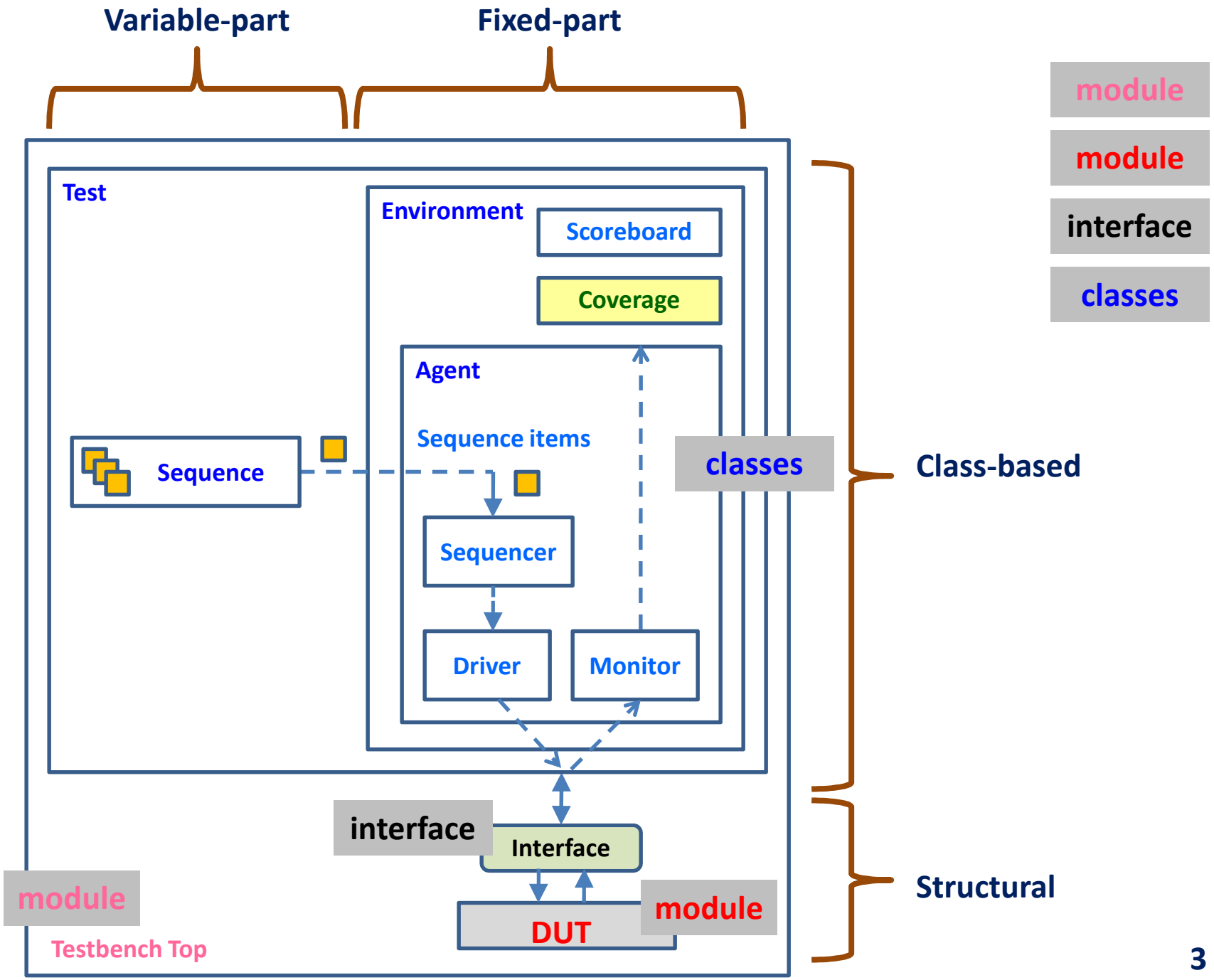
UVM (Universal Verification Methodology)

