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

A Generic Functional Safety Vector UVC

Siril Roy, Cadence Design System, Bengaluru, India (*sirilr@cadence.com*)
Raghav Sharma, Cadence Design System, Noida, India (*raghav1@cadence.com*)
Kilaru Vamsikrishna, Cadence Design System, Bengaluru, India (*kilaruv@cadence.com*)
Sushrut B Veerapur, Cadence Design System, Bengaluru, India (*sushrut@cadence.com*)

***Abstract*—Functional Safety (FuSa) standards are mandatory in modern semiconductor chips since the industry evolves, and chips are largely used in automotive industry (Automotive FuSa standard – ISO26262). Failure analysis is required to be completed throughout the RTL design and necessary safety mechanisms needs to be added in the RTL design. Which means, FuSa begins with the design intellectual properties (IPs). FuSa related RTL must be added for legacy IPs based on the analysis and new IPs must be architected with FuSa considerations. This paper demonstrates the methodology to verify FuSa RTL design by using reusable UVC (UVM verification component) in constrained random UVM TB (Module or IP Level TB)**

Keywords—functional safety; UVM; UVC; SystemVerilog; verification; fault/error injection; interrupts; ASIL

I. INTRODUCTION

This paper proposes a convenient approach for design verification of semiconductor designs which are compliant with Functional Safety (FuSa) standards. To ensure FuSa compliance, RTL designs must be capable of identifying any faults or unintended behavior that occur within the design and ensure design integrity to maintain a safe state for the system. The detected interrupts are further forwarded to other design layers or other system components based on the failure integrity. The proposed UVC simplifies the process of stress testing a design's ability to inject and handle interrupts, also ensuring that fault injection is performed concurrently with functional design aspects while also verifying compliance with the FuSa standards.

A. Functional Safety (FuSa) Overview

Functional safety refers to the concept that an overall system will remain dependable and function as intended even in the event of an unplanned or unexpected occurrence. It provides the assurance that the safety-related systems will offer the necessary risk reduction required to achieve safety for the equipment. FuSa is based on the concept of risk reduction [1][2][3].

Two types of requirements are necessary to achieve functional safety:

1. Identification of the safety function
2. The safety integrity level – Measure of the risk level

In automotive domain, risk is quantified as ASIL (Automotive Safety Integrity Level). ISO 26262 classifies the risk levels from ASIL - A to D based on the most to least stringent safety measures [1].

In ISO 26262, SEooC (Safety element out of context) typically applies to items developed by Tier 2 or Tier 3 suppliers, who may not have the complete system information. Semiconductor components can be developed as an SEooC. SEooC developer makes assumptions how the element will be used and detailed safety requirements are derived from these assumptions.

The SEooC flow involves a series of activities and deliverables that ensure the component or software element meets the required safety standards. Some of the key steps in the SEooC flow include:

1. Safety Requirements Specification: Define the safety requirements for the component or software element, based on the intended use and application.

2. FMEDA (Failure Modes and Effects Analysis): Perform a failure modes and effects analysis to identify possible failure modes, their causes, and potential consequences.
3. FMEA (Failure Modes and Effects Analysis) validation: Validate the FMEA results by identifying the critical items that must be controlled to prevent or mitigate failure.
4. Safety Manual: Develop a safety manual that provides detailed information on the safe use and integration of the component or software element.
5. Assessment and Qualification: Assess the component or software element against the specified safety requirements and perform qualification activities to ensure compliance with the relevant standards.

B. FuSa compliance RTL Design architecture Overview

Technical safety requirements (TSRs) are developed from the assumed functional safety requirements. Design microarchitecture includes fault instances, such as Fault Generators or Fault Checkers, associated with each TSR item. These fault instances serve as protection mechanisms in the event of a fault occurrence. To ensure thorough testing, it is recommended to implement fault injection support for all safety-critical items. This internal fault injection interface enables real-time testing, allowing for the detection and analysis of faults [4][5][6].

