2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE MUNICH, GERMANY OCTOBER 14-15, 2025

# Unified UVM Testbench: Integrating Random, Directed and Pseudo-Random Verification Capabilities

Kilaru Vamsikrishna, Cadence Design System, Bengaluru, India (*kilaruv@cadence.com*)

Amitav Mitra, Cadence Design System, Noida, India (amitav@*cadence.com*)

Salehabibi Shaikh, Cadence Design System, Bengaluru, India (*ssalehab@cadence.com*)

Sushrut B Veerapur, Cadence Design System, Bengaluru, India (*sushrut@cadence.com*)

***Abstract*- Most testbench environments use separate setups for random, pseudo-random, and directed verification strategies, leading to duplicate efforts and limited reusability. This fragmentation results in redundant development, inconsistent methodologies, and delays in verification cycles. As projects progress—from directed tests early on to random exploration in the middle and pseudo-random patterns for targeted coverage closure toward the end—maintaining isolated environments becomes inefficient. The proposed solution is a unified UVM-based testbench that integrates all verification modes into a single configurable environment. By supporting mode selection through configuration, dynamic layering of sequences, and utilizing a reusable testbench library, this approach reduces overhead, enhances reusability from IP to SoC levels, and streamlines test development throughout the verification lifecycle.**

***Keywords— Unified UVM Testbench, UVM Random Testbench, UVM Pseudo Random Testbench, UVM Directed Testbench; UVM; Design Verification***

## I. INTRODUCTION

As hardware designs grow more complex, verification, particularly in UVM-based testbenches, faces increasing challenges. Traditionally, verification teams maintain separate testbenches for different phases—directed, random, and pseudo-random testing—each requiring its own infrastructure. While effective in isolation, this approach leads to redundant development, increased maintenance, and limited reuse. Furthermore, transitioning between verification stages becomes cumbersome, and coverage gaps can arise, leaving potential bugs undetected.

To address these issues, we propose a unified UVM-based testbench that integrates directed, random, and pseudo-random verification modes into a single, configurable environment. This unified architecture supports dynamic configuration of stimulus modes, modular sequence layering, and a reusable testbench library, streamlining the transition between verification phases. By reducing overhead, improving resource utilization, and enhancing reusability across projects and verification stages, this solution accelerates coverage closure and simplifies the verification process.

## II. METHODOLOGY

The verification approach typically evolves across three main phases of the project—starting with simple directed testing, progressing to random stimulus generation, and ending with a focused mix of pseudo-random and directed techniques aimed at closing functional and code coverage.

Using separate operation block also produces a rather useful side effect, it isolates operations from each other reducing interdependencies and thus keeps debugs isolated to each operation block.

**Pseudo-Random Testbench methodology:** Pick and push operations randomly from the list of available legal operations. This is controlled via protocol specific constraints based on element positions in the operation queue.

**Directed Testbench flow:** User specifies what operation to push in the operation queue. This allows the user to create any and every kind of testcase scenario with the available operations.

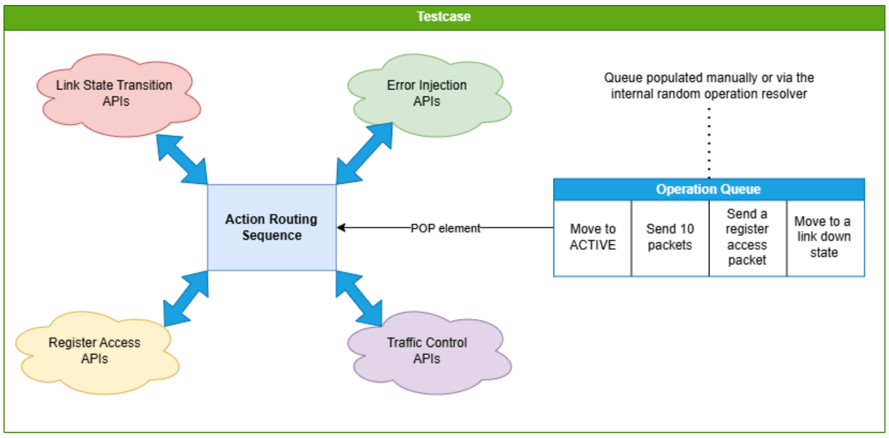

Testcase
Link State Transition APIs
Error Injection APIs
Queue populated manually or via the internal random operation resolver
Action Routing Sequence
POP element
Operation Queue
Move to ACTIVE
Send 10 packets
Send a register access packet
Move to a link down state
Register Access APIs
Traffic Control APIs

*Figure 1: Operation queue approach directed/pseudo random*

**Constraint Randomization flow:** Constrained randomization ensures that all valid, legal scenarios are tested randomly. This approach helps mimic real-world conditions, allowing the design to be stressed under various unpredictable situations. By generating a wide range of possible input combinations, this technique helps uncover edge cases that might not be found through traditional deterministic testing.