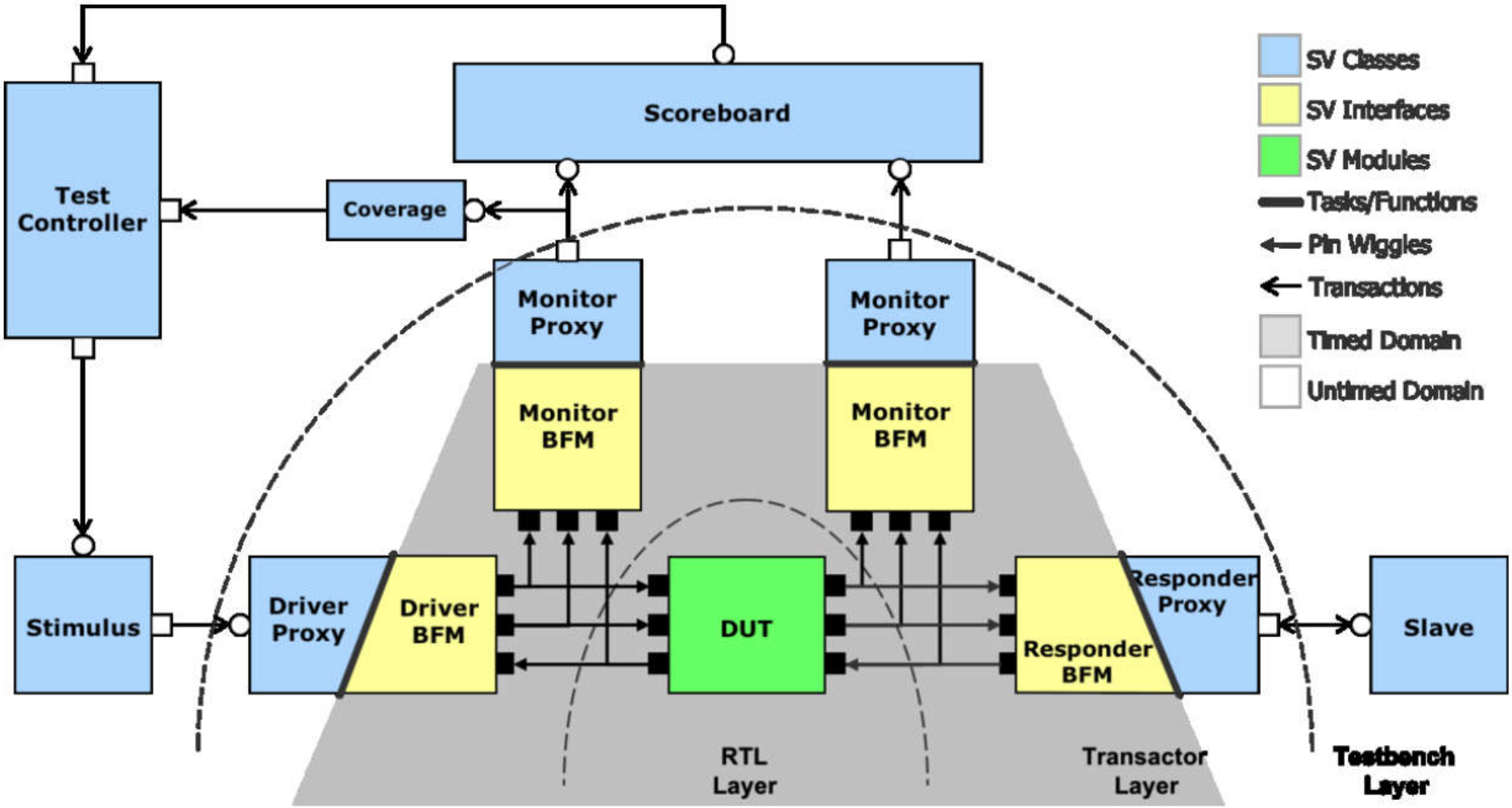
For whom can know how to program

Tuan Nguyen-viet

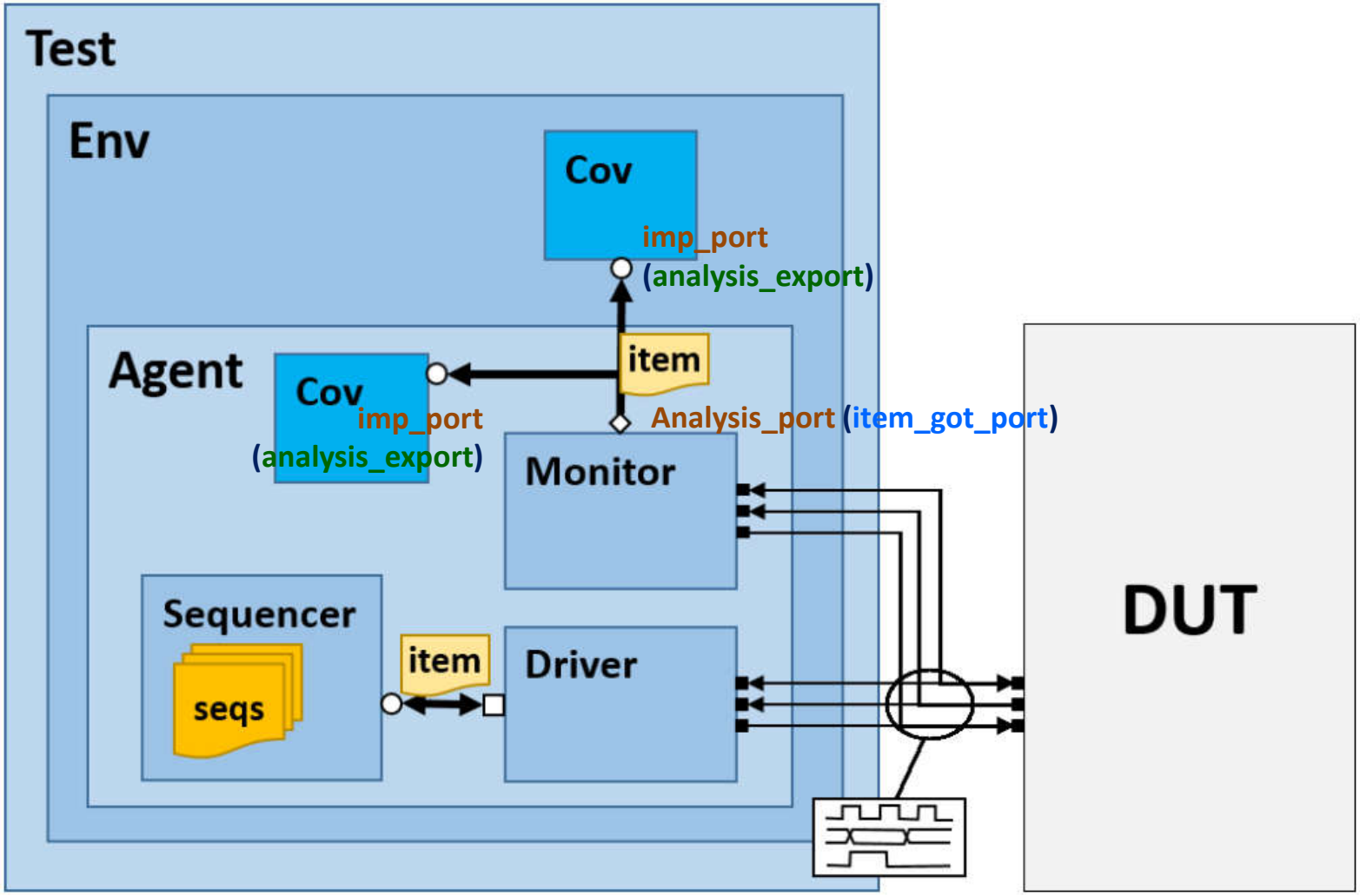
UVM Testbench Basic


Variable-part
Fixed-part
module
module
interface
classes
Test
Environment
Scoreboard
Coverage
Agent
Sequence items
Sequence
classes
Class-based
Sequencer
Driver
Monitor
interface
Interface
module
DUT
module
Testbench Top
Structural

UVM Testbench Architecture (Cookbook)



Coverage Collectors (Cookbook)



Coverage Collectors (2)

- The **uvm_monitor** is responsible for passively observing the pin-level behavior on the **DUT interface**,
  - converting it into **sequence items**
  - and providing those **sequence items** to analysis components
    1. in the **Agent**
    2. or elsewhere in the **testbench**
       - such as
         - » **coverage collectors**
         - » or **scoreboards**.