Some common fault instance designs used is given below.

1. Data and address parity generators and checkers.
2. Control and status register (CSR) duplicate error checkers.
3. Transaction timeout error checkers.
4. CRC generators and checkers
5. SRAM protection (single-bit or two-bit ECC error checkers).
6. FSM integrity (One-hot error) checkers.

Figure 1 is an example of a parity fault checker RTL module. In this module, the data_in and parity_in signals serve as inputs that carry actual functional information. Conversely, data_in_fault_inj and parity_in_fault_inj are fault injection signals corresponding to data_in and parity_in, respectively. These two fault injection signals are utilized for real-time testing purposes [4][5][6]. Notably, the signal width can be controlled using input parameters, and an output parity_err signal is generated to report detected parity errors.

```
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

In complex digital designs, numerous interfaces, Control and Status Registers (CSRs), RAMs, timeout conditions, Finite State Machine (FSM) states, and other components are often intertwined. According to the safety manual, a fault instance (Which helps to move to safe state.) is required for every safety-critical item (TSRs). Fortunately, most fault instance modules inside the RTL design can be reused and instantiated multiple times with the required parameters and inputs. For instance, every interface data signal is typically parity-protected, requiring

either a parity generator or parity checker within the module, depending on the interface signal direction. The same parity checker module can be instantiated several times, accepting interface data signals, parity signals as inputs, and data width and parity width as input parameters.

Usually, registers are available in the design for internal fault injection, and the application layer can program these registers to inject faults. However, maintaining injection and reporting registers in every part of the design can be challenging. To address this issue, most microarchitectures incorporate a Fault logger module, through which internal fault injections can be exercised. This module is then connected to all the fault instances inside the surrounding modules through wiring.

C. Design Verification for FuSa Certification

In the context of Functional Safety (FuSa) certification, the Verification team plays a crucial role in providing or helping to create work products that are essential for the certification process.

The three primary verification work products required for FuSa certification are:

1. Verification Plan [4][5].
2. Verification Specification [5][6].
3. Verification Report [5][6].

The Verification team is also involved in creating the Requirements Traceability Matrix (RTM), which ensures that each safety requirement is linked to its corresponding verification test case and verification result. Tools like Jama Connect specialize in requirements management and provide a platform for managing RTMs effectively. An example of RTM is shown in Figure 2. As part of this process, the Verification team must map verification test cases, verification results, functional coverage, and other relevant data to the applicable Technical Safety Requirements (TSRs) or Technical Requirements (TRs). This meticulous mapping ensures clarity and accountability throughout the verification process. Safety items mentioned in the safety manual must be tested by injecting Transient Faults and Latent Faults. These test cases must demonstrate a connection to the TSRs using the RTM. Additionally, Single Fault Interruption (SFI) and Double Fault Interruption (DFI) testing is required for all identified safety items within the Device Under Test (DUT).

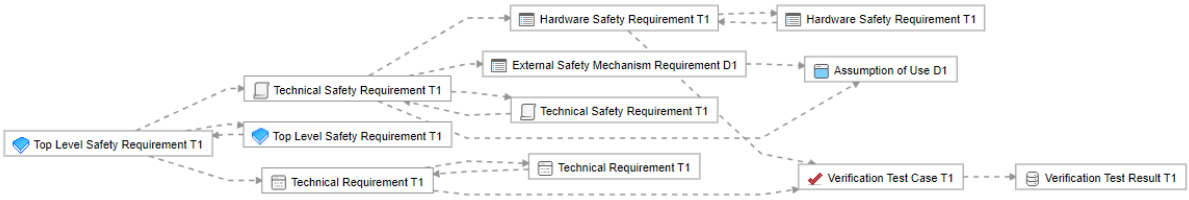

Figure 2: Requirements Relationship diagram from TSR to Test result

Testbench for FuSa Design verification must support following items, in addition to other functional verification methods.

1. Injection and checking of both latent and transient faults.
2. Testcases or Functional coverage for every safety-related item.
3. Unnecessary fault interrupt assertion checks to prevent false triggers.
4. Injection and checks for SFI and DFI.
5. Checks to ensure proper fault reporting by the design.

