2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

Handling Asynchronous Reset(s) Testing by building reset-awareness into UVM testbench components

Wei Wei Cheong, Xilinx, Edinburgh, United Kingdom (weiweic@xilinx.com)

Katherine Garden, Xilinx, Edinburgh, United Kingdom (*kgarden@xilinx.com*)

Ana Sanz Carretero, Xilinx, Dublin, Ireland (asanzcar@xilinx.com)

***Abstract*— Reset testing is a very common and crucial procedure to be carried out on the Device-Under-Test (DUT) in order to verify that the DUT is able to enter and exit reset phase cleanly and its performance still conforms to its specification after the reset process. Conventionally, an asynchronous reset that happens at a random point in the simulation, where the DUT maybe actively processing some in-flight traffic, is considered one of the most disruptive reset testing to the DUT and it is certainly one of the most problematic events to the testbench as well. Although there are some existing testbench architecture and techniques designed to handle reset testing at different complexity levels, this paper aims to present an alternative approach in scheduling one or more asynchronous resets throughout a simulation and handling these random reset events in the UVM testbench components by using the fork-disable_fork structure.**

Keywords*— *UVM; asynchronous reset; random reset; on the fly reset; reset awareness; testbench; UVM components;

I. INTRODUCTION

Typically, whenever a reset is asserted, the DUT is expected to halt its operations and return its internal states to default states, whereas on the other hand, the testbench needs to be able to synchronize with the DUT's behavior and able to react appropriately upon detecting the reset. Among the existing testbench solutions, there are mainly 2 testbench approaches in handling the reset events, one is by utilizing the UVM phasing and phase jumping method [4][5] and another one is by building reset awareness into UVM testbench components[1][2][3]. Although UVM has a built-in reset phase that is originally meant for tackling the reset testing, the lack of examples and documentation around UVM phasing methodology makes this method only suitable for verification engineers who have advanced UVM knowledge and the ability to do a deep dive into the UVM source code in order to implement it. The technique introduced by this paper has some similarities with [1][2][3] where all these techniques handle the reset events by making the UVM components and sequences aware of the reset events so that the testbench is able to respond to a random reset in a synchronized way. However, from the implementation point of view, this paper introduces an alternative approach in building the reset-aware mechanism. This approach has been implemented and used in a testbench that has a structure as shown in Figure 1. Other than the clock agent, all the elements and interfaces shown in Figure 1 reside in the same reset domain. From the DUT's reset specification point of view, any incomplete in-flight traffic, which is seen at the reset assertion point, will be discarded.

II. RESET INTERRUPTION ON SEQUENCE EXECUTION FLOW

Figure 2 shows a conventional sequence execution flow a testbench would have. First, a sequence is started at the test or virtual sequence level to generate stimulus in a UVM testbench. One or multiple sequence_items can be setup in the sequence, and they will be sent to the driver via the driver-sequencer communication mechanism. The driver then converts the information carried in a sequence_item into interface level activities. After consuming the sequence_item, the driver calls *item_done* and with that it completes the driver-sequencer handshake. It is very important to note that any uncompleted sequence threads, i.e. driver-sequencer handshakes, will lock up the sequencer and leave the driver-sequencer flow in deadlock. As a random reset event can be injected at any point of the simulation, it is very likely that a reset happens at the middle of a driver-sequencer handshake. Therefore, when

building the reset handling element, some attentions on the driver-sequencer flow are required to ensure that both sequencer and driver can response to the reset in a synchronized way and able to drive stimulus again once reset is lifted.