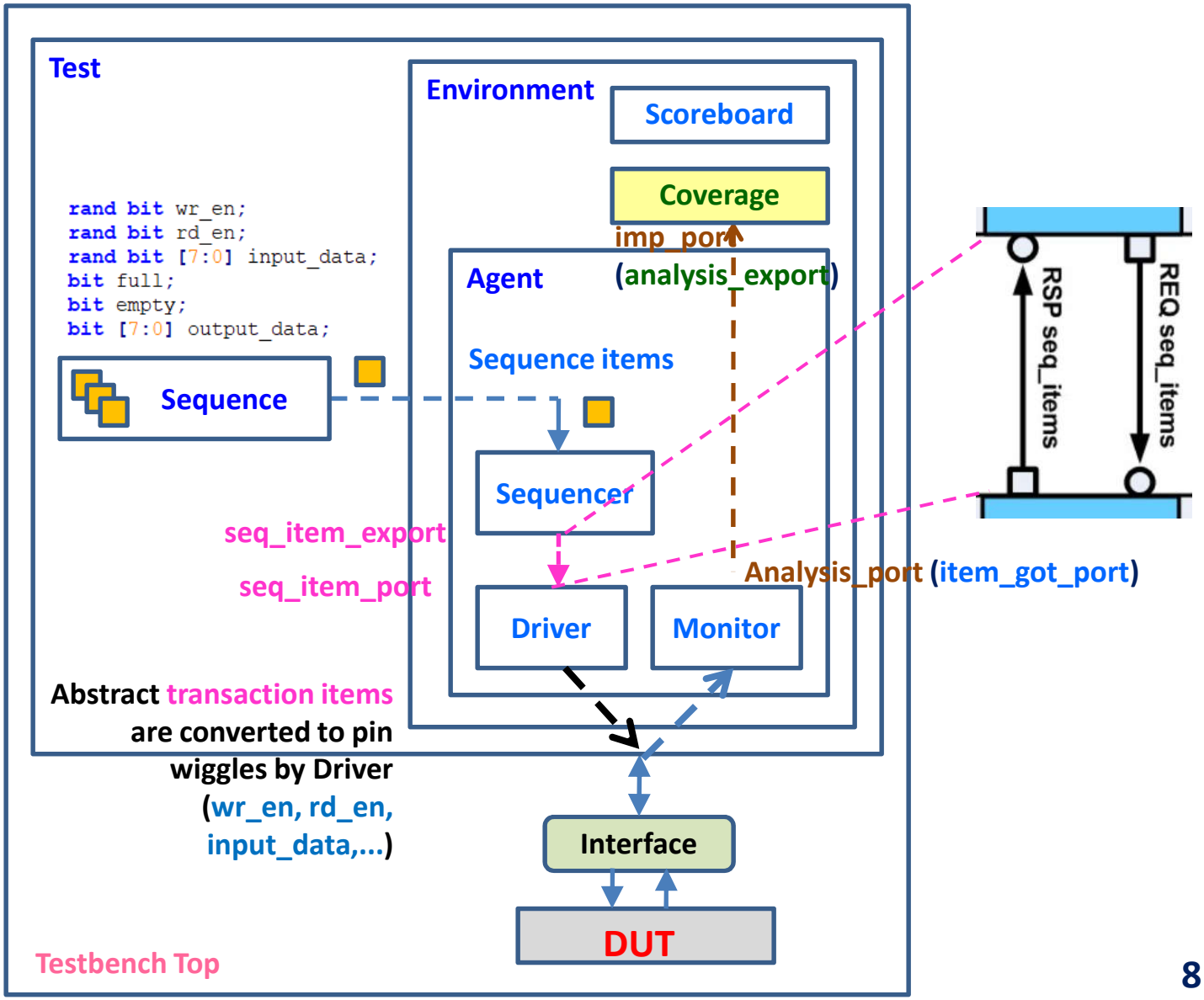
Coverage Collectors (3)

- **Coverage information** is paramount to answer the questions
  - "Are we done testing yet?"
  - or "Have we done enough testing yet?”

- **Coverage collectors** are *analysis port* **subscribers** that sample
  - observed **transactions**
  - and activity into SystemVerilog *functional coverage groups*.

