QUIZ UVM

EL DUT

- Este es un detector de patrones simple escrito en Verilog para identificar un patrón en un flujo de valores de entrada.

- En cada reloj, hay una nueva entrada al diseño y cuando coincide con el patrón "1011 ', la salida (out) se configurará en 1. Para este propósito, el diseño se implementa como una máquina de estado que se mueve a través de diferentes etapas. a medida que avanza a través de la secuencia de identificación de patrones.

PRUEBA BÁSICA

El banco de pruebas de verificación se desarrollará en UVM:

- La secuencia genera un flujo aleatorio de valores de entrada que se pasarán al controlador como uvm_sequence_item
- El driver recibe el artículo y lo conduce al DUT a través de una interfaz virtual
- El monitor captura valores en el pin de entrada y salida del DUT, crea un paquete y lo envía al scoreboard
- El cuadro de indicadores es el principal responsable de verificar la corrección funcional del diseño en función de los valores de entrada y salida que recibe del monitor.

El flujo de valores de entrada debe ser aleatorio para lograr la máxima eficiencia. Debería poder captar los siguientes escenarios:

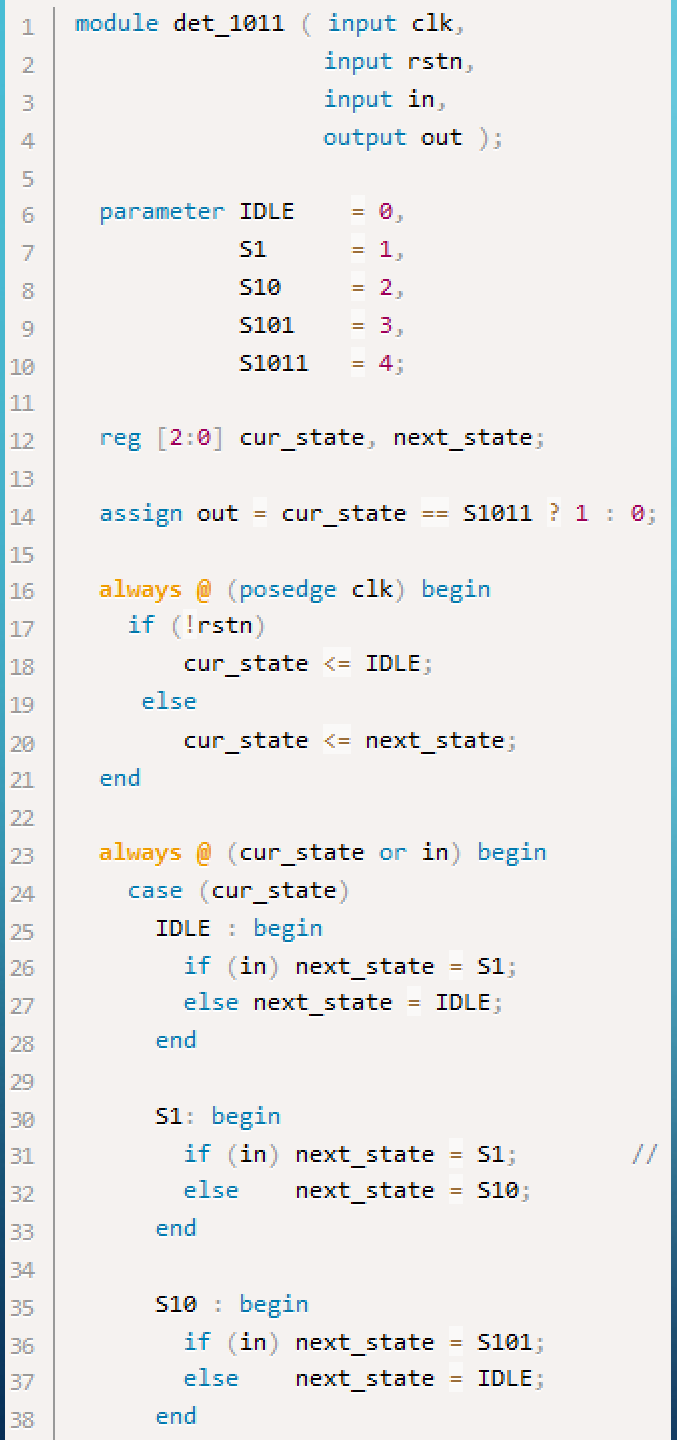
- 011011011010
- 101011100
- 111011011

PARA CORRER UNA PRUEBA QUE USA UVM EN VCS

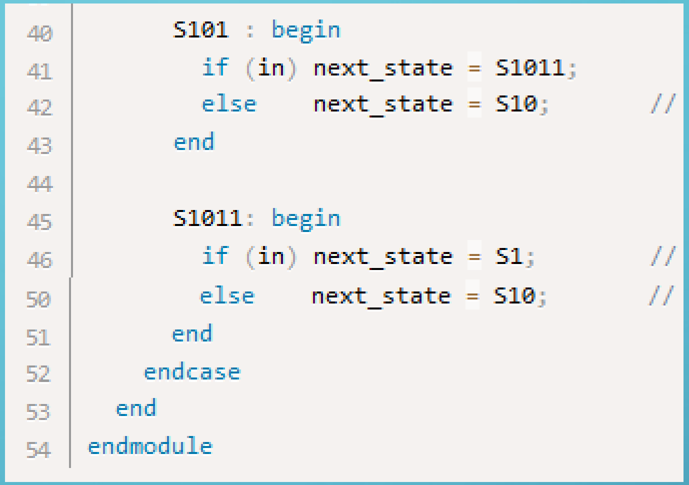
Añada el switch:

