Self Checking TestBench (Example -4)

```
// BCD Counter
module bcdcounter(clk, reset, enable, Q, max);
  input clk, reset, enable;
  output reg [3:0] Q;
  output max;

  assign max = (Q == 9) & enable;

  always @( posedge clk )
  begin
    if (reset)
      Q <= 4'b0;
    else if (enable)
      begin
        if (max)
          Q <= 0;
        else
          Q <= Q + 1;
     end
  end
endmodule
```

Self Checking TestBench (Example -4)

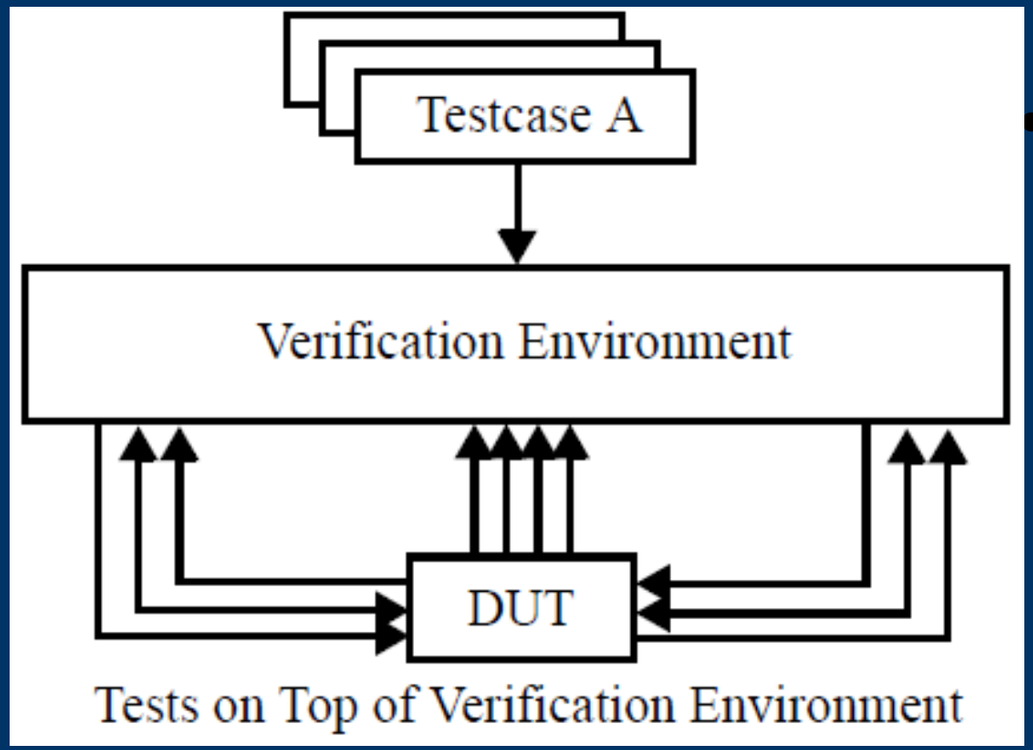
```
module bcdcounter_test;

  // signals for connecting the counter
  reg         clk;
  reg         reset;
  reg         enable;
  wire [3:0] Q;
  wire        carry;

  // testbench variables;
  integer   i;

  // counter instance
  bcdcounter DUT(.clk(clk),.reset(reset),.enable(enable),.Q(Q),.carry(carry));

  task check;
      input [3:0] Q, check_Q;
      input carry, check_carry;

      begin
        if (Q != check_Q)
          $display("Error at time %t: Expected Q=%d, Actual Q=%d",$time, check_Q, Q);
        if (carry != check_carry)
          $display("Error at time %t: Expected carry=%d, Actual carry=%d", $time, check_carry, carry);
      end
    endtask
```

Self Checking TestBench (Example -4)

```
// clock drives both counter and bench
  always #10 clk = ~clk;

  initial begin
    clk = 0;  // necessary to set it to a known value!
    reset = 0;
    enable = 0;
    @(posedge clk);
      // do a reset and check that it worked
      reset = 1;
    @(posedge clk);
      check(Q,0,carry,0);

    // now try counting a few cycles
    #1 reset = 0;
    #1 enable = 1;
    for (i=0; i<9; i=i+1)
      begin
        @(posedge clk);
          check(Q,i,carry,0); //Try inducing error here!
      end
```