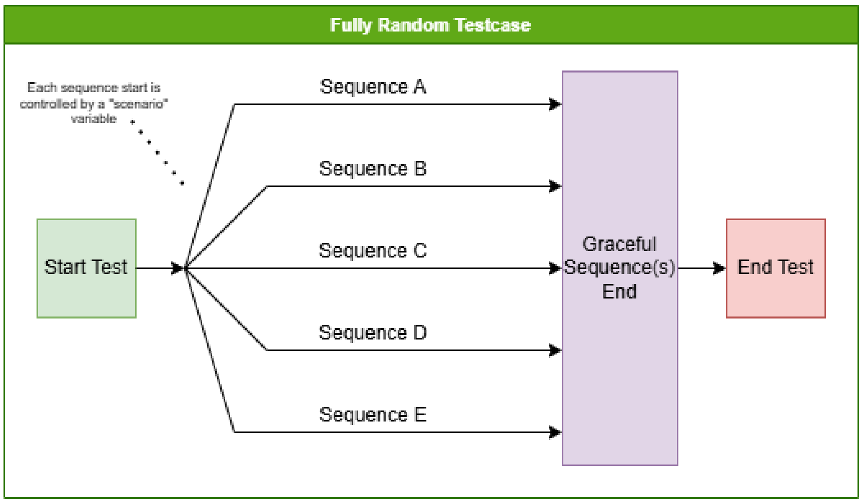
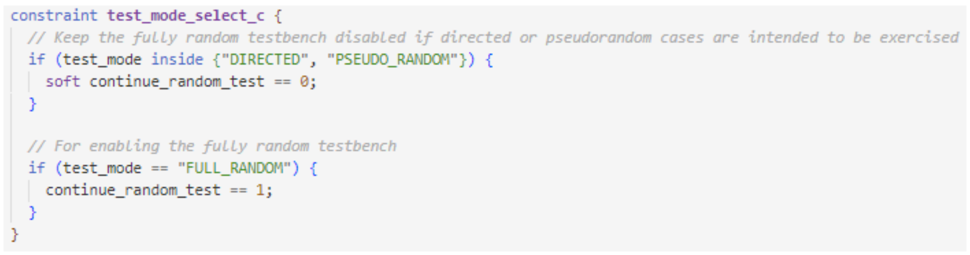


*Figure 2: Fully Random Parallel Sequence Control*

Start with directed tests early in the project, use fully random test cases during development, and switch to pseudo-random or directed test at the end to close FC/CC gaps.

This approach breaks down the testbench functionality into multiple granular blocks of operation independent of each other as sub-sequences running either sequentially or parallel to each other with the help of a dynamic queueing mechanism. Unified control architecture used to support directed, random, and pseudo-random test scenarios. At the core is the Action Routing Sequence, which reads operations from a queue and calls the appropriate protocol scenario specific APIs. The operation queue can be filled manually (for directed tests), through constrained random generation (for random tests), or a mix of both (for pseudo-random tests). This design allows a single testbench to handle all types of verification needs efficiently, while maintaining modularity and protocol-specific behavior.



```
constraint test_mode_select_c {
  // Keep the fully random testbench disabled if directed or pseudorandom cases are intended to be exercised
  if (test_mode inside {"DIRECTED", "PSEUDO_RANDOM"}) {
    soft continue_random_test == 0;
  }

  // For enabling the fully random testbench
  if (test_mode == "FULL_RANDOM") {
    continue_random_test == 1;
  }
}
```

*Figure 3: Constraint for allowing Radom testbench mode to be enabled or disabled*

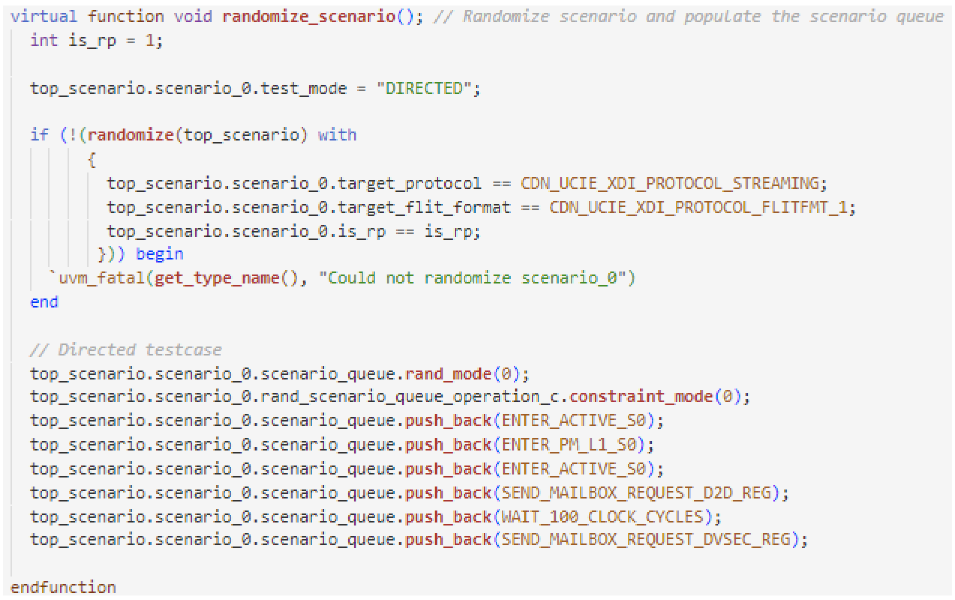
Each block of operation corresponds to a dedicated sequence in the testbench's sequence library. For forming a testcase, encodings with respect to each required block of operation are then pushed into a global queue. Expanding functionality in terms of adding more operation in cases where new features are added is as simple as creating a sub sequence type API and defining it in the operation list.