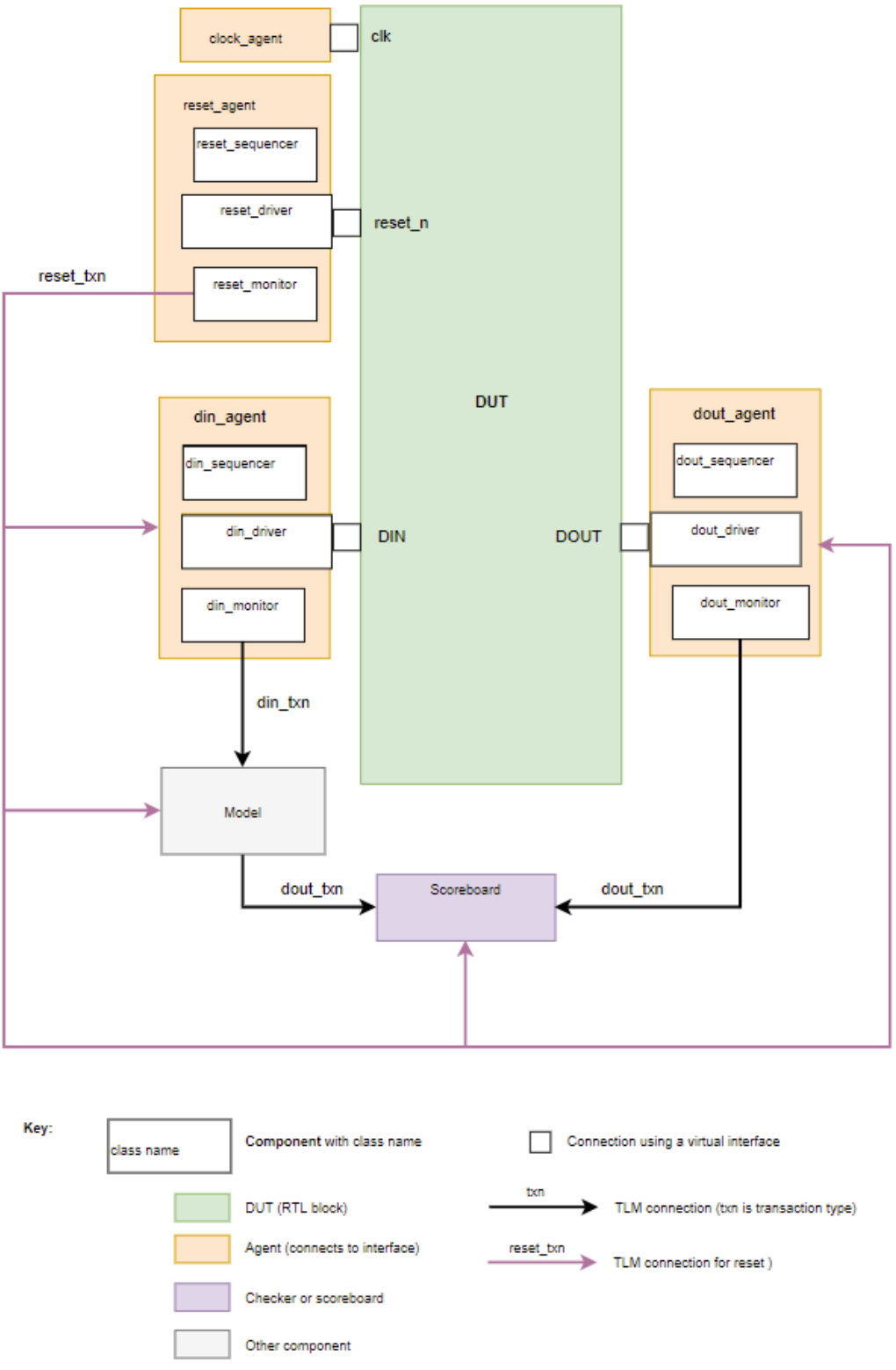

Figure 1 Testbench block diagram used by the reset handling technique proposed

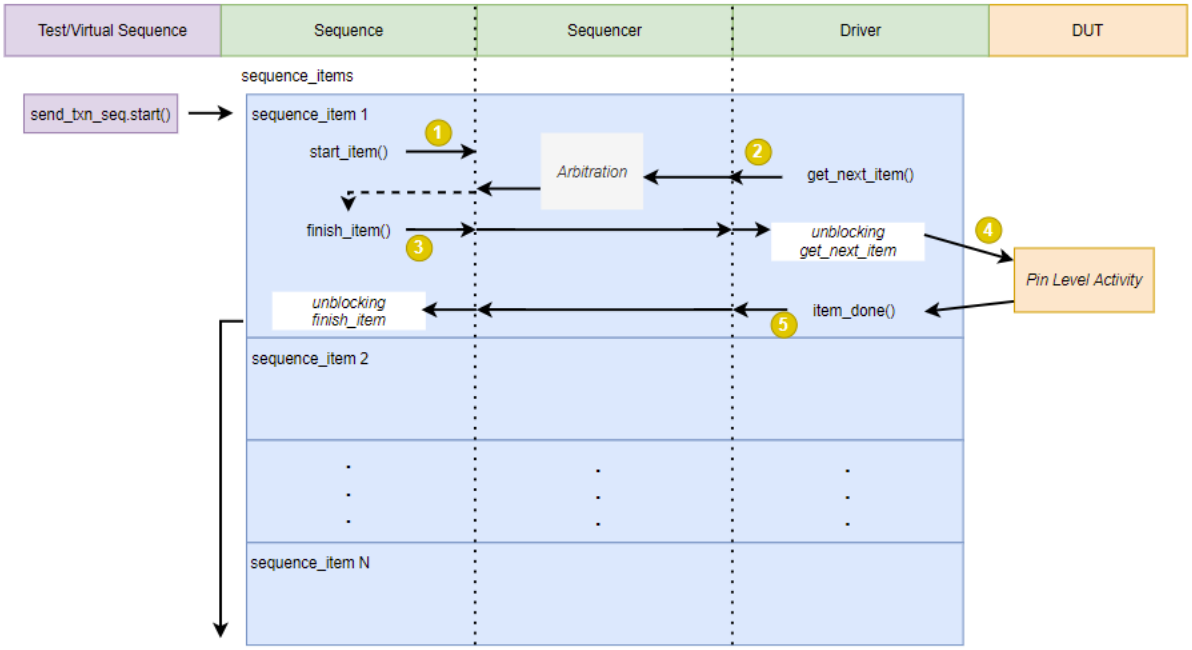

Figure 2 Sequence execution flow

III. RESET HANDLING IMPLEMENTATION OVERVIEW

This section focuses in explaining the reset handling technique from these perspectives below:

A. *Stimulus generation flow*

A top virtual sequence at the test level is used to contain the stimulus generation flow. In the virtual sequence, there are mainly 2 threads running, the reset thread and the main stimulus generation thread. The reset thread contains the reset sequence that drives a reset event at a random point in time. The number of reset events to be scheduled, randomly but sequentially, in a simulation can be controlled by a knob defined in the test configuration. The main stimulus generation thread contains the sequence that drives all the input transactions. The virtual sequence is structured in a way that upon the completion of a reset sequence in the reset thread and the reset is asserted, the main stimulus thread will be killed cleanly by the *disable_fork* and restarted again after the reset is lifted. And this process will be repeated as many times as the number of reset events set in the simulation. Then eventually the virtual sequence ends when all the reset events have happened, all the input transactions have been sent and all the expected output transactions have been received. Code example is shown in Figure 3.

B. *Reset generation flow*

In a UVM testbench, there are a few ways to generate a reset event, to detect and propagate a reset event, then react to it. In this methodology, a reset agent (which contains a reset driver, sequencer and a monitor) and sequences are used to generate and drive the reset sequence item. It is a requirement that all the other testbench components that needs to be reset-aware to have a reset TLM port or TLM FIFO to receive the reset sequence item that has been detected and broadcasted by the reset monitor.

