Universal Verification Methodology (UVM)

1. UVM Testbench Architecture:

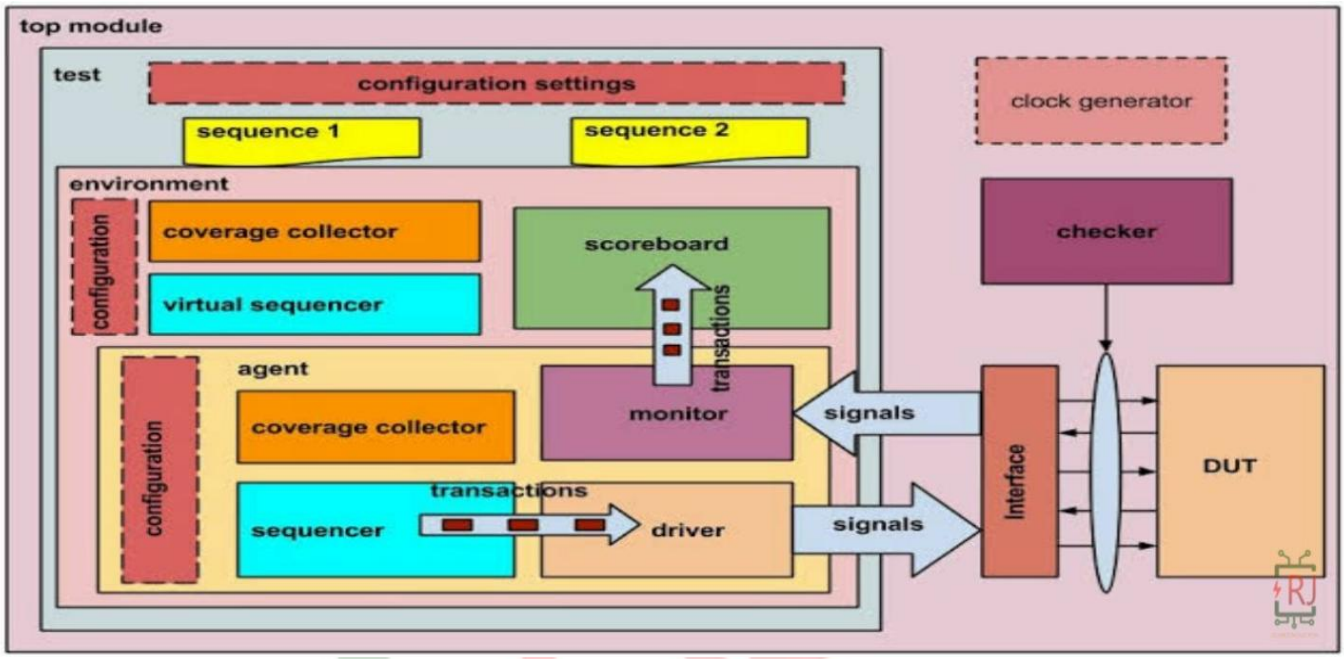

Figure 1: Generic UVM TB to verify DUT

2. UVM Testbench/ Top Module:

The UVM Testbench typically instantiates the Design under Test (DUT) module and the UVM Test class, and configures the connections between them. If the verification collaterals include module-based components, they are instantiated under the UVM Testbench as well. The UVM Test is dynamically instantiated at run-time, allowing the UVM Testbench to be compiled once and run with many different tests.

3. UVM Test:

The UVM Test is the top-level UVM Component in the UVM Testbench. The UVM Test typically performs three main functions: Instantiates the top-level environment, configures the environment (via factory overrides or the configuration database), and applies stimulus by invoking UVM Sequences through the environment to the DUT. Typically, there is one base UVM Test with the UVM Environment instantiation and other common items. Then, other

individual tests will extend this base test and configure the environment differently or select different sequences to run.

4. UVM Environment:

The UVM Environment is a hierarchical component that groups together other verification components that are interrelated. Typical components that are usually instantiated inside the UVM Environment are UVM Agents, UVM Scoreboards, or even other UVM Environments. The top-level UVM Environment encapsulates all the verification components targeting the DUT.

---

5. UVM Scoreboard:

The UVM Scoreboard's main function is to check the behavior of a certain DUT. The UVM Scoreboard usually receives transactions carrying inputs and outputs of the DUT through UVM Agent analysis ports (connections are not depicted in Figure 1), runs the input transactions through some kind of a reference model (also known as the predictor) to produce expected transactions, and then compares the expected output versus the actual output. There are different methodologies on how to implement the scoreboard, the nature of the reference model, and how to communicate between the scoreboard and the rest of the testbench.