## III. WORKING MODEL

**Directed Testing:** Directed testing is the first step in the verification process, where known input patterns are used to exercise specific DUT features or bring up the environment. This approach uses fixed stimulus and predictable responses to validate basic functionality and ensure that the testbench infrastructure is correctly set up. It is especially useful in the early stages of the project when both the DUT and the environment are still stabilizing

The primary goal is environment bring-up and basic DUT validation. Here, simple directed testcases are employed to verify known, deterministic behavior.

- Smoke/Sanity tests are used to confirm end-to-end connectivity.
- The scenario queue is made non-random and all the constraints which are applied on the scenario queue are disabled. This ensures a completely directed behavior with no randomness involved.
- This phase ensures that the testbench components (drivers, monitors, scoreboard, sequencers) are correctly implemented and that the DUT responds to basic transactions
- Although the scenario queue is declared as random to support later test phases, in directed testing we disable its rand_mode and turn off all related constraints. This ensures the queue is filled manually with fixed operations, providing fully deterministic behavior. It helps maintain strict control during environment bring-up and makes debugging easier in early project stages.



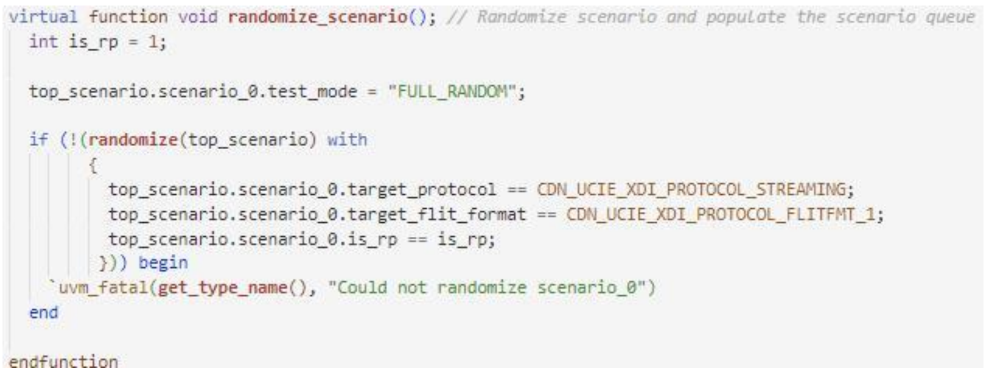
```
virtual function void randomize_scenario(); // Randomize scenario and populate the scenario queue
  int is_rp = 1;

  top_scenario.scenario_0.test_mode = "DIRECTED";

  if (!(randomize(top_scenario) with
        {
          top_scenario.scenario_0.target_protocol == CDN_UCIE_XDI_PROTOCOL_STREAMING;
          top_scenario.scenario_0.target_flit_format == CDN_UCIE_XDI_PROTOCOL_FLITFMT_1;
          top_scenario.scenario_0.is_rp == is_rp;
        })) begin
    `uvm_fatal(get_type_name(), "Could not randomize scenario_0")
  end

  // Directed testcase
  top_scenario.scenario_0.scenario_queue.rand_mode(0);
  top_scenario.scenario_0.rand_scenario_queue_operation_c.constraint_mode(0);
  top_scenario.scenario_0.scenario_queue.push_back(ENTER_ACTIVE_S0);
  top_scenario.scenario_0.scenario_queue.push_back(ENTER_PM_L1_S0);
  top_scenario.scenario_0.scenario_queue.push_back(ENTER_ACTIVE_S0);
  top_scenario.scenario_0.scenario_queue.push_back(SEND_MAILBOX_REQUEST_D2D_REG);
  top_scenario.scenario_0.scenario_queue.push_back(WAIT_100_CLOCK_CYCLES);
  top_scenario.scenario_0.scenario_queue.push_back(SEND_MAILBOX_REQUEST_DVSEC_REG);

endfunction
```

*Figure 4: configure TB to from the test to run in "DIRECTED" mode*

**Constrained-Random Testing:** Once the environment is stable and the testbench components are fully functional the goal shifts from basic validation to exploring the entire design space by generating a wide range of valid input scenarios. Randomization is guided by protocol-aware constraints, enabling effective coverage of both typical and corner-case behaviors. It enables all the constraints and scenario class is fully randomized. This approach is essential for uncovering issues that may not be visible during directed testing. The key aspects of constrained-random testing include:

- Broad coverage of legal input combinations.
- Verification of corner cases and uncommon state transitions.
- Identification of issues not easily captured through directed testing.

