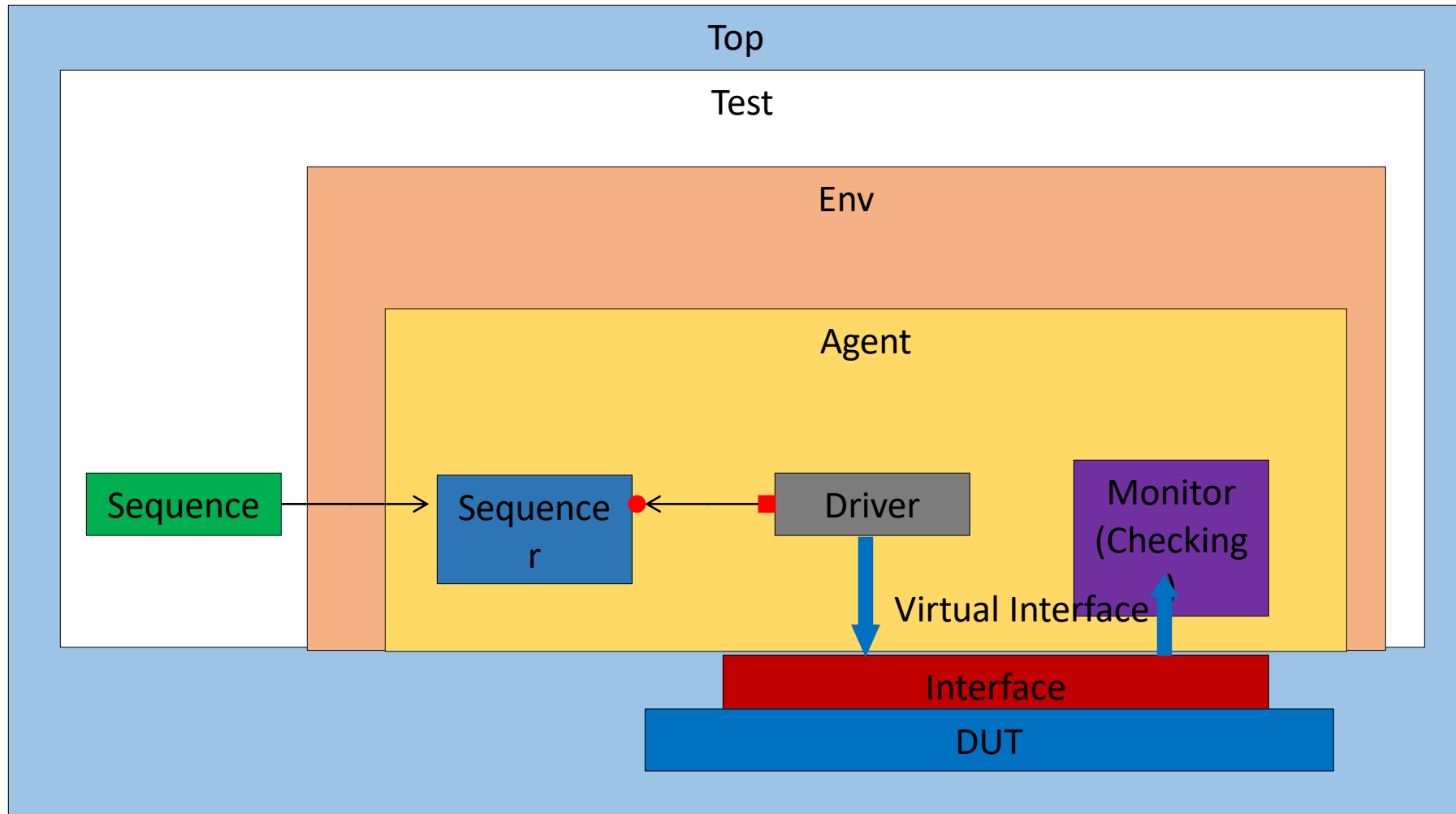
.....//On Next Slide
```

# Adder – agent

