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

A Generic Functional Safety Vector UVC

Siril Roy, Lead Design Engineer

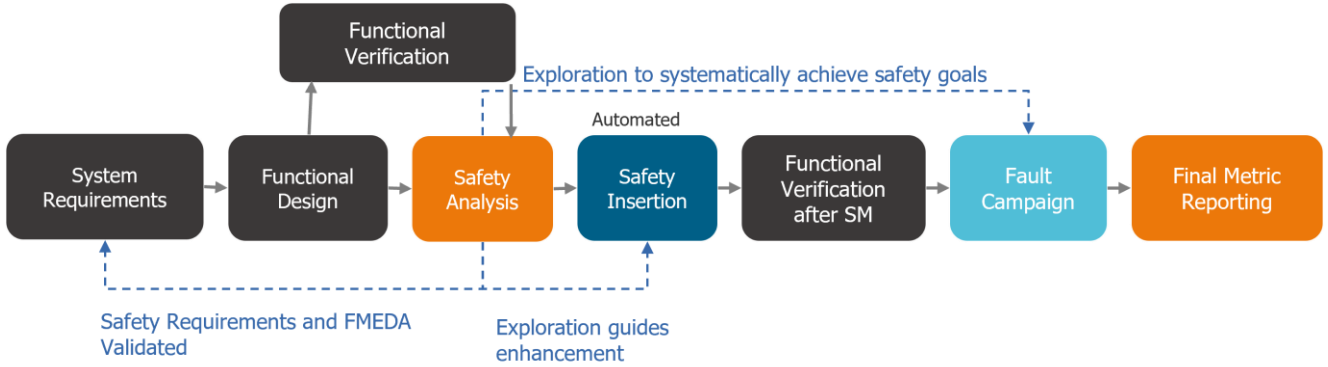
Kilaru Vamsikrishna, Design Engineering Architect

Raghav Sharma, Design Engineer 1

cadence®

accellera SYSTEMS INITIATIVE

FuSa RTL Design & Verification Flow



1. Requirements & Design

Define system-level safety requirements.

Develop RTL with safety points and functional design.

2. Safety Integration & Analysis

Insert safety mechanisms into RTL.

Perform safety analysis to identify fault scenarios.

Plan and execute fault campaigns.

3. Verification & Metrics

Conduct functional verification before and after safety mechanism insertion.

Report final safety metrics and validate ASIL compliance.

Validate FMEDA and safety requirements.

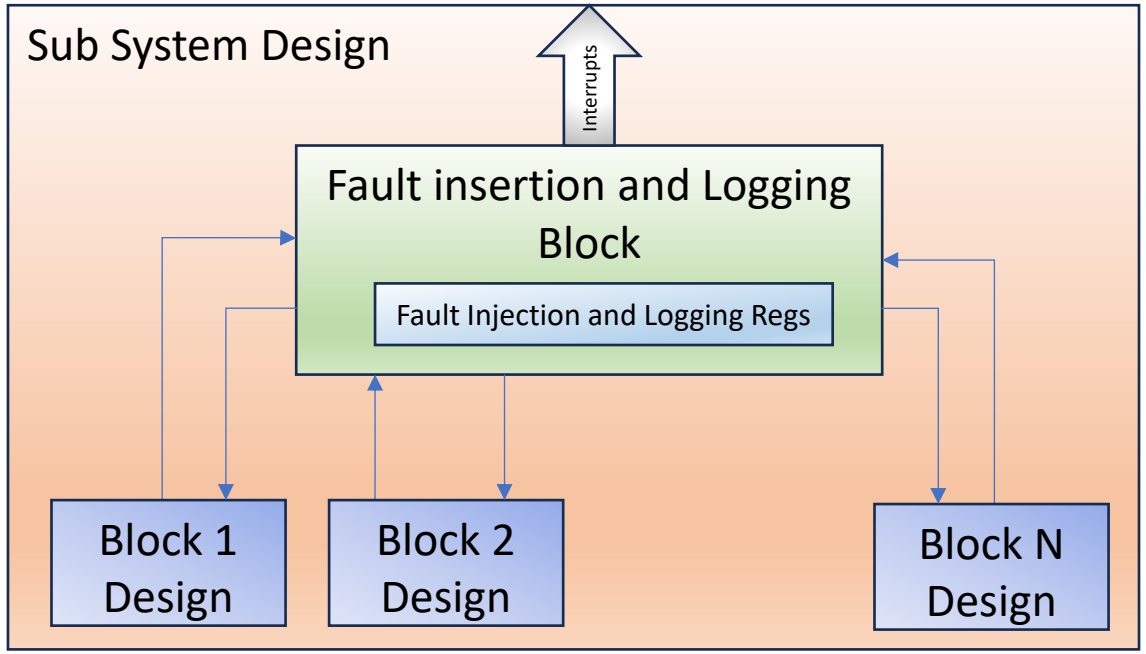
4. Exploration & Automation

Use exploration to guide architectural enhancements.

Systematically achieve safety goals through iterative refinement.

Automate flow for efficiency and repeatability.

