An Introduction to Universal Verification Methodology

[1]*Bhaumik Vaidya* [2]*NayanPithadiya*

[1 2] **Department of Electronics Engineering,**
Gujarat Technological University,
Gandhinagar, Gujarat, India.

[1]*vaidya.bhaumik@gmail.com*
[2]*pithadiya.nayan@yahoo.com*

ABSTRACT : Due to advancement in fabrication technology more and more logic is being placed on a single silicon die. More and more components are reused to improve time to market. Overall, more than 70 percent of the time is spent on verification. So there is a need for constructing a reusable and robust verification environment. Universal verification methodology was introduced to fulfill that goal. Another feature of UVM is that it is supported by all major simulator vendors, which is not the case with earlier methodology. This methodology is new so goal of this paper is to introduce the basic terminology used in UVM and how a simple verification environment can be constructed using UVM.

KEY WORDS: *UVM, driver, monitor, sequence, test, environment*

1. Introduction

The UVM (Universal Verification Methodology) was introduced in December 2009, by a technical subcommittee of Accellera. UVM uses Open Verification Methodology as its foundation. Accellera released version UVM 1.0 EA on May 17, 2010. UVM Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments. It uses system Verilog as its language. All three of the simulation vendors (Synopsys, Cadence and Mentor) support UVM today which was not the case with other verification methodology.

Today, more and more logic is being integrated on the single chip so verification of it is a very challenging task. More than 70 percent of the time is spent on the verification of the chip. So it is a need of an hour to have an common verification methodology that provide the base classes and framework to construct robust and reusable verification environment. UVM provides that.

In this paper , all the terminology related to UVM is introduced along with the sample example. In first phase uvm components are introduced. In second phase some of the features related to UVM are introduced and in final phase small environment is built using UVM from the scratch.

2. Test Bench architecture :

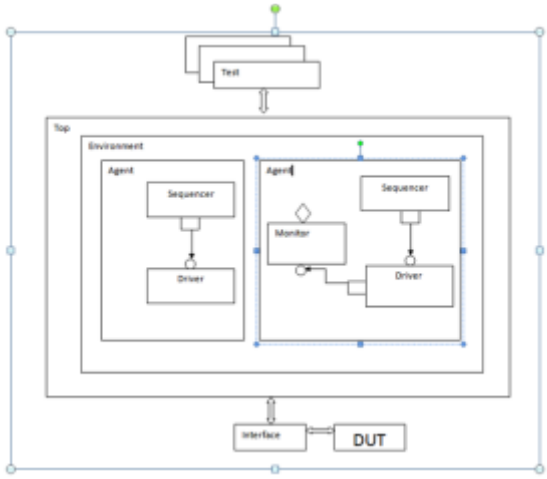


Figure 1Test bench architecture

The following subsections describe the components of a verification component.

Data Item (Transaction)

Data item are basically the input to the device under test. All the transfer done between different verification components in UVM is done through transaction object. Networking packets, instructions for processor are some examples of transactions. From the top level test many data items are generated and applied to the dut so by intelligently randomizing the data items object we can check corner cases and maximize the coverage on the device under test.

Driver (BFM)

Driver as the name suggest, drive the dut signals. It basically receives the transaction object from the sequencer and converts it in to the pin level activity. So for example it can generate read or write signal, write address and data to be transferred. It is the active part of the verification logic.

Sequencer

Sequencer is the component on which the sequences will run. The dut needs to be applied a sequence of transaction to test its behavior. So sequence of transaction is generated and it is applied to driver whenever it demands by the sequencer.

Monitor

A monitor is the passive element of the verification environment. It just sample the dut signal from the interface but does not drive them. It collect the pin information , package it in form of a packet and then transfer it to scoreboard or other components for coverage information.

Agent

Agent is basically a container. It contains driver, monitor and sequencer. Driver and sequencer are connected in agent. Agent has two modes of operation: passive and active. In active mode it drives the signal to the dut. So driver and sequencer are instantiated in active mode. In passive mode it just sample the dut signals does not drive them. So only monitor is instantiated in passive mode. Normally there is one agent per interface like AHB or APB.

Scoreboard:

Scoreboard is a verification component that checks the response from the dut against the expected response. So it keeps track of how many times the response matched with the expected response and how many time it failed.

Environment

Environment is at the top of the test bench architecture, it will contain one or more agents depend on design. If more than one agents are there then it will be connected in this component. Agents are also connected to other components like scoreboard in this component.