- -ntb_opts **uvm-1.2**

DISEÑO



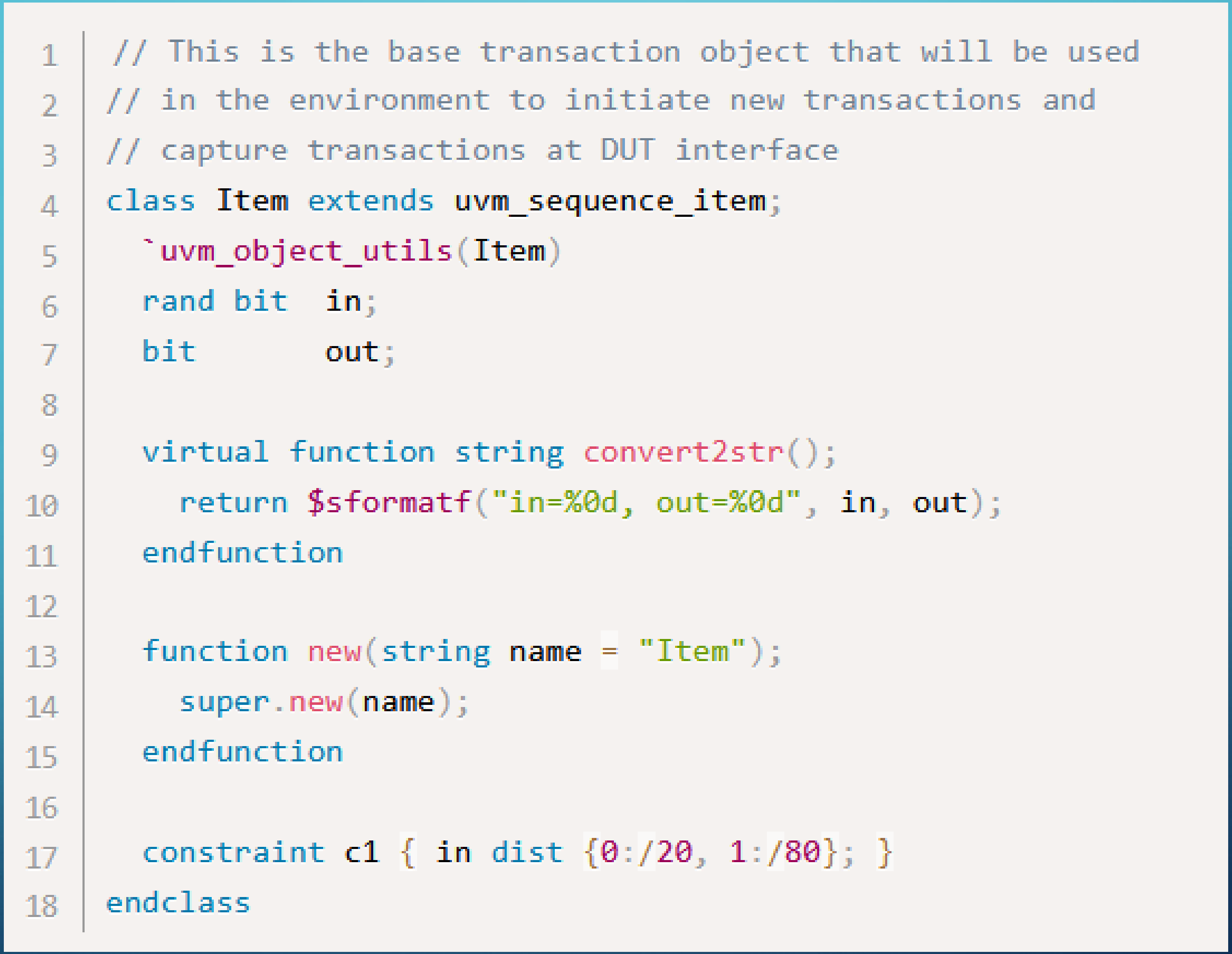
```verilog
module det_1011 ( input clk,
                  input rstn,
                  input in,
                  output out );

  parameter IDLE    = 0,
            S1      = 1,
            S10     = 2,
            S101    = 3,
            S1011   = 4;

  reg [2:0] cur_state, next_state;

  assign out = cur_state == S1011 ? 1 : 0;

  always @ (posedge clk) begin
    if (!rstn)
        cur_state <= IDLE;
     else
        cur_state <= next_state;
  end

  always @ (cur_state or in) begin
    case (cur_state)
      IDLE : begin
        if (in) next_state = S1;
        else next_state = IDLE;
      end

      S1: begin
        if (in) next_state = S1;       //
        else     next_state = S10;
      end

      S10 : begin
        if (in) next_state = S101;
        else     next_state = IDLE;
      end
```



```verilog
      S101 : begin
        if (in) next_state = S1011;
        else    next_state = S10;       //
      end

      S1011: begin
        if (in) next_state = S1;        //
        else    next_state = S10;       //
      end
    endcase
  end
endmodule
```

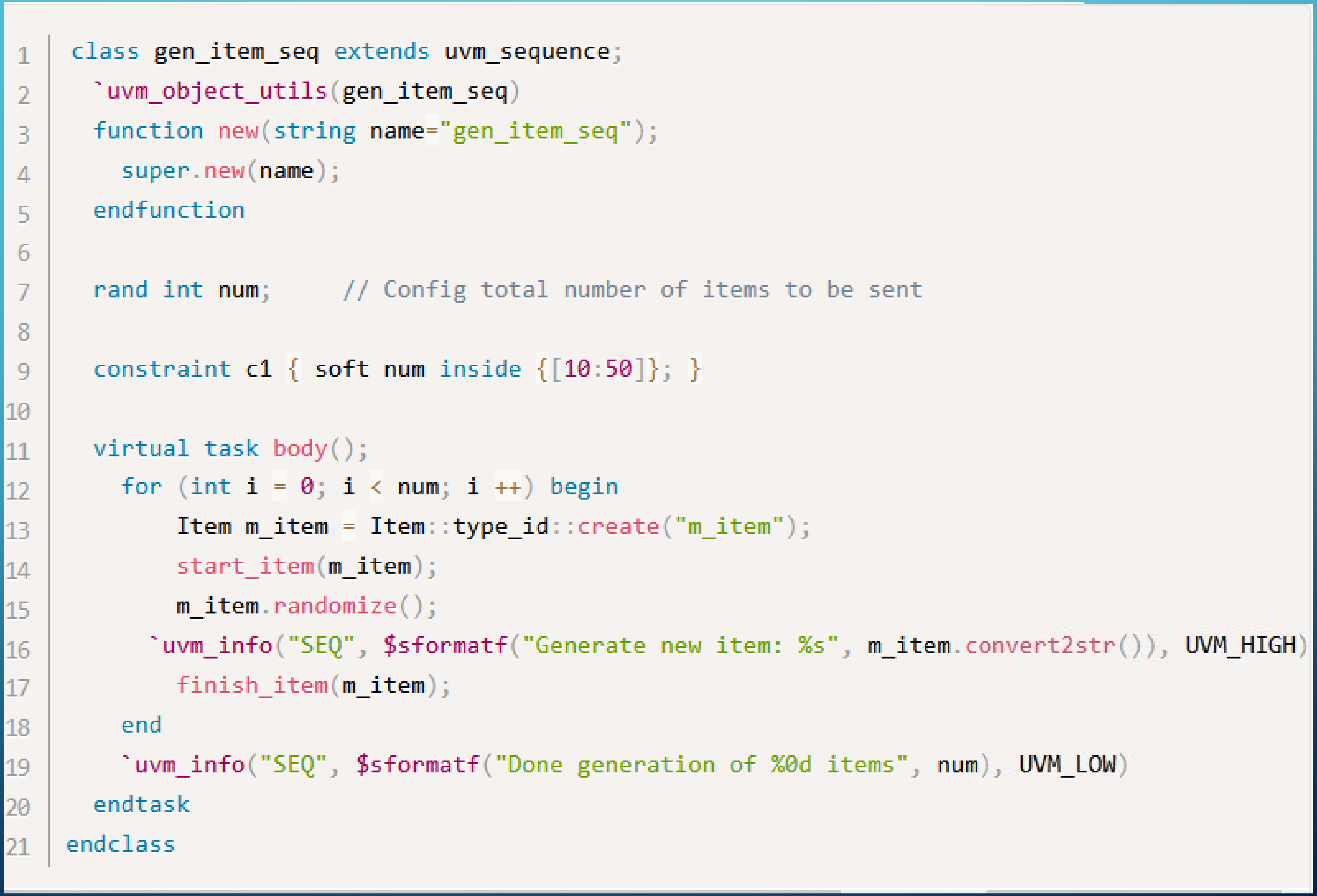
RESPUESTA

SEQUENCE ITEM



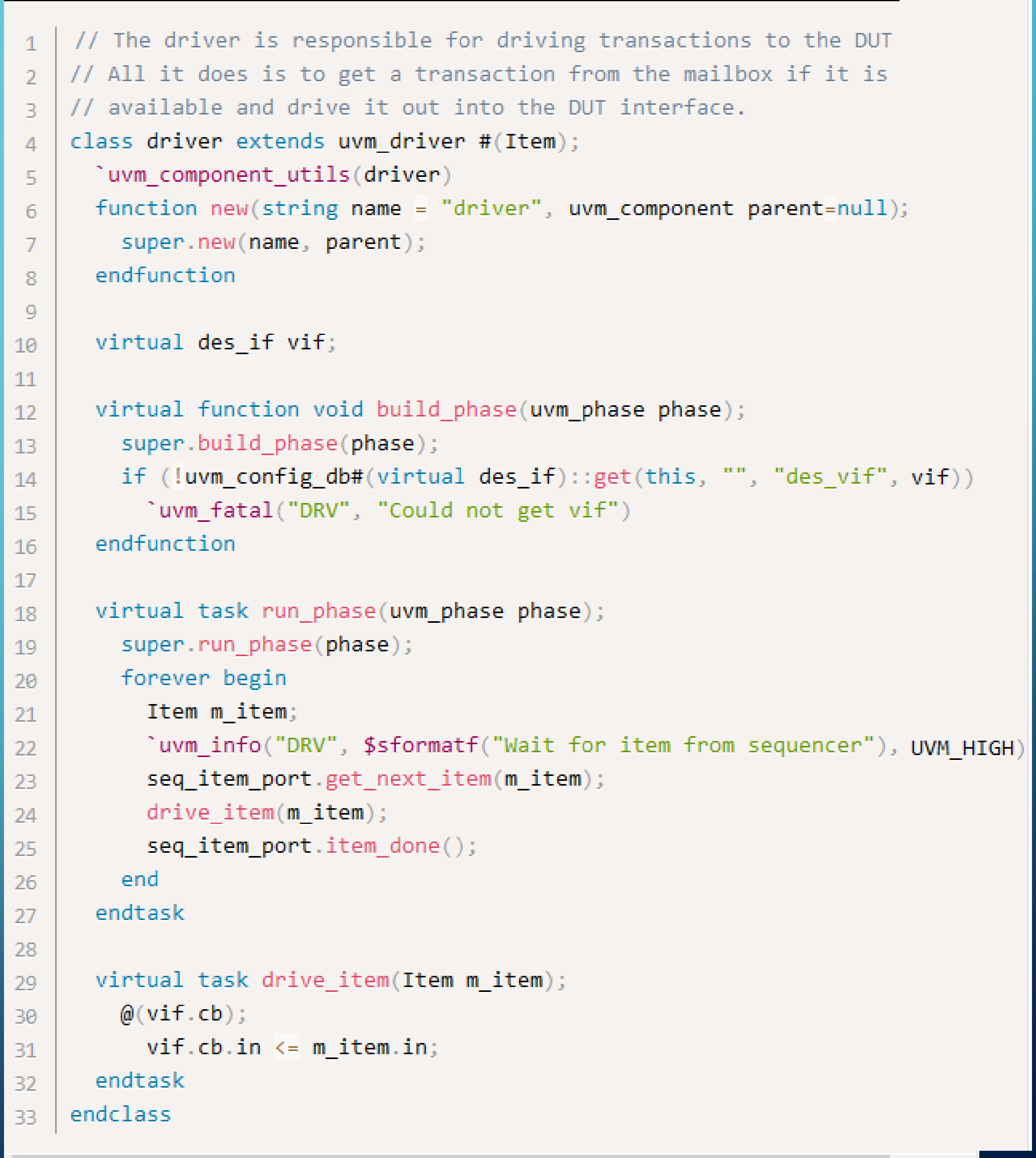
```systemverilog
// This is the base transaction object that will be used
// in the environment to initiate new transactions and
// capture transactions at DUT interface
class Item extends uvm_sequence_item;
  `uvm_object_utils(Item)
  rand bit  in;
  bit       out;

  virtual function string convert2str();
    return $sformatf("in=%0d, out=%0d", in, out);
  endfunction

  function new(string name = "Item");
    super.new(name);
  endfunction

  constraint c1 { in dist {0:/20, 1:/80}; }
endclass
```

SEQUENCE



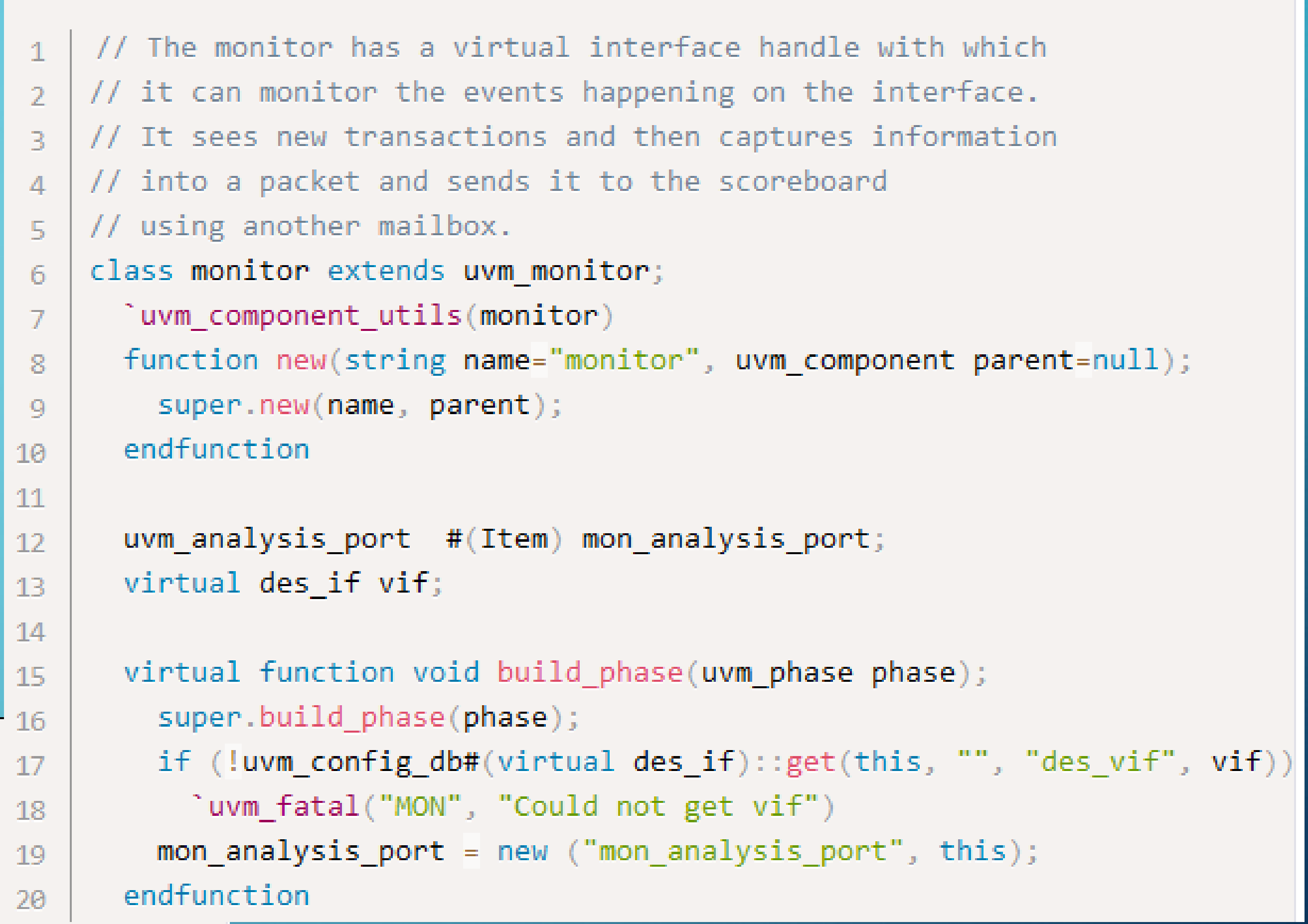
```
class gen_item_seq extends uvm_sequence;
  `uvm_object_utils(gen_item_seq)
  function new(string name="gen_item_seq");
    super.new(name);
  endfunction

  rand int num;     // Config total number of items to be sent

  constraint c1 { soft num inside {[10:50]}; }

  virtual task body();
    for (int i = 0; i < num; i ++) begin
        Item m_item = Item::type_id::create("m_item");
        start_item(m_item);
        m_item.randomize();
      `uvm_info("SEQ", $sformatf("Generate new item: %s", m_item.convert2str()), UVM_HIGH)
        finish_item(m_item);
    end
    `uvm_info("SEQ", $sformatf("Done generation of %0d items", num), UVM_LOW)
  endtask
endclass
```

DRIVER

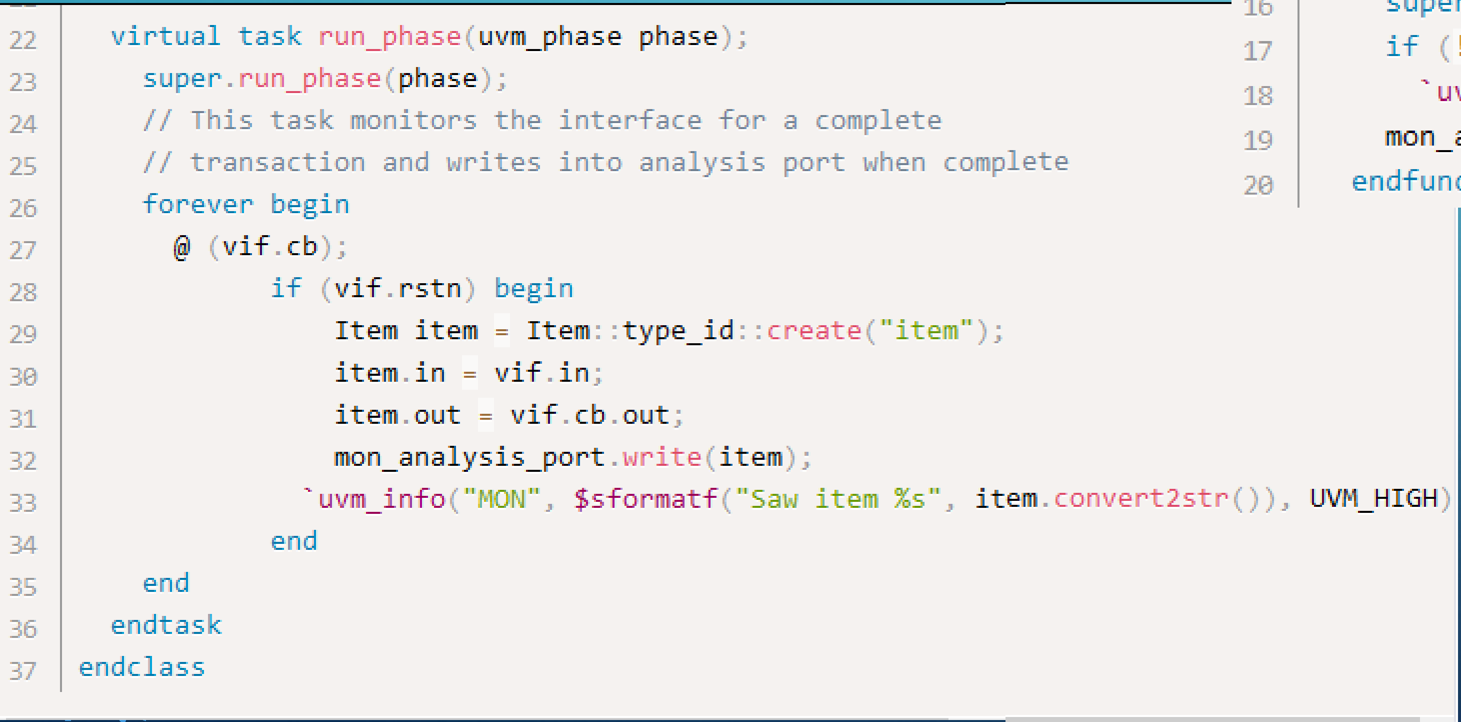


```
// The driver is responsible for driving transactions to the DUT
// All it does is to get a transaction from the mailbox if it is
// available and drive it out into the DUT interface.
class driver extends uvm_driver #(Item);
  `uvm_component_utils(driver)
  function new(string name = "driver", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  virtual des_if vif;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(virtual des_if)::get(this, "", "des_vif", vif))
      `uvm_fatal("DRV", "Could not get vif")
  endfunction

  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    forever begin
      Item m_item;
      `uvm_info("DRV", $sformatf("Wait for item from sequencer"), UVM_HIGH)
      seq_item_port.get_next_item(m_item);
      drive_item(m_item);
      seq_item_port.item_done();
    end
  endtask

  virtual task drive_item(Item m_item);
    @(vif.cb);
      vif.cb.in <= m_item.in;
  endtask
endclass
```

MONITOR

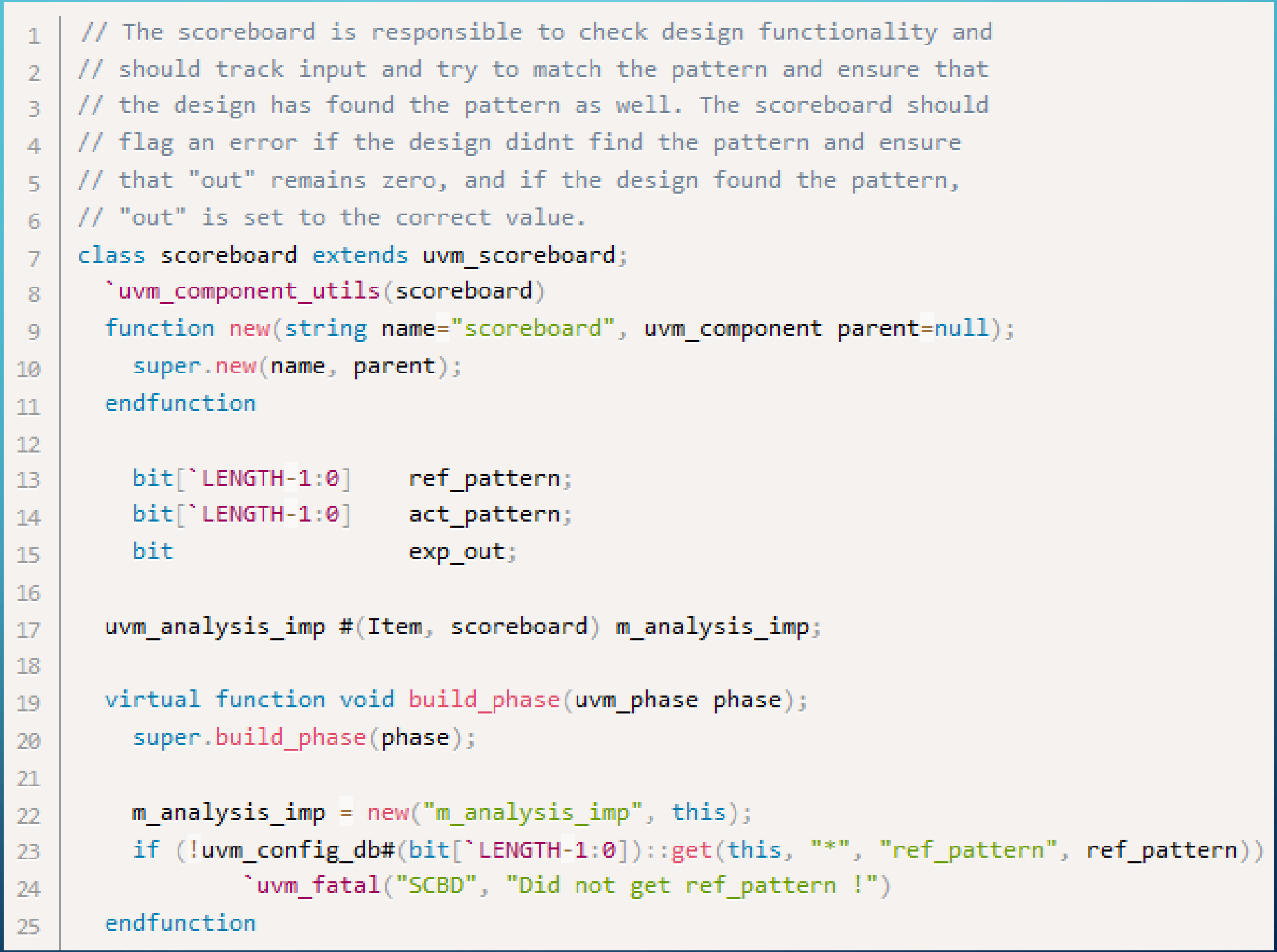


```
// The monitor has a virtual interface handle with which
// it can monitor the events happening on the interface.
// It sees new transactions and then captures information
// into a packet and sends it to the scoreboard
// using another mailbox.
class monitor extends uvm_monitor;
  `uvm_component_utils(monitor)
  function new(string name="monitor", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  uvm_analysis_port  #(Item) mon_analysis_port;
  virtual des_if vif;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(virtual des_if)::get(this, "", "des_vif", vif))
      `uvm_fatal("MON", "Could not get vif")
    mon_analysis_port = new ("mon_analysis_port", this);
  endfunction
```



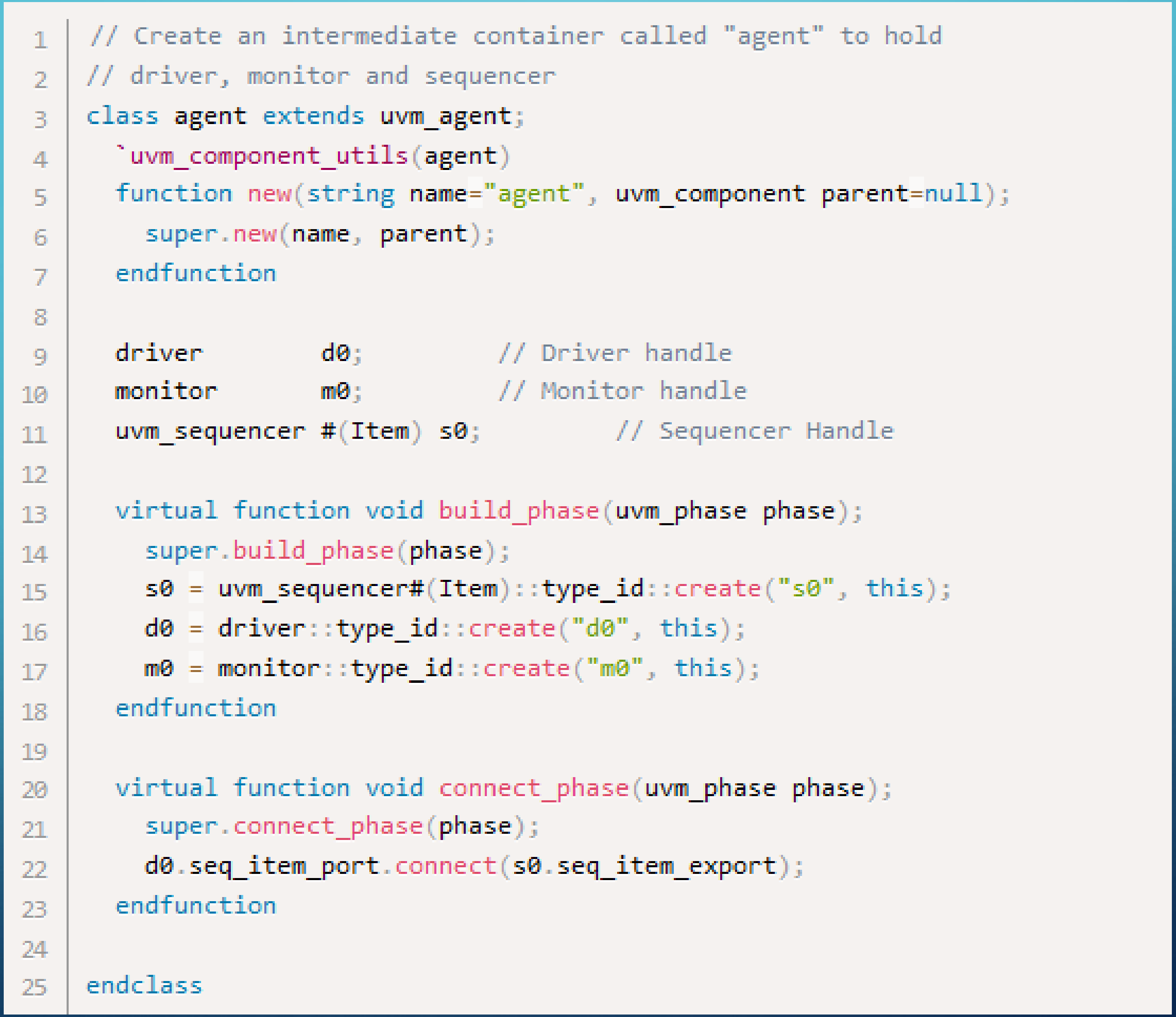
```
  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    // This task monitors the interface for a complete
    // transaction and writes into analysis port when complete
    forever begin
      @ (vif.cb);
        if (vif.rstn) begin
            Item item = Item::type_id::create("item");
            item.in = vif.in;
            item.out = vif.cb.out;
            mon_analysis_port.write(item);
          `uvm_info("MON", $sformatf("Saw item %s", item.convert2str()), UVM_HIGH)
        end
    end
  endtask
endclass
```

SCOREBOARD



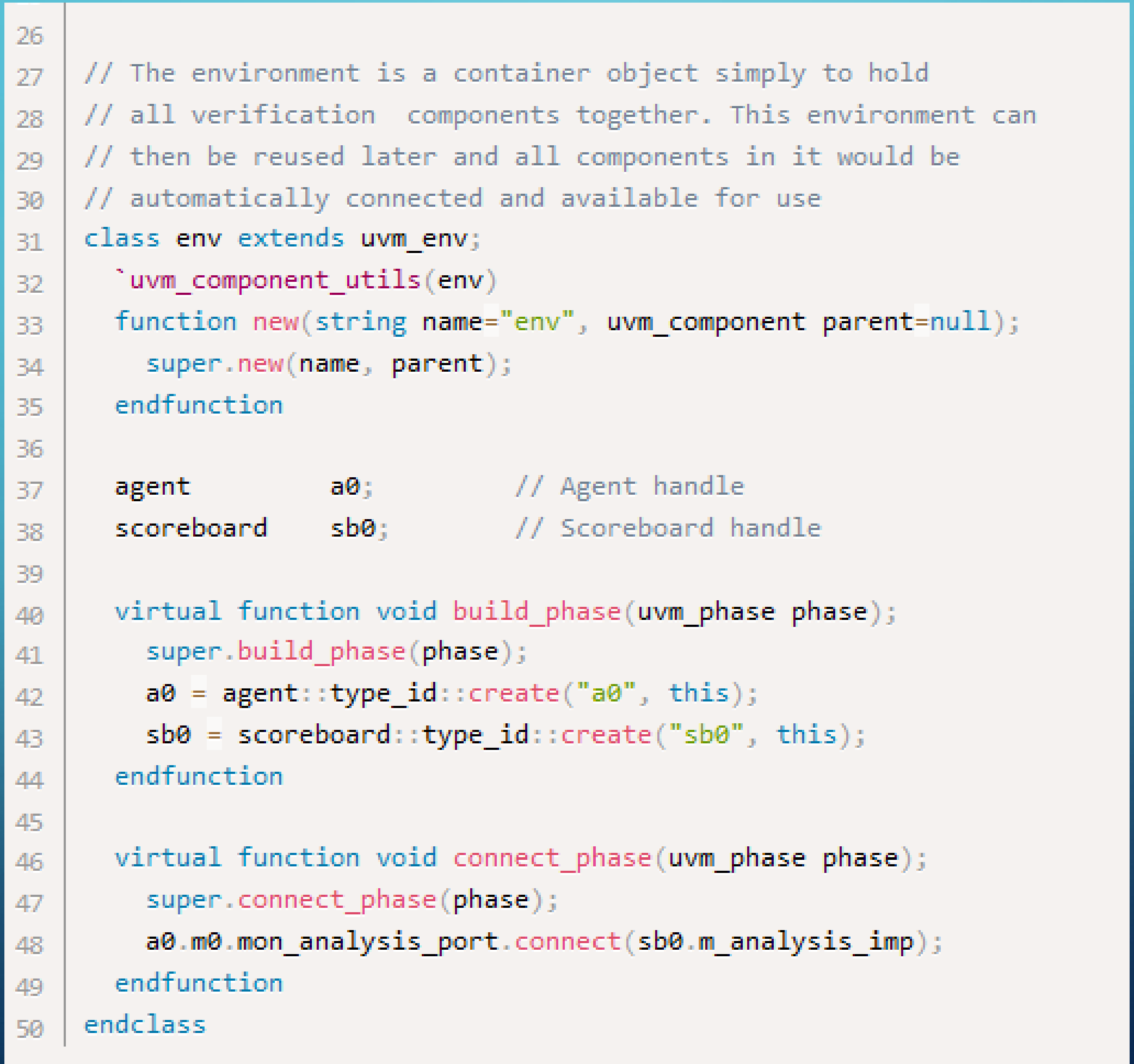
```
// The scoreboard is responsible to check design functionality and
// should track input and try to match the pattern and ensure that
// the design has found the pattern as well. The scoreboard should
// flag an error if the design didnt find the pattern and ensure
// that "out" remains zero, and if the design found the pattern,
// "out" is set to the correct value.
class scoreboard extends uvm_scoreboard;
  `uvm_component_utils(scoreboard)
  function new(string name="scoreboard", uvm_component parent=null);
    super.new(name, parent);
  endfunction

    bit[`LENGTH-1:0]    ref_pattern;
    bit[`LENGTH-1:0]    act_pattern;
    bit                 exp_out;

  uvm_analysis_imp #(Item, scoreboard) m_analysis_imp;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    m_analysis_imp = new("m_analysis_imp", this);
    if (!uvm_config_db#(bit[`LENGTH-1:0])::get(this, "*", "ref_pattern", ref_pattern))
            `uvm_fatal("SCBD", "Did not get ref_pattern !")
  endfunction
  virtual function write(Item item);
    act_pattern = act_pattern << 1 | item.in;

    `uvm_info("SCBD", $sformatf("in=%0d out=%0d ref=0b%0b act=0b%0b",
                               item.in, item.out, ref_pattern, act_pattern), UVM_LOW)

    // Always check that expected out value is the actual observed value
    // Since it takes 1 clock for out to be updated after pattern match,
    // do the check first and then update exp_out value
    if (item.out != exp_out) begin
      `uvm_error("SCBD", $sformatf("ERROR ! out=%0d exp=%0d",
                                                item.out, exp_out))
    end else begin
      `uvm_info("SCBD", $sformatf("PASS ! out=%0d exp=%0d",
                                  item.out, exp_out), UVM_HIGH)
    end

    // If current index has reached the full pattern, then set exp_out to be 1
    // which will be checked in the next clock. If pattern is not complete, keep
    // exp_out to zero
    if (!(ref_pattern ^ act_pattern)) begin
      `uvm_info("SCBD", $sformatf("Pattern found to match, next out should be 1"), UVM_LOW)
        exp_out = 1;
    end else begin
      exp_out = 0;
    end

  endfunction
endclass
```

AGENT



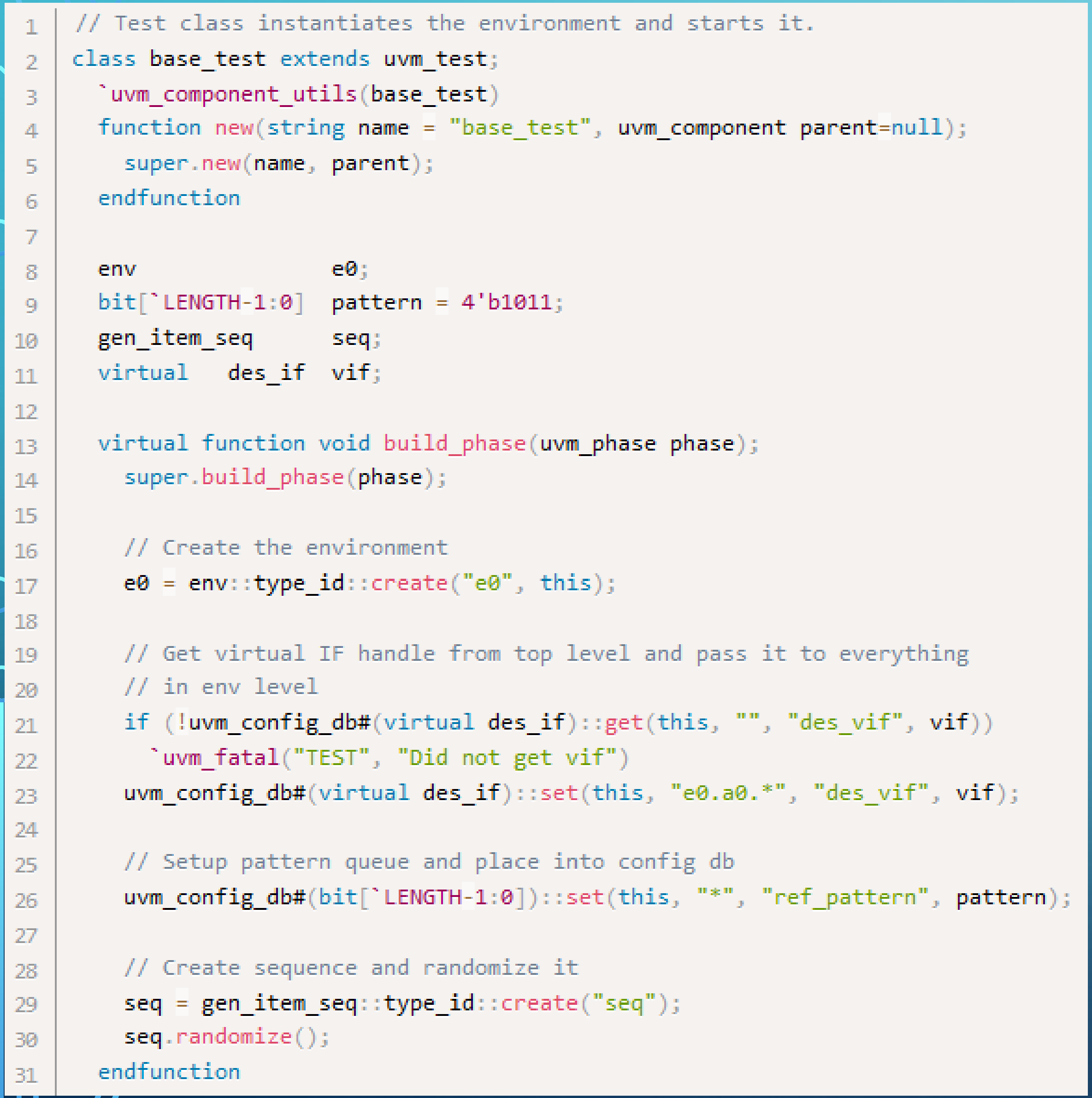
```
// Create an intermediate container called "agent" to hold
// driver, monitor and sequencer
class agent extends uvm_agent;
  `uvm_component_utils(agent)
  function new(string name="agent", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  driver         d0;          // Driver handle
  monitor        m0;          // Monitor handle
  uvm_sequencer #(Item) s0;          // Sequencer Handle

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    s0 = uvm_sequencer#(Item)::type_id::create("s0", this);
    d0 = driver::type_id::create("d0", this);
    m0 = monitor::type_id::create("m0", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    d0.seq_item_port.connect(s0.seq_item_export);
  endfunction

endclass
```

AMBIENTE

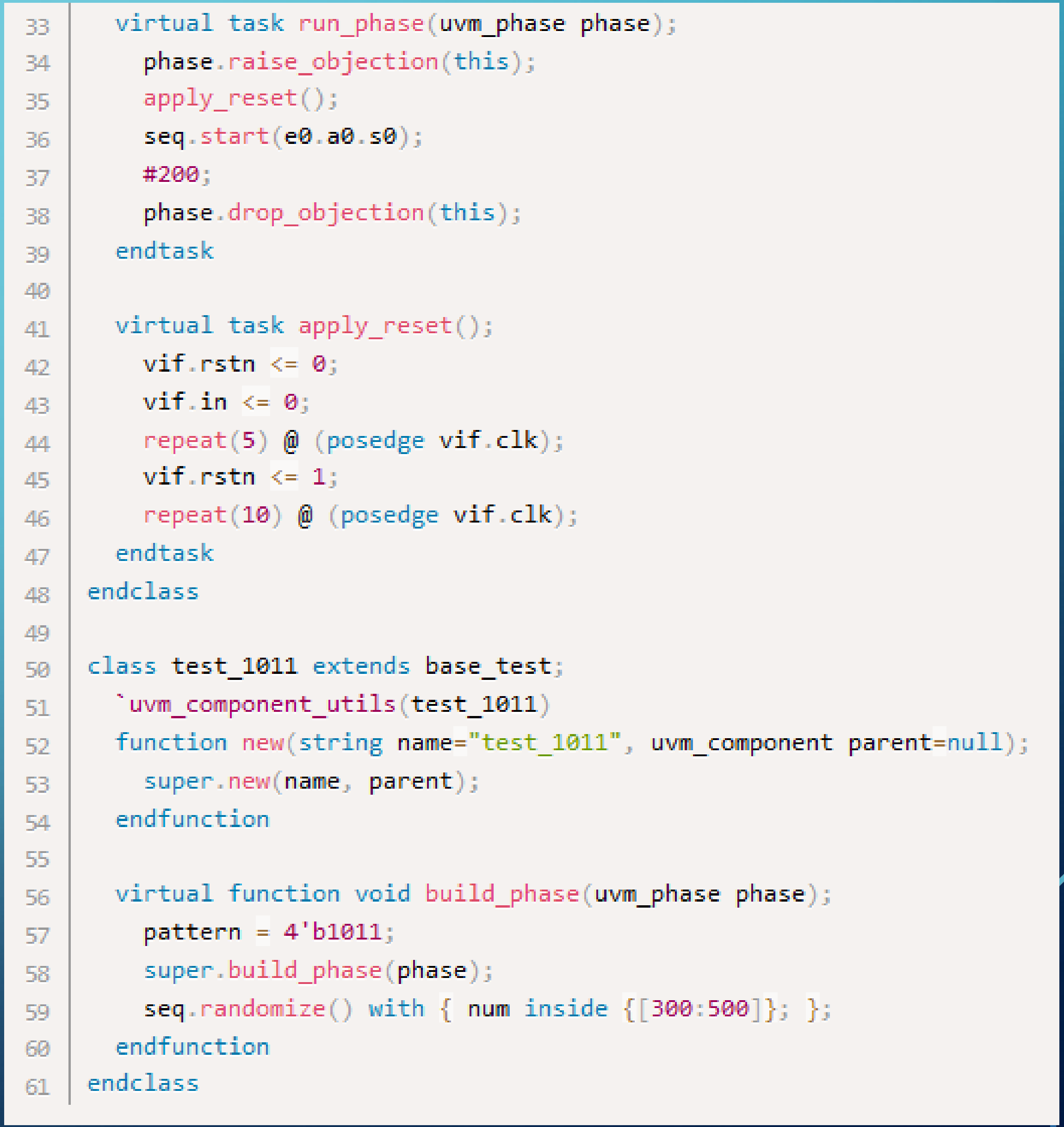


```

// The environment is a container object simply to hold
// all verification  components together. This environment can
// then be reused later and all components in it would be
// automatically connected and available for use
class env extends uvm_env;
  `uvm_component_utils(env)
  function new(string name="env", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  agent          a0;          // Agent handle
  scoreboard     sb0;         // Scoreboard handle

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    a0 = agent::type_id::create("a0", this);
    sb0 = scoreboard::type_id::create("sb0", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    a0.m0.mon_analysis_port.connect(sb0.m_analysis_imp);
  endfunction
endclass
```

PRUEBA

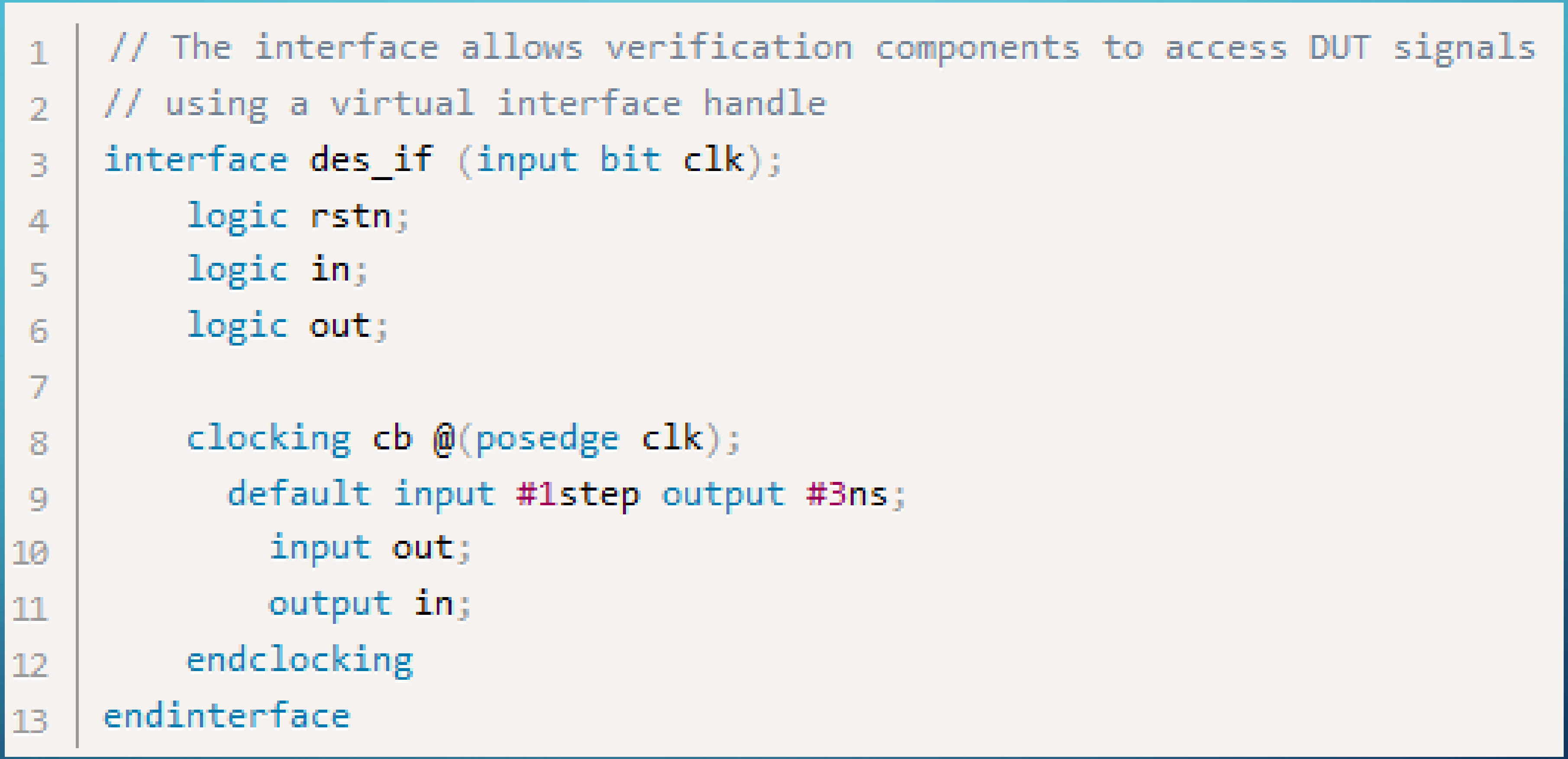


```
// Test class instantiates the environment and starts it.
class base_test extends uvm_test;
  `uvm_component_utils(base_test)
  function new(string name = "base_test", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  env                e0;
  bit[`LENGTH-1:0]   pattern = 4'b1011;
  gen_item_seq       seq;
  virtual   des_if   vif;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Create the environment
    e0 = env::type_id::create("e0", this);

    // Get virtual IF handle from top level and pass it to everything
    // in env level
    if (!uvm_config_db#(virtual des_if)::get(this, "", "des_vif", vif))
      `uvm_fatal("TEST", "Did not get vif")
    uvm_config_db#(virtual des_if)::set(this, "e0.a0.*", "des_vif", vif);

    // Setup pattern queue and place into config db
    uvm_config_db#(bit[`LENGTH-1:0])::set(this, "*", "ref_pattern", pattern);

    // Create sequence and randomize it
    seq = gen_item_seq::type_id::create("seq");
    seq.randomize();
  endfunction
```



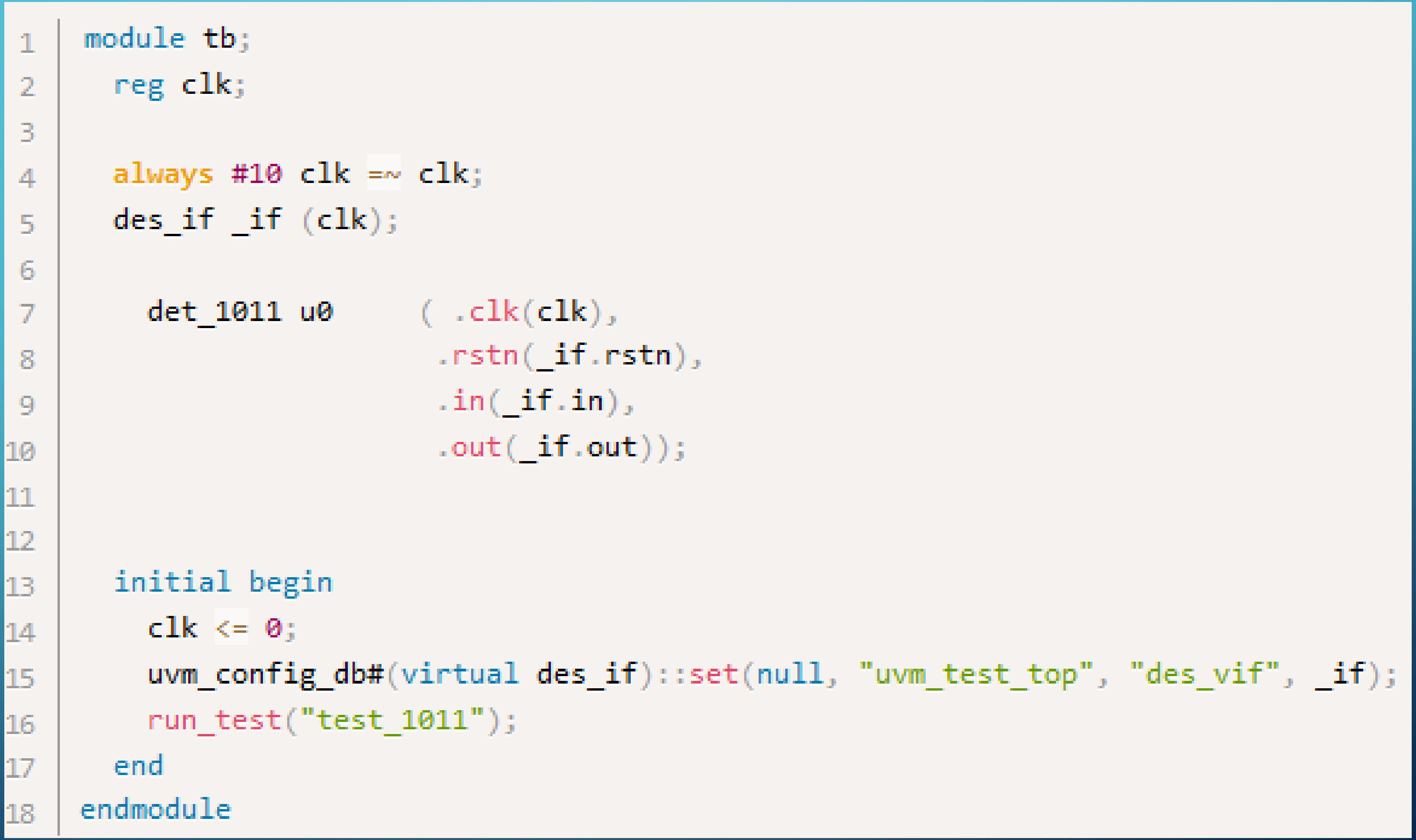
```
  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    apply_reset();
    seq.start(e0.a0.s0);
    #200;
    phase.drop_objection(this);
  endtask

  virtual task apply_reset();
    vif.rstn <= 0;
    vif.in <= 0;
    repeat(5) @ (posedge vif.clk);
    vif.rstn <= 1;
    repeat(10) @ (posedge vif.clk);
  endtask
endclass

class test_1011 extends base_test;
  `uvm_component_utils(test_1011)
  function new(string name="test_1011", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    pattern = 4'b1011;
    super.build_phase(phase);
    seq.randomize() with { num inside {[300:500]}; };
  endfunction
endclass
```

INTERFACE



```
// The interface allows verification components to access DUT signals
// using a virtual interface handle
interface des_if (input bit clk);
    logic rstn;
    logic in;
    logic out;

    clocking cb @(posedge clk);
      default input #1step output #3ns;
        input out;
        output in;
    endclocking
endinterface
```

TEST BENCH



```
module tb;
  reg clk;

  always #10 clk =~ clk;
  des_if _if (clk);

    det_1011 u0     ( .clk(clk),
                     .rstn(_if.rstn),
                     .in(_if.in),
                     .out(_if.out));


  initial begin
    clk <= 0;
    uvm_config_db#(virtual des_if)::set(null, "uvm_test_top", "des_vif", _if);
    run_test("test_1011");
  end
endmodule
```