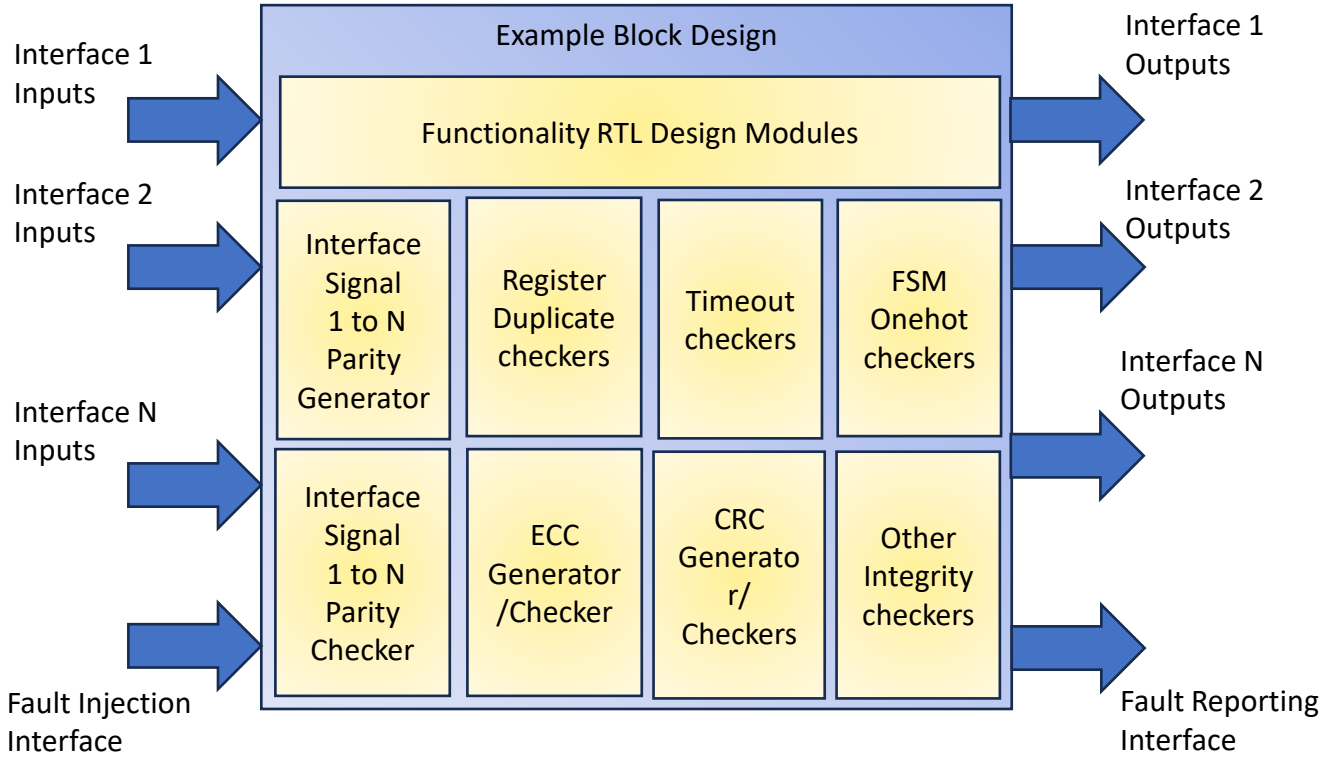
Subsystem Design with Fault Insertion & Logging



- RTL subsystem integrates multiple functional modules.
- Aggregation logic consolidates fault signals and safety responses.
- Dedicated fault insertion and error logging modules are included.
- Each module is designed with safety instrumentation.
- Supports scalable fault injection and traceability.

Block DUT With Fault Checkers and Generators

- **Interface & IO Design**
  - The RTL module includes multiple interfaces (Interface 1 to N) with defined inputs and outputs.
  - Dedicated interfaces for injecting faults and capturing error responses.
- **Example Safety Mechanisms**
  - Safety mechanisms are implemented to detect and mitigate faults in data paths and control logic, ensuring robust functional safety compliance within the module.



RTL snippet showing parity check logic with fault injection support

- Compares input parity with calculated parity.
- Flags parity errors via a dedicated output.
- This module can be reused as it required only width parameter.

```
module parity_checker_example #(parameter DATAPATH_WD = 1024)
  ( input                          odd_par,
    input [DATAPATH_WD-1:0]        data_in,
    input [(DATAPATH_WD/8)-1:0]    parity_in,
    input [DATAPATH_WD-1:0]        data_in_fault_inj,
    input [(DATAPATH_WD/8)-1:0]    parity_in_fault_inj,
    output                         parity_err)

  wire [(DATAPATH_WD/8)-1:0] calc_parity;

  // Parity Generator for comparision
  parity_generator(.odd_par(odd_par),
                   .data(data_in^data_in_fault_inj),
                   .parity_out(calc_parity));

  // Error output
  parity_err = |(calc_parity^(parity_in^parity_in_fault_inj));

endmodule
```

Figure 1: Parity Checker Verilog RTL Module

- Supports fault injection into data or parity logic for safety validation.
- Validates safety instrumentation and detects transient.

