1 Introduction

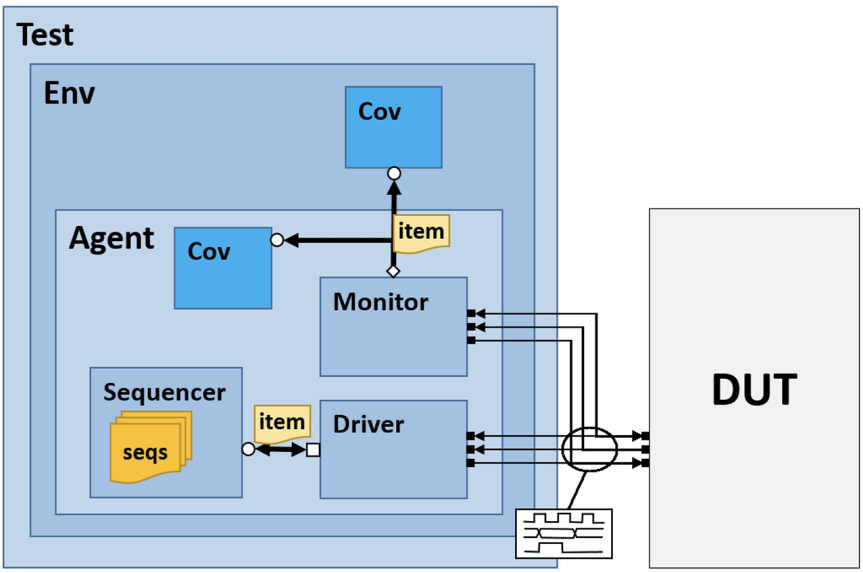


Figure 1: UVM testbench

The main focus of this document is to create an easy and complete implementation of a generic testbench according to the *Dual-Top pattern*. UVM employs a layered, object-oriented approach used to design a modular and reusable verification environment. This paradigm entails the separation of concerns while building the testbench architecture and the specific test that can be performed: multiple tests can be applied to the same architecture just by modifying the input stimuli sequence (and constraints) and by selecting different Design Under Test (DUT) properties to be observed. Alternatively, same tests can be applied to different testbench architectures.
The Dual-Top architecture consists of splitting the architecture into a verification environment (TB), based on SystemVerilog (SV) classes that rely on TLM (Transaction Level Modeling) protocols, and the HDL domain containing the DUT and its interface, used with Drivers and Monitors to translate transactions into pin signals and vice-versa.

For all the DUTs, namely the simple adder, the MBE-Dadda multiplier and the pipelined floating-point (FP) multiplier, the same TB architecture was used. The interface example was modified to clarify the Master and Slave modports and it was also reused in all the tests.
The DUT module provided in the example is exploited, which is actually a wrapper that includes a Finite-State Machine (FSM) processing the valid-ready signals. The additional latency of the FP multiplier leaded to extend the simple interface provided in the assignment with a trivial pipeline for the handshake between Master and Slave sides. However, for the combinational units only, the wrapper was marginally modified by adding an output register to consider also the case in which the slave is not ready to sample the results and they must be retained. Different solutions for DUT/Driver pairs were evaluated to test the design correctness, by using the DUT module provided in the example and by modifying the Driver BFM to support the delayed comparison. We also provide a slightly different version of the DUT that take advan-