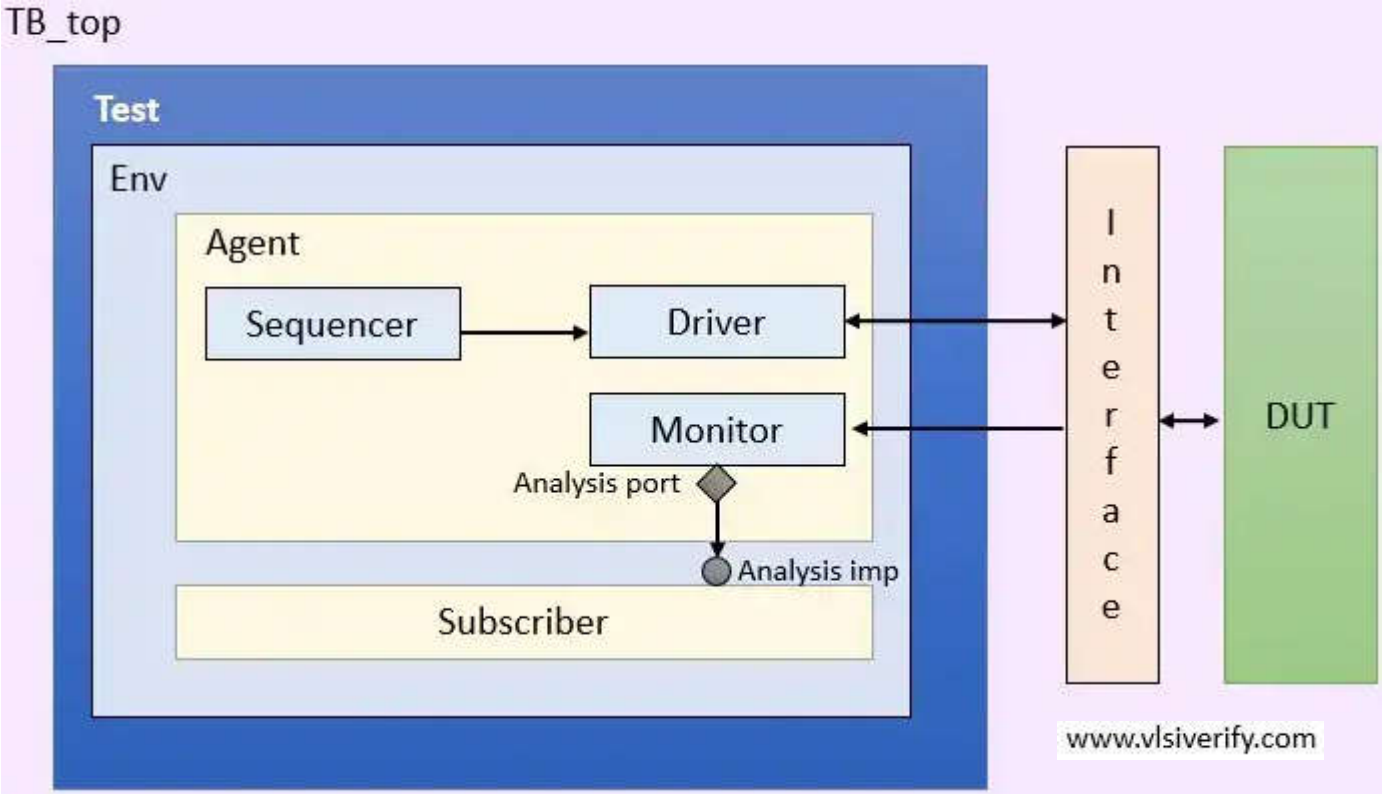
- The ***coverage data*** collected from each test
  - is stored into a *shared coverage database*
    - used to determine overall verification progress.

Transaction / Sequence Item: Functional Coverage



Subscriber Class

- The **uvm_subscriber** class provides an ***analysis export*** (Analysis imp) that connects with the *analysis port*. **Subscribers** are basically listeners of an *analysis port*.
- **They** subscribe to a broadcaster (i.e. **monitor**'s *analysis port*)
  - and receive objects/broadcasted transactions whenever an item is broadcasted via the connected *analysis port*. (see the figure below).



Subscriber Class (2)

- A **uvm_component** class does not have an built-in *analysis port*,
  - while a **uvm_subscriber** is an extended version
    - with an *analysis port* named **analysis_export**.

```
1  virtual class uvm_subscriber #(type T=int) extends uvm_component;
2      typedef uvm_subscriber #(T) this_type;
3
4      uvm_analysis_imp #(T, this_type) analysis_export;
5
6      function new (string name, uvm_component parent);
7          super.new (name, parent);
8          analysis_export = new ("analysis_imp", this);
9      endfunction
10
11     pure virtual function void write (T, t);
12 endclass
```

REF: https://www.chipverify.com/uvm/uvm-subscriber

Coverage Class

- The **uvm_subscriber** is derived from **uvm_component**
  - and adds up the ***analysis_export*** port in the class

```
virtual class uvm_subscriber #(type T=int) extends uvm_component;
uvm_analysis_imp #(T, this_type) analysis_export;
```

- The **uvm_subscriber** class defines the write method

```
pure virtual function void write(T t);
```

- Since **uvm_subscriber** has built-in ***analysis_export***, it is generally used to implement a functional coverage monitor.
  - The user-defined subscriber (**sfifo_coverage.svh**) is derived from **uvm_subscriber**.

```
class sfifo_coverage extends uvm_subscriber #(sfifo_seq_item);
```

- An **agent** has a TLM *analysis port* for it's **monitor**
  - to share the data object, which is collected on the **agent**'s *interface*,
    - with other testbench components, e.g. user-defined coverage class:

```
// Monitor behaves as a broadcaster.
f_agt.f_mon.item_got_port.connect(f_cov.analysis_export);
```

Coverage Class (2)

➔ Summarily, for the use case, it is easier:

1. to create a **user-defined class** (**sfifo_coverage**) inherited from **uvm_subscriber** (extends from **uvm_component**)
2. and use the built-in *analysis_export* implementation
   - to connect to the *analysis port* of the **agent** at the environment (see sfifo_environment.svh)

- Implementation:
  - See the file named **sfifo_coverage.svh**.