Example Block DUT Design with Safety Instrumentation

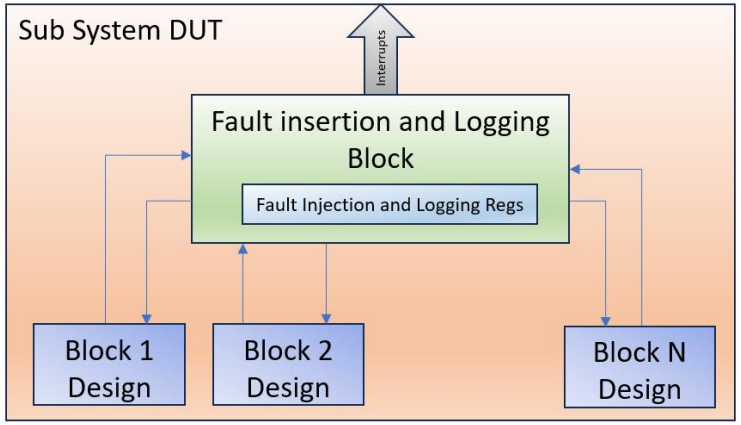
- This block design integrates multiple safety mechanisms and supports fault injection and error logging for subsystem-level validation.
- Integrates multiple safety mechanisms
- Supports fault injection and detection.
- Interfaces support traceability and fault response capture

```
module sample_rtl
#( parameter DATAPATH_WD = 1024
  ,parameter ADDR_WD = 128 )
(  input                          clk
  ,input                          rst
  ,input  [ADDR_WD-1:0]           addr_ingress
  ,input  [ADDR_WD-1:0]           addr_ingress_fault_inj
  ,input  [(ADDR_WD/8)-1:0]       addr_ingress_par
  ,input  [(ADDR_WD/8)-1:0]       addr_ingress_par_fault_inj
  ,output                         addr_ingress_error_out
  ,output [ADDR_WD-1:0]           addr_egress
  ,input  [ADDR_WD-1:0]           addr_egress_fault_inj
  ,output [(ADDR_WD/8)-1:0]       addr_egress_par
  ,input  [(ADDR_WD/8)-1:0]       addr_egress_par_fault_inj
  ,output                         addr_egress_error_out
  ,input  [DATAPATH_WD-1:0]       data_ingress
  ,input  [DATAPATH_WD-1:0]       data_ingress_fault_inj
  ,input  [(DATAPATH_WD/8)-1:0]   data_ingress_par
  ,input  [(DATAPATH_WD/8)-1:0]   data_ingress_par_fault_inj
  ,output                         data_ingress_error_out
  ,output [DATAPATH_WD-1:0]       data_egress
  ,input  [DATAPATH_WD-1:0]       data_egress_fault_inj
  ,output [(DATAPATH_WD/8)-1:0]   data_egress_par
  ,input  [(DATAPATH_WD/8)-1:0]   data_egress_par_fault_inj
  ,output                         data_egress_error_out

  //-- Other I/Os for the functionality
)

  // RTL Functional logic

  // Fault instance : Parity Chekers instance for ingress address
  parity_checker_example #(.DATAPATH_WD(DATAPATH_WD)) checker_addr_ingress
  (  .odd_par             (1'b1)
    ,.data_in             (addr_ingress)
    ,.parity_in           (addr_ingress_par)
    ,.data_in_fault_inj   (addr_ingress_fault_inj)
    ,.parity_in_fault_inj (parity_in_fault_inj)
    ,.parity_err          (addr_ingress_error_out));

 // Similar connections for other signals
 parity_checker_example #(.DATAPATH_WD(DATAPATH_WD)) checker_addr_egress(//*port connections*//);
 parity_checker_example #(.DATAPATH_WD(DATAPATH_WD)) checker_data_ingress(//*port connections*//);
 parity_checker_example #(.DATAPATH_WD(DATAPATH_WD)) checker_data_engress(//*port connections*//);
endmodule
```

