9.3 Testbench approaches

9.3.1 Top-down or bottom-up?

In a forum discussion, the following questions were asked:[69]
Before implementation of testbench ,
1) how to start the testbench architecture?
2) Should it be Top-down or Bottom-up approach? I mean start the top testbench and begin then building the class libraries for test, or test class library first then the testbench top-level?

Though the questions are more generic about testbenches in general, these questions are also applicable to the testing of assertions. A bottom-up approach is preferred, particularly since there are templates or other automation tools to build the higher level structures.[70] We're also in favor of building something fast, and then refining it. Thus, the steps involved include:

1) Define the interfaces: This step defines the signals that need to be driven for an RTL, desidn; this step can be automated,
2) Define the transactions (*aka sequence item, in the world of UVM***) and the constraint:** That defines what will be driven and with which constraints. The constraints can be refined later on, but easy known ones could be defined at the early stages.
3) Define the transaction sequences: This steps addresses the sequences of transactions (*e.g., READ, then WRITE, then IDLE, then WRITE, then ..);* they could be defined separately from the drivers, a la UVM style, or they can be integrated as a step just prior to the drivers. Initially, one may chose a simple constraint random set of transaction sequences.
4) Define the drivers: This step addresses the application of those tests.
5) Define top: This is the top-level module to run the simulation.
6) Do a sanity check. This is a quick verification tests to ensure that all the pieces are working correctly, and there are no major error in the test environment.
7) Refine the testbench units: This includes constraints, scenarios (or sequences), monitors and checkers if needed.

9.3.2 Elements of a testbench

A popular testbench design is UVM. UVM is a transaction-based, class-based, and constrained-randomization of stimulus style of testbench using a library of classes to facilitate some automation (*with the cycling through the various phases of setup and configurations*) and reuse.

Should this UVM verification environment be used to test assertions?
To answer this question, let's examine a typical UVM architecture and the elements really needed to drive the assertions. The Figures below demonstrate a typical UVM architecture; it includes the following elements:

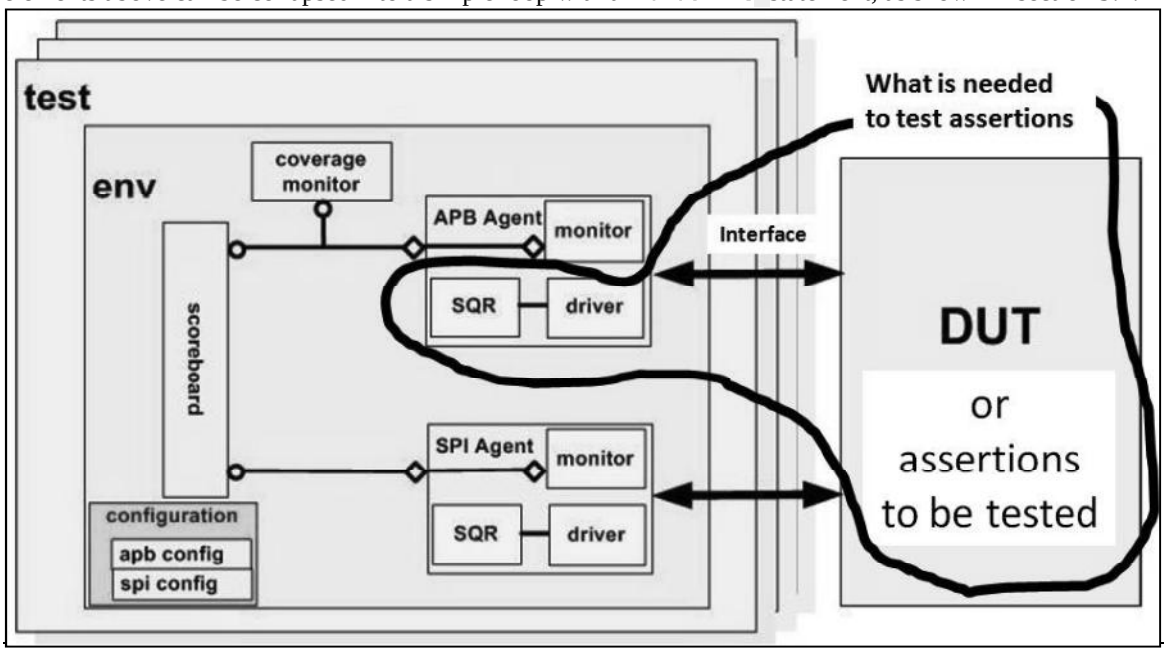
1. **An interface** to connect to the DUT of the module or interface that holds the assertions.
2. **A driver** for converting the data inside a series of high-level transactions (called *sequence_items*, e.g., READ, WRITE, RESET, IDLE, etc) into pin level wiggles.
3. **A sequencer** to route sequence_items from a sequence (*e.g., a READ, followed by 2 IDLEs, followed by a WRITE, and then a WRITE, etc...*). The sequence may contain constrained-random transactions.

