**Universal verification Methodology**

UVM is standard methodology framework to build the verification component.

There are so many built in classes like uvm components and uvm objects.

TLM facility to store and retrieve data

Configuration data -> uvm\_config\_db

Uvm agents and uvm factory

Achieving more reusability compared to sv in soc level.

1. **Uvm component and uvm object**

**UVM component** is a base class in uvm for creating hierarchy is called as uvm component

Uvm components is present entire uvm testbench

Components are connected using tlm ports

For uvm component, function new constructor has two arguments one is string name and uvm component parent. Below is example code.

**Function new(string name = “Uvm\_commponet”, uvm component parent);**

**Super.new(name, parent);**

**Endfunction**

**UVM object** is a base class for creating stimulus and it doesnot longer entire uvm testbench . it doesnot have any hierarchy.

For uvm object, new constructor has only one argument

**Function new(string name = “Uvm\_commponet”);**

**Super.new(name);**

**Endfunction**

**UVM agent** is base class. It is also configurable component. Also called as UVC (universal verification component).

Agent has also active and passive agents

Active agent means driver, monitor and sequencer

Passive means only monitor

**UVM factory**

Is a class that creates a component and objects during runtime.

Need of uvm factory is to override components with other components.

In SV polymorphism, we will override methods only.

Use of factory means for example, there is a one driver code, according to our specification, we need to implement another driver, in that case without erasing existing driver code, by using factory overriding, we will override components. So that we must achieve more reusability.

For that purpose, we need to class with factory, we have uvm macros

`Uvm\_component\_utils(component name) // for component

`uvm\_object\_utils(object name) // for object

From this register, it will create type id also. By using type id, we can override components and create components and objects with type id.

Create method

Drv = driver::type\_id::create(“drv”, this); for component

Seq = sequence::type\_id::create(“seq”); for object

Factory overriding has 2 types

1. Global override:
2. Instance override

Global override: we can override globally,

Set\_type\_override\_by\_type(original class::get\_type(), substistute class::get\_type());

Instance override: by using specific instance only, we ca override,

Ex: we have 2 agents. Agent1 and agent2, we need to override driver component in agent2, in that case we will specify path in the first argument.

Set\_inst\_override\_by\_type(“agent1”, original class::get\_type(), substistute class::get\_type());

**UVM phases**

Phases are required to synchronize between multiple agents. We have too many components and objects, while creating stimulus and send data to the DUT. Without phases, stimulus is which is generated, it is not send by proper synchronously, it will lead to race free condition. In multiple agents we must drive more stimulus, that is why we need phases. In each phase, we will send stimulus.

Uvm phases are 9 phases

1. Build phase: in this phase, we will build components and objects. It is function and executes zero-time sim. Top-down approach

**Function build phase(uvm phase phase);**

**Super.build\_phase(phase);**

**Endfunction**

1. Connect phase : bottom up, it used for connecting components in agent and environment.
2. Start of simulation phase
3. End of elaboration phase to make any final adjustments

Above of 2 phases, we will print the topology, both are bottom up

1. Run phase; time consuming task, executing parallel an take simulation time.bottom up approach.
2. Extract phase ; retrieves information from scoreboard and coverage and monitors. Buttom up approach

Before coming to extract phase, it will check all in run phase activities are completed or not. That means raise\_objection and drop\_objection.

All raise objections are dropped only , it will move from run phase to extract phase.

1. Check phase: if dut behaving correcting or not and identifies errors
2. Report phase : display results of the simulation, buttom up approach
3. Final phase: any outstanding activities, it will complete in this phase, it is a top-down approach.

**Sequence -driver communication**

Start\_item get\_next\_item

Randomize user task

Finish\_item item\_done

Driver initiates request by sending get\_next\_item which is blocking method, sequence will start item and generate sequences and randomizes and sent to driver which is drive data to DUT and send response back to sequence by using item\_done.

Sequence will tell finish item.

**UVM\_config\_db --🡪 database**

It is a built-in class. Used for configure data from top level and retrive from low level components by using set and get

We can configure no\_of agents etc.

Uvm\_config\_db #(int) :: set( this, “\*”, “int”, agent );

If(!uvm\_config\_db #(int) :: get(this, “”, “int”, agent)

`uvm\_fatal(“driver”, not getting config db);