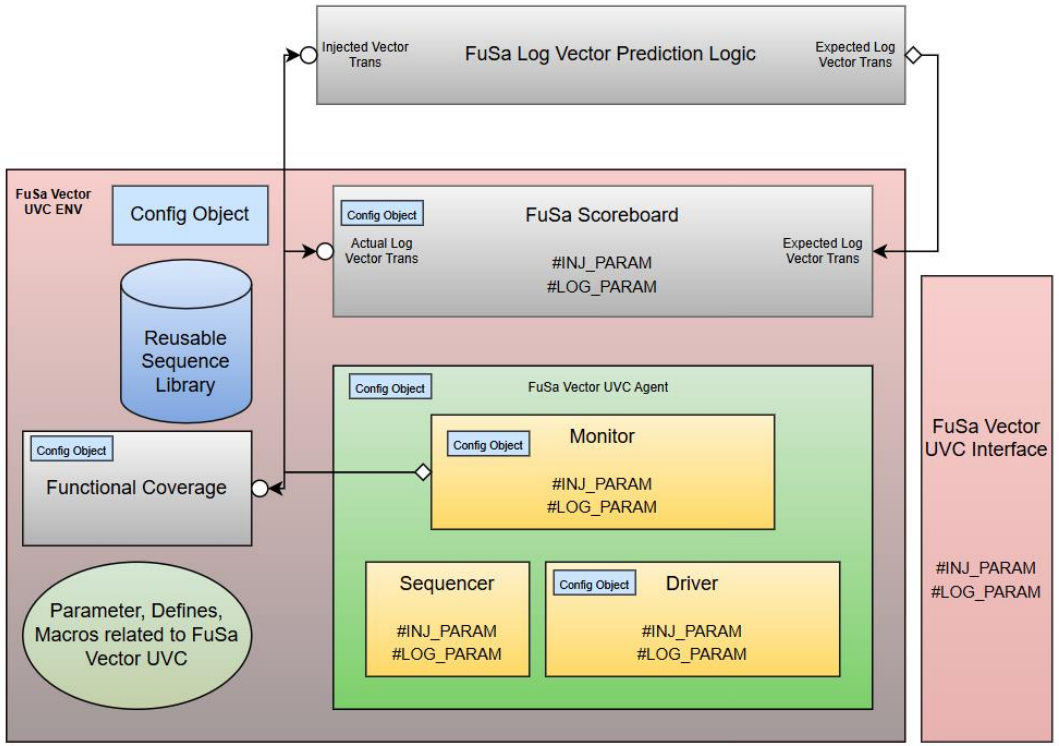
Figure 4: DUT with some fault instances

Problem statement

- Validating Safety Mechanisms
  - Each module must have safety logic that operates correctly and independently.
  - Reusing safety RTL across modules makes isolation and correctness harder to verify.
- Scalable Testbench Design
  - Creating a testbench that supports multiple DUTs with shared safety RTL is complex.
  - It demands significant resources and careful architecture.
- Testcase Overhead
  - Every safety mechanism needs a dedicated testcase.
  - This leads to increased effort, longer regression cycles, and slower verification closure.
- Fault Injection with Functional Traffic
  - Injecting faults while running functional traffic adds layers of complexity.
  - Debugging becomes more time-consuming and error-prone.
- Cross-Module Fault Propagation
  - Faults in one module can affect shared safety RTL and propagate to others.
  - Verifying these interactions is non-trivial and requires robust coverage.



FuSa Vector UVC Env



- Includes standard UVM elements:
  - Agent, Driver, Monitor, and Sequencer, enabling modular and reusable verification.
- Built-in Scoreboard and Coverage Collectors
  - Ensures automated checking and coverage tracking for fault injection and safety responses.
- Reusable Transaction Item, Interfaces and Sequences.
  - Promotes consistency and reuse across different DUTs and testbenches.
- Configurable UVC
  - Designed to connect with parameterized interfaces and defined control logic, allowing flexibility across varying DUT configurations

FuSa Vector UVC – Interface and Trans item

- Parametrized interface and transaction item for flexibility.
- Designed for modularity and scalability across different DUTs.
- Promotes consistent safety verification methodology.

```
interface fusa_vector_uvc_if ( input bit clk, input bit rst_n);

  logic [FUSA_VECTOR_UVC_FAULT_INJ_VEC_WD-1 : 0]     fault_inj_vector;
  logic [FUSA_VECTOR_UVC_FAULT_LOG_VEC_WD-1 : 0]     fault_log_vector;

  // Monitor clocking block
  clocking monitor_cb @(posedge clk);
    input fault_inj_vector;
    input fault_log_vector;
  endclocking

  // Clocking block for driver
  clocking driver_cb @(posedge clk);
    output fault_inj_vector;
  endclocking // producer_cb

endinterface
```

Figure 3: FuSa Vector UVC Interface with fault signals

```
// Parameters are user defined
class fusa_vector_uvc_seq_item #(int INJ_VECTOR_SIZE = 100, int LOG_VECTOR_SIZE = 100 ) extends
uvm_sequence_item;

  // Variable : Vector for Fault Injection,
  rand logic [INJ_VECTOR_SIZE-1:0] fault_inj_array;

  // Variable : Vector for Fault Reporting
  rand logic [LOG_VECTOR_SIZE-1:0] fault_log_array;

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

  // Tasks and functions to be performed over transcation

endclass
```

Figure 5: FuSa Vector UVC Transaction item

FuSa Vector UVC - Monitor

- The monitor captures interface signals, including injection and reporting vectors.
- Monitor creates and writes transaction items to the analysis port whenever there is a bit-level change in any of the interface signals.

```
forever begin
  // Create a new monitor object
  monitor_trans = fusa_vector_uvc_seq_item::type_id::create("monitor_trans", this);

  // Indicate the start of a monitor transaction
  void'(begin_tr(monitor_trans));

  monitor_trans.fault_inj_array = vif.monitor_cb.fault_inj_vector;
  monitor_trans.fault_log_array = vif.monitor_cb.fault_log_vector;

  // Indicate the end of a monitor transaction & trigger callback
  void'(end_tr(monitor_trans));
  monitor_ap.write(monitor_trans);

  // Wait until a transaction has started.
  do begin
    @(vif.monitor_cb);
  end while (( monitor_trans.fault_inj_array === vif.monitor_cb.fault_inj_vector) &&
          ( monitor_trans.fault_log_array === vif.monitor_cb.fault_log_vector));
end
```