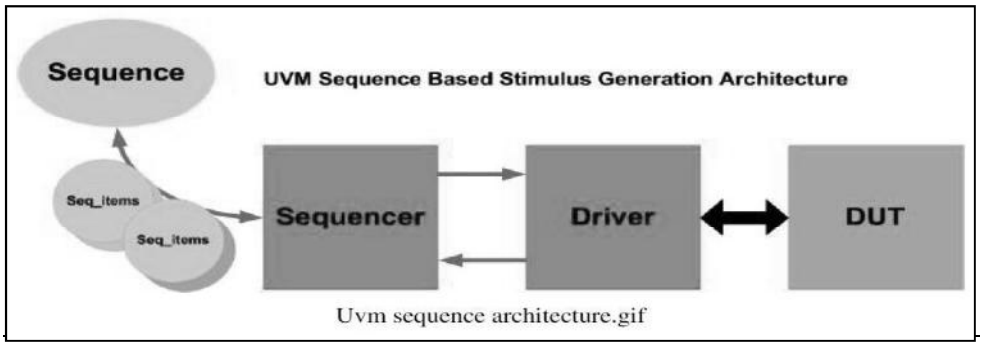
---

69 https://verificationacademy.com/forums/uvm/tb-architecture

70 https://verificationacademy.com/forums/uvm/tb-architecture
http://verifworks.com/products/dvcreate-uvm/
http://verifworks.com/products/dvcreate-svi/

To just stimulate a set of assertions for manual analysis, all the remaining elements of UVM are not needed, including: the monitor, scoreboard, agent, and configurations. In fact, in many situations, all three elements above can be collapsed into a simple loop with a `randomize` statement, as shown in section 9.4.





Uvm sequence architecture.gif

The following sections address various techniques to verify assertions embedded into modules, checkers, and interfaces.

9.4 Simple unconstrained randomization of test vectors in a test module

The simplest approach to verify assertions is to insert the assertions directly into a verification module or to bind an assertion module or checker into that verification module. This is then followed by an unconstrained randomization of the variables used in the assertions. This is equivalent to combining the sequence, sequencer and driver into one loop statement. UVM verbosity levels (see 4.2.3.2) are used to display messages. For example:

```
import uvm_pkg::*; `include "uvm_macros.svh"
module simple; // /ch9/9.4/testbench_ex.sv
   bit clk, req, ack;
   string log_id="SVA";
   ap_req_ack: assert property(@(posedge clk)
                     req |=> ##[1:5] ack) else
            `uvm_error(log_id, $sformatf(": Error in req ack %b %b", req, ack));
```

```
   initial forever #5 clk=!clk;
   initial begin
     repeat(200) begin
       @(posedge clk);
       if (!randomize(req, ack))
       `uvm_error(log_id, $sformatf("%m : error in randomization req ack %b %b", req, ack));
       $display("%t %b %b", $time, req, ack);
       end
      $finish;
    end
endmodule
```

The sequence of transactions is defined by the randomize. The driver is implicit because the variables are directly modified by the **randomize**.