By combining intelligent constraint definition with functional coverage feedback, constrained-random testing helps accelerate coverage closure and ensures a deeper, more robust verification process.



```
virtual function void randomize_scenario(); // Randomize scenario and populate the scenario queue
  int is_rp = 1;

  top_scenario.scenario_0.test_mode = "FULL_RANDOM";

  if (!(randomize(top_scenario) with
        {
          top_scenario.scenario_0.target_protocol == CDN_UCIE_XDI_PROTOCOL_STREAMING;
          top_scenario.scenario_0.target_flit_format == CDN_UCIE_XDI_PROTOCOL_FLITFMT_1;
          top_scenario.scenario_0.is_rp == is_rp;
        })) begin
    `uvm_fatal(get_type_name(), "Could not randomize scenario_0")
  end

endfunction
```

*Figure 5: configure TB to from the test to run in “DIRECTED” mode*

**Dynamic Sequence Management Using Scenario Control in UVM:** With using a modular and runtime-configurable UVM testbench using a centralized scenario control knob (Figure 6: do_mailbox_access). This architecture supports seamless integration of directed, random, and pseudo-random test modes.

1. Sequence Initiation with Scenario Control

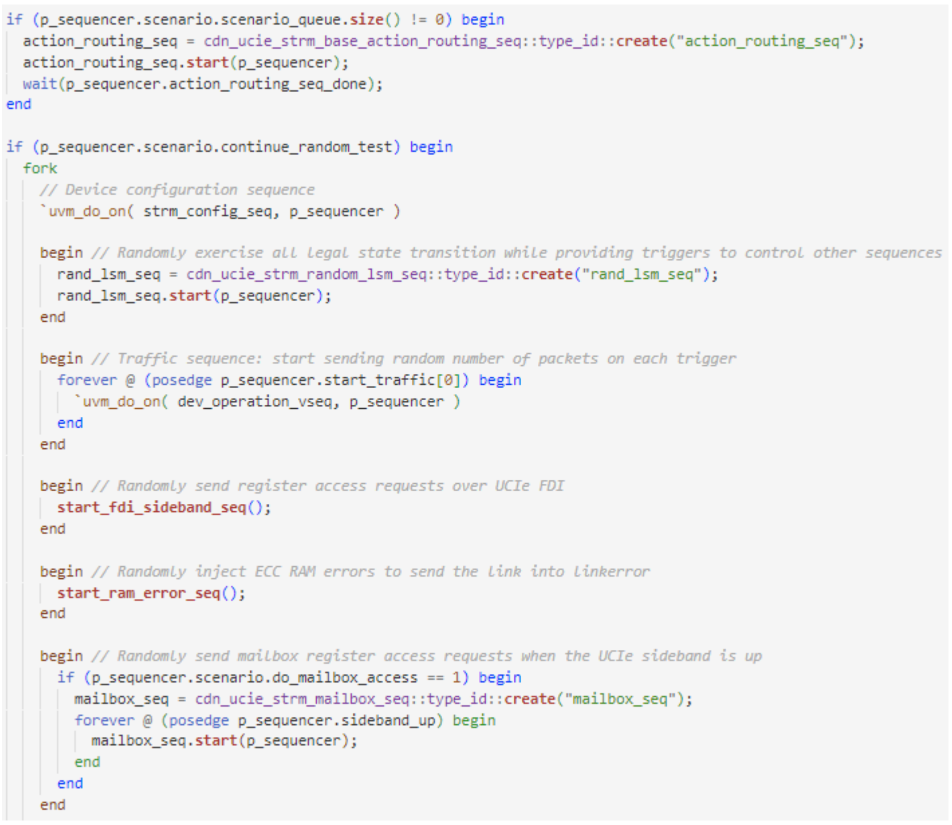
    - Sequences are enabled conditionally using scenario control flags (scenario.<knob>).
    - The type and number of sequences are protocol-specific or requirement-driven.
    - In our implementation, we illustrate this using the UCIE protocol.
    - All sequences are launched inside a fork...join_none block, they run as parallel background threads.
    - Allows runtime selection of test behavior without modifying test code.

2. Graceful Termination of Parallel Sequences

    - Each sequence uses a “*_done” (eg: mailbox_seq_done) flag for lifecycle control.
    - Flag is set to 0 at start and updated to 1 upon completion.
    - Main control waits for all “*_done” signals before completing the top-level sequence.
    - Ensures deterministic test completion and avoids premature exits.
    - Prevents interleaved or inconsistent behavior in complex protocols like UCIE.