Figure 6: FuSa Vector UVC Monitor Sampling logic

FuSa Vector UVC – Structs and Params

- Define structs and parameters to support fault injection and reporting in the design.
- Use separate vectors for injection and reporting to improve scalability of scoreboarding, coverage, and analysis.
- Apply these parameters consistently across all UVC components to ensure modularity and reuse.
- These definitions can also be reused in the top-level testbench for passive monitoring.

```
//-- Fault Injection packed Struct
typedef struct packed {
  logic [ADDR_WD-1:0]         addr_ingress_fault_inj      ;
  logic [(ADDR_WD/8)-1:0]     addr_ingress_par_fault_inj  ;
  logic [ADDR_WD-1:0]         addr_egress_fault_inj       ;
  logic [(ADDR_WD/8)-1:0]     addr_egress_par_fault_inj   ;
  logic [DATAPATH_WD-1:0]     data_ingress_fault_inj      ;
  logic [(DATAPATH_WD/8)-1:0] data_ingress_par_fault_inj  ;
  logic [DATAPATH_WD-1:0]     data_egress_fault_inj       ;
  logic [(DATAPATH_WD/8)-1:0] data_egress_par_fault_inj   ;
} fault_inj_t;

//-- Fault Log packed Struct
typedef struct packed {
  logic                       addr_ingress_error_out      ;
  logic                       addr_egress_error_out       ;
  logic                       data_ingress_error_out      ;
  logic                       data_egress_error_out       ;
} fault_log_t;


//-- 2 addr signals, 2 addr_par signals, 2 data signals, 2 data_par signals
parameter FUSA_VECTOR_UVC_FAULT_INJ_VEC_WD = (ADDR_WD + (ADDR_WD/8) + DATAPATH_WD +
                                              (DATAPATH_WD/8)) * 2;

//-- 4 report signals
parameter FUSA_VECTOR_UVC_FAULT_LOG_VEC_WD = 4;
```

Figure 8: FuSa Vector UVC related Structures and Parameters

FuSa Vector UVC – TB Arch

- Parallel functional and fault simulation support.
- Monitored fault transactions shared across the testbench.
- Fault prediction support based on functional verification.
- Reusability of this passive logic in Top level TB.