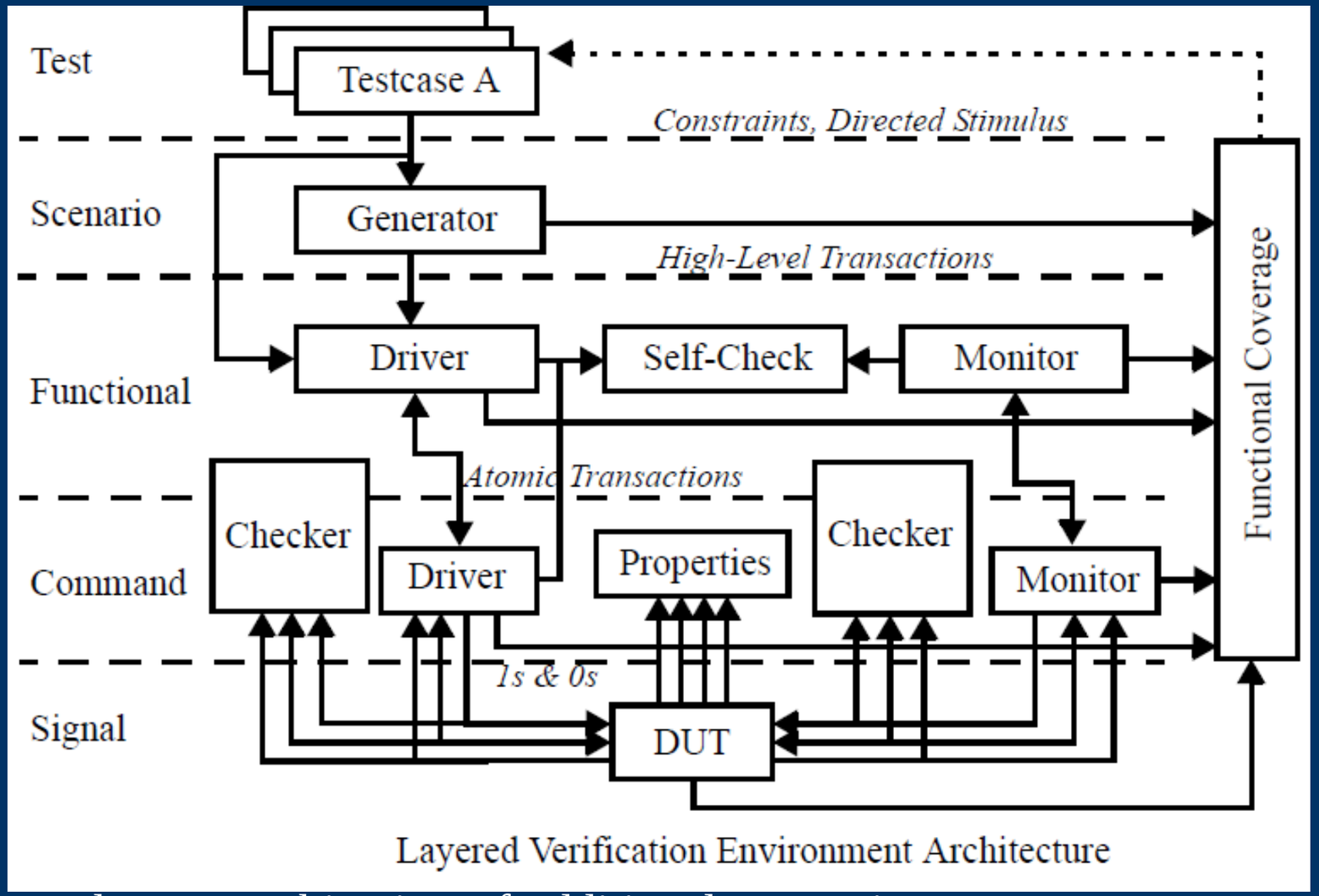
```
// now check the carry count should be 9!
      @(posedge clk);
        check(Q,9,carry,1);
        // now check the rollover
      @(posedge clk);
        check(Q,0,carry,0);
        // intentional error - count !=2, carry != 1
      @(posedge clk)
        check(Q,2,carry,0);
        check(Q,1,carry,1);
        repeat (7) @(posedge clk);
          #1 check(Q,9,carry,1);
          #5 enable = 0;
          #2 check(Q,9,carry,0);
      repeat (3) @(posedge clk);
      $stop();  // all done!
  end // initial

endmodule
```

Testbench Architecture



- Verification environments are composed of many layers
- Verification Environment
  - Implements the abstraction and automation functions that help minimize the number and details of testcases that need to be written
  - Will also be reused, without modifications, by as many testcases as possible to minimize the amount of code required to verify the DUT