C. *Input and Output agents*

Both input and output agents consist of a driver, a monitor and a sequencer. Each of these agents and their UVM components forks the *run_phase* into 2 threads, one is used as a reset detection thread and another one is used to execute normal operation of that component. When a reset is detected, the normal operation thread together with the reset polling thread will be killed by the *disable_fork*, then some pre-defined non-blocking clean-up activities can be serviced, and all these threads will be restarted after reset is lifted. The reset detection in the drivers and monitors are based on the reset assertion on the pin level whereas at the agent and sequencer level, the reset is

```
class base_vseq extends uvm_sequence;
     `uvm_object_utils(base_vseq)
     `uvm_declare_p_sequencer(base_vseqr)
     bit                     all_done;
     test_config             test_cfg_h;
     initial_reset_seq       init_rst_seq;
     single_reset_seq        during_reset_s;
     send_txn_seq            send_txn_s;
     protected int unsigned  reset_count = 0;
     rand int unsigned       reset_delay; // Waiting time before asserting reset
     rand int unsigned       reset_clk_duration; // The duration that reset staying asserted

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

     function void config_setup ();
       test_cfg_h = p_sequencer.env_cfg_h.test_cfg_h;
     endfunction : config_setup

     // Task: body
     task body();
       config_setup();
       do_poweron_reset();
       do begin
         fork
           begin : reset_thread
             if (reset_count < test_cfg_h.resets_during_test) begin
               schedule_reset_during_test();
             end else begin
               wait(0);  // no more resets to drive
             end
           end
           begin : main_thread
             // Do data processing as required by the test configuration
             send_transactions();
             // Wait for all output transactions to be received from the DUT, with a timeout
             wait_for_output_transactions(all_done);
           end
         join_any
         // Either the main stimulus generation sequence has finished, or a reset occured
         disable fork;
       end while (!all_done);
     endtask : body

     // Task: send_transactions
     task send_transactions();
       send_txn_s = send_txn_seq::type_id::create("send_txn_s",, get_full_name());
       if (!send_txn_s.randomize())
         `uvm_fatal("RANDOMIZE_FAIL", "Failed to randomize send_txn_s")
       send_txn_s.start(p_sequencer.din_seqr_h, this);
     endtask : send_transactions
```

```
   // Task: schedule_reset_during_test
     task schedule_reset_during_test();
       if (!this.randomize(reset_delay, reset_clk_duration))
         `uvm_fatal("RAND_ERR", "Failed to randomize reset_delay and reset_clk_duration")
       during_reset_s = single_reset_seq::type_id::create("during_reset_s", , get_full_name());
       if (!during_reset_s.randomize() with {seq_pre_reset_clk_duration == reset_delay;
                                             seq_reset_clk_duration     == reset_clk_duration;})
         `uvm_fatal("RANDOMIZE_FAIL", "Failed to randomize during_reset_s")
       during_reset_s.start(rst_sqr_h, this);
       reset_count++;
     endtask : schedule_reset_during_test
```

```
      // Task: wait_for_output_transactions
      task wait_for_output_transactions (output bit all_done);
        all_done = 0
        fork begin
          fork begin
            wait (p_sequencer.txn_scoreboard_h.all_done);
            all_done = 1;
          end
          begin
            #(test_cfg_h.timeout_from_last_input);
            `uvm_fatal("TIMEOUT", $sformatf("All outputs not received within %t after the
last input was sent", test_cfg_h.timeout_from_last_input))
          end
          join_any
          disable fork
        end join
      endtask : wait_for_output_transactions
    endclass : base_vseq
```

Figure 3 Example code of stimulus generation that includes reset events generation and handling

detected by receiving a reset sequence item broadcasted by the reset monitor. In the sequencer, upon detecting the reset, *stop_sequences* is executed to stop any running sequence. It is worth noting that there should be no hanging and uncompleted driver-sequencer handshakes after reset services in both the driver and sequencer are executed and the stimulus generator sequence should only be started again after the reset is lifted. Code examples for driver, monitor and sequencer are shown in Figure 4, Figure 5, and Figure 6, respectively.

```
    class din_driver extends uvm_driver #(din_txn,din_txn);
      `uvm_component_utils(din_driver)
      din_vif vif;
     ... ...
      task run_phase(uvm_phase phase);
        vif.reset(); // Reset the DUT din interface to its reset state. Note that this
doesn't drive the reset
        vif.wait_posedge_aclk();
        vif.wait_areset_deassert();
        while (1) begin
          fork
            begin : ACTIVE
              // the main task that executes get_next_item and item_done in a forever-loop
              run_active();
            end
            begin : RESET_SERVICE
              // Detects a reset at pin level
              vif.wait_areset_asserted();
            end
          join_any
          disable fork;

          // Reset the DUT din interface to its reset state. Note that this doesn't drive
the reset
          vif.reset();
          // Wait until reset is lifted
          vif.wait_areset_deassert();
          `uvm_info(get_name(), $sformatf("RESET Released"),UVM_LOW)
        end
      endtask
     ... ...
    endclass : din_driver
```