3. **Features of UVM**

Base classes in UVM

uvm_object is the base class for all components and sequences in UVM. uvm_component class is derived from this class and all uvm components listed above extends the uvm_component class. Transaction class is derived from uvm_object class and sequence_item and sequence extends the uvm_transaction class.

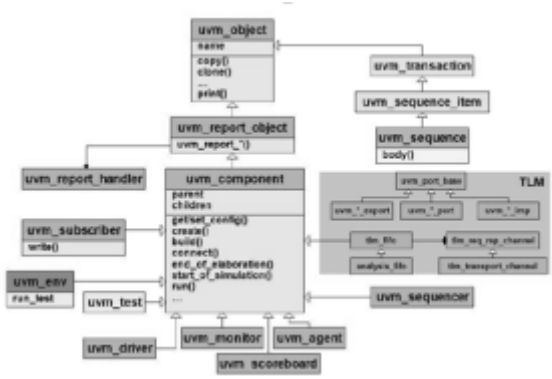


Figure 2 Class hierarchy in UVM

UVM Phases:

In UVM simulation runs in predefine phases so all the components used in the verification environment need to implement phase methods. So this phase methods will be called in order defined in the figure below.

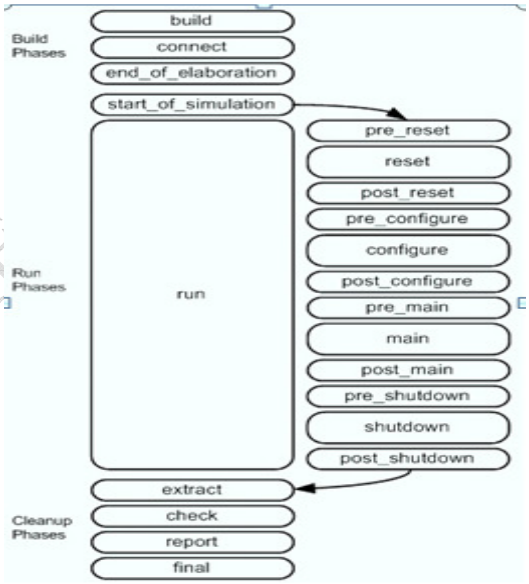


Figure 3 UVM phases

Now, brief description of all the phases:

Build phase: It is used to instantiate the child component instance as well as parent one.

Connect phase: It is used to connect ports to exports, exports to ports and ports to ports on the child components.

End_of_elaboration phase:It is used for fine-tuning the test bench. The verification environment has been completely assembled. Also used for print the topology and opening the files

Start_of_simulation phase: It is used to notify DUT for simulate. It shows that verification environment is completely configured and ready to start.

Run phase: It is used runs simulation. It is divided in to several run phases. It is the only phase that consumes time so task is used to define run phase

method. All other phases use function because they run in zero simulation time.

Extract phase: It is used to extract data from different points of the verification environment. It will take all data from scoreboard and extract it.

Check phase: It is used for check any unexpected condition in verification environment.

Report phase: It shows the report of the particular test.

Final phase: It shows that all the phase are completed and terminate the simulation.

All phases run top down in the hierarchy only the connect phase is bottom up.

UVM resource and configuration data base:

The configuration and resources classes provide access to a centralized database where type specific information can be stored and received. The uvm_resource_db is the low level resource database which users can write to or read from. The uvm_config_db#(T) is layer on top of the resource database and provides a typed interface for configuration setting that is consistent with the uvm_component. Configuration interface information can be read from or written to the database at any time during simulation. A resource may be associated with a specific hierarchical scope of a uvm_component or it may be visible to all components regardless of their hierarchical position.

```
uvm_resource_db #(virtual dut_if) ::set (
        “dut_ifs”, “ dut_vi”  dut_if1) ;
```

Where,
Virtual dut_if - type of value
“Dut_ifs” - scope
“dut_vi” - field name
Dut_if1 - value

By this syntax, it automatically call the set method of the resource database.

Same , way data can be set in the configuration data base. Configuration data base in the layer on top of the resource databse.

Create a new resource, write a *Val* to it, and set it into the database using *name* and *scope* as the lookup parameters.

```
uvm_config_db #(virtual dut_if) ::
 set (     this , “*”, “dut_vi”, dut_vi) ;
```

Where,
Virtual dut_if - type of value
this - prefix
“*” - path
“Dut_ifs” - scope
“dut_vi” - field name
Dut_vi - value

Both have the same sturucter for the get methods , path in get and set method should be match also the field name should be same.