9.4.1 Simple constrained randomization of test vectors in a test module

In this methodology, constraints are added inline with the randomize function. Those constraints adjust the distribution of the values of the variables to emulate more realistic test cases.

```
import uvm_pkg::*; `include "uvm_macros.svh"
module simple_ct; // /ch9/9.4/simple_ct.sv  /ch9/9.4/testbench_ex.sv
   bit clk, req, ack;
   string log_id="SVA";
   ap_req_ack: assert property(@(posedge clk)
                     req |=> ##[1:5] ack) else
      `uvm_error(log_id, $sformatf(": Error in req ack %b %b", req, ack));

   initial forever #5 clk=!clk;
   initial begin
     repeat(200) begin
       @(posedge clk);
       if (!randomize(req, ack)  with
           { req dist {1'b1:=1, 1'b0:=3};
             ack dist {1'b1:=1, 1'b0:=5};})
               `uvm_error("MYERR", "This is a randomize error")
       // `uvm_info("simple", $sformatf("%t %b %b", $time, req, ack), UVM_MEDIUM);
       end
       $finish;
    end
endmodule
```

The sequence of transactions is defined by the randomize. The driver is implicit because the variables are directly modified by the **randomize**.

9.4.2 Class-based randomization of test vectors

Using classes to define the variables to be constrained not only allows for the collocation of the definition of the constraints, but they also provides a greater flexibilities in making changes through the extension of classes. Thus,

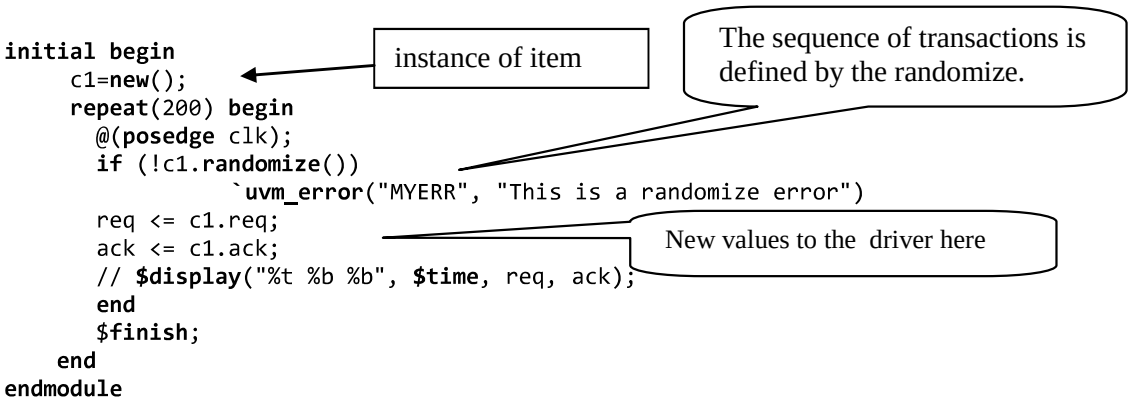
```
import uvm_pkg::*;   `include "uvm_macros.svh"
class c;    // /ch9/9.4/testbench_ex.sv
       rand bit req, ack;
       constraint req_ct { req dist {1'b1:=1, 1'b0:=3};}
       constraint ack_ct { ack dist {1'b1:=1, 1'b0:=5};}
endclass

module class_based; /ch9/9.4/testbench_ex.sv
   bit clk, req, ack;
   string tID="simple";
   c c1;
   ap_req_ack: assert property(@(posedge clk)
                     req |=> ##[1:5] ack) else
        `uvm_error(log_id, $sformatf(": Error in req ack %b %b", req, ack));

   initial forever #5 clk=!clk;
```

sequence item

item declaration



9.4.3 Transaction-based defintion of test sequences

This methodology raises the level of abstraction by considering transactions that are then decoded into values for individual signals (e.g., the *rd, wr, data* signals) This methodology encompasses several steps as demonstrated in the figure below :

1) Definition of transactions to be randomized (AKA sequence item, or item).

A *transaction* is as an operation that represents the job to be performed, such as Read / Write / Idle. Transactions are best implemented in a class, along with other variables that need to be constrained. For example, a *transaction* may consist of the following:

1. **Instruction**. This represents the high-level tasks to be executed, such as a READ, WRITE, NO-OP, LOAD, etc.
2. **Data**. This represents information such as address, data, number of cycles, etc.
3. **Parameters**. This can represent a mode, a size, path, etc.