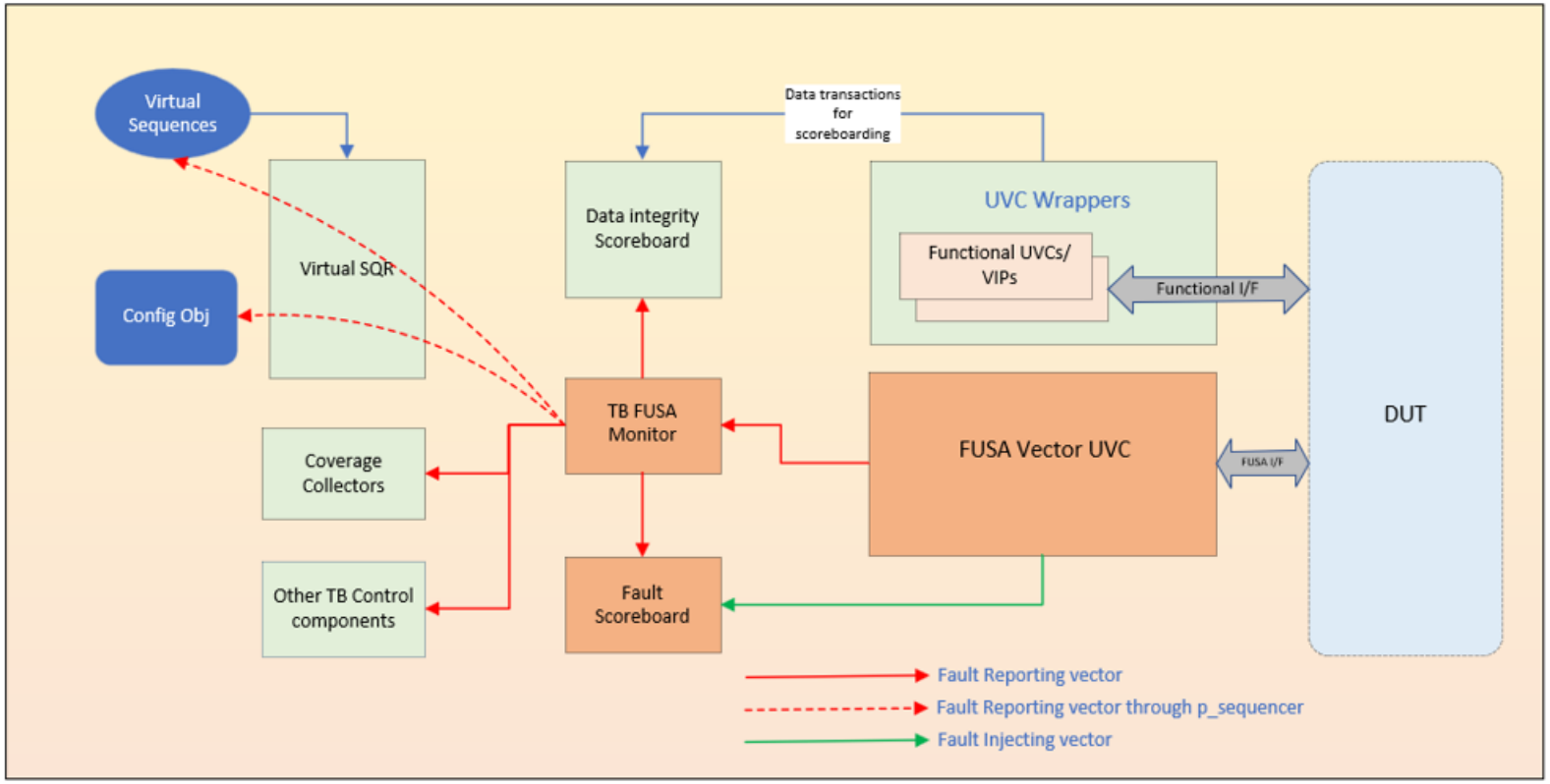


Figure 7. Testbench architecture diagram

FuSa Vector UVC – TB Top

- Block DUT instantiation and connection to the typedef structs.
- Reducing the complexity of width-based connection by using SV structs.
- Interface instantiation of FuSa Vector UVC interface and connecting it to the design through structs.

```
module fusa_top_tb_example()

  //--- FuSa UVC Interface handle ---
  fusa_vector_uvc_if uvc_if ( intf_wrapper.clk, intf_wrapper.rst );

  //--- Structure with Fault Injection Information ---
  fault_inj_t fault_inj;

  //--- Structure with Faults Logged Information ---
  fault_log_t fault_log;

  //--- Connections between FuSa Vector and UVC interface ---
  assign fault_inj              = uvc_if.fault_inj_vector;
  assign uvc_if.fault_log_vector = fault_log;

  //--- Controller Interface ---
  tb_intf_wrapper intf_wrapper();

  //--- RTL Instance ---
  sample_rtl #(.DATAPATH_WD(1024)) DUT
  (  .clk                          (intf_wrapper.clk)
    ,.rst                          (intf_wrapper.rst)
    ,.addr_ingress                 (intf_wrapper.addr_ingress)
    ,.addr_ingress_fault_inj       (fault_inj.addr_ingress_fault_inj)
    ,.addr_ingress_par             (intf_wrapper.addr_ingress_par)
    ,.addr_ingress_par_fault_inj   (fault_inj.addr_ingress_par_fault_inj)
    ,.addr_ingress_error_out       (fault_log.addr_ingress_error_out)
    ,.addr_egress                  (intf_wrapper.addr_egress)
    ,.addr_egress_fault_inj        (fault_inj.addr_egress_fault_inj)
    ,.addr_egress_par              (intf_wrapper.addr_egress_par)
    ,.addr_egress_par_fault_inj    (fault_inj.addr_egress_par_fault_inj)
    ,.addr_egress_error_out        (fault_log.addr_egress_error_out)
    ,.data_ingress                 (intf_wrapper.data_ingress)
    ,.data_ingress_fault_inj       (fault_inj.data_ingress_fault_inj)
    ,.data_ingress_par             (intf_wrapper.data_ingress_par)
    ,.data_ingress_par_fault_inj   (fault_inj.data_ingress_par_fault_inj)
    ,.data_ingress_error_out       (fault_log.data_ingress_error_out)
    ,.data_egress                  (intf_wrapper.data_egress)
    ,.data_egress_fault_inj        (fault_inj.data_egress_fault_inj)
    ,.data_egress_par              (intf_wrapper.data_egress_par)
    ,.data_egress_par_fault_inj    (fault_inj.data_egress_par_fault_inj)
    ,.data_egress_error_out        (fault_log.data_egress_error_out)
);
endmodule
```

Figure 9: Interconnections between DUT and FuSa Vector UVC inside TB top

FuSa Vector UVC – Fault Generation

- An example UVM sequence for fault generation.
- Constraints are created based on the test intention.
- Parallel fault injection to multiple safety points.
- Can be wait for the fault reporting based on the fault monitored by the UVC.

```
class fusa_vector_uvc_seq extends uvm_sequence;

  rand fault_inj_t        err_inj_array;

  // Testbench config object for creating test scenarios
  fusa_tb_scenario        tb_sceanrio;

  // Variable declarations for other sequence control

  `uvm_object_utils(fusa_vector_uvc_seq)
  `uvm_declare_p_sequencer(vsequencer)

  // Constraint to control Fault Injection Types and Scenarios
  constraint addr_egress_par_fault_inj_c {
    if(tb_sceanrio.addr_egress_corruption_en) {
      $countones({err_inj_array.addr_egress_par_fault_inj,
                  err_inj_array.addr_egress_fault_inj} > 0) ;
    }
  }

  // Constructor other methods

  // Task: body
  virtual task body();
    fusa_vector_uvc_seq_item #( INJ_VECTOR_SIZE, LOG_VECTOR_SIZE) fusa_uvc_seq;

   // Create Fusa_fault_uvc sequence item //TODO add parameters
   `uvm_create_on(fusa_uvc_seq(), p_sequencer.fusa_vector_uvc_sqr)

    // Pre-Error injection process ( Register Configurations( Mask, Severity, Control etc.) )
    pre_err_inj_config();

    // Sequence start
    `uvm_rand_send_with( fusa_uvc_seq, { fault_inj_array == local::err_inj_array; })

    // wait for the Interrupt log vector from Fusa vector UVC monitor
    wait_for_interrupt();

    // Post-Error detection process ( Interrupt checks and CSR Checks )
    post_err_inj_config();

  endtask // body

endclass // fusa_vector_uvc_seq
```