Testbench Architecture

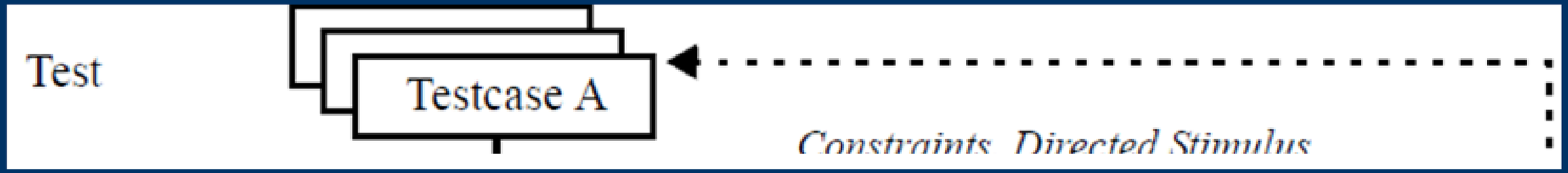


Layered Verification Environment Architecture

- Testcases are implemented as a combination of additional constraints on generators, new random scenario definitions, synchronization mechanisms between transactors, error injection enablers, DUT state monitoring and directed stimulus

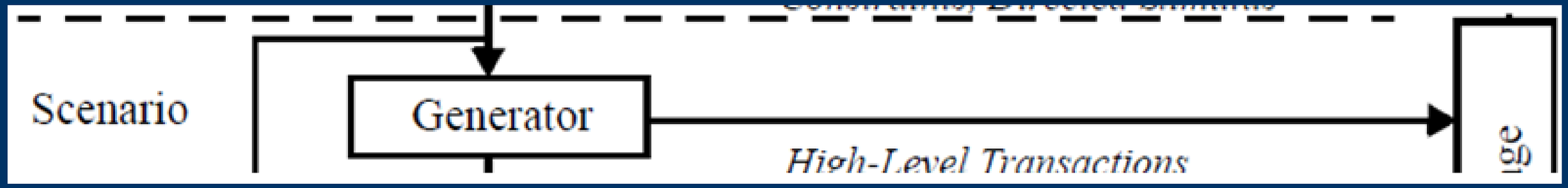
Testbench Architecture

- Test layer
  - Tests are located in this layer.
  - Test layer can interact with all the layers.
  - This layer allows to pass directed commands to functional and command layer.



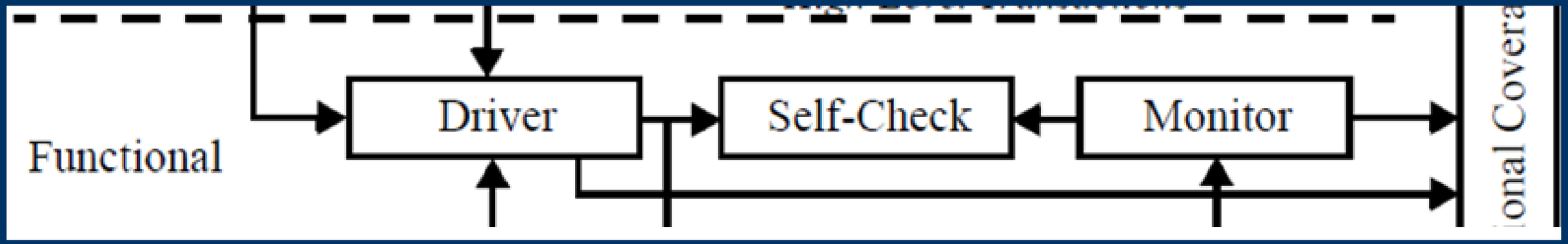
Testbench Architecture

- Scenario layer
  - Uses generators to produce streams or sequences of transactions that are applied to the functional layer.
  - The generators have a set of weights, constraints or scenarios specified by the test layer.
  - The randomness of constrained-random testing is introduced within this layer.