3. Synchronization Across Threads

    - Certain threads need coordination, e.g., pause traffic when state machine transitions to linkdown/reset.
    - Achieved using uvm_event for inter-thread signaling.
    - Enables clean and modular synchronization between parallel sequences without tight coupling.



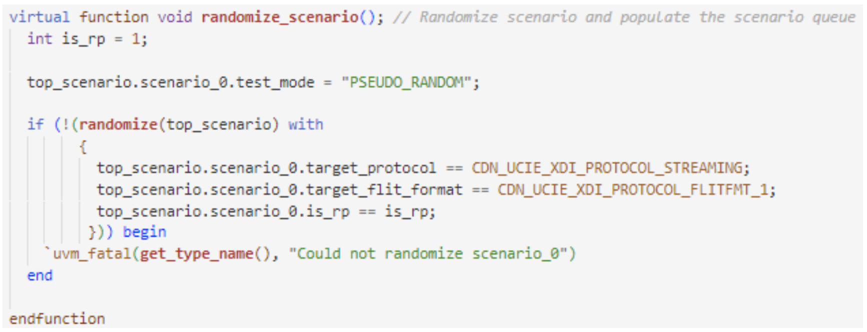
```
if (p_sequencer.scenario.scenario_queue.size() != 0) begin
  action_routing_seq = cdn_ucie_strm_base_action_routing_seq::type_id::create("action_routing_seq");
  action_routing_seq.start(p_sequencer);
  wait(p_sequencer.action_routing_seq_done);
end

if (p_sequencer.scenario.continue_random_test) begin
  fork
    // Device configuration sequence
    `uvm_do_on( strm_config_seq, p_sequencer )

    begin // Randomly exercise all legal state transition while providing triggers to control other sequences
      rand_lsm_seq = cdn_ucie_strm_random_lsm_seq::type_id::create("rand_lsm_seq");
      rand_lsm_seq.start(p_sequencer);
    end

    begin // Traffic sequence: start sending random number of packets on each trigger
      forever @ (posedge p_sequencer.start_traffic[0]) begin
        `uvm_do_on( dev_operation_vseq, p_sequencer )
      end
    end

    begin // Randomly send register access requests over UCIe FDI
      start_fdi_sideband_seq();
    end

    begin // Randomly inject ECC RAM errors to send the link into linkerror
      start_ram_error_seq();
    end

    begin // Randomly send mailbox register access requests when the UCIe sideband is up
      if (p_sequencer.scenario.do_mailbox_access == 1) begin
        mailbox_seq = cdn_ucie_strm_mailbox_seq::type_id::create("mailbox_seq");
        forever @ (posedge p_sequencer.sideband_up) begin
          mailbox_seq.start(p_sequencer);
        end
      end
    end
```

*Figure 6: Parallel Sequence Operation for random testing*

**Pseudo-Random and Directed Testing for Coverage Closure:** In the final phase of the verification cycle, the primary focus shifts to closing remaining Functional Coverage (FC) and Code Coverage (CC) gaps. At this stage, pseudo-random testcases are crafted by tightening constraints or applying bias to guide the stimulus toward uncovered or hard-to-hit scenarios. In parallel, directed tests are reused or newly added to specifically target uncovered bins or logic paths that constrained-random tests may have missed. This targeted approach improves regression efficiency and ensures that even rare or corner-case conditions are fully validated. Key aspects of this phase include:

- Pseudo-random stimulus is generated by tightening constraints or adding bias to existing random sequences.
- Testcases are tuned to target hard-to-reach protocol behaviors or corner-case conditions.
- Directed testcases are reused or created to explicitly hit specific functional coverage points or unexercised code paths.
- Helps reduce regression runtime by avoiding broad random testing where focused effort is more effective.
- Ensures thorough coverage closure by combining the precision of directed tests with the reach of controlled randomness.