Figure 4 Example code of inserting reset handling in a driver

```
class din_monitor extends uvm_monitor;
  `uvm_component_utils(din_monitor)
  din_vif vif;
... ...
  virtual task run_phase(uvm_phase phase);
    vif.wait_posedge_aclk();
    vif.wait_areset_deassert();
    while (1) begin
      fork
        begin : ACTIVE
          run_active();
        end
        begin : RESET_SERVICE
          // Polling for a reset
          vif.wait_areset_asserted();
        end
      join_any
      disable fork;
      `uvm_info(get_name(), $sformatf("RESET DETECTED"),UVM_LOW)

      // Waiting until reset is lifted
      vif.wait_areset_deassert();
    end
  endtask : run_phase
... ...
endclass : din_monitor
```

Figure 5 Example code of inserting reset handling in a monitor

```
`uvm_analysis_imp_decl(_reset)

class din_seqr extends uvm_sequencer #(din_txn);
  `uvm_sequencer_utils(din_seqr)
  // TLM analysis imp to receive resets from the reset agent
  uvm_analysis_imp_reset #(bit, din_seqr) resetflag_export;
... ...

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Build TLM components
    resetflag_export = new("resetflag_export", this);
  endfunction : build_phase

  // Handle reset
  virtual function void write_reset(bit t);
    // Stop any running sequence
    stop_sequences();
    // Reset any variables that should be reset e.g. the din_txn count
    din_txn_count = 0;
  endfunction : write_reset

endclass : din_seqr
```

Figure 6 Example code of inserting reset handling in a sequencer

D. Scoreboard

The scoreboard detects the reset by receiving a reset transaction broadcasted by the reset monitor. Upon detecting the reset, it resets all the necessary elements in scoreboard such as clearing the queue that storing the predictions generated by the model. A code example for this section is shown in Figure 7.

E. Other UVM components such as Model

The reset handling in the model is the same as the scoreboard.

F. An example in extending this reset handling method in a layered agent

Previously, the input and output agents are viewed as simple agents that schedule and drives sequence items at their simplest form without any translation from another higher abstracted sequence item. However, in the real testbench with this reset handling technique implemented, layered agents and sequences have been used. For example, the sequence_item created at the *send_txn_seq* is with din_txn type and it is handled by an upper layer abstracted din_agent and din_sequencer. The din_txn sequence item is then being translated into an AXI-Stream type via a translator sequence. Then the AXI-Stream type sequence_item is sent to an AXI-S driver via an AXI-S type sequencer. When a reset event happens, the din_txn sequencer and AXI-S sequencer should both be reset and their *stop_sequences* function should be executed. A code example for this section is shown in Figure 8.

```
  class scbd extends uvm_component;
    `uvm_component_utils(scbd)
    // TLM port for flags from the reset monitor (core has been reset)
    uvm_analysis_export #(bit)   resetflag_export;
    // Receive monitored reset events
    uvm_tlm_analysis_fifo #(bit) resetflag_fifo;
  ... ...
    virtual function void build_phase(uvm_phase phase);
      resetflag_export = new ("resetflag_export", this);
      resetflag_fifo   = new ("resetflag_fifo", this);
    endfunction : build_phase

  virtual function void connect_phase(uvm_phase phase);
      resetflag_export.connect(resetflag_fifo.analysis_export);
    endfunction : connect_phase

    virtual task run_phase(uvm_phase phase);
      forever begin
        fork
          begin : ACTIVE
            run_active();
          end
          begin : RESET_SERVICE
            handle_reset();
          end
        join_any
        disable fork;
      end
    endtask : run_phase

    task handle_reset();
      bit resetflag;

      // Wait until a reset occurs
      resetflag_fifo.get(resetflag);

      // Display the transactions that will be flushed
      `uvm_info(get_name(), $sformatf("Reset occurred: [ matches = %0d, mismatches = %0d]:
removing    %0d    unmatched    transactions    (+    %0d        unmatched    transactions)",
m_matches, m_mismatches, received_data.size(),actual_received_data.size()), UVM_LOW)

      // Reset the variables that needed to be reset
      received_data.delete();
      actual_received_data.delete();
      exp_count = 0;
      act_count = 0;
      all_done = 0;
    endtask: handle_reset
  ... ...
  endclass : scbd
```