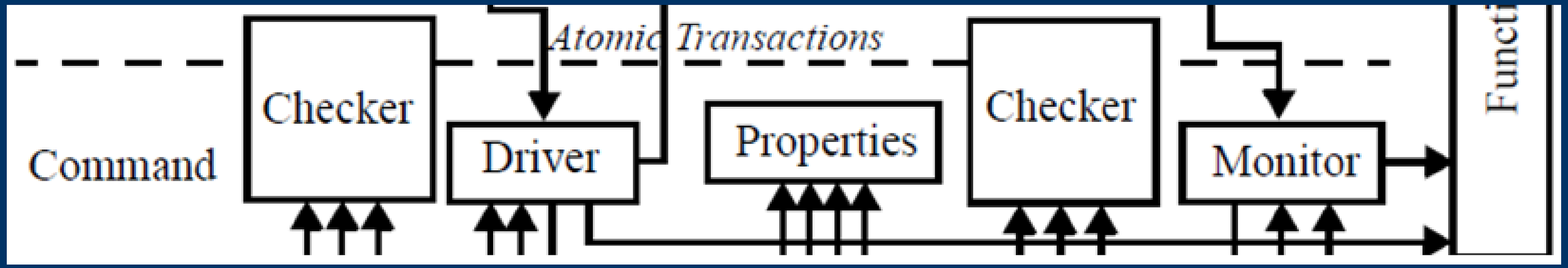
Testbench Architecture

- Functional layer
  - Contains higher-level driver and monitor components, as well as the self-checking structure (scoreboard/tracker).



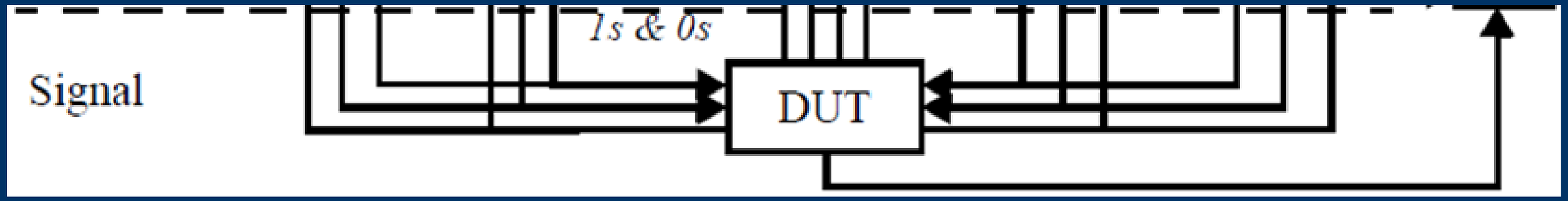
Testbench Architecture

- Command layer
  - Contains lower-level driver and monitor components, as well as the assertions.
  - Provides a transaction-level interface to the layer above and drives the physical pins via the signal layer.