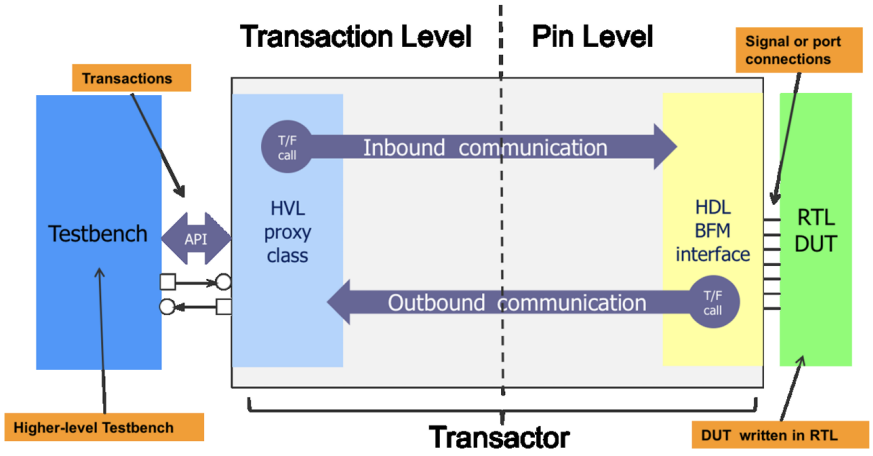


Figure 2: UVM Dual-Top pattern

■ **modport** Master

- Inputs
  - ▷ clk
  - ▷ rst
  - ▷ ready
- Outputs:
  - ◁ res
  - ◁ valid

■ **modport** Slave

- Inputs
  - ▷ clk
  - ▷ rst
  - ▷ a
  - ▷ b
  - ▷ valid
- Outputs:
  - ◁ ready

Figure 3: DUT Interface

tages of an output register to retain data and facilitate the handshake procedure. In practice, we tried to move the synchronization procedure from the driver BFM to the low-level design under test. We effectively verified the equivalence of different solutions.

For the testbench's components, a *Scoreboard* class grouping both *Predictor* (reference model) and *Comparator* was developed, too. The comparator itself was modified by using the *uvm_tlm_analysis_fifo* as queue mechanism to buffer transactions from the reference model and the DUTs with different latency. Furthermore, an **objection** mechanism was implemented to ensure that all transactions driven by the test would have been evaluated.
After the analysis of the example code, the use of SV events was avoided and *record* functions were neglected.

By reusing almost all components for different DUTs, it was implemented the parametrisation of input and output data size to be propagated through the hierarchy of both HDL and TB domains. They are set in a top-level package and they can be overridden from this command-line options during simulation's launching. However, the configuration possibilities were limited

by neglecting UVM configuration objects and common OOP feature such as polymorphism. Instead, for minor changes in our project, separate objects were created.

1.1 Project organization

Files that could be reused by HDL and TB domains, are collected in the same `src` and `tb` directories respectively. Then, for each design to be verified, appropriate subdirectories contain all the remaining files to be used.

Within `tb` directory:

- ▷ `common/` contains all UVM classes shared by different DUTs.
- ▷ `add/` , `mbe/` and `fpm/` contain the different reference models and, eventually, a sequence item class to specify different constraints according to the test case.

`sim/` folder is used to collect simulation results. In this directory are also placed all **Makefiles** and the scripts needed to run simulations.

The following tree clarifies project organization in a simple, graphical way.

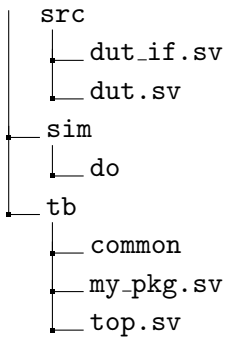


Figure 4: Project organization: directories and files

2 Testbench architecture

TB's components and objects hierarchy is reported hereafter in Fig. 5. All SV classes are included in a customed package named *my_pkg.sv*, that is going to be imported in the top-level testbench module. In general, all classes to be included in the package have ".svh" extension, whereas source files to be compiled have ".sv" one.

2.1 Top-level Testbench

The top-level *top.sv* is a SV module that instantiates the HDL top module, with DUT and its interface. It generates clock and reset stimuli, it sets the virtual interface handling to the UVM *configuration database* and it finally starts UVM testbench by calling the static method `run_test()`. This method constructs the root node of the testbench hierarchy, which defines the

```
my_test.svh
  my_env.svh
    my_agent.svh
      my_driver.svh
      my_monitor.svh
    my_agent_o.svh
      my_driver_o.svh
      my_monitor_o.svh
    my_scoreboard.svh
      my_predictor.svh
      my_comparator.svh
my_packet_o.svh
my_packet_i.svh
my_sequence.svh
my_sequencer.svh
```

Figure 5: UVM objects/components hierarchy

test case to be executed by specifying the configuration of the testbench components and the stimuli. The root node from run_test is usually a factory registered *uvm_test*. Additionally, the top module sets a UVM static variable that allows terminating the simulation run from the root node after that all UVM phases are completed: `uvm_top.finish_on_completion = 1;`

2.2 Factory registry

From an high-level view can be distinguished two different domains with SV-OOP and HDL features. In UVM, a *virtual* interface object is used to refer to the same Bus Functional Model (BFM) from both domains. To do so the Factory pattern is exploited. In general, the purpose of the UVM factory is to enable objects replacement without changing the testbench structure or even the testbench code. The mechanism used is referred to as an override. It ensures the implementation of a configuration database that is used to build the testbench hierarchy. The configuration database is exploited to create references to different objects inside the hierarchy and to tune all the configuration parameters by simple static methods' call.