Figure 10: FuSa Vector UVC Sequence Staring and Handling interrupts

FuSa Vector UVC – Fault Scoreboarding

- Fault Scoreboarding after Internal and external fault injection.
- Default checks present in scoreboards to flag unexpected fault reporting.

```
virtual function void write_exp_fusa_uvc (fusa_vector_uvc_seq_item trans);
  fault_inj_t fault_inj_vector;
  fault_log_t fault_log_vector;

  if( !$cast(fault_inj_vector, trans.fault_inj_array))
    `uvm_error(get_name(), "Dynamic casting failed")
  // -- Predicting the log vector based on injection
  if ( $countones(trans.fault_inj_array) > 0 ) begin
    fault_log_vector = generic_fault_prediction(fault_inj_vector);
    exp_fault_log_array_q.push_back(fault_log_vector);
  end
  //-- Other function codes --
endfunction

//-- Predicting err reporting vector based on error inj vector
virtual function fault_log_t generic_fault_prediction( input fault_inj_t err_inj );
  fault_log_t err_log;

  err_log.addr_ingress_error_out   = (|{err_inj.addr_ingress_fault_inj,
                                        err_inj.addr_ingress_par_fault_inj}) ? 'h1 : 'h0;
  err_log.addr_egress_error_out    = (|{err_inj.addr_egress_fault_inj ,
                                        err_inj.addr_egress_par_fault_inj }) ? 'h1 : 'h0;
  err_log.data_ingress_error_out   = (|{err_inj.data_ingress_fault_inj,
                                        err_inj.data_ingress_par_fault_inj}) ? 'h1 : 'h0;
  err_log.data_egress_error_out    = (|{err_inj.data_egress_fault_inj ,
                                        err_inj.data_egress_par_fault_inj }) ? 'h1 : 'h0;

  return err_log;
endfunction
```

Figure 11: Scoreboarding approach for Internal fault injection

```
  virtual function void write_dut_ingress_trans (dut_trans_item trans);
    dut_trans_item pred_trans;
    fault_log_t    fault_log_vector;

    //-- created pred_trans. then updating the parity based on rcvd addr signal
    pred_trans.addr_ingress_par = calc_parity(trans.addr);

    //-- Example External error prediction
    fault_log_vector.addr_ingress_error_out = ( trans.addr_ingress_par !=
pred_trans.addr_ingress_par ) ?  1'b1 : 1'b0;

    //-- Other fault vector predictions

    //-- Sending the item to the expected Q after all predictions
    if ( $countones(fault_log_vector) > 0 ) begin
      exp_fault_log_array_q.push_back(fault_log_vector);
    end

    //-- Other function codes --

  endfunction
```

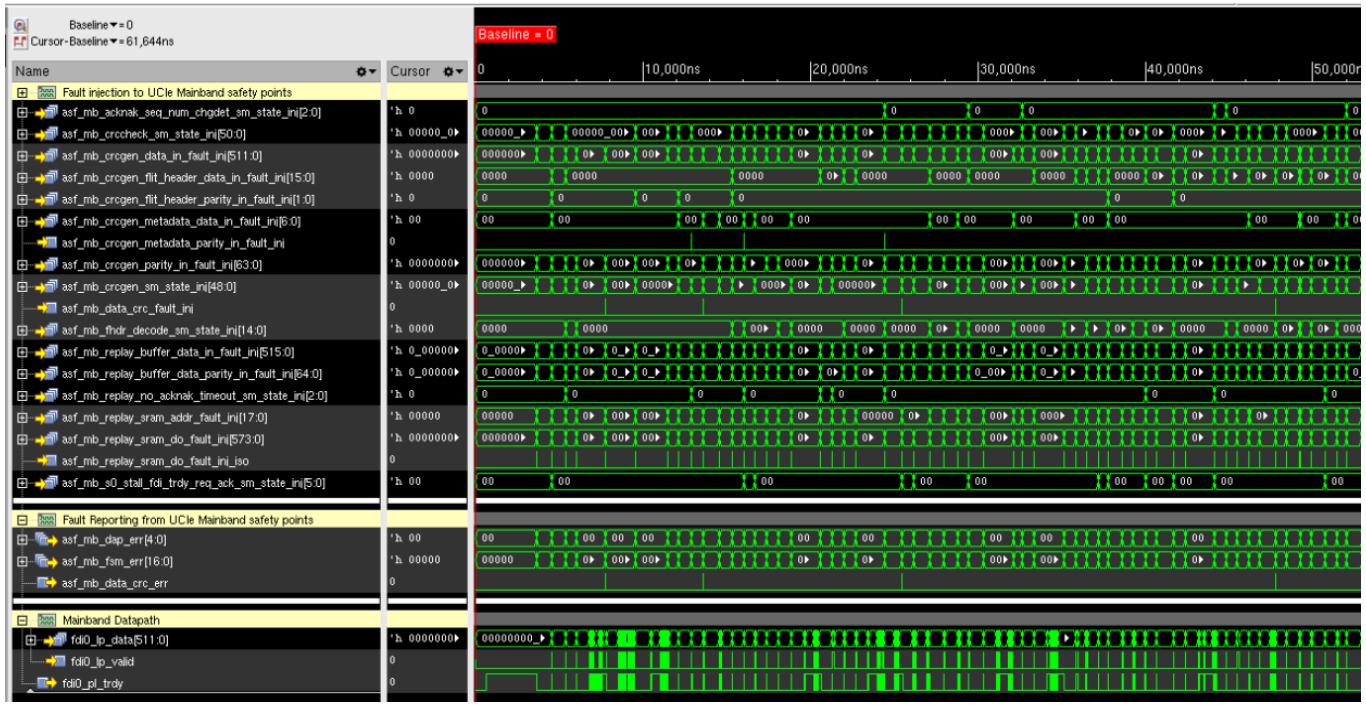
Figure 12: Scoreboarding approach for External fault injection

Results

- Reused across 3 Block level testbenches with different parameter configurations.
  - Two-man week timeframe for this UVC integration into UCIe Adapter Block TB.
  - Early bug identification and verification closure.
- Passive monitoring and checking enabled in Top TB by reusing this module TB Envs.
- Reduced TB complexity significantly with parallel traffic and fault injection support.

Results

- RTL module with IO’s and Safety points
- Stress fault injection to the safety instances to identify corner bugs.
- 100% FC and CC achieved in short time.



Results

- Example monthly bug rate. 90% bugs identified and closed in 3 months.
- Filtered FSM CC for a DUT safety instances.

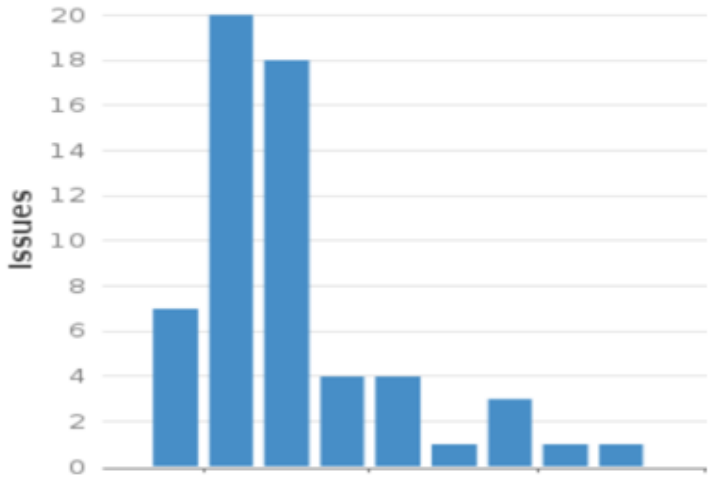


Figure 15: Graph representing UCIe Controller monthly ASF RTL bug rate

Name	Combined Cov
u_dut.gen_ucie_cxs.i_ucie_cxs.i_ucie_fdi_sm.one_hot_stall_state	23 / 23 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.i_s0_fdi_arbmux.ob_fdi_arbiter.fdi_protocol_select.mb_ob_ctrl_sm_one_hot_err_detect	43 / 43 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.i_s0_fdi_arbmux.ob_fdi_arbiter.fdi_protocol_select.raw_fdi_sm_one_hot_err_detect	31 / 31 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.ob_stack_arb_retry.stack_arb_mux_ctrl_stg1.stack_arb_sm_one_hot_err_detect	38 / 38 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.ob_stack_arb_retry.stack_arb_mux_ctrl_stg1.stack_arb_nop_idle_count_sm_one_hot_err_detect	33 / 33 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.ob_stack_arb_retry.replay_cmd_gen.acknak_seq_num_chgdet_sm_one_hot_err_detect	33 / 33 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.ob_stack_arb_retry.seq_num_sync_sm.sync_sm_state_one_hot_err_detect	33 / 33 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.ob_rdi_flit_packer.ob_flit_pack_crcgen_DP512B_256B_128B.i_ob_flit_pack_crcgen.crcgen_sm_one_hot_err_detect	86 / 86 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.ob_rdi_flit_packer.ob_flit_pack_pds_68b.ob_flit_pack_pds_68b.packer_ofvl_ctrl_sm_one_hot_err_detect	31 / 31 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.ob_rdi_flit_packer.ob_flit_pack_pds_68b.ob_flit_pack_pds_68b.irdy_inactive_det_sm_one_hot_err_detect	31 / 31 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.i_mb_misc.i_s0_stall_fdi_trdy_req_ack_sm.stall_fdi_trdy_req_ack_sm_one_hot_err_detect	39 / 39 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.i_mb_misc.rdi_stall_req_ack_sm_one_hot_err_detect	38 / 38 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.i_ib_rdi_flit_unpack.ib_flit_unpack_pds_68b.i_ib_flit_unpack_pds_68b.unpacker_hreg_ctrl_sm_one_hot_err_detect	35 / 35 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.i_ib_rdi_flit_unpack.ib_flit_unpack_crc_check_DP512B_DP256B_DP128B.i_ib_flit_unpack_crc_check.ib_sf_fifo_rd_sm_one_hot_err_detect	37 / 37 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.i_ib_rdi_flit_unpack.ib_flit_unpack_crc_check_DP512B_DP256B_DP128B.i_ib_flit_unpack_crc_check.crccheck_sm_one_hot_err_detect	90 / 90 (100%)
u_dut.i_ucie_adapter_top.i_ucie_adapter_core.i_ucie_adapter_mainband.i_ib_replay_decode_stack_demux.i_ib_fhdr_replay_decode.replay_no_acknak_timeout_sm_one_hot_err_detect	33 / 33 (100%)

Figure 14: Example Code Coverage Results for UCIe Adapter FSM fault instances

Questions