Functional Coverage (FC)

- *Functional Coverage* checks the correctness of the design by collecting values (or sets of values) of design variables during the simulation.

- ➔ It is a **user-written code** that observes the execution of the test plan.
  - It ensures that a test did what was intended,
    - particularly with *randomisation*.
  - When the test plan has been executed,
    - testing can be declared as complete.

Code Coverage (CC)

- *Code Coverage* is mainly described
  - as **simulator** (e.g. Modelsim/Questasim, etc.) collected metrics
    - on code that has been run.
- While it identifies what has ***not*** been run,
  - it cannot identify the code
    - ➔ that was ***not*** implemented as per the **specification**.
- By nature, *Code Coverage* analyses
  - the **structural correctness** of the code,
    - ➔ ***not*** the **functional correctness**.

Code Coverage and Functional Coverage

- Combining *Code Coverage* and *Functional Coverage* provides us with an overall set of testing metrics for the **DUT**.

- *Code Coverage* can be used as a catch-all
  - to indicate whether something got missed in *Functional Coverage*.

- Therefore, we can safely say that:

  Test Done = **100%** *Code Coverage* + **100%** *Functional Coverage*

Functional Coverage and Assertion Coverage

- An **assertion** in Verilog/SystemVerilog/VHDL is a statement or condition that is checked during the *simulation or formal verification* of a digital circuit design.
- The main purpose of **assertions** is to **validate** the functionality and behavior of a digital circuit design.
- When a signal of the design is **asserted** that means
  - it is active,
  - it is doing what its name says.
- To **assert** a **signal** means to bring it to its activated state. We can use **assertion** to describe the state of both active high or active low signals.
  - For example,
    - the signal RESET is **asserted** when it is active, which is HIGH logic.
    - the signal RESET_n is **asserted** when it is active, which is LOW logic.
- **Assertion** level is the voltage level in a **logic circuit** that represents a logical "1".
  - Common level for High = +5v/1.2v/etc. (**process**) and Low = 0v.
- Implementation:
  - See file named **sfifo_sequence.svh** for assertion of signals

Functional Coverage Report

File D:/Questasim_projects/sfifo_uvm_6e2/covhtmlreport/pages/__frametop.htm

Testplan | Design | DesUnits

- tb_top (no coverage)
- uvm_pkg (no coverage)
- sfifo_agent_pkg (no coverage)
- sfifo_sequence_pkg
- sfifo_environment_pkg
- sfifo_test_pkg (no coverage)
- tb_top_sv_unit (no coverage)
- questa_uvm_pkg (no coverage)

Questa Coverage Report

Number of tests run:	1
Passed:	1
Warning:	0
Error:	0
Fatal:	0

List of tests included in report...

List of global attributes included in report...

Coverage Summary by Structure:

Design Scope	Coverage
sfifo_sequence_pkg	100.00%
sfifo_sequence	100.00%
sfifo_environment_pkg	100.00%
sfifo_coverage	100.00%

Coverage Summary by Type:

Total Coverage:					100.00%	100.00%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Covergroups	20	20	0	1	100.00%	100.00%
Assertions	3	3	0	1	100.00%	100.00%

Report generated by Questa on Thursday 12 September 2024 04:23:44

Code Coverage (simulator)

First **compile** the design with *cover options*:

b – Collect branch statistics.

c – Collect condition statistics.

e – Collect expression statistics.

s – Collect statement statistics. Default.

t – Collect toggle statistics.

- Cannot be used if 'x' is specified.

x – Collect extended toggle statistics

- Cannot be used if '**t**' is specified(toggle coverage)

f – Collect FSM statistics.