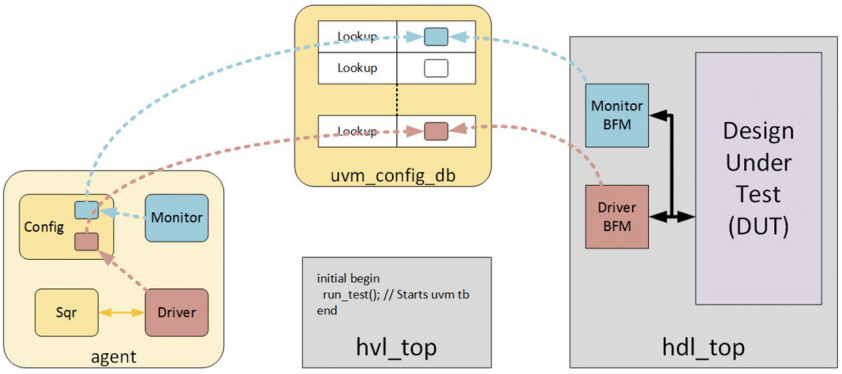


Figure 6: Configuration database look-up for virtual interfaces

In order to effectively exploit the Factory, some coding conventions are followed for all TB classes:

1. **Factory Registration**: every UVM component or transactions must contain Factory registration code generated by using Factory registration macros, as shown below in the code.

```
    // Registration for a component
    class my_comp extends uvm_component;
        `uvm_component_utils(my_comp)
        ...
    endclass: my_comp
    // Registration for a sequence
    class my_seq extends uvm_sequence #(my_txn);
        `uvm_object_utils(my_seq)
        ...
    endclass: my_seq
    // Registration for a transaction
    class my_txn extends uvm_sequence_item;
          `uvm_object_utils(my_txn)
        ...
    endclass: my_txn
```

   Factory registration macros contain the necessary code to implement the factory override mechanism:

   - A *uvm_component_registry_wrapper, which is proxy for the component registration* of generic type. It removes the need for an instance of the component itself.
   - A static function to get the type_id (override by instance)
   - A function to get the type name (override by name)

2. **Constructor defaults**: defaults constructor allows a factory registered class to be built inside the factory using the defaults and then the class properties can be re-assigned to the arguments passed via the create method of the *uvm_component_registry* wrapper class.

```

class my_comp extends uvm_component;
    `uvm_component_utils(my_comp)

    // Component Default Constructor
    function new (string name = "my_comp", uvm_component parent = null);
        super.new(name, parent);
    endfunction: new

endclass: my_comp

class my_seq extends uvm_seqeuence #(my_txn);
    `uvm_object_utils(my_seq)

    // Sequence Default Constructor
    function new (string name = "my_seq");
        super.new(name);
    endfunction: new

```

```
endclass: my_seq

class my_txn extends uvm_sequence_item;
    `uvm_object_utils(my_txn)

    // Transaction Default Constructor
    function new (string name = "my_txn");
        super.new(name);
    endfunction: new

endclass: my_txn
```

3. **Component and Object Creation**: testbench components are created during the *build_phase* using the create method of the *uvm_component_registry*. This, first constructs the class, then assigns the handle to the class to its declaration handling. For components the build process is top-down, which allows higher level components and configuration to control what actually gets built.

   Object classes are created as require, again using the create method.

```
class my_agent extends uvm_agent;
    `uvm_component_utils(my_agent)

    my_driver drv; // components handle != object
    my_monitor mon;

    // Component Constructor
    function new (string name = "my_agent", uvm_component parent = null);
        super.new(name, parent);
    endfunction: new

    // Build Phase
    function void build_phase (uvm_phase phase);
        super.build_phase(phase);
        // object creation and handle assignment
        drv = my_driver::type_id::create("drvr", this);
        mon = my_monitor::type_id::create("moni", this);

    endfunction: build_phase

endclass: my_agent
```

   When component construction takes place using `<my_type>::type_id::create` what happens is that the `type_id` is used to pick up the Factory component wrapper for the class, construct its contents and pass the resultant handle back again to the LHS.

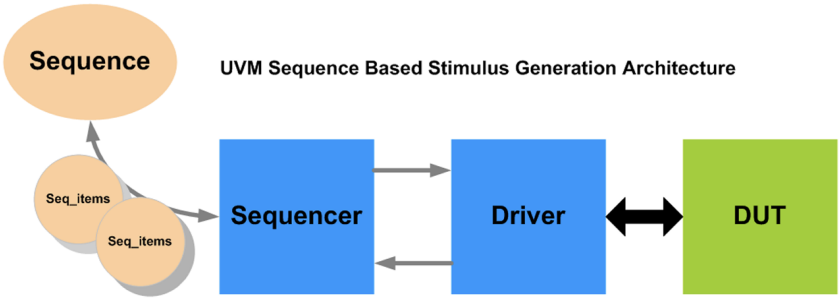
2.3 Test

Test object is characterized by two main entities:

1. *quasi*-static components hierarchy, defined by a particular UVM environment and configuration object.