```
function void build_phase (uvm_phase phase);  
    super.build_phase (phase);  
    mon=monitor :: type_id :: create("mon", this); .....//call get method  
    if(is_active) begin  
        dvr=driver :: type_id :: create("dvr", this);  
        sqr=sequencer :: type_id :: create("sqr", this);  
    end  
endfunction
```

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

# Test Bench – Env



# Adder – env

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

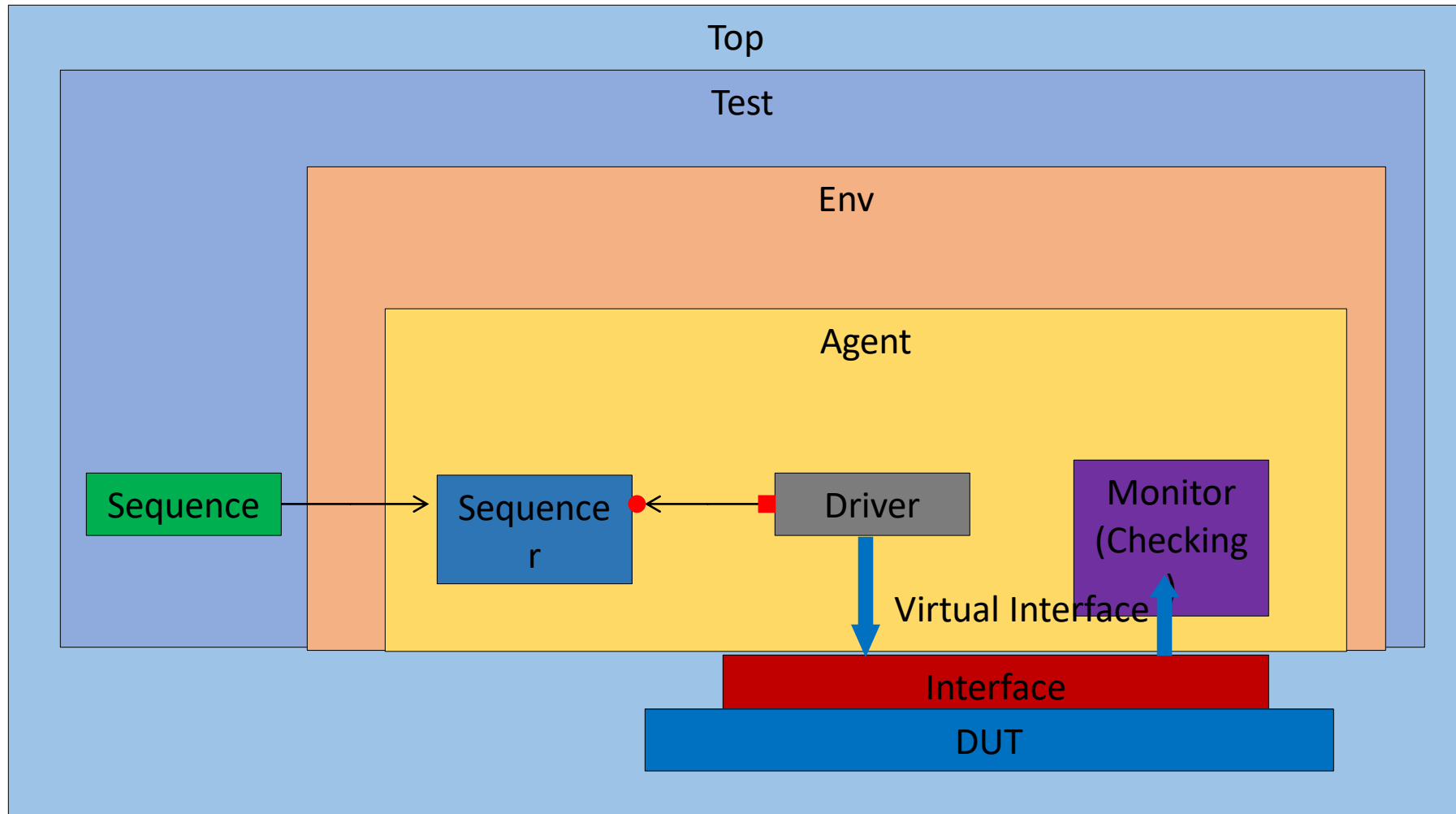
  agent ag;

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

  function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db #(bit) :: set (this, "ag", "is_active", 1);
    ag=agent :: type_id :: create("ag", this);
  endfunction
endclass
```



# Test Bench – Test



# Adder – test

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

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

.....// On Next Slide
```

# Adder – test

```
task run_phase (uvm_phase phase);  
sequence1 seq1;  
phase.raise_objection(this);      //Raise objection  
seq1=sequence1:: type_id :: create("seq1");  
seq1.randomize;  
seq1.start(e.ag.sqr);              //start sequence on sequencer  
phase.drop_objection(this);       //Drop Objection  
endtask  
  
endclass
```

# “myfiles.sv” and run test

//Order is based on dependency

```
`include “interface.sv”
```

```
`include “adder.sv”
```

```
`include “transaction.sv”
```

```
`include “sequence.sv”
```

```
`include “sequencer.sv”
```

```
`include “driver.sv”
```

```
`include “monitor.sv”
```

```
`include “agent.sv”
```

```
`include “env.sv”
```

```
`include “test.sv”
```

Pass plus argument in command line to run test name called test

+UVM\_TESTNAME=test