4. Example of simple verification environment of sequence detector fsm in UVM :

Now , simple UVM environment is constructed step by step to verify the fsm using all the components defined above.

First step is to construct the interface that will connect the verification environment with the dut. In this all the dut signals are defined along with clk and reset. You can define clocking block to avoid timing violations.

Before moving to next step , it is important to learn few syntax that will help in configuring the environment from top. UVM has one facility called factory. So every component created in the environment must be registered with the factory for the configuration from the top. Factory is basically , you can defer the creation of object to run time so can modify the object created from the top level.

Any component that extends the uvm_component base class needs `uvm_component_utils macro to register it with factory and any class that extends the uvm_transaction class will need `uvm_object_utils macro to register it with factory. All the objects are created using type_id::create method from the factory that is the syntax difference from the oop languages which use new keyword. Also to be noted is that every class has its constructor. There is predefined syntax for constructors of object classes and component classes.

Next step is to create sequence item class. Sequence item or transaction is the group of inputs to be provided to device under test. It is a unit of transfer between different uvm components. So other components need to parameterize by transaction class object. In this example only one input is there so it is defined in sequence item using rand keyword for giving random inputs to dut. Reset is not randomize and defined as bit.

```
import uvm_pkg::*;

class Transaction extends uvm_sequence_item;

rand bit in;

bit rst=0;


`uvm_object_utils_begin(Transaction)

    `uvm_field_int(in, UVM_ALL_ON)

`uvm_object_utils_end

function new(string name = "Transaction");
    super.new(name);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
endfunction : new

endclass : Transaction
```

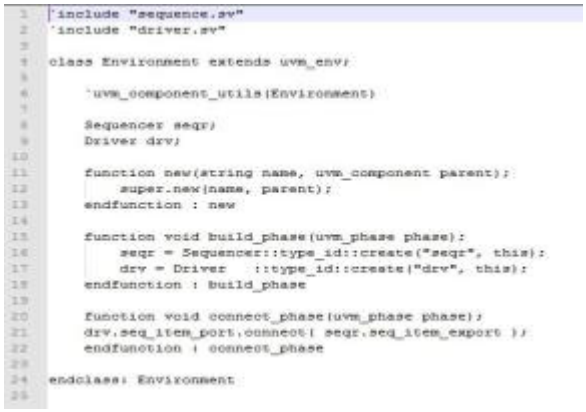
Next step is generate a stream of transaction to be applied to dut so this is done in sequence class. This

class will provide sequence item to driver as and when it ask for it. Here we are generating 20 random inputs to be applied to dut.

It can be noted that every sequence class has a body task synonymous to run_phase in other components so logic should be define in body task.
The object of uvm_sequencer class is also created that will run the sequences and communicate with the driver.

Next step is to create driver , that will drive the signal to the dut via interface. It will handshake with the sequencer for the transaction. get_next_item method called in driver will in tern call start_item method in sequence and item is returned to driver by finish_item method. Then driver will generate pin signals using the item received from sequencer and then call the item_done method as shown below. This process will continue till all objections are released or simulation is finished.

Now, we need to connect the driver with the sequencer usually it is done in agent. But this is a simple example which does not have scoreboards and other components so we are connecting driver and sequencer in top level environment. The connection is done using the connect method where port on driver is connected to export of sequencer.



Next step is to create a test class that will start the sequence on the sequencer in its run phase. It will also instantiate environment in its build phase. This class is connected to dut and interface in the top module.

In the top module , dut and test are instantiated and connected using the interface. Interface is also passed to the verification environment using configuration database.

Results :

After building whole environment , It is run on VCS tools from synopsys that will result in following result:

5. Conclusion:

It can be concluded that it is not easy to build a robust and reusable verification environment from the scratch. We need support from the base classes and a proper framework to construct a verification environment that can be understood and reused by others. UVM provided just that. It provide a rich set of base class library and features required for efficient verification.

6. Acknowledgement:

This work was greatly supported by Gujarat Technological University and Seer Akademi by providing state of the art tools like VCS and Design Compiler from Synopsys

7. References:

[1] Young-Nam Yun, Jae-Beom Kim, Nam-Do Kim, Byeong Min ," Beyond UVM for practical Soc Verification" , year 2011,pp 158-162.

[2]http://www.accellera.org/downloads/standards/uvm/uvm_users_guide_1.1.pdf

[3]http://www.doulos.com/content/events/easierUVM.php

[4] A Prectical Guide to Adopting UVM

[5] UVM cookbook