Fault injection can be performed through two primary methods: the Fault Injection Interface (FII) or the functional path. In this document, fault injection via the Fault Injection Interface is referred to as Internal Fault Injection, whereas injection through the functional path is referred to as External Fault Injection. Examples of External Fault Injection include sending incorrect parity over an interface, sending an incorrect Cyclic Redundancy Check (CRC) in a packet, or presenting incorrect Error-Correcting Code (ECC) in the read data of a Random Access Memory (RAM).

II. DESIGN VERIFICATION OF FUSA RTL DESIGN

A. *Existing verification Methodology*

Directed Testbench Development: This involves creating a testbench environment to validate specific Functional Requirements of the Design Under Test (DUT). Directed tests focus on checking specific behavior or transactions of the design. Testcases written in this stage are designed to function independently, having specific expectations and checking for those expectations only. Many numbers of FuSa related directed testcases created to exercise each fault instances and fault logging and interrupt assertions is tested in each test.

Fault Logging Monitors and checkers: In a classic FUSA methodology, the monitor elements are added inside the Design Under Verification (DUV) only when specific tests are written to capture the signal values required for the test. The post-simulation results are verified and compared to the expected results manually or with script-based help. A pin level UVM monitor logic created for monitoring every DUT (Design Under Test) interrupt ports. This is not a reusable monitor and makes plenty lines of code when interrupt/error pins are more.

Constrained-Random UVM Testbench: The testbench developed with a set of constraints to limit random test cases to only feasible cases. Constrained-random verification helps in ensuring coverage of specific corner cases, conditionals, or parameterizable conditions while regulated via constrained directives. Fault injection makes plenty of issues in the testbench and fault injection handling logic in the testbench leading to a complex testbench. Usually, fault injection can be done through registers, corrupting interface bus, external fault injection input signals, HDL forces, etc. For all these cases, a lot of testbench logic require to make the smooth test simulation with proper checkers, data traffic, FSM state changes, etc. This complexity increases significantly if the design is compliance to FuSa.

Formal Verification Checks: Verification engineers created assertion checks for specified functionality. Significant time and effort required for creating formal checks for all DUT fault instances.

B. *Introduction to FuSa Vector UVC*

Both creating new FuSa compliance testbenches and enhancing existing testbenches to meet FuSa compliance requirements involve significant time and effort, requiring multiple debug cycles. The design under test (DUT) must recover from a fault state through a smooth transition, with each part of the testbench synchronized with the recovery steps. The complexity of this process increases substantially as the number of safety items, also known as TSRs, grows.

FuSa compliance RTL modules typically comprise hundreds of fault instances, depending on the complexity of the DUT. This leads to thousands of fault injection or reporting input/output (I/O) ports, assuming that the error or fault injecting and logging registers are not part of the DUT module. Usually, the fault logger takes input from various modules, logs the status inside a register, and provides a few output interrupts.

This paper presents a reusable FuSa Vector UVC (Universal Verification Component) designed to be integrated into any constraint-random UVM (Universal Verification Methodology) testbench that involves fault injection and reporting I/Os.

The primary benefit of this UVC is the simplification of testbench complexity, particularly for FuSa compliant RTL designs that feature fault injection and fault logging I/Os. By utilizing the FuSa Vector UVC, users can generate constraint-random fault injection vectors, which can then be employed to inject faults into any part of the safety requirements. Furthermore, this UVC streamlines testbench logic related to fault injection and reporting within scoreboards, sequences, coverage collectors, and other components.

Figure 3 shows UVC interface example with injection and logging signals vectors declared. This signal width must be decided by the UVC user based on the DUT interface.