6. UVM Agent:

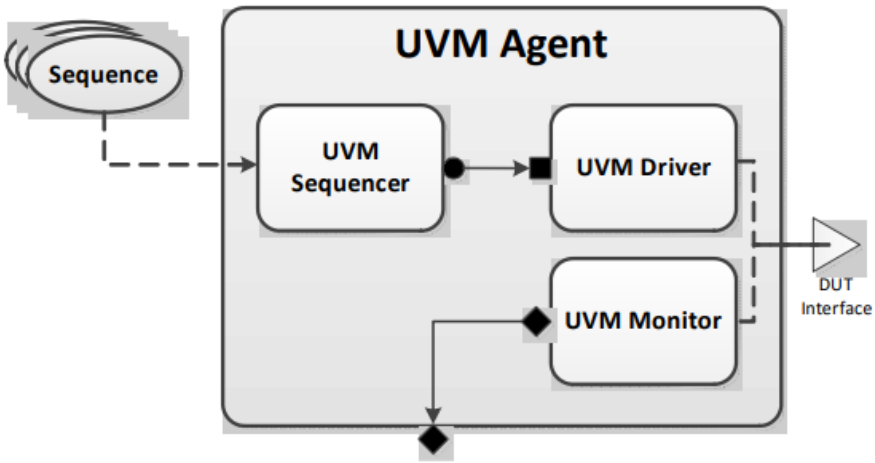

Figure 2: UVM agent

The UVM Agent is a hierarchical component that groups together other verification components that are dealing with a specific DUT interface (see Figure 2). A typical UVM Agent includes a UVM Sequencer to manage stimulus flow, a UVM Driver to apply stimulus on the DUT interface, and a UVM Monitor to monitor the DUT interface. UVM Agents might include other components, like coverage collectors, protocol checkers, a TLM model, etc. The UVM Agent needs to operate both in an active mode (where it is capable of generating stimulus) and a passive mode (where it only monitors the interface without controlling it).

7. UVM Sequencer:

The UVM Sequencer serves as an arbiter for controlling transaction flow from multiple stimulus sequences. More specifically, the UVM Sequencer controls the flow of UVM Sequence Items transactions generated by one or more UVM Sequences.

8. UVM Sequence:

A UVM Sequence is an object that contains a behavior for generating stimulus. UVM Sequences are not part of the component hierarchy. UVM Sequences can be transient or persistent. A UVM Sequence instance can come into existence for a single transaction, it may drive stimulus for the duration of the simulation, or anywhere in-between. UVM Sequences can operate hierarchically with one sequence, called a parent sequence, invoking another sequence, called a child sequence. To operate, each UVM Sequence is eventually bound to a UVM Sequencer. Multiple UVM Sequence instances can be bound to the same UVM Sequencer.

9. UVM Driver:

The UVM Driver receives individual UVM Sequence Item transactions from the UVM Sequencer and applies (drives) it on the DUT Interface. Thus, a UVM Driver spans abstraction levels by converting transaction-level stimulus into pin-level stimulus. It also has a TLM port to receive transactions from the Sequencer and access to the DUT interface in order to drive the signals.

10. UVM Monitor:

The UVM Monitor samples the DUT interface and captures the information there in transactions that are sent out to the rest of the UVM Testbench for further analysis. Thus, similar to the UVM Driver, it spans abstraction levels by converting pin-level activity to transactions. In order to achieve that, the UVM Monitor typically has access to the DUT interface and also has a TLM analysis port to broadcast the created transactions through. The UVM Monitor can perform internally some processing on the transactions produced (such as coverage collection, checking, logging, recording, etc.) or can delegate that to dedicated components connected to the monitor's analysis port.

11. UVM Virtual sequencer:

A virtual sequencer is a top-level sequencer that connects and controls multiple lower-level sequencers (e.g., for different master agents) in a UVM testbench. It allows coordinated stimulus generation across multiple interfaces or modules, enabling complex test scenarios like parallel or interleaved transactions. It doesn’t directly drive a driver but helps in managing multiple sequence flows using a virtual sequence.

12. UVM Sequence Item/Transaction:

A uvm_sequence_item represents a single transaction or operation to be driven on the DUT interface. It is a class that defines the fields (such as address, data, control signals) used in communication and is extended to model protocol-specific operations like AXI read/write or APB transfers. It's the core element passed from sequence → sequencer → driver.

13. Checker:

A checker is a functional block, typically implemented using SystemVerilog Assertions (SVA) or procedural checks, to ensure that protocol rules and expected behaviors are followed during simulation. Checkers can validate handshake timing, ordering constraints, and response rules, and are often placed near interfaces or transaction monitoring points in the testbench.

14. Interface:

An interface in SystemVerilog is a construct that groups related signals (like clock, reset, read/write controls) and optionally includes clocking blocks, modports, or tasks/functions. In UVM, interfaces connect the DUT and testbench, allowing drivers and monitors to access and drive/sense signals consistently and cleanly.

15. Coverage:

Coverage is used to measure and track how thoroughly a testbench has exercised the DUT's functionality. In UVM, coverage can be functional (e.g., burst types, ID combinations) or code-based (e.g., line, toggle coverage). Covergroups are used to define coverage models, and they help ensure corner cases and all protocol behaviors are tested.

16. Configuration Block:

A configuration block is a user-defined class in UVM that holds settings or parameters (like address ranges, burst types, ID width, etc.) that are shared across testbench components. It enables flexible and reusable testbenches by allowing components to fetch their configuration using uvm_config_db, without tight coupling.

The UVM Class Library:

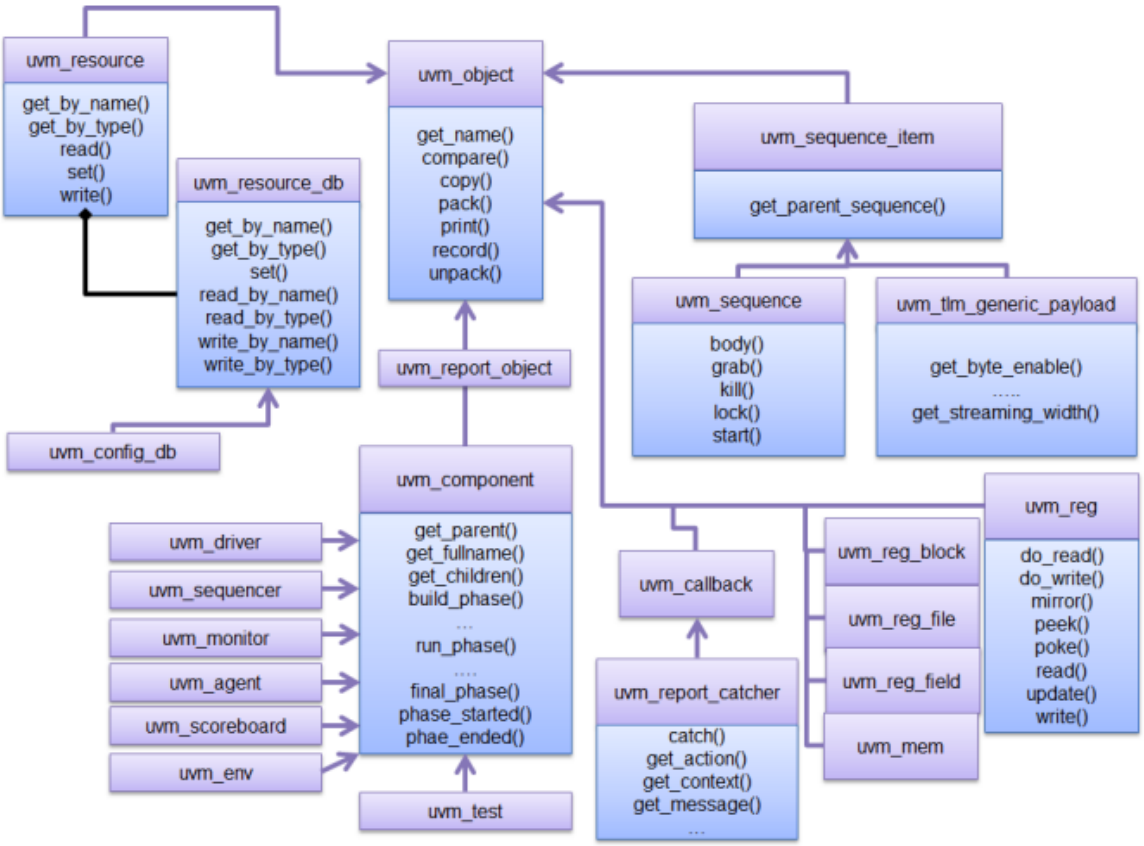

Figure 3: UVM Class Library

The UVM Class Library provides all the building blocks you need to quickly develop well-constructed, reusable, verification components and test environments. The library consists of base classes, utilities, and macros. Figure 3 shows a subset of those classes. Components may be encapsulated and instantiated hierarchically and are controlled through an extendable set of phases to initialize, run, and complete each test. These phases are defined in the base class library, but can be extended to meet specific project needs

The advantages of using the UVM Class Library include:
a) A robust set of built-in features—The UVM Class Library provides many features that are required for verification, including complete implementation of printing, copying, test phases, factory methods, and more.
b) Correctly-implemented UVM concepts—Each component in the block diagram in Figure 1 and Figure 2 can be derived from a corresponding UVM Class Library component. Using these baseclass elements increases the readability of your code since each component's role is predetermined by its parent class.

The UVM Class Library also provides various utilities to simplify the development and use of verification environments. These utilities support configurability by providing a standard resource sharing database. They support debugging by providing a user-controllable

messaging utility for failure reporting and general reporting purposes. They support testbench construction by providing a standard communication infrastructure between verification components (TLM) and flexible verification environment construction (UVM factory). Finally, they also provide macros for allowing more compact coding styles.