Figure 7 Example code of inserting reset handling in a scoreboard

```
class din_agent extends uvm_agent;
  `uvm_component_utils(din_agent)
... ...
  din_cfg cfg;

  // TLM port for flags from the reset monitor (core has been reset)
  uvm_analysis_export #(bit)    resetflag_export;

  // Receive monitored reset events
  uvm_tlm_analysis_fifo #(bit) resetflag_fifo;

  // Higher level sequencer to create din_txn which contains stimulus with higher level of
abstraction
  // This is a just a handle. The instance will be created in the env and then pass into
this handle.
  // This is the same sequencer being used in base_vseq to start the send_txn_seq
  din_seqr    din_sequencer_h;

  // AXIS transaction sequencer which handles sequence that
  // creates axis_transaction based on stimulus info translated from din_txn
  axis_mst_sequencer axis_sequencer;

  // AXI-S Data Input interfaces - this is a "Translator Sequence"
  // start() is called on this sequence here and it runs forever
  axis_mst_din_base_seq axis_mst_din_s;

  // AXI-Stream agent environments
  axis_mst_env  axis_env;

  virtual function void build_phase(uvm_phase phase);
... ...
    // Build TLM components
    resetflag_export = new ("resetflag_export", this);
    resetflag_fifo   = new("resetflag_fifo", this);

    // Build layered axis agent env
    axis_env = axis_mst_env::type_id::create("axis_env", this);

    // Create the Translator Sequence
    axis_mst_din_s =
axis_mst_din_base_seq::type_id::create("axis_mst_din_s",,get_full_name());
    axis_mst_din_s.din_sequencer_h = din_sequencer_h;
  endfunction : build_phase

  virtual function void connect_phase(uvm_phase phase);
    // Hook up sequencer used by axis agent and sequence
    axis_sequencer = m_din.agent.sequencer;
    // Connect reset to agent components
    resetflag_export.connect(mon.resetflag_fifo.analysis_export);
    resetflag_export.connect(resetflag_fifo.analysis_export);
  endfunction :connect_phase

  // Start the translator sequence, which runs forever
  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);

    forever begin
      fork
        begin : ACTIVE
          run_sequence();
        end
        begin : RESET_SERVICE
          handle_reset();
        end
      join_any
      disable fork;
    end
  endtask : run_phase
```

```
    task run_sequence();
      // Start the translator sequence using axis sequencer
      axis_mst_din_s.start(axis_sequencer);
    endtask: run_sequence

    task handle_reset();
      bit resetflag;
      // Wait until a reset occurs
      resetflag_fifo.get(resetflag);

      // Stop sequences on axis agent sequencers
      axis_sequencer.stop_sequences();
    endtask: handle_reset
  ... ...
endclass : din_agent
```

Figure 8 Example code of reset handling in a layered agent

IV. ACKNOWLEDGMENT

We would like to extend our appreciation to our senior colleague at Xilinx, Chris Clegg who owned the original version of this reset methodology, for sharing his testbench knowledge and allowing us to present this methodology in this paper.

V. CONCLUSIONS

This paper presents a reset testing flow that has these benefits: (a) enables the testbench to trigger one or more reset events in some random points of a simulation, and (b) ensures the various parts of the testbench are capable to terminate their active threads cleanly when a reset event is detected, and (c) enables the normal operation to be restarted in a synchronized way after the DUT exits from reset.

VI. REFERENCES

[1] Mark Peryer, “On the fly reset,” Mentor Graphics. (*references*)

[2] Cristian Slav, “How to handle reset in UVM,” http://cfs-vision.com/2016/04/18/systemverilog-how-to-handle-reset-in-uvm/ (online references).

[3] Cristian Slav, “How to handle reset in UVM part2,” http://cfs-vision.com/2017/06/14/systemverilog-how-to-handle-reset-in-uvm-part-2/ (online references).

[4] Brian Hunter, Ben Chen, Cavium. Rebecca Lipon, Synopsys, “Reset Testing Made Simple with UVM Phases,”.

[5] “How to handle Reset in UVM,” www.learnuvmverification.com (online references).

[6] “UVM cookbook”, Verification Academy (references)