```
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

```
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

The integration and verification of FuSa Vector UVC are elucidated through example design and testbench codes. Figure 4 illustrates an example DUT that incorporates four parity checker instances to perform parity checking on interface input data. The parity checker instance module is depicted in Figure 1. Each checker has input injection vectors for fault injection and a corresponding output wire for fault reporting, facilitating comprehensive fault analysis and detection.

The FuSa Vector UVC is a comprehensive verification component that consists of a UVM sequencer, driver, and monitor, all encapsulated within a UVM agent.

The sequence item is designed to incorporate two primary vector variables: fault injection and fault detection. The size of these vectors is parameterized, allowing for flexibility and scalability in various verification environments. At the time of integration, the UVC user must define this parameter to adapt the UVC to specific module requirements. Additionally, the sequence item can be extended to include customized control variables based on unique verification needs, providing further flexibility in test scenario development. To accommodate specific requirements, users can extend the FuSa Vector UVC sequence item class to create a custom class and override the parent class as needed. Furthermore, the interface of this UVC is designed to maintain consistency by incorporating the same parameterized vectors, ensuring alignment between the sequence item and the interface.

```
// 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

```
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

This UVC monitor is designed such that any toggling of injection or reporting vectors will create a transaction item and send it into the analysis port. Users can then take the transaction item and perform the required processing based on the values. Figure 6 illustrates an example of the monitor's run phase logic.

C. *Integration of FuSa Vector UVC in to the Testbench*

Figure 7 illustrates an example Testbench architecture diagram with an integrated FuSa Vector UVC. The FuSa Vector UVC initiates fault vector transactions using a virtual sequencer. Monitored transactions are sent to a fault scoreboard and other necessary components through an analysis port, depicted by the green line. When a fault is detected, monitored transaction item is shared across the Testbench components via the red line. Some sequences wait for the reported fault, which can be accessed through the p_sequencer handle (shown as red dotted line).

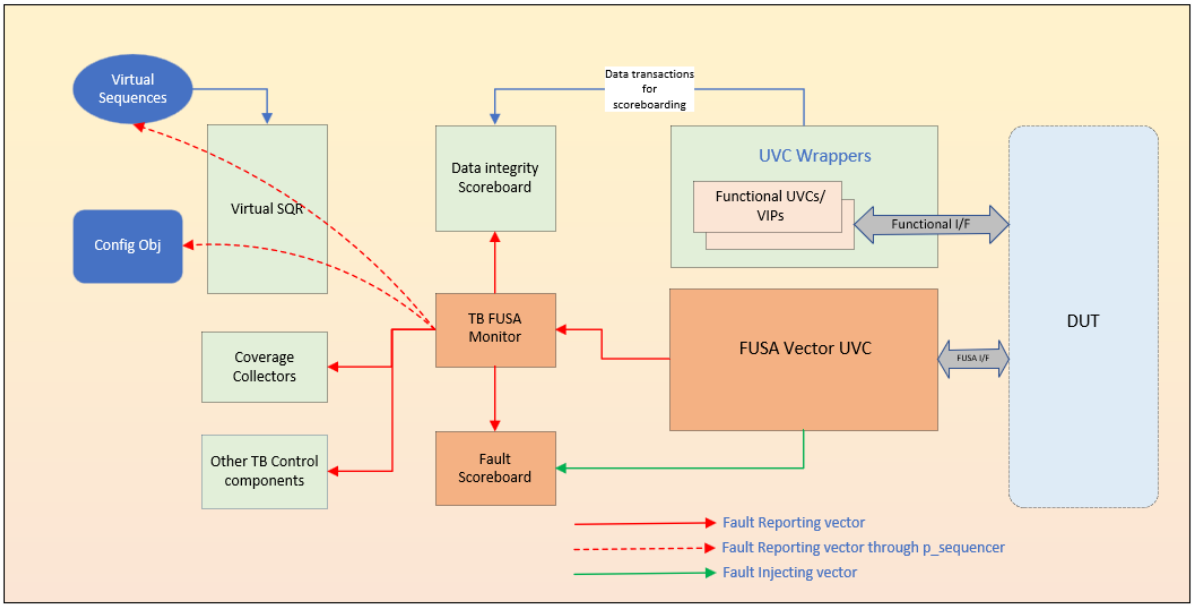

Figure 7. Testbench architecture diagram