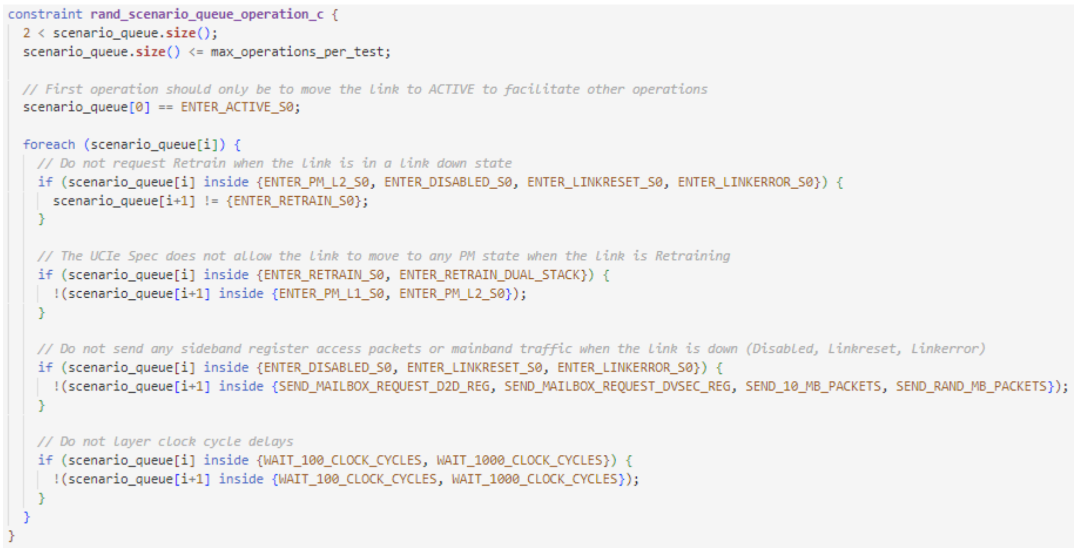
This hybrid strategy ensures that both common and rare use-cases are validated thoroughly, maximizing coverage efficiency in the final stages of verification. Setting test_mode = "PSEUDO_RANDOM" within the scenario object is a key configuration that informs the testbench which verification mode is being used. This value acts as a flag to enable pseudo-random test generation logic throughout the environment. Based on this mode, the testbench selectively activates constraints, biases, or custom sequences tailored to pseudo-random behavior. This allows the same testbench infrastructure to adapt dynamically to different verification phases such as directed, random, or pseudo-random, without requiring structural changes.



```
virtual function void randomize_scenario(); // Randomize scenario and populate the scenario queue
  int is_rp = 1;

  top_scenario.scenario_0.test_mode = "PSEUDO_RANDOM";

  if (!(randomize(top_scenario) with
        {
          top_scenario.scenario_0.target_protocol == CDN_UCIE_XDI_PROTOCOL_STREAMING;
          top_scenario.scenario_0.target_flit_format == CDN_UCIE_XDI_PROTOCOL_FLITFMT_1;
          top_scenario.scenario_0.is_rp == is_rp;
        })) begin
    `uvm_fatal(get_type_name(), "Could not randomize scenario_0")
  end

endfunction
```

*Figure 7: Configure TB from test to run in "PSEUDO_RANDOM" mode*

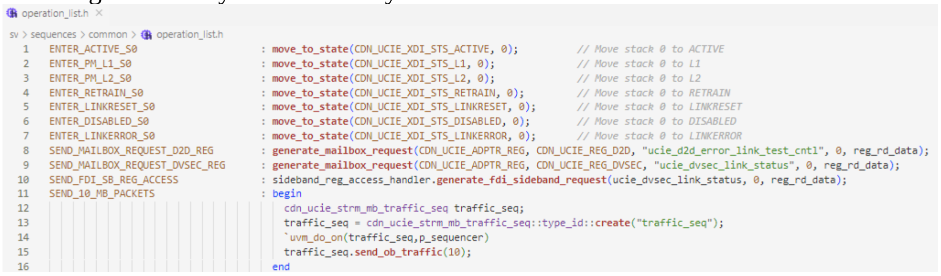
The constraints shown in the code are protocol- or specification-specific, tailored here for the UCIE protocol. These rules guide how operations are allowed to sequence based on valid link and power state behaviors. However, they are designed in a modular way, allowing users to modify or extend them easily. Depending on the protocol being verified, users can plug and play their own constraint logic into the same structure-making the testbench flexible, reusable, and adaptable across various designs.



```
constraint rand_scenario_queue_operation_c {
  2 < scenario_queue.size();
  scenario_queue.size() <= max_operations_per_test;

  // First operation should only be to move the link to ACTIVE to facilitate other operations
  scenario_queue[0] == ENTER_ACTIVE_S0;

  foreach (scenario_queue[i]) {
    // Do not request Retrain when the link is in a link down state
    if (scenario_queue[i] inside {ENTER_PM_L2_S0, ENTER_DISABLED_S0, ENTER_LINKRESET_S0, ENTER_LINKERROR_S0}) {
      scenario_queue[i+1] != {ENTER_RETRAIN_S0};
    }

    // The UCIe Spec does not allow the link to move to any PM state when the link is Retraining
    if (scenario_queue[i] inside {ENTER_RETRAIN_S0, ENTER_RETRAIN_DUAL_STACK}) {
      !(scenario_queue[i+1] inside {ENTER_PM_L1_S0, ENTER_PM_L2_S0});
    }

    // Do not send any sideband register access packets or mainband traffic when the link is down (Disabled, Linkreset, Linkerror)
    if (scenario_queue[i] inside {ENTER_DISABLED_S0, ENTER_LINKRESET_S0, ENTER_LINKERROR_S0}) {
      !(scenario_queue[i+1] inside {SEND_MAILBOX_REQUEST_D2D_REG, SEND_MAILBOX_REQUEST_DVSEC_REG, SEND_10_MB_PACKETS, SEND_RAND_MB_PACKETS});
    }

    // Do not layer clock cycle delays
    if (scenario_queue[i] inside {WAIT_100_CLOCK_CYCLES, WAIT_1000_CLOCK_CYCLES}) {
      !(scenario_queue[i+1] inside {WAIT_100_CLOCK_CYCLES, WAIT_1000_CLOCK_CYCLES});
    }
  }
}
```

