ENCS5337: Chip Design Verification

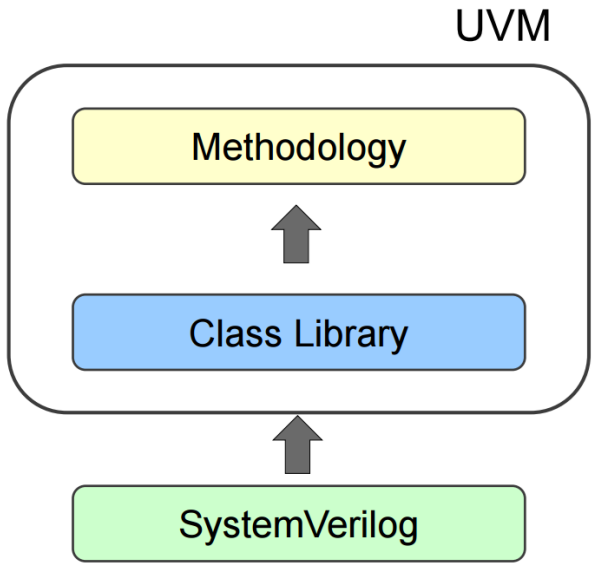
Spring 2023/2024

UVM (Universal Verification Methodology) TestBench Structure

Dr. Ayman Hroub

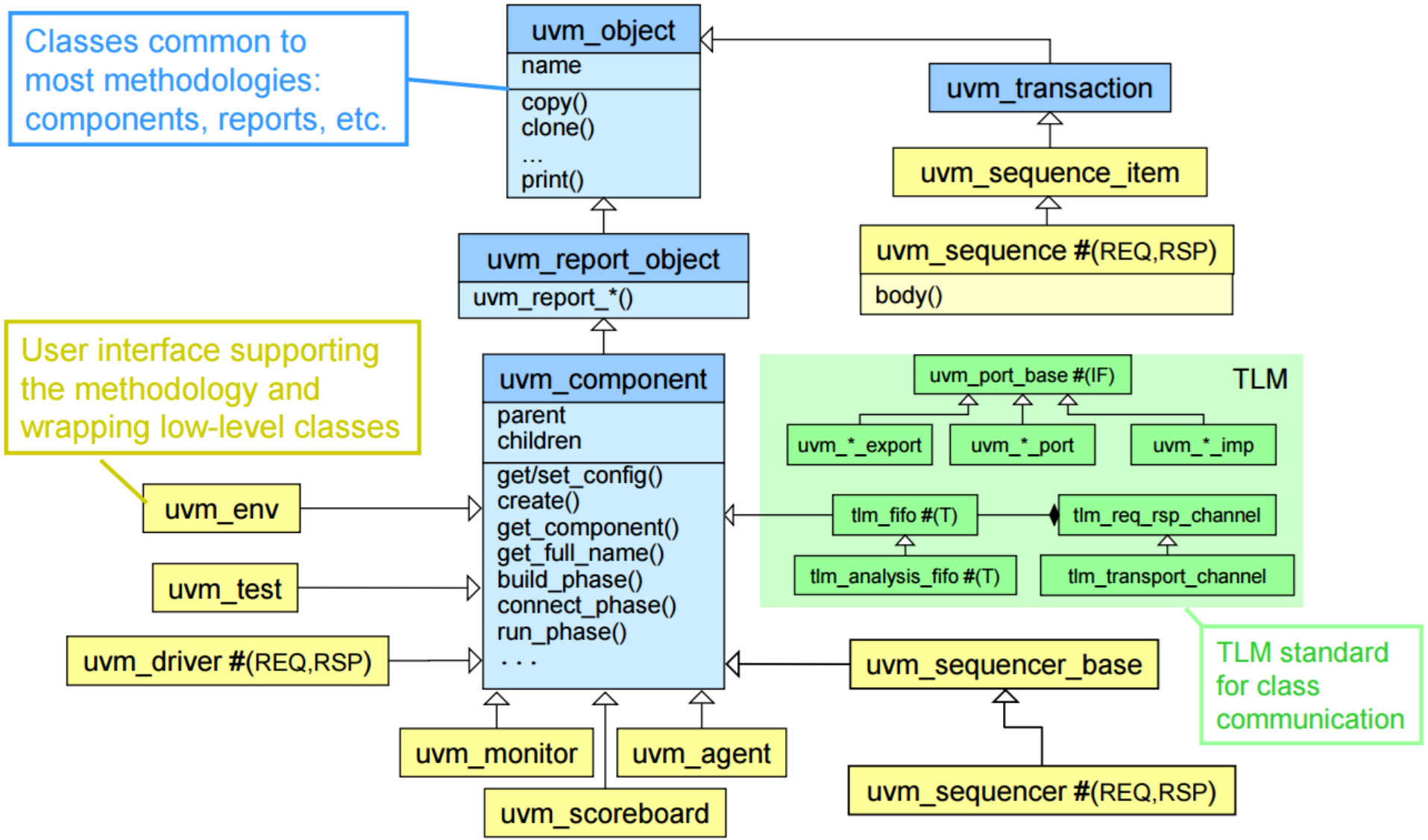
What is UVM?

- A class **library** of verification building blocks
  - Written in standard IEEE1800 SystemVerilog

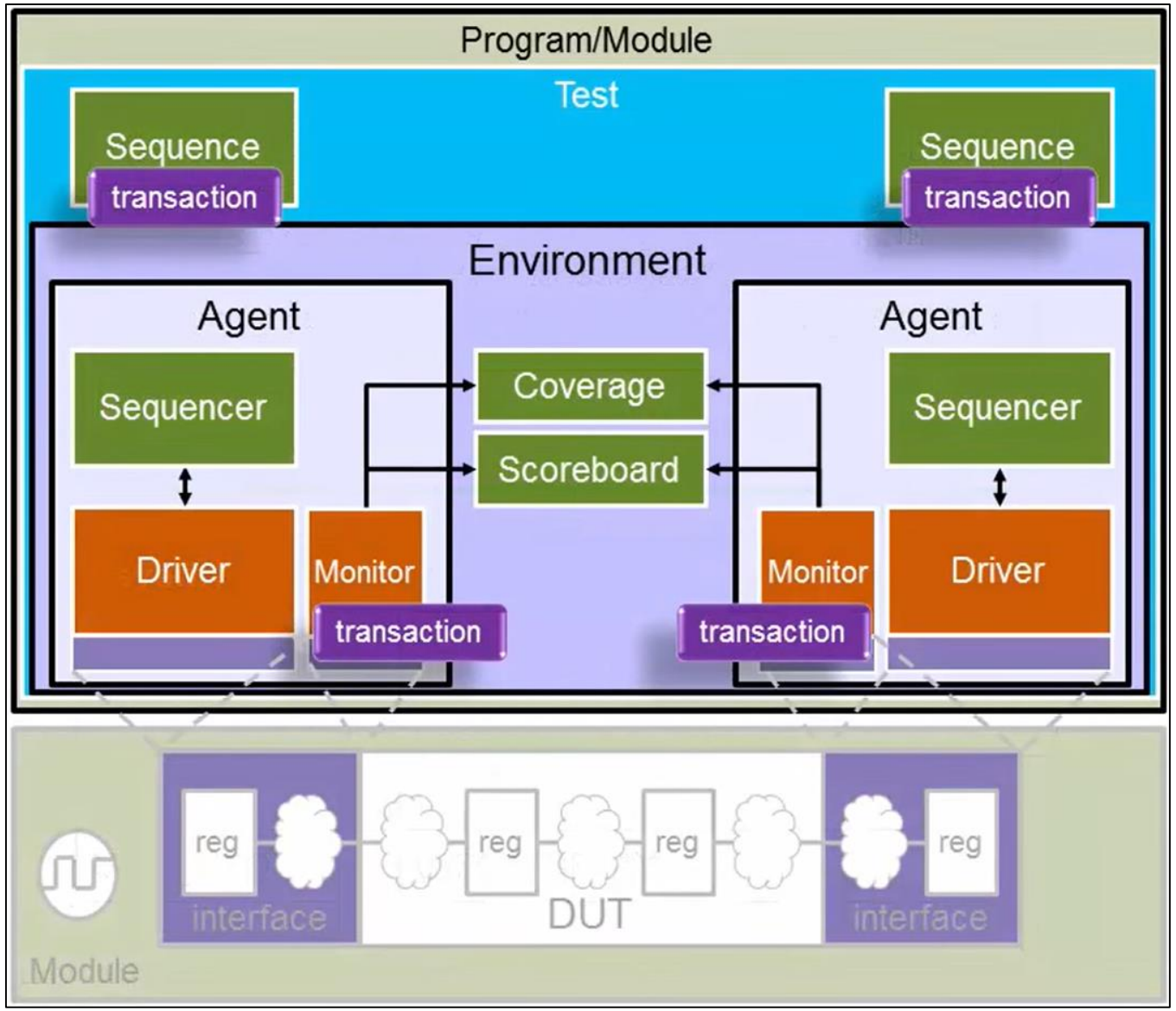


- A proven verification **methodology**
  - Defines how to use the class library
  - Scalable from block-level to system-level verification

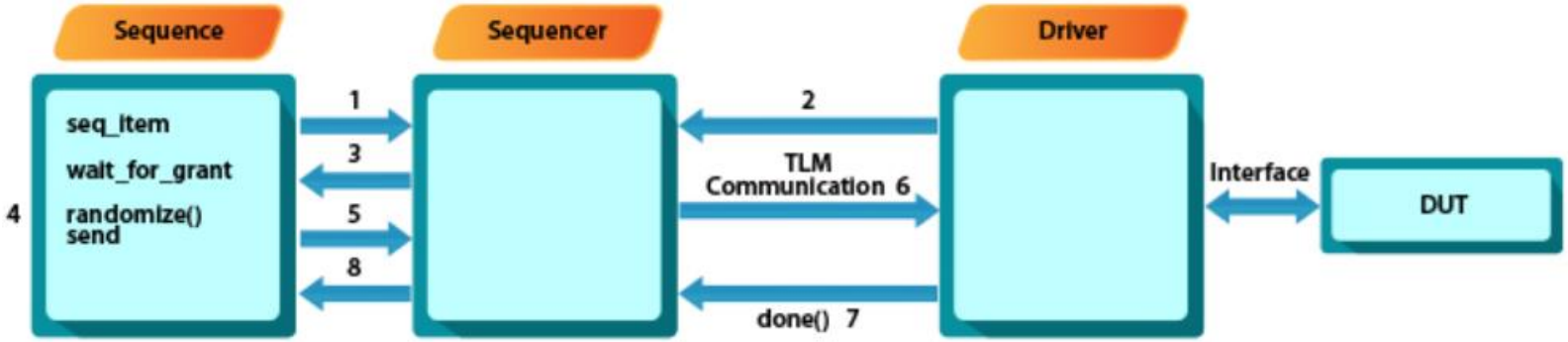
Simplified UVM Class Hierarchy



UVM Testbench Structure

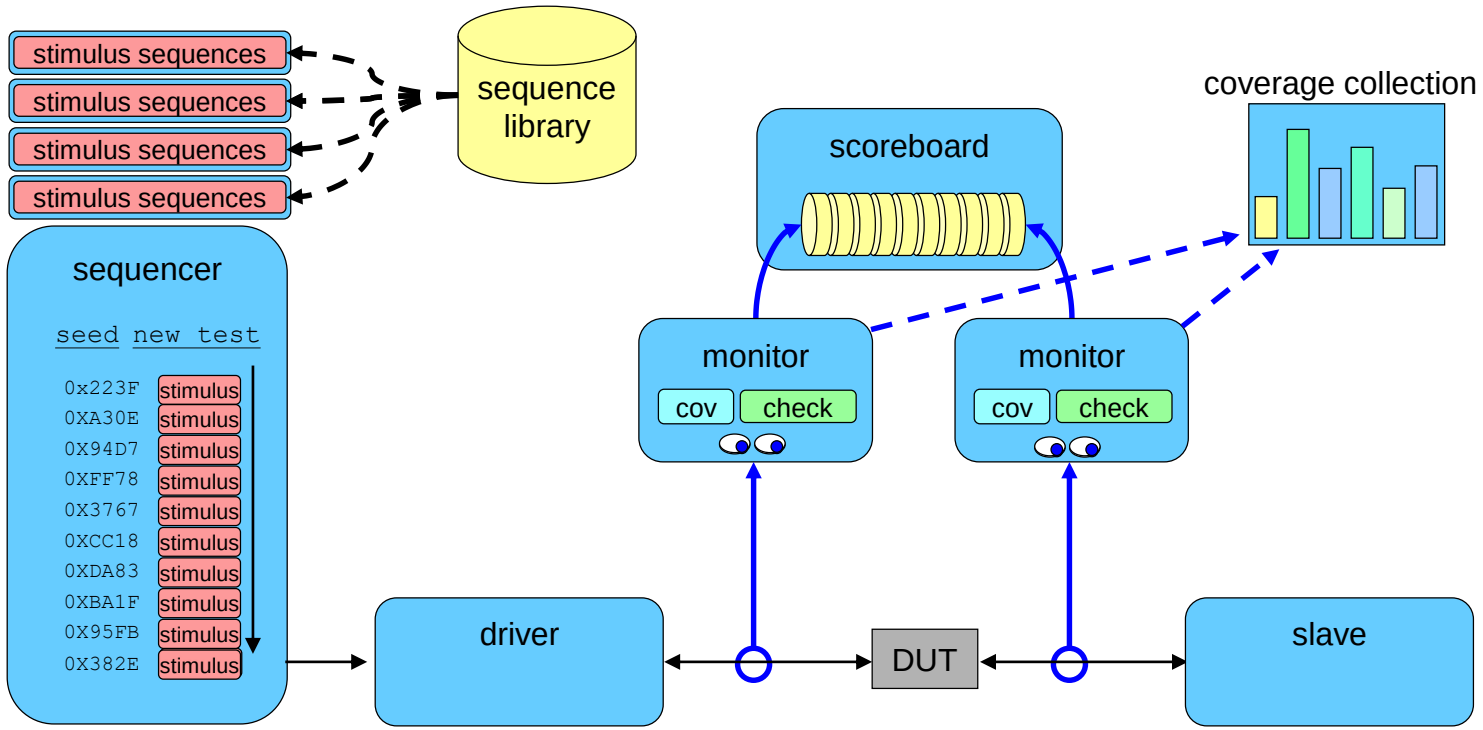


Communication between Sequence, Sequencer, and Driver



1. send request to sequencer
2. data_request from driver
3. grant request to sequence
4. randomize seq_item
5. send seq_item to sequencer
6. send seq_item to driver
7. DUT response via driver to sequencer (item_done)
8. item_done signal to sequence

Coverage Driven Environment



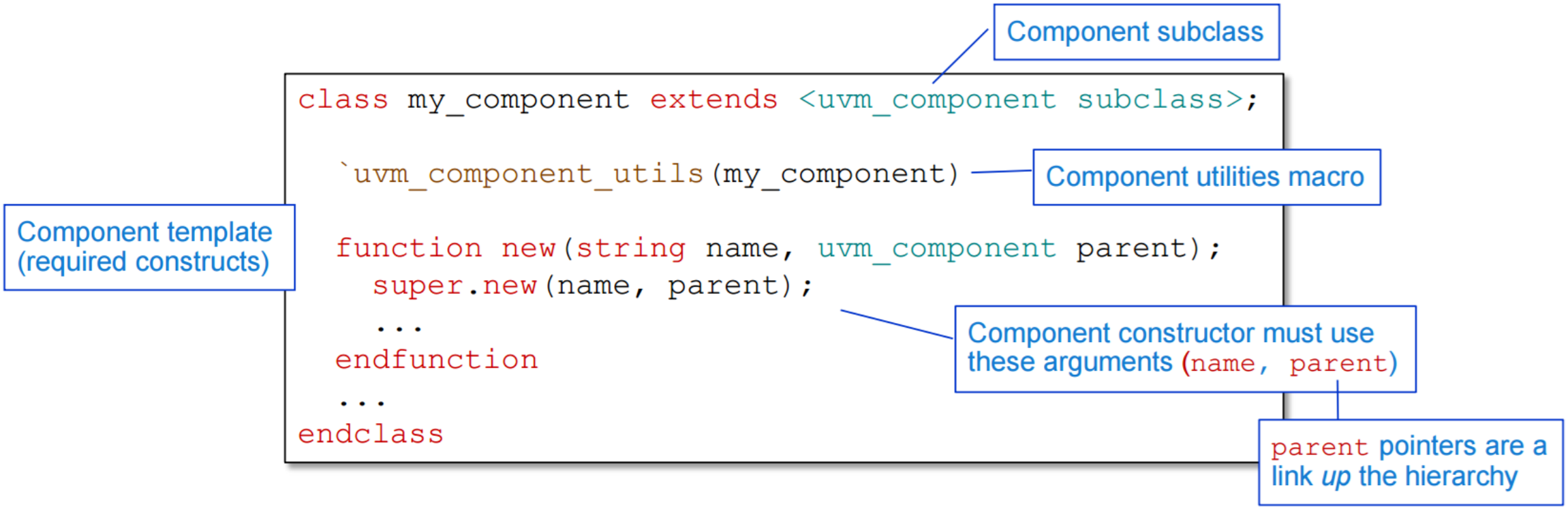
UVM Simulation Phase Names and Descriptions

build_phase	Build top-level testbench topology
connect_phase	Connect environment topology
end_of_elaboration_phase	Post-elaboration activity (e.g., print topology)
start_of_simulation_phase	Prepare for simulation
run_phase	Task – run-time execution of the test
extract_phase	Gathers details on the final DUT state
check_phase	Processes and checks the simulation results
report_phase	Simulation results analysis and reporting
final_phase	Tie up loose ends, close files

Why UVM?

- Reusability
- Scalability
- Maintainability

UVM Component Template



Sequence Item (Transaction)

- The **sequence item** is a class-based abstract transaction representing the lowest level of stimulus passed from a **sequence** to a **driver**

- The **sequence item** is also known as a **transaction**.

Sequence

- The sequence is a class-based representation of one or more stimulus items which are executed on a driver.

- Sequences can represent temporal succession of stimulus, or parallel tracks of competing or independent stimulus on more than one interface.

Sequencer

- The sequencer is a component responsible for coordinating the execution of stimulus in the form of sequences and sequence items from a parent sequence
- It ultimately feeds a driver component with transactions.

- At its simplest, a sequencer can be thought of as an arbiter. It arbitrates who gets access to the driver, which represents who gets access to the interface.

Driver

- The **Driver** is a verification component responsible for taking transactions written at a particular level of abstraction and converting them to a lower-level of abstraction, according to the protocol being verified.

- A typical driver would accept a transaction and convert it to signal changes.

Monitors (1)

- Monitors are TB components that observe the inputs, outputs, or internals of the DUV.
  - Monitors watch activity of the DUV.
    - Black box: DUV inputs and outputs
    - Grey box: potentially selected internals
  - Monitors can convert low-level signals to transactions.
  - Monitors can flag simple timing and protocol errors.
  - Monitors collect functional coverage.
  - Monitors update the scoreboard.
  - Monitors don’t drive DUV pins; they are “passive”.

Monitors (2)

- In other words, the monitor is a component responsible for watching pins wiggle on a bus and converting those pin wiggles back into transactions.
- The transactions are then sent out through an **analysis_port**, that may be connected to a scoreboard, a predictor, a coverage class, etc.
- The monitor doesn't do checking directly.

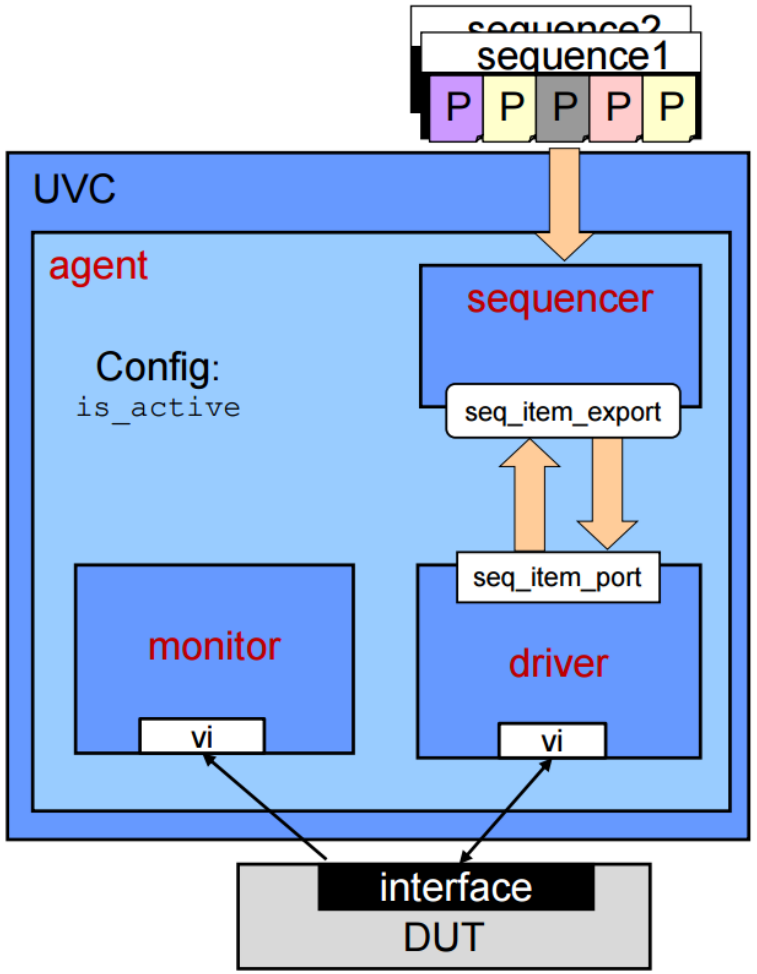
Types of Monitors

- Input monitors:
  - Collect inputs to the DUV and pass them to scoreboard.
  - Can have checker components.
- Output monitors:
  - Observe the outputs from the DUV and pass them to the scoreboard.
  - Can have checker components.
- Coverage monitors:
  - Collects inputs, outputs and selected internal signals.

Agent (1)

- An **Agent** is a verification component for a specific logical interface (it can be a SystemVerilog interface)
- Agents can be configured as either active or passive
- Active agent
  - generate stimulus and drive to DUT
  - It consists of all the three components driver, sequencer, and monitor
- Passive agent
  - It monitor activity on the interface, but do not drive the DUV
  - It consists of only the monitor

Agent (2)


sequence2
sequence1
P P P P P
UVC
agent
Config:
is_active
sequencer
seq_item_export
seq_item_port
monitor
vi
driver
vi
interface
DUT

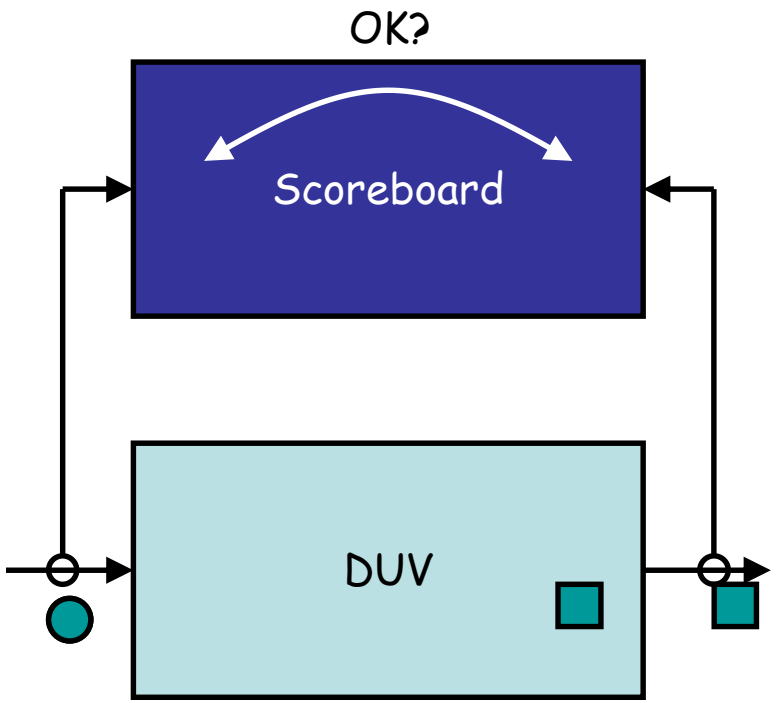
Scoreboards (1)

- Scoreboards are smart data structures that keep track of events in the DUV during simulation
- Usually, scoreboards are global
  - One scoreboard per verification environment
- Scoreboards are not checking mechanisms, but
  - The main purpose of using scoreboards is for checking
  - In practice, many checkers are implemented inside scoreboards
  - There are many typical checks that are done with scoreboards
- The scoreboard checks the operation of the DUT by collecting input and output data, and checking that the outputs received are compatible with the inputs sent.
- The scoreboard may also record statistical information and report that information back at the end of a simulation.

Scoreboards (2)

- Scoreboards source information from
  - the inputs and outputs of the DUV, and
  - occasionally also from internal events in the DUV.
- Types of checks enabled using a scoreboard:
  - Matching outputs with inputs
    - No loss of data
      - Detect inputs with no matching output.
    - No creation of data
      - Detect output with no matching input.
    - No unintended modification of data
  - Timing specification
    - Delay from input to output remains within specified limits.
  - Data order

Scoreboard Operation



Test (1)

- A **Test** is a class-based representation of a verification scenario.
- It is the top level verification component in UVM
- There are usually many test classes for each DUT, applying different stimulus or configurations to achieve different verification goals.
- Many test classes are declared in the same file (**Test library**)
- We can select which test class to execute by passing the test type name as a string argument to the **`run_test()`** method.

Test (2)

- There is no test class handle declared, or test class instance created explicitly
- When the **`run_test ()`** task is executed from an **`initial`** block in the topmost SystemVerilog module, implicitly creates a declaration and an instance of the test class type which is passed to it.
- If the test class cannot be found or is undefined, a fatal runtime error is issued
- Also, you can use command line to select a test. This command takes precedence over an argument to **`run_test()`**

```
%xrun ... +UVM_TESTNAME=mytest
```

Report Macros

```
`uvm_info(string id, string message, int verbosity)
`uvm_warning(string id, string message)
`uvm_error(string id, string message)
`uvm_fatal(string id, string message)
```

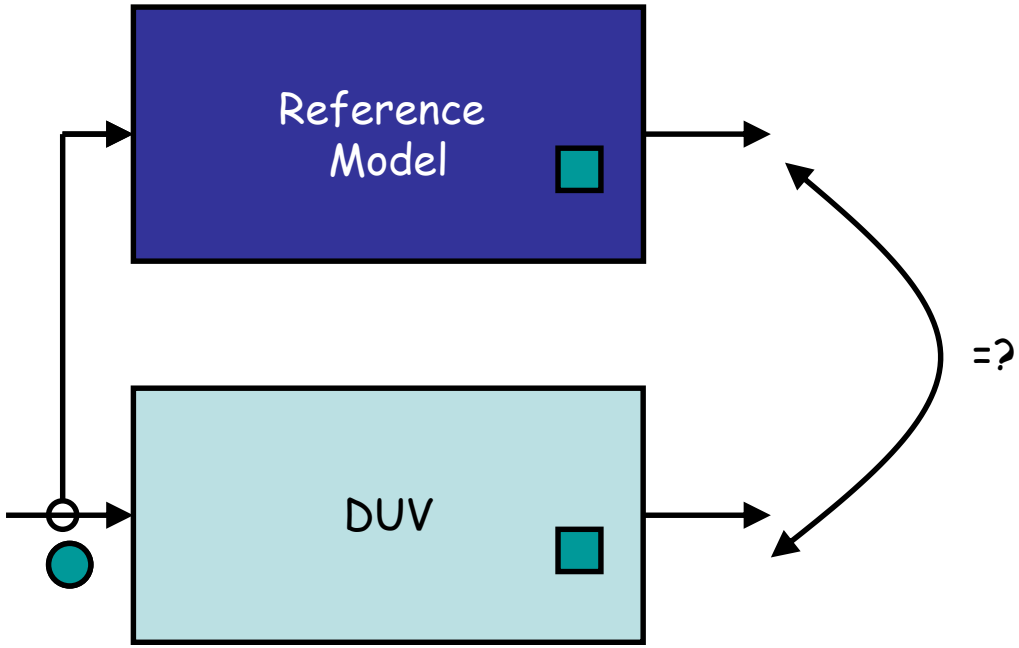
Reference Models (1)

- A reference model is an executable specification - a golden model that predicts how the DUV should behave
  - Usually in the form of an alternative implementation
- It runs in parallel to the DUV, using the same inputs and provides the checking mechanisms with information about the expected behavior
  - Checking is done by comparing the expected behavior to the actual one
- Pure reference models can run independently of the DUV
  - But not all reference models are pure.

Reference Models (2)

- Reference models have many uses
  - Checking
  - Aids for stimuli generation *(When?)*
  - “Smart” protocol models – imitate the function of the DUV
  - Vehicles for SW development
- What can we check with a reference model
  - In principle, anything
  - In practice it depends on the level of detail and accuracy of the reference model
    - And how much of its behavior we are willing to expose

Reference Model Operation

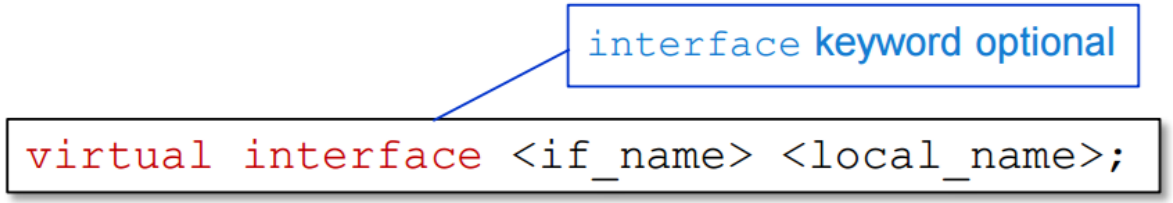


Levels of Abstraction

- The level of abstraction in a reference model dictates the type of information we can get out of it for checking
  - Functionally accurate models can be used only to check correctness of data, usually at the end of the test or at well defined points in time
    - Timing, order, and other checks need other means
  - Cycle accurate models can be used for checking all aspects of I/O behavior
  - Cycle accurate and latch accurate models can be used also for checking the internal state of the DUV
    - This type of model is sometimes called deep function reference model

SystemVerilog Virtual Interface

- UVM components cannot be directly connected to interface instances
  - Breaks reusability
  - Interface instances are static
- SystemVerilog virtual interface:
  - An interface variable that can be connected to an interface instance
  - Can be declared as a class property
  - Access interface signals using virtual interface as a prefix
  - Needs to be connected to an actual interface



Complete Tutorial

- UVM TestBench architecture - Verification Guide
- https://www.edaplayground.com/x/5r89