The basic idea of a transaction-level methodology, such as UVM, is to separate the *transaction* from the sequence, sequencer and drivers. This approach enables the class to easily be extended and adjusted without modifying the other elements of the testbench.

2) **Definition of assertions grouped into module(s) or checkers**

Co-locating the assertions into separate modules or checkers provides several benefits[71]

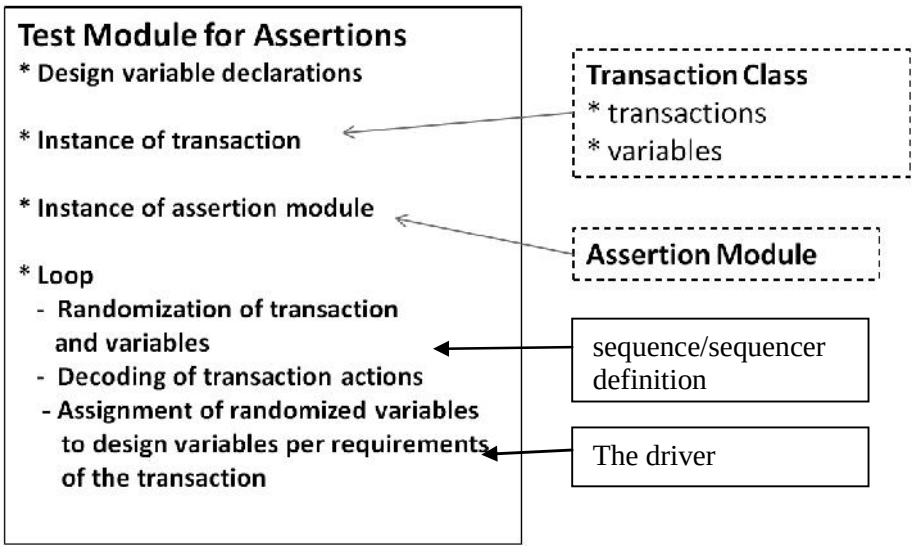
- Isolates assertions from DUT
- Provides timestamp isolation.
- Enables easy modifications to the assertions
- Enables easier review
- Enables tools to perform static checks on assertions (*a la lint*)

3) **Definition of test module**

The test module includes the following:

- The design variable declarations, which defines the values to be applied to the assertions.
- A declaration and an instance of the transaction class (referred to as the sequence item).
- An instance of the assertion module.
- A loop that provides the following functions, typically on a clocked basis:
  * Randomizes items within the transaction class
  * Decoding of the transaction to be exercised.
  * Assignment of the randomized variables in the class instance into the design variables.
- Initiation of a finish task to end the simulation.

[71] (See Stuart Sutherland SNUG 2015 paper)



9.4.3.1 The sequence item

The stimulus generation process is based on sequences controlling the behavior of drivers by generating sequence_items and sending them to the driver via a sequencer. The framework of the stimulus generation flow is built around the sequence structure for control, but the generation data flow uses sequence_items as the data objects.

The content of the sequence_item is closely related to the needs of the driver. The driver relies on the content of the sequence_items it receives to determine which type of pin level transaction to execute. The sequence items property members will consist of data fields that represent the following types of information:

- Control - i.e. What type of transfer, what size
- Payload - i.e. The main data content of the transfer
- Configuration - i.e. Setting up a new mode of operation, error behavior etc
- Analysis - i.e. Convenience fields which aid analysis - time stamps, rolling checksums etc