```
//-- 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

It is highly recommended to create a fault injection structure that includes all the input signals of the DUT where faults can be injected, as well as a fault logging structure that comprises all the output signals of the DUT that require monitoring and reporting. Additionally, it is essential to calculate the fault injection vector size and fault

logging vector size parameters. Figure 8 illustrates an example of how these structures can be created and the corresponding parameter calculations can be performed.

D. Random Fault generation and fault checks using FuSa Vector UVC

Random fault injection data is driven to the virtual interface by the UVC driver, enabling dynamic and realistic fault simulation. This provides a robust methodology for testing the reliability and fault tolerance of the design. To connect the interface signals to the DUT, DUT-specific fault signal structures explained in the Figure 8 is used. This enables the verification environment to mimic real-world scenarios and test the DUT's response to faults. Figure 9 illustrates an example of DUT instantiation in the TB top-level module and demonstrates the connection method between the FuSa vector UVC interface and DUT inputs/outputs.

```
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

This UVC designed for FuSa applications allows sequences to randomize FuSa Vector sequence items and generate constrained random fault vectors. Additionally, transaction items from the monitor can be utilized to verify fault assertions in various functional (External) fault injection scenarios, including parity fault injection over interface data, cyclic redundancy check (CRC) fault injection, error-correcting code (ECC) fault injection, finite state machine (FSM) one-hot fault injection and Timeout fault injection scenarios. This UVC provides monitored

vectors through an analysis port, enabling testbenches to leverage these transaction items for scoreboarding, coverage analysis, testbench control, and other purposes.

Figure 10 shows an example of fault vector generation based on an example constraint. Some tasks also shown in the figure related to pre fault injection configurations and post fault detection checks.

```
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

The testbench (TB) with integrated fusa Vector UVC can generate an expected transaction item based on the fault injection and further compare the actual fault from the DUT with the expected transaction item. These fault queues can further be used to perform end of simulation checks to make sure all intended fault injections happened successfully and no unintended faults were detected by the DUT.