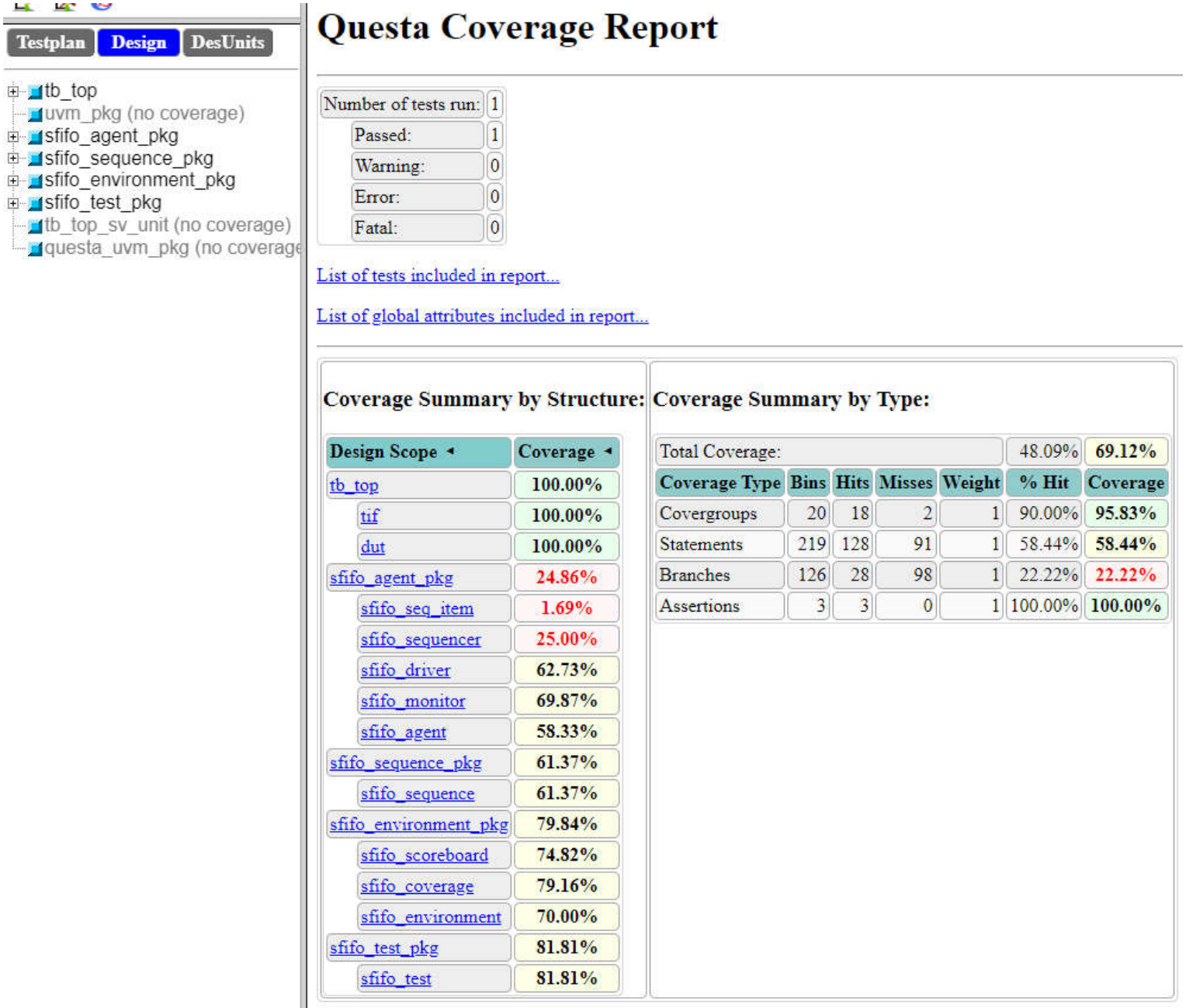
1. Run vlog, for two examples:

```
vlog -cover bcs *.v
vlog -cover bcs work.tb_top
```

2. Start vsim with coverage switch, for two examples :

```
vsim -coverage test_design
vsim -coverage work.tb_top
```

Functional Coverage Report (2)



Testplan | Design | DesUnits

- tb_top
  - uvm_pkg (no coverage)
  - sfifo_agent_pkg
  - sfifo_sequence_pkg
  - sfifo_environment_pkg
  - sfifo_test_pkg
  - tb_top_sv_unit (no coverage)
  - questa_uvm_pkg (no coverage

Questa Coverage Report

Number of tests run:	1
Passed:	1
Warning:	0
Error:	0
Fatal:	0

List of tests included in report...

List of global attributes included in report...

Coverage Summary by Structure:

Design Scope	Coverage
tb_top	100.00%
tif	100.00%
dut	100.00%
sfifo_agent_pkg	24.86%
sfifo_seq_item	1.69%
sfifo_sequencer	25.00%
sfifo_driver	62.73%
sfifo_monitor	69.87%
sfifo_agent	58.33%
sfifo_sequence_pkg	61.37%
sfifo_sequence	61.37%
sfifo_environment_pkg	79.84%
sfifo_scoreboard	74.82%
sfifo_coverage	79.16%
sfifo_environment	70.00%
sfifo_test_pkg	81.81%
sfifo_test	81.81%

Coverage Summary by Type:

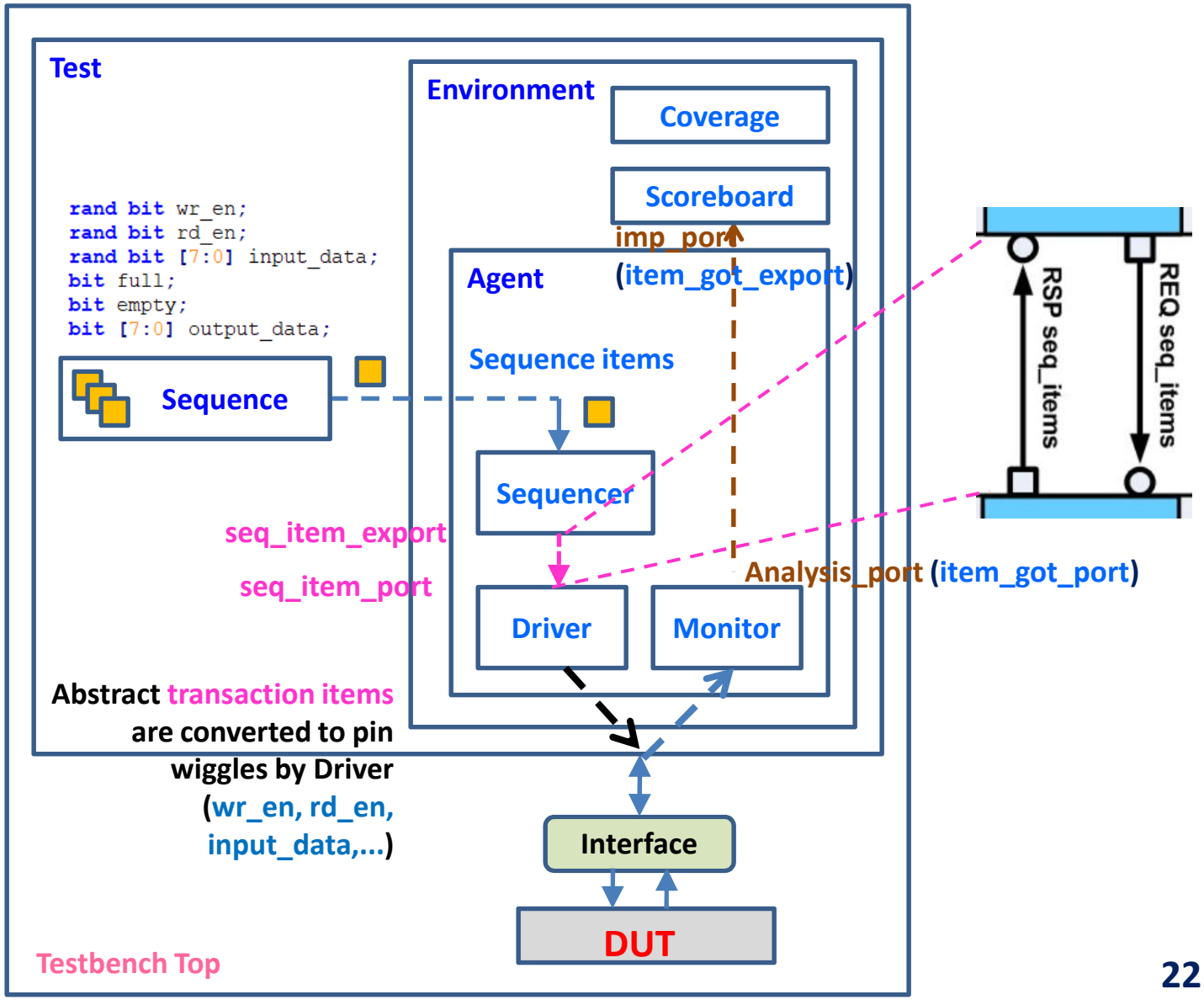
Total Coverage:					48.09%	69.12%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Covergroups	20	18	2	1	90.00%	95.83%
Statements	219	128	91	1	58.44%	58.44%
Branches	126	28	98	1	22.22%	22.22%
Assertions	3	3	0	1	100.00%	100.00%

To improve UVM Assertion and Coverage skills

- Books
  1. SystemVerilog Assertions and Functional Coverage by Ashok B. Mehta
  2. Coverage Cookbook by Mentor Graphics Verification Methodology Team
- Need to practice and learn from different sources and examples
  1. Online resources such as
     - blogs,
     - forums,
     - tutorials,
     - and documentation
  2. Use sample codes such as those provided by
     - Mentor Graphics,
     - Cadence,
     - or Synopsys,
- ➔ to see how different test/scenarios and challenges are handled by UVM experts

BACKUP SLIDE

Transaction / Sequence Item: Scoreboard



Thank You