2. *dynamic* transaction sequences, for the object-oriented stimuli generation.

Following the main phases of a UVM simulation, it first creates environment and sequence objects by calling the *uvm_factory_registry* create method, then, it starts the execution of *sequences* through the *sequencer* by calling the *start()* method of the Sequence API.
UVM Sequences provide an object-oriented mechanism to specify stimuli generation at the transaction level. Sequences are not able to directly access testbench resources, they are responsible for the stimuli generation flow and send sequence_items to a driver via a sequencer component. The driver is then responsible for converting the information contained within sequence_items into pin level activity. The sequencer is an intermediate component which implements communication channels and arbitration mechanisms to facilitate interactions between sequences and drivers. The flow of data objects is bidirectional, request items will typically be routed from the sequence to the driver and response items will be returned to the sequence from the driver.



In UVM we make a distinction between *components*, such as tests, environments, agents, and *objects*, based on how these classes are handled during the simulation. In fact, components are described as *quasi*-static objects, in the sense that they can only be instantiated at the beginning of the simulation, in the build phase during TB hierarchy creation. They exist for all the simulation period. On the other hand, objects like sequences, have limited simulation life-time and can therefore be described as transient objects. They are created and discarded dinamically during UVM run_phase. For this reason, *my_env* handling is assigned during the build_phase() whereas *my_sequence* is created in the run_phase() task.
Moreover, the run_phase includes the objection mechanism that allow the end of test once the sequence is ended.
In fact, a UVM testbench has a number of time consuming phases (run) associated to each component/object that is used to coordinate the simulation. End of tests occurs when all time-consuming phases have ended and each phase ends when there are no longer any pending objections to that phase. So the end-of-test in the UVM is controlled by managing phase objections. The *uvm_objection* class provides a mean for sharing a counter between participating components and sequences. Each participant may "raise" and "drop" objections asynchronously, which increase or decrease the counter value. When the counter reaches zero (from a non-zero value), "all dropped" condition occurs.

Listing 1: my_test::run_phase

```
1 task run_phase(uvm_phase phase);
```

```
  my_sequence_h = my_sequence::type_id::create("my_seq_h");

  phase.raise_objection(this);

  assert( my_sequence_h.randomize() with { n inside {[16:64]}; } )
  // start sequencer
  my_sequence_h.start(my_env_h.my_agent_h.my_sequencer_h);

  phase.drop_objection(this);

endtask // run_phase
```

2.3.1 Constrained-random verification

Listing 1 highlights the random-constrained randomization, that is usually performed for the stimuli generation. n sequences length and sequence_items properties are declared as `rand` data type to be randomized. SV constraints are applied to force specific values when different test cases must be evaluated, such as *valid* and *ready* signals for the Master-Slave protocols, or to simply define the legal data range for the DUT. SVA (SystemVerilog assertions) are used to provide a basic Functional coverage method for the testbench built.
For example, as highlighted in the code snippet, the test forces the sequence length to be `inside` the specified range. The assertion mechanism allows to interrupt the simulation whenever the randomization process is not fulfilled.

All simulation-based verification suffers from the issue that enough test vectors cannot be run to exhaustively test the whole design. One way to fix this issue is using constrained random stimuli. The use of random stimuli brings very significant benefits to discover unexpected bugs: giving enough time and resources allows to explore the entire state space of the design, free from the selective biases defined by a human test-writer. Of course, pure random stimuli would be nonsensical, so adding constraints to make random stimuli legal is an important part of the verification process, and is explicitly supported by SystemVerilog and UVM.
The best way to approach verification process is starting with simple focused (non-random) tests to bring up the design, then moving to fully random tests in order to explore the state space in a broad fashion and flush out as many bugs as possible with minimum human effort devoted to test writing. This can typically achieve much less than 100% functional coverage, and the remainder of the verification process is spent defining a series of tests, each of which constrains and shapes the random stimuli is a different way to push the design into interesting corner cases. The state space of a typical design is so vast that random stimuli alone is not enough to explore all the key use cases, yet directed or highly constrained tests can be too narrow to give good overall coverage. Constrained random stimuli is a compromise between the two extremes, but effective usage comes down to making a series of good engineering judgements. The solution is to use the priorities set in the verification plan to direct verification resources to the key areas.