*Figure 8: Constraints for pseudo-random testing*

This operation list acts as a unified execution layer that maps abstract operations from the scenario queue to specific testbench tasks or APIs. Pseudo-random tests apply guided constraints to influence which operations get pushed. Once the scenario is resolved, this file ensures all modes invoke the exact same behavior for any given operation, maintaining consistency and modularity across the verification flow.



operation_list.h

sv > sequences > common > operation_list.h

```
ENTER_ACTIVE_S0                  : move_to_state(CDN_UCIE_XDI_STS_ACTIVE, 0);          // Move stack 0 to ACTIVE
ENTER_PM_L1_S0                   : move_to_state(CDN_UCIE_XDI_STS_L1, 0);              // Move stack 0 to L1
ENTER_PM_L2_S0                   : move_to_state(CDN_UCIE_XDI_STS_L2, 0);              // Move stack 0 to L2
ENTER_RETRAIN_S0                 : move_to_state(CDN_UCIE_XDI_STS_RETRAIN, 0);         // Move stack 0 to RETRAIN
ENTER_LINKRESET_S0               : move_to_state(CDN_UCIE_XDI_STS_LINKRESET, 0);       // Move stack 0 to LINKRESET
ENTER_DISABLED_S0                : move_to_state(CDN_UCIE_XDI_STS_DISABLED, 0);        // Move stack 0 to DISABLED
ENTER_LINKERROR_S0               : move_to_state(CDN_UCIE_XDI_STS_LINKERROR, 0);       // Move stack 0 to LINKERROR
SEND_MAILBOX_REQUEST_D2D_REG     : generate_mailbox_request(CDN_UCIE_ADPTR_REG, CDN_UCIE_REG_D2D, "ucie_d2d_error_link_test_cntl", 0, reg_rd_data);
SEND_MAILBOX_REQUEST_DVSEC_REG   : generate_mailbox_request(CDN_UCIE_ADPTR_REG, CDN_UCIE_REG_DVSEC, "ucie_dvsec_link_status", 0, reg_rd_data);
SEND_FDI_SB_REG_ACCESS           : sideband_reg_access_handler.generate_fdi_sideband_request(ucie_dvsec_link_status, 0, reg_rd_data);
SEND_10_MB_PACKETS               : begin
                                     cdn_ucie_strm_mb_traffic_seq traffic_seq;
                                     traffic_seq = cdn_ucie_strm_mb_traffic_seq::type_id::create("traffic_seq");
                                     `uvm_do_on(traffic_seq,p_sequencer)
                                     traffic_seq.send_ob_traffic(10);
                                   end
```

Figure 9: Operation list

**Operation Execution Using Action Routing Sequence:** To support a unified testbench that can handle directed, random, and pseudo-random tests, we use a centralized control sequence called cdn_ucie_strm_base_action_routing_seq(Figure 10). This sequence is responsible for reading the operations from a scenario queue and executing them one by one or in parallel.

1. Reading and Executing the Scenario Queue:

- The body() task is the main entry point of this sequence.
- It reads the scenario_queue from the sequencer.
- This queue contains a list of operations that need to be performed.
- The queue can be filled by any type of test—directed, random, or pseudo-random—depending on the verification phase.
- The sequence keeps processing operations from the queue until it is empty, using the launch_operations() task.