This UVC can also be integrated at various levels such as sub-system, System-on-Chip (SOC), and top-level testbenches as a passive agent for monitoring the faults. For clarity on internal fault and external fault scoreboarding, example code is available for illustration, such as in figure 11 and figure 12, which distinguish between these two types of faults.

```
  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

III. RESULTS

The FuSa vector UVC has been successfully integrated into multiple testbenches at both the module and top-level TB and efficiently handling thousands of fault injection vectors and reporting vectors. Due to its modular and scalable architecture, the FuSa vector UVC has been effectively utilized for dynamic fault injection, comprehensive checker coverage, and monitoring purposes across various use cases.

The presented approach has been successfully used in UCIe (Universal Chiplet Interconnect Express) Controller IP (FuSa certified IP) project, which is a highly configurable and complex design. FuSa vector UVC reused across five module level testbenches as an active component and it is utilized as a passive component for various interfaces

within the top-level testbench. The stability and effectiveness of this UVC have been thoroughly verified, yielding successful outcomes across all testbenches involved in the project.

Please find the testbench simulation snapshot of UCIe Adapter Module testbench where random fault injection across different safety points along with the data traffic. A wave diagram of fault injection along with traffic to few UCIe adapter mainband safety points is illustrates in Figure 13. Output fault assertion and data traffic signals also highlighted in the Figure.

Figure 13: Simulation Result representing multiple fault injections and fault reporting along with UCIe mainband traffic

Considering the criticality of FuSa verification, a separate code coverage analysis was performed with an auditor for all FuSa safety points. This was achieved in a remarkably short timeframe. Below are a few examples of the results obtained for the UCIe Adapter mainband FSM one-hot RTL instance's Code Coverage (CC).

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

Hundred percent functional coverage (FC) for all safety items (TSRs), is required to be presented for verification completeness. Notably, this UVC supports direct conversion of monitored fault-reported vectors with TSR interface signals using the Verilog structures. As a result, we were able to sample all FC elements related to TSRs immediately after receiving the transaction item from the FuSa vector UVC monitor. This approach enables us to achieve and report 100% FC in the FuSa verification report work product.

To create and manage the FC plan, we utilize Cadence vPlan (Verification Plan) within the vManager tool. This plan is then seamlessly mapped to Jama Connect using REST APIs, thereby establishing a connection between verification results and corresponding TSRs in a remarkably short timeframe. FC closeness metric is illustrating

verification test results as a pass in Jama Connect, indicating that the test suite has achieved a satisfactory level of functional coverage.

A total of bugs related to Cadence UCIe Controller Functional Safety (FuSa) RTL design were detected within a 6-month timeframe. A significant 75% of these bugs were identified in less than 3 months. The graph illustrating this trend is derived from data extracted from Jira software, specifically filtered to focus on bugs associated with the Cadence UCIe Controller FuSa functionality.

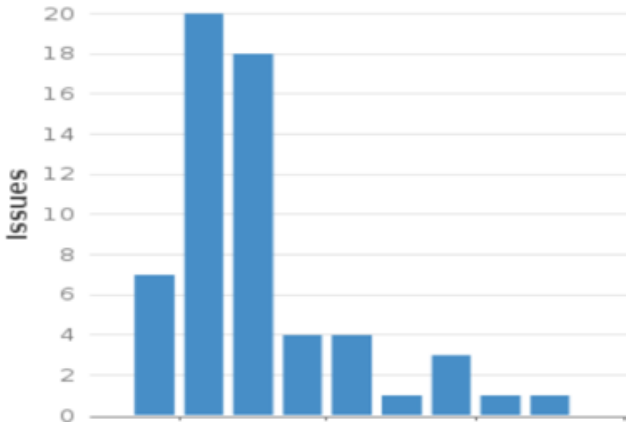

Figure 15: Graph representing UCIe Controller monthly ASF RTL bug rate

Cadence UCIe controller got ISO26262 ASIL B certification after submitting FuSa work products on November 2024. Verification report created with the 100% regression and coverage metrics. All the safety points verified with internal and external fault injection and mapped all these metrics to Jama Software for review.

A. Other Benefits of using FuSa Vector UVC

- This UVC integration into the testbench required less effort and enabled simulation of the design within a two-man-week timeframe.
- This approach helps simplify all types of fault injection and checking methods (both external and internal faults) by enabling predictions for fault reporting signals after fault injections within the scoreboards by analyzing the monitored transactions.
- Most of the RTL safety mechanisms (E.g. Parity Generator or checker modules, ECC generators and checkers, Duplication fault checkers) for TSR's are reused across the design modules. These RTL instances are differed only with respect to I/O connection and parameters passed. All the parameter (E.g. Signal width, odd or even parity, etc.) issues, I/O connection issues identified easily and in quick time.
- Fault injection testing with and without traffic is not a complex task since fault injection transactions are send from this UVC sequence in parallel to other traffic sequences.
- Most of the fault injection testcases controlling the testbench (E.g. Pause and resume of traffic, moving to different states, applying the reset, Flushing the scoreboards, etc.) is a complex task. By using FuSa Vector UVC, testbench could wait for transactions from this UVC monitor and then took necessary action based on the transaction item field values.
- This approach helped to reduce significant amount of time for closing the Code Coverage (CC) analