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

Amitav Mitra, Kilaru Vamsikrishna, Salehabibi Shaikh, Sushrut B Veerapur

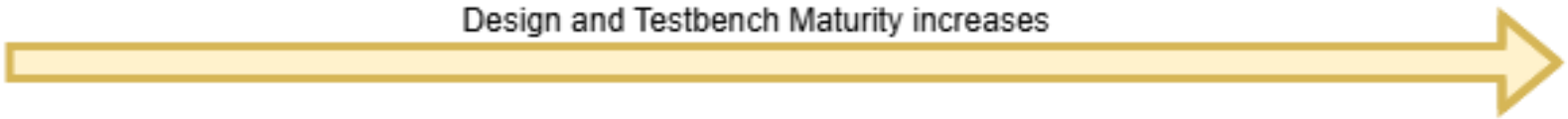
cadence

accellera
SYSTEMS INITIATIVE

Agenda

- Problem Statement
- Solution
- Implementation
- Results

Problem Statement



Verification Phase 1	Verification Phase 2	Verification Phase 3
Simple Directed tests	Fully flushed out complex randomized tests	Directed tests to cover corner case scenarios
• Initial Design Bring-up scenarios. • Test targeting specific features • Step-by-step thread execution in a pre-defined order with minimum randomization	• Constrained random verification • Protocol compliance • Design Stress testing • Modeling real world scenarios • Thread execution is random, with each thread running its operation independently and randomly in the background	• Edge conditions creation (FIFO full and stress testing) • Corner case Error Verification • FC and CC closure • Thread randomization and execution order is controlled with user specified constraints

Problem Statement (Cont)

- Testbench should scale and fine tune the it’s components (sequences, etc.) to operate in all these phases is challenging
- Sometimes, these requirements end up creating multiple different testbenches that leads to:
  - Maintenance
  - Cost
  - resources
- Reusing these verification components across all these phases requires unified testbench architecture with the capabilities of random, pseudo-random and directed testcases into single testbench.

Solution

Unified UVM Testbench Architecture for Verification

- **Centralized Control of Randomization**: Enables precise modeling of specific scenarios by managing randomization across the entire testbench.
- **Layered Architecture**: Introduces modular layers that progressively unfold during different phases of the verification cycle, enhancing clarity and scalability.
- **Maximum Reusability**: Promotes reuse of components, sequences, and configurations across multiple projects and verification stages.
- **Hybrid Testcase Integration**: Seamlessly incorporates deterministic testcases within the UVM framework, while maintaining extensibility for constrained random verification.

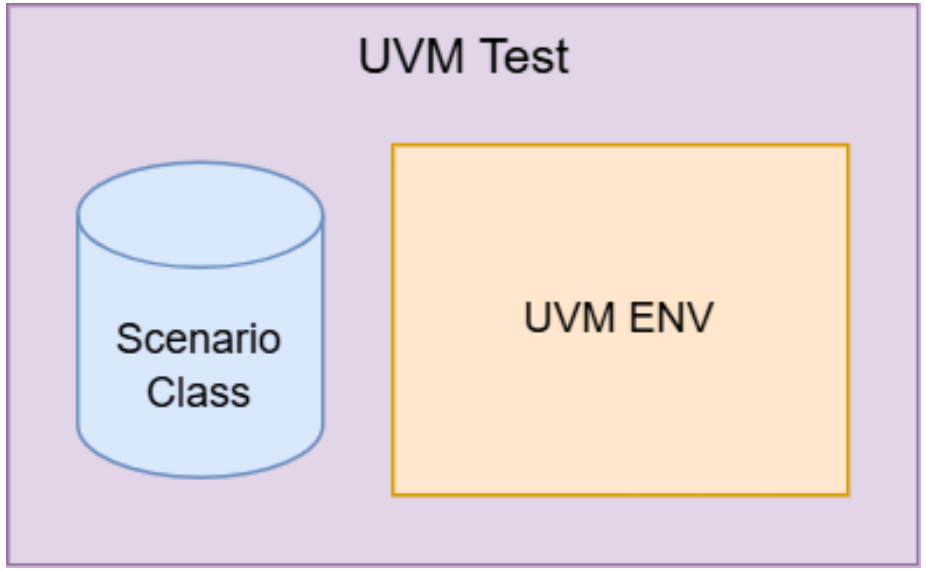
How do we implement this ?

accellera
SYSTEMS INITIATIVE

Implementation – Test Considerations

- To scale base test for all 3 flavors requires a control object called “scenario”
  - Test Scenario
    - Global common configuration class
    - Controls the execution of the threads
    - Acts as an interface between TB and the testcase writer allowing the control on randomization and the execution
  - Each test mode configures the scenario, and the unified TB performs the actions based on the selected scenario
  - Has all random and non-random variables that control the TB and Specification
- Base Test with scenario is the Key to the Unified TB Architecture

Implementation - Test Scenario Example




```
1  //------------------------------------------------------------------------------
2  // Class: cdn_ucie_strm_scenario
3  //------------------------------------------------------------------------------
4  class cdn_ucie_strm_scenario extends uvm_object;
5
6    // Queue: scenario_queue
7    // Hold all actions to be done sequentially before executing the randomized portion of the TB.
8    rand operation_type_e scenario_queue [$];
9
10   string test_mode = "FULL_RANDOM";
11
12   // Variable: target_flit_format
13   rand cdn_ucie_xdi_protocol_flitfmt_e target_flit_format;
14
15   // Variable: linkTransition_entry_condition
16   rand linkTransition_entry_condition_e linkTransition_entry_condition;
17
18   // Variable: min_state_transition_delay
19   // Minimum delay after which the next state transition will be requested
20   // Actual state transition delay will be randomized between min and max values
21   rand int min_state_transition_delay;
22
23   // Variable: max_state_transition_delay
24   // Maximum delay after which the next state transition will be requested
25   // Actual state transition delay will be randomized between min and max values
26   rand int max_state_transition_delay;
27
```

Implementation - Test mode configuration

The test can be configured in any of the 3 modes from the scenario class:

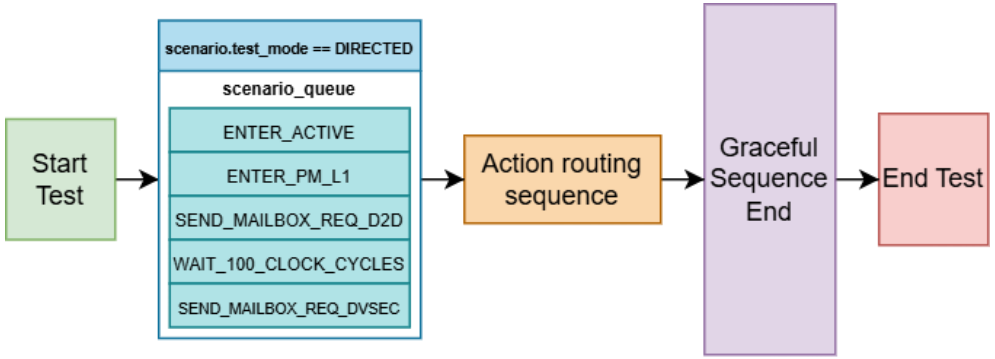
- Directed:
  Simple user-defined testcases
- Random:
  Fully randomized constrained flow with parallel running sequences according to user specification
- Pseudo-random:
  Queue operations populated randomly however user controls number of operations and enabled features

```
virtual function void randomize_scenario();

  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.test_mode = "DIRECTED";
```

```
virtual function void randomize_scenario();

    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_6;
            top_scenario.scenario_0.is_rp == is_rp;
          })) begin
      `uvm_fatal(get_type_name(), "Could not randomize scenario_0")
    end

    top_scenario.scenario_0.test_mode = "FULL_RANDOM";
  endfunction
```

```
virtual function void randomize_scenario();

    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_6;
            top_scenario.scenario_0.is_rp == is_rp;
          })) begin
      `uvm_fatal(get_type_name(), "Could not randomize scenario_0")
    end

    top_scenario.scenario_0.test_mode = "PSEUDO_RANDOM";
  endfunction
```

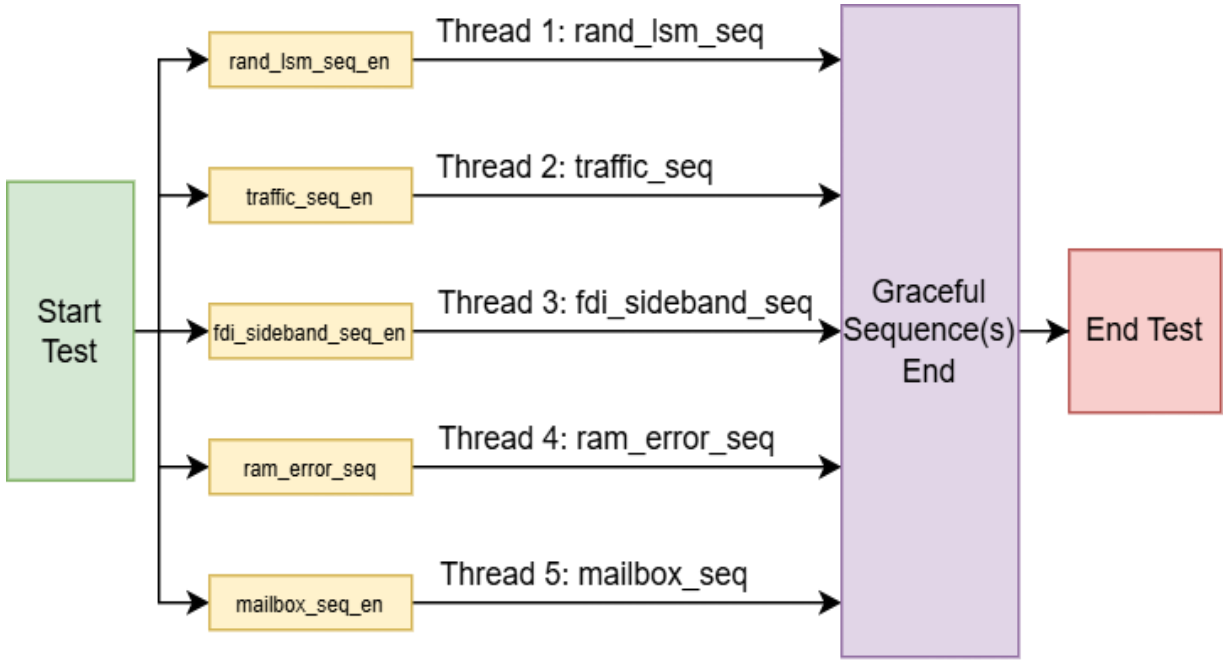
Implementation - Directed Testcases



- **User-defined operations**: The user specifies which operations to push into the queue.
- **Flexible test creation**: Enables creation of diverse and custom test scenarios using available operations.
- **Non-random execution**: The scenario queue is made non-random. All applied constraints on the scenario queue are turned off to ensure fully directed behavior.

```
virtual function void randomize_scenario();

  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.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_6;
          top_scenario.scenario_0.is_rp == 1;
        })) begin
    `uvm_fatal(get_type_name(), "Could not randomize scenario_0")
  end

  // Directed testcase
  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
```

Implementation - Random Testcases




```
fork
  if (p_sequencer.scenario.test_mode != "FULL_RANDOM") begin
    action_routing_seq = cdn_ucie_strm_base_action_routing_seq::type_id::create("action_routing_seq");
    action_routing_seq.start(p_sequencer);  // Action routing sequence
    wait(p_sequencer.action_routing_seq_done);
  end

  if (p_sequencer.scenario.continue_random_test) begin
    // Device configuration sequence
    `uvm_do_on( strm_config_seq, p_sequencer )
    fork
      begin
        rand_lsm_seq = cdn_ucie_strm_random_lsm_seq::type_id::create("rand_lsm_seq");
        rand_lsm_seq.start(p_sequencer);  // Random Link State Machine handling sequence thread
      end

      begin
        start_dev_operation_vseq();  // traffic sequences
      end

      begin
        start_fdi_sideband_seq();  // fdi sideband sequence thread
      end

      begin
        ram_err_seq();  // ASF: ECC RAM Error Sequence
      end

      begin
        if (p_sequencer.scenario.do_mailbox_access == 1) begin
          mailbox_seq.start(p_sequencer); // Sideband mailbox register access sequence
        end
      end
    join
  end
join
```

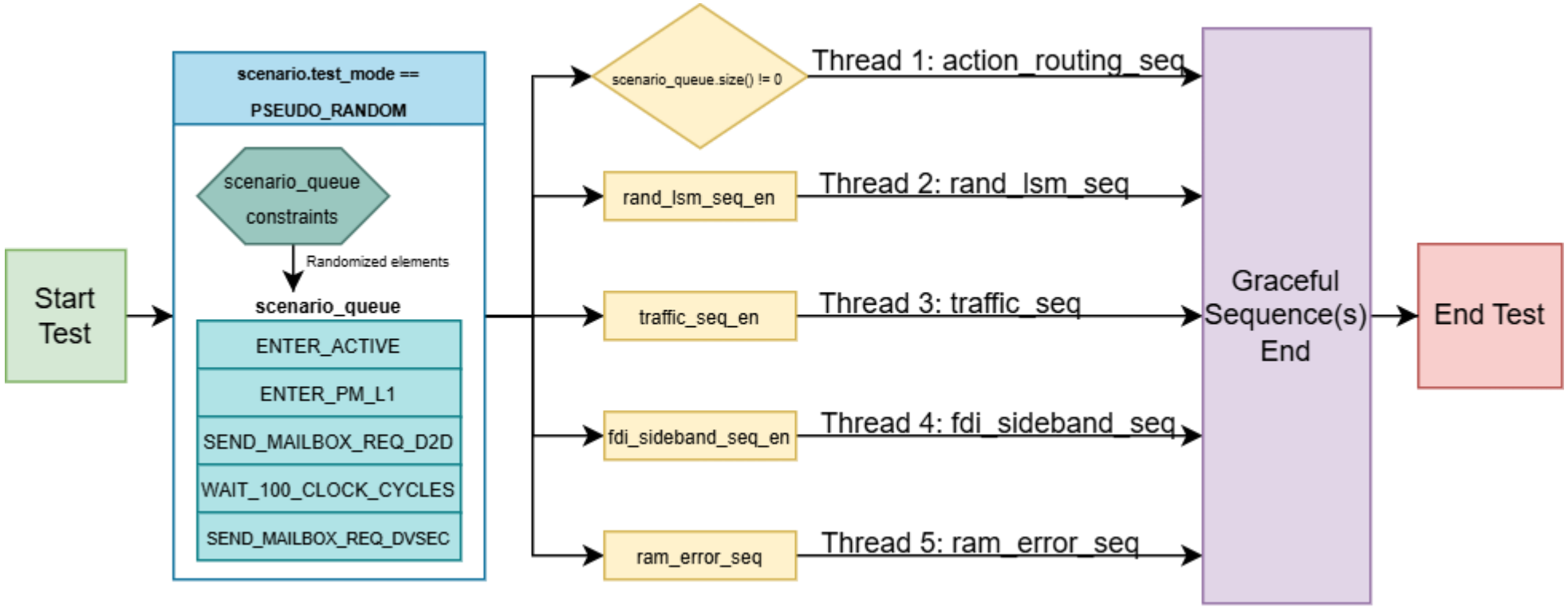
Random testcases do not rely on the operation queue or the action routing sequence. Instead, it launches multiple independent sequences in parallel and exercises the design with a random set of valid stimuli

Implementation - Pseudo-Random Testcases

- **Dynamic execution**: Pick and push operations randomly from the list of available legal operations.
- **Protocol-driven control**: Selection is guided by protocol-specific constraints based on element positions in the operation queue.
- **Protocol-specific logic**: Constraints are tailored for the UCIe protocol to guide operation sequencing based on valid link and power states, yet designed modularly for easy customization

```
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});
      }
    }
  }
```

Implementation - Pseudo-Random Testcases

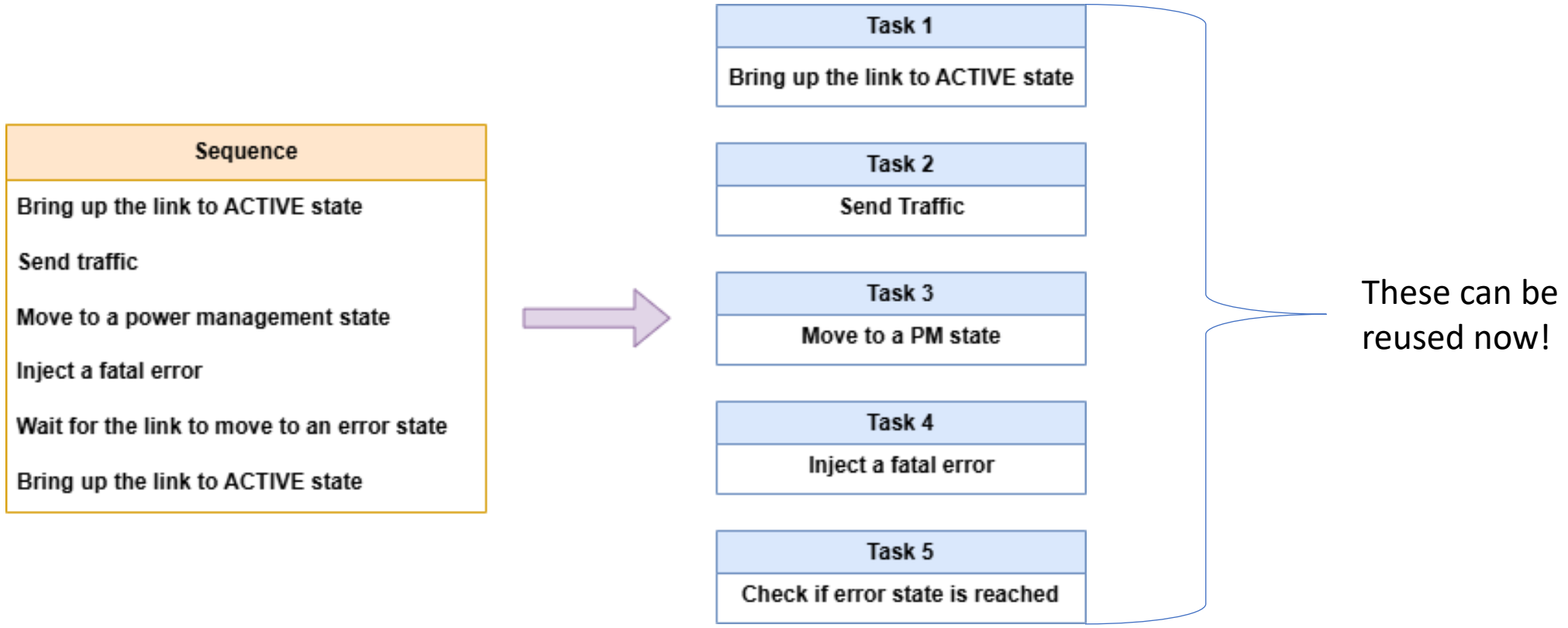


Implementation – Sequence Considerations

- Granularizing or breaking down the main sequences into reusable unit tasks/functions(APIs)
  - Each task/APIs can start sequences
  - Layering the sequence body into APIs allows better control (Polymorphism) and reusability (Inheritance)
  - Granularizing helps to achieve fine control thread execution(Events, Timeouts..etc.)
  - APIs are synchronized based on the functional requirements
- Example Granularization
  - state transitions, traffic, error injection..etc.

Implementation – APIs

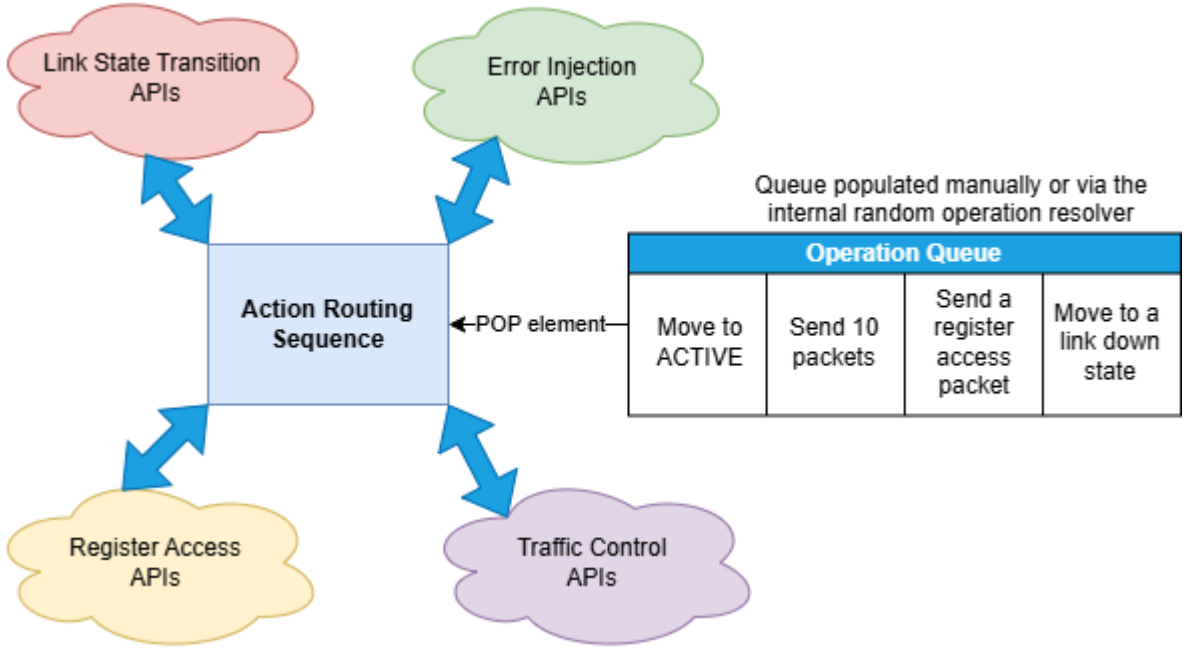
Example sequence broken down to its constituent unit operations



Implementation – APIs Execution and Control

The action routing sequence performs the following steps to selectively call APIs from various parts of the TB:

1. Accept populated queue from the scenario class.
2. Pop queue element.
3. Route popped element to its relevant API call from the **operation list**.
4. Wait for the operation performed by the API to finish.
5. Perform steps 2 to 4 till queue elements are exhausted



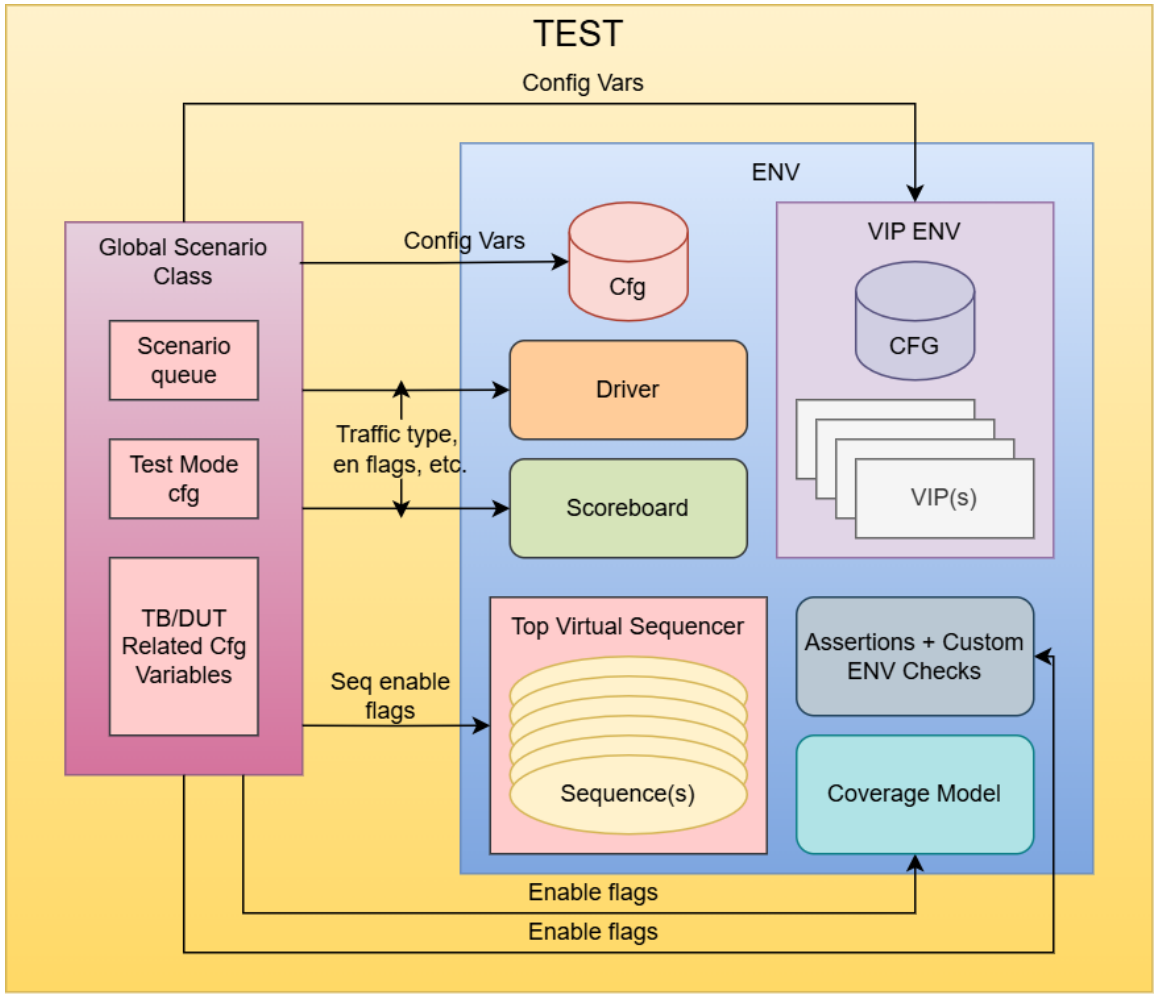
Implementation – APIs Execution and Control

```
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)
    perform_operation(operation_queue.pop_front());
  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

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

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
```

Segregated tasks/functions kept in their specific API classes are labelled and stored in a header file enclosed in a case statement.

```
ENTER_RESET_S0                : move_to_state(CDN_UCIE_XDI_STS_RESET, 0);          // Move stack 0 to RESET
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);
WAIT_100_CLOCK_CYCLES         : p_sequencer.misc_if.wait_cycle(100);
WAIT_1000_CLOCK_CYCLES        : p_sequencer.misc_if.wait_cycle(1000);
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
```

Implementation - Test Flow Architecture



Results

For UCIe Link State Machine (LSM) verification,

- Initial bringup was done through directed testcases
- Over 50 LSM cases + transition arcs were covered by constrained random verification
- Final corner case verification (about 5 arcs) was completed using the pseudo-random flow

All the while using the same set of reused sequences and codebase.

Resulting in 100 percent functional and code coverage closure.

Q&A

accellera
SYSTEMS INITIATIVE

Thank You

accellera
SYSTEMS INITIATIVE