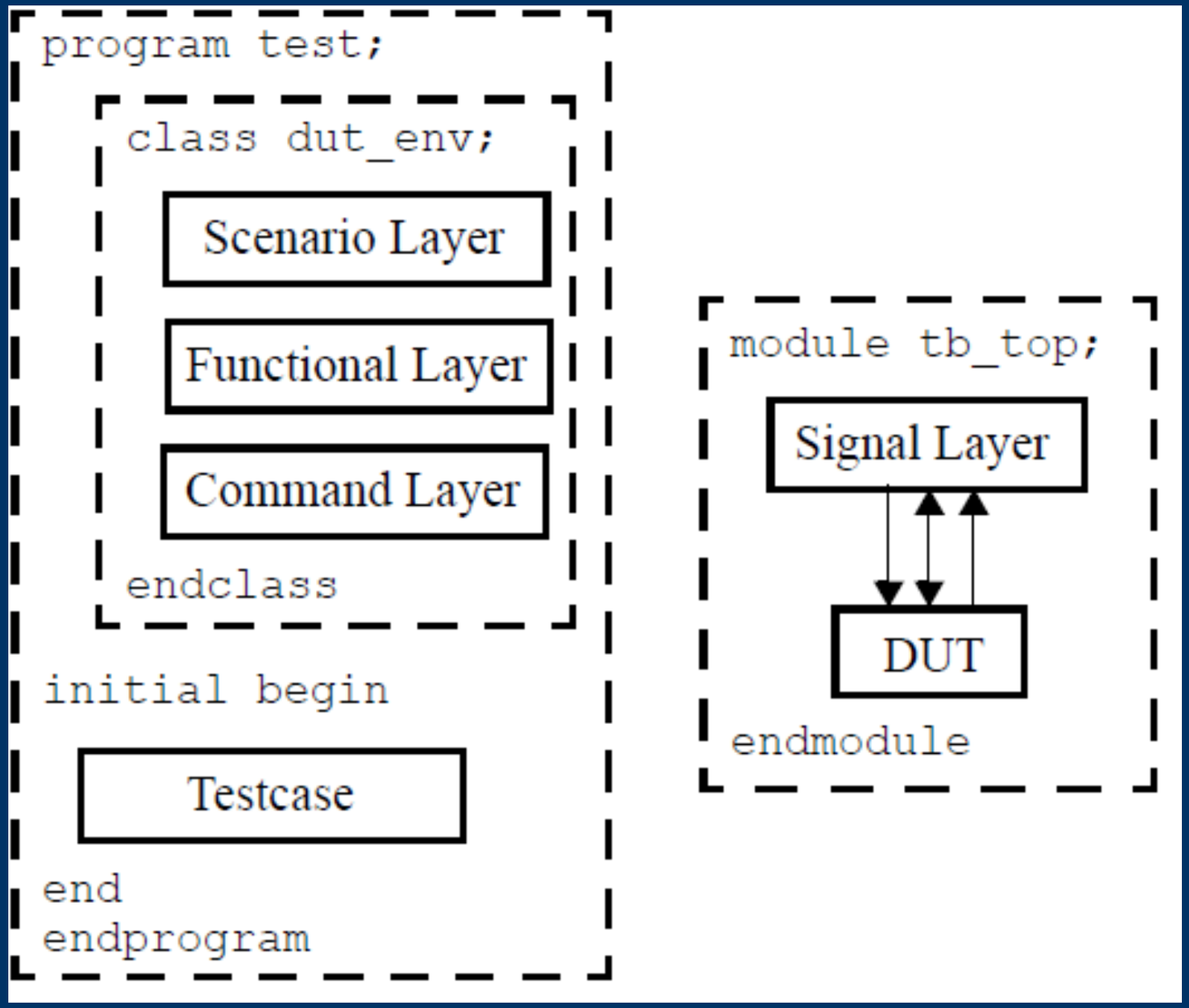
Testbench Architecture

- Signal layer
  - Connects the testbench to the RTL design
  - Consists of interface, clocking, and modport constructs



Testbench Architecture

- General structure to implement complete testbenches



Testbench Architecture

- Complete verification environments are never implemented in one shot.
- They evolve to meet the increasingly complex requirements of the testcases being written and responses being checked.
- Initially supporting only a trivial directed testcase with no self-checking, layers are added to evolve them into full-fledged self-checking constrained-random verification environment.
- The layered architecture makes no assumption about the DUT model.
- It can be an RTL or gate-level model as well as a transaction-level model.

Testbench Architecture

- Rule 1: The signal layer shall be implemented and the DUT shall be instantiated in a top-level module
  - This top-level module will contain the design portion of the simulation.
  - Any element of the signal layer or DUT will be accessible via cross-module references through the top-level module.
  - There is no need to instantiate the top-level module anywhere.
- Rule 2: The verification environment shall be implemented in a top-level class.
  - The environment will leverage generic functionality from a verification environment base class.
  - It will be able to refer to the signal layer or any DUT element via cross module references into the top-level module.
  - The environment will be instantiated by each testcase.

Testbench Architecture

- Rule 3: Testcases shall be implemented in an initial block in a program block
  - The testcase procedure can refer to any public members in the verification environment via hierarchical references through the top-level environment class.
  - It will also be able to access any element of the signal layer or DUT via cross-module references through the top level module.
  - The verification environment will be designed to minimize the number of statements in testcases.

Testbench Architecture

- Rule 4: Interfaces shall be packaged in the same file as the transactors that use them
  - Command-layer transactors have a physical-level interface composed of individual signals.
  - All signals pertaining to a physical protocol are bundled in a single interface construct.
  - The interface declarations and the class declarations for transactors operating on those interfaces are intimately related.
  - They shall be packaged together in the same file.
- Rule 5: Interfaces shall be named with a prefix that matches the associated component name prefix or package name.
- Rule 6: All interface signals shall be declared as wire
- Rule 7: Synchronous interface signals shall be sampled and driven using a clocking block.

Testbench Architecture

- Rule 8: Set-up and hold time in clocking blocks shall be defined using parameters.
- Rule 9: Individual modports shall be declared for each type of proactive, reactive and passive transactors.
- Rule 10: The direction of asynchronous signals shall be specified in the modport declaration
- Rule 11: The direction of synchronous signals shall be specified in the clocking block declaration
- Rule 12: The clocking block shall be included in modports port list instead of individual clock and synchronous signals
- Rule 13: The design and all required interfaces and signals shall be instantiated in a module with no ports

Testbench Architecture

- Rule 14: Signals in different interface instances implementing the same physical interface shall be mapped to each other
- Rule 15: Clock generation shall be done in the top-level module
- Rule 16: There shall be no clock edges at time 0.
  - Wait for the duration of a few periods of the slowest clock in the system before generating clock edges
- Rule 17: The bit type shall be used for all clock and reset signals