2.4 Sequence

The sequence object is inherited from base class *uvm_sequence* and is parametrised with the particular transaction object on which operates. It is an example of a software development "functor" concept, that is an object which is used as a method. A UVM sequence contains a task called body. When a sequence is used, this is created, its body method is executed, and then the sequence can be discarded.
We firstly create our custom sequence_item *my_packet_i*. Then Sequence API is exploited to initiate the transaction communication via the sequence. By using the SV `repeat` block, *n* sequence_items are created and pushed to the driver by using the `start_item` method. *late randomization* is performed on the sequence item with a constraint that forces to 1 the valid signal, so that every transaction that is generated can be sampled by the DUT when ready. Now the handshake mechanism with the driver is started and the transaction is transmitted as soon as both the request from driver and start from sequence are called. It must be noted that there is no particular order by which the function calls must be made. The `finish_item` method waits for the driver response before it returns, blocking the next transaction.

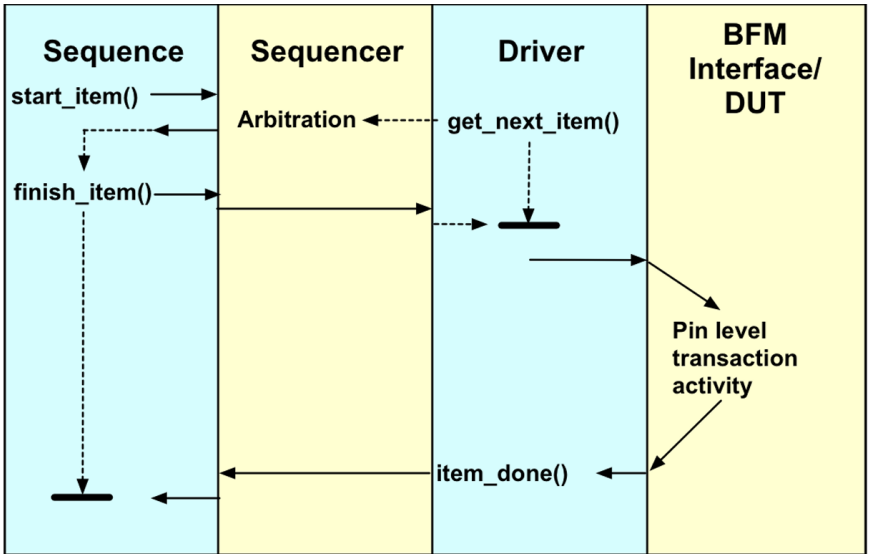


Figure 7: Execution flow for Sequence-Driver API

Listing 2: my_sequence::body

```
task body();
    repeat(n) begin
        start_item(tx);
        assert( tx.randomize() with { tx.valid > 0; });
        `uvm_info("Packet_in", $sformatf("START_ITEM A=%0h B=%0d VALID=%h", tx.a, tx.b, tx.
            valid), UVM_MEDIUM)
        finish_item(tx);
        end

    // closing

    tx.valid = 0;
    start_item(tx);
```

```
    `uvm_info("Packet_in", $sformatf("FINISH_SEQUENCE"), UVM_MEDIUM)
    finish_item(tx);

endtask // body
```

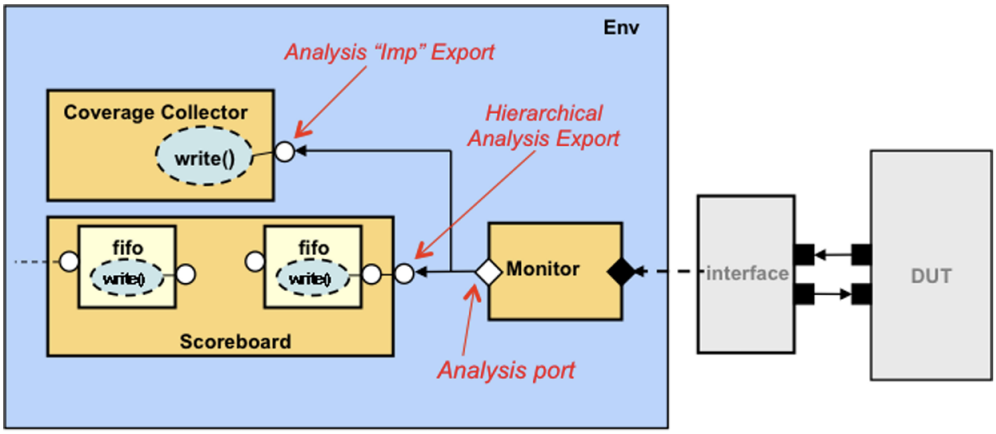
After that n samples are processed by the input agent, a *valid 0* closing sequence is sent. This is necessary because of the objection mechanism implemented in the scoreboard, so that it can be processed any sequence's length without explicitly set a drain time when the DUT has some latency.

2.5 Sequence items

It is an UVM sequence_item developed to group all i/o transactions with the DUT. As mentioned before, input stimuli are declared as `rand` 2-state data type to be used in a constrained-random verification test. The DUT output (res) is simply declared as 4-state *logic* variable. Sequence item classes only include a standard constructor and utility macros that implement data operations, such as utility functions to print a formatted string with the UVM messaging system, or a comparison method.

2.6 Environment

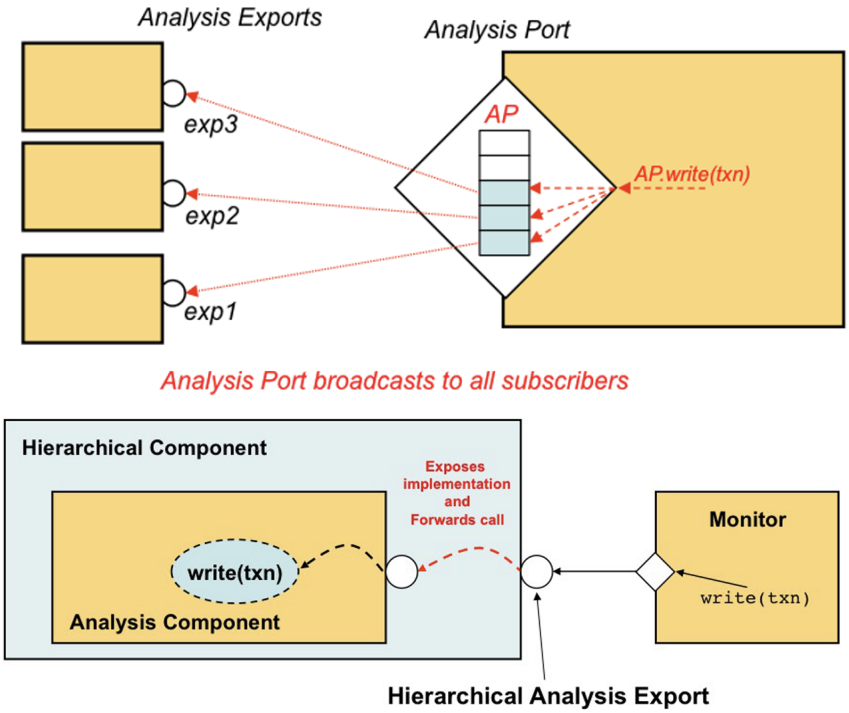
The environment class simply creates the agents and scoreboard through the usual factory method. In the connect phase it is mapped the TLM analysis Ports of both agents to two distinct analysis Exports of the scoreboard.



2.6.1 TLM Ports and Exports

Ports and Exports are the primary objects that ensure the transactional-level communication between components. In practice, these are required to establish the function call framework that is used to "pass" one transaction to other components in the design. In the analysis section, often it can be useful to broadcast items to many components (one-to-many topology) that should not interfere with DUT operations (non-blocking). UVM provides three object to meet these requirements:

- ▷ Analysis Port
- ▷ Analysis Export
- ▷ Analysis FIFO



In particular, Ports provide a local object through which the code can call a function: write requests are forwarded to other objects. Exports *actually* implement the write method. For instance, in order to connect an agent to the scoreboard, it is connected an analysis port of the monitor to an export of the predictor or comparator, so that the transaction can be stored. If transactions must be buffered, either SV queue of Exports objects, implementing custom write() functions, or the base Analysis FIFO with its methods can be used. Both buffers objects are unbounded so that the non-interference requirement is satisfied; hence, the write is always a non-blocking operation.
TLM connections are built with a standard constructor outside the Factory mechanism, since they do not have configuration parameters.

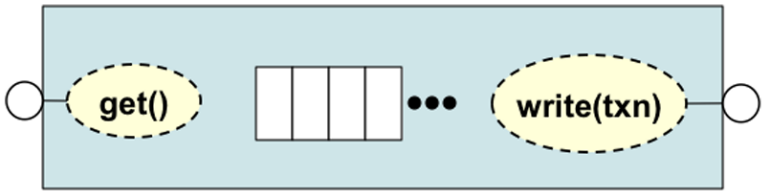


Figure 8: Analysis FIFO

Since the environment only acts as a container, TLM port/export can be directly accessed from one hierarchical component to the other one.

Listing 3: my_env::connect_phase

```
function void connect_phase(uvm_phase phase);

   my_agent_h.aport.connect(my_scoreboard_h.axport_i);
   my_agent_o_h.aport.connect(my_scoreboard_h.axport_o);

endfunction // connect_phase
```

2.7 Agents

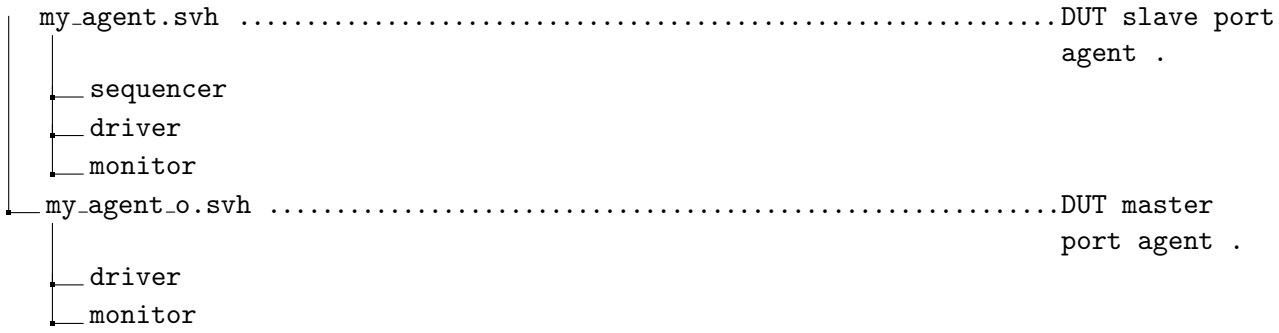


Figure 9: UVM objects/components hierarchy

The agents contain the code needed for the factory mechanism to be applied. They instantiate drivers, monitors and sequencer by calling `create` method and assign the analysis_port *aport* handle for the respective transactions (my_packet_i, my_packet_o). In the connect phase the monitor analysis_port is forwarded to the *aport* object, which was connected to the scoreboard in the environment class. Then, for the input agent the driver is connected to the sequencer. An active agent is provided for the DUT output also by defining a deterministic signal behavior to the DUT slave. In this case, no sequence is implemented. In fact, a precise test case was developed: while forcing from the sequence object an always valid transaction, *my_driver_o* simulates a slave unit that can be alternatively ready to sample DUT results or busy, performing its own elaborations.

2.7.1 Drivers

As mentioned before, the driver is responsible for the translation of sequence_items to signal activity, making the DUT interface able to understood this data. To do so, the driver retrieves the virtual interface object from the configuration database. In the run phase, DUT pins are driven by calling the `drive()` task. This task was declared as *virtual* and *protected* to eventually exploit *polymorphism* when creating new drivers for a different BFM. This solution can be further extended by separating the BFM with the actual driver object.

To decouple the agent from the actual interface it could have been implement the same write() task directly into the DUT interface. Doing so, if the verification environment to be used includes hardware-assisted acceleration platform (e.g. FPGA), driver (and monitor) implementations could have been reused. It is only the interface code which must be changed to be synthesised. This, again, enlarges the possibility that a common verification architecture can be adapted to different test cases.

Following the handshake mechanism explained in the sequence section (Fig. 7), in drive() a new sequence_item request is put to the sequencer by using the embedded `seq_item_port`. The `get_next_item()` blocking function will return as soon as the sequencer receives the Sequence request. Then, the actual pin assignment is performed.
At first, it is ensured that the driver is stalled until the reset sequence imposed by the top-level module is completed. When the test can start, the input transaction is sampled at each clock cycle only if the DUT is ready to process new data. After that, `item_done()` function is called sending a response to the sequencer which permits the sequence to return from finish_item (Fig. 7). After that each transaction is written to the DUT interface, we wait for the circuit to be ready again. In this case we keep the valid signal asserted, since the sampling is automatically handled by the DUT.