When the assertions address more complex scenarios, a transaction-based definition of test sequences is preferred. This approach borrows from the UVM stimulus generation process, but is simpler in that it does not encompass all of UVM, and can be built relatively quickly. With this approach, a transaction defines, at a higher level, the types of operations to be applied to the assertions. These high level operations are supplemented with constraint variables to help in the definition of the values applied to the assertions. Examples of transactions include RESET, READ. WRITE, ADD, MULTIPLY, SEND, etc. The transactions are typically specified as enumeration of a variable. The transaction is typically randomized with constraints, and it is used by tasks to decode the type of activity to be acted upon in that randomization step (e.g., a WRITE); The driver tasks then executes the steps necessary to implement that transaction. An example of a class that includes transaction type definitions and items to be randomized is shown below.

```
package bus_pkg; // /ch9/9.4/txexample.sv
       timeunit 1ns; timeprecision 100ps;
       typedef enum {BS_READ, BS_WRITE, BS_READ_WRITE, BS_IDLE} BS_scen_e;
endpackage : bus_pkg
```

```
import bus_pkg::*;
class bus_seq_item;
    rand BS_scen_e kind;
    rand logic[31:0] addr;
    rand logic[31:0] data;
    rand bit x, y;
    rand int delay;
    constraint kind_ct {kind dist
        {BS_READ:=30, BS_WRITE:=30, BS_READ_WRITE:=30, BS_IDLE:=10 };}
    constraint at_least_1 { delay inside {[1:20]};}

       // 32 bit aligned transfers
    constraint align_32 {addr[1:0] == 0;}

    constraint xy_ct { {x, y} dist
              {2'b00:=15, 2'b01:=2, 2'b10:=2, 2'b11:=1};}
endclass: bus_seq_item
```

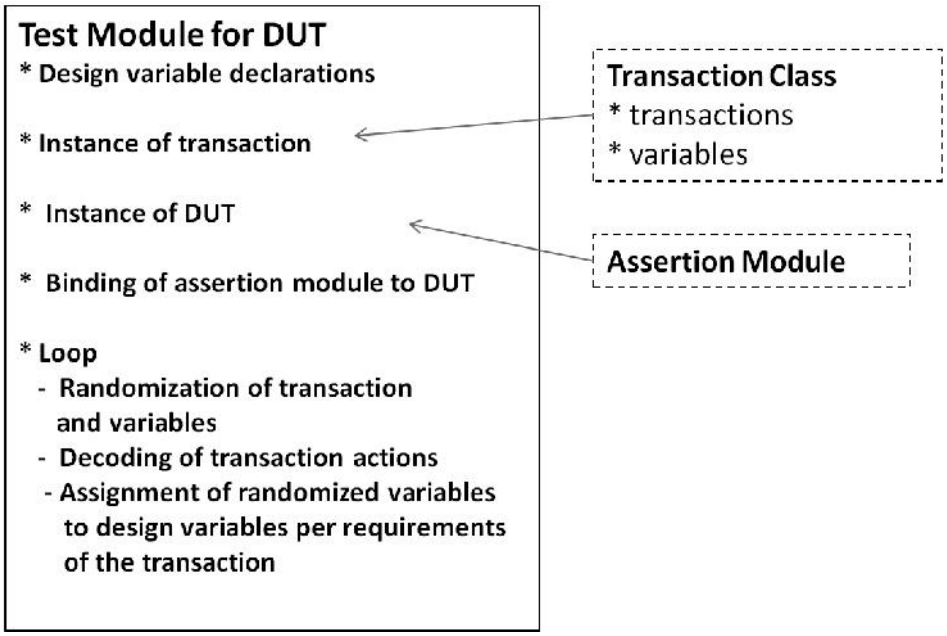
transaction to be randomized, and then applied per defintion in the driver protocols.

variables to be used once a value of kind item is determined

constraints to control sequences

9.4.3.2 Transitioning from testbenching assertions to quick testbenching DUT.

The methodologies between verifying assertions and performing sanity testing on the DUT are not that far apart. Below is a quick summary of the methodology. This is followed by an example for a counter with some strange requirements, defined as such to demonstrate the capabilities of the assertions and the testbench environment.



The design problem:

- Design and write assertions for a loadable synchronous up-counter
- Loadable counter, reset to MIN_COUNT if rst_n==0
  - Min load value == MIN_COUNT (default ==2)
  - Max count value == MAX_COUNT (default ==9)
- holds the count when it reaches the maximum value
- Must change value at least every 9 clocks

Increments If ld==1'b0 and the counter != MAX_CO