2. Launching Operations:

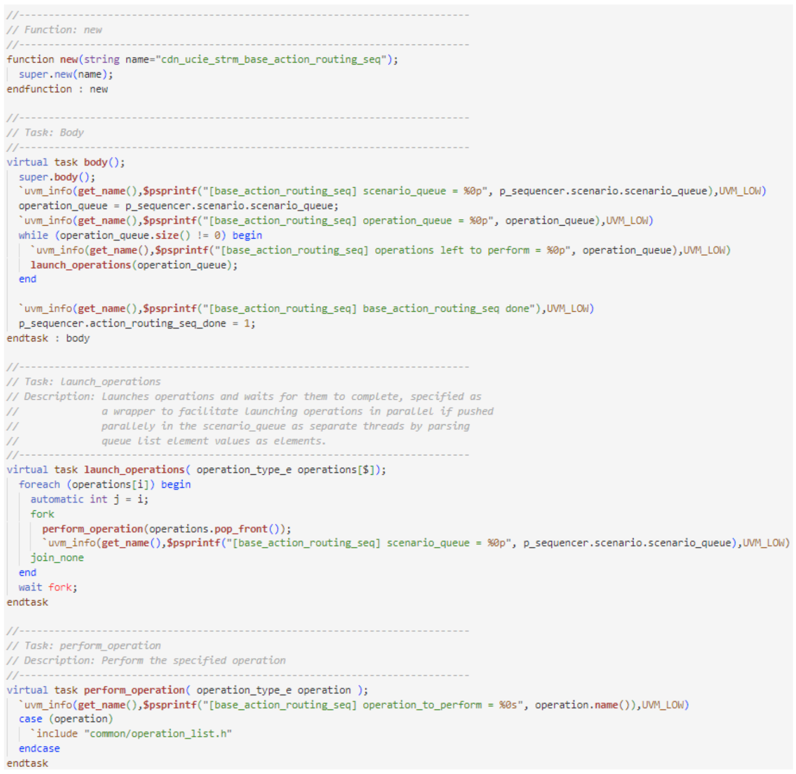
- The launch_operations() task goes through each operation in the queue.
- It calls the perform_operation() task to execute each one.
- All operations are started inside a fork...join_none block so that they can run in parallel if needed.
- After launching all threads, a wait fork is used to wait until all operations are completed. This helps in running multiple tasks at the same time when required and ensures proper test behavior.

3. Performing the Operation:

- The perform_operation() task takes a single operation and performs the required action.
- The actual list of supported operations is defined in a separate include file (operation_list.h) using a case statement.
- This makes the code easy to maintain—new operations can be added without modifying the main sequence.

4. Completion Signal:

- After completing all operations in the queue, the sequence sets a flag action_routing_seq_done = 1, this will be set when all the background threads explained above is completed.



```
//-----------------------------------------------------------------------
// Function: new
//-----------------------------------------------------------------------
function new(string name="cdn_ucie_strm_base_action_routing_seq");
  super.new(name);
endfunction : new

//-----------------------------------------------------------------------
// Task: Body
//-----------------------------------------------------------------------
virtual task body();
  super.body();
  `uvm_info(get_name(),$psprintf("[base_action_routing_seq] scenario_queue = %0p", p_sequencer.scenario.scenario_queue),UVM_LOW)
  operation_queue = p_sequencer.scenario.scenario_queue;
  `uvm_info(get_name(),$psprintf("[base_action_routing_seq] operation_queue = %0p", operation_queue),UVM_LOW)
  while (operation_queue.size() != 0) begin
    `uvm_info(get_name(),$psprintf("[base_action_routing_seq] operations left to perform = %0p", operation_queue),UVM_LOW)
    launch_operations(operation_queue);
  end

  `uvm_info(get_name(),$psprintf("[base_action_routing_seq] base_action_routing_seq done"),UVM_LOW)
  p_sequencer.action_routing_seq_done = 1;
endtask : body

//-----------------------------------------------------------------------
// Task: launch_operations
// Description: Launches operations and waits for them to complete, specified as
//              a wrapper to facilitate launching operations in parallel if pushed
//              parallely in the scenario_queue as separate threads by parsing
//              queue list element values as elements.
//-----------------------------------------------------------------------
virtual task launch_operations( operation_type_e operations[$]);
  foreach (operations[i]) begin
    automatic int j = i;
    fork
      perform_operation(operations.pop_front());
      `uvm_info(get_name(),$psprintf("[base_action_routing_seq] scenario_queue = %0p", p_sequencer.scenario.scenario_queue),UVM_LOW)
    join_none
  end
  wait fork;
endtask

//-----------------------------------------------------------------------
// Task: perform_operation
// Description: Perform the specified operation
//-----------------------------------------------------------------------
virtual task perform_operation( operation_type_e operation );
  `uvm_info(get_name(),$psprintf("[base_action_routing_seq] operation_to_perform = %0s", operation.name()),UVM_LOW)
  case (operation)
    `include "common/operation_list.h"
  endcase
endtask
```

*Figure 10: To take the operation from the scenario queue*

## IV. CONCLUSION

The proposed unified UVM testbench methodology consolidates directed, random, and pseudo-random testing into a single, adaptable framework. This integration not only reduces the need for multiple testbenches, thereby conserving engineering resources, but also simplifies maintenance and enhances scalability. By dynamically adjusting to various verification needs, the methodology facilitates the creation of directed test cases while also addressing challenging corner cases that are often difficult to cover. This approach significantly accelerates coverage closure, leading to a more efficient and thorough verification process. Ultimately, it streamlines the verification workflow, improves resource utilization, and enhances the overall quality of the design. This reduces debug effort when compared to a fully random testbench which runs interdependent layered sequences which leads to a lot of unwanted debug effort. Addition of this random and pseudo random is straight forward which re-use existing APIs which will save execution and coding time.