Listing 4: my_driver::drive

```
task run_phase(uvm_phase phase);
    drive();
endtask

virtual protected task drive();
    forever begin

        seq_item_port.get_next_item(tx);

        if (dut_vi_i.rst) begin
            // reset virtual interface signals
            wait(!dut_vi_i.rst);
        end
        else begin
            @(posedge dut_vi_i.clk);
            // assign virtual interface signals from tx properties
        end // else: !if(dut_vi_i.rst)
        @(posedge dut_vi_i.ready);

        seq_item_port.item_done();
    end
endtask
```

Similarly, for the output driver *my_driver_o*:

Listing 5: my_driver_o::drive

```
virtual protected task drive();
    forever begin
        if (dut_if_o.rst) begin
            dut_if_o.ready <= 1'b0;
            wait(!dut_if_o.rst);
         end
```

```
7           else begin
8              @(posedge dut_if_o.clk)
9                  dut_if_o.ready <= 1;
10                 //#DELAY dut_if_o.ready <= ~dut_if_o.ready;
11          end
12      end
13  endtask // drive
```

2.7.2 Monitors

Monitors perform the inverse function of drivers. run_phase task calls a custom `read()` method which creates a transaction object at each clock cycle, if a valid-ready pair is observed at the interface. Before receiving input data, the DUT ready and a valid transaction from the sequence have to occur. For the output results, the signals to be checked are a ready signal eventually asserted by my_driver_o and a valid flag produced by the DUT. In both cases, at the end of the same `read()` method, the transaction is written to the Scoreboard through the analysis port.

2.8 Scoreboard

The scoreboard component simply groups the prediction task and the evaluation task. It forwards analysis port write() to the Predictor and Comparator exports from the input and output agents respectively. Of course, it also provides the connection from Predictor to the Comparator, so that expected and actual outputs are evaluated.

2.8.1 Predictor

A predictor is a verification component that represents a "golden" reference model. The predictors are typical analysis component that are subscribers to transaction streams: they take input transactions and process them to produce expected output transactions. Those outputs are broadcast through analysis ports to the evaluation part of the scoreboard. The predictors inherits from the uvm_subscriber class, which provide an embedded implementation of the export required by the input transaction. Indeed, it is parametrised with my_packet_i sequence item.

run_phase task only performs the arithmetic computation from the input data and write on the output analysis port that has been created to pass output transactions (my_packet_o) to the comparator component. In our work, since each arithmetic unit requires a different reference model, three predictors were implemented: they differ just for their write function. Details of these functions are provided in the last sections with the simulation results.

2.8.2 Comparator

Comparators are simple UVM components implementing an outputs' accurate comparison. Matching transactions will appear in the same order from both expected and actual streams. Transactions arrive independently and are buffered from the "before" side, to be compared only when a transaction arrives on the "after" side. Synchronization work is performed by

instantiating two analysis FIFOs and two hierarchical exports. The comparator behavior is implemented in the run() task of the component and the FIFOs blocking operations are exploited to wait a matching pair of items. The evaluation is easily obtained by calling the transaction's compare() method implemented through their field macros.

Considering the DUT latency with respect to the predictor results, an objection mechanism must be implemented to ensure that the "before" (expected) FIFO is drained. To do so, an objection is raised after the blocking `get()` method of FIFO_before. Then, after certain delay, the actual result will be written to FIFO_after and its get method returns. After the evaluation the objection is dropped waiting for the end-of-test. As explained previously in the section, because of the objections raised inside the comparator a closing sequence must be sent to avoid deadlock condition. When the sequence ends with a valid transaction, the predictor writes data to the before side and the get method returns waiting for a valid transaction to the after side, which is in turn waiting for a valid output results from the DUT. However, this is a consequence of our decision to not de-assert the valid signal from the driver as soon as a valid-ready pair is sampled at the DUT slave port. In fact, a valid signal is logically bounded to the input packet and it does not change until a new transaction is obtained from the sequence.

Listing 6: my_comparator::run_test

```
1
2   task run_phase(uvm_phase phase);
3       my_packet_o before_tx;
4       my_packet_o after_tx;
5
6       forever begin
7           before_fifo.get(before_tx);
8           `uvm_info("FIFO_before", $sformatf("RES=%0h", before_tx.res), UVM_MEDIUM);
9           phase.raise_objection(this);
10
11          after_fifo.get(after_tx);
12          `uvm_info("FIFO_after", $sformatf("RES=%0h", after_tx.res), UVM_MEDIUM);
13
14          if( !after_tx.compare(before_tx) ) begin
15              my_mismatches++;
16          end
17          else begin
18              my_matches++;
19          end
20
21          phase.drop_objection(this);
22
23      end // forever begin
24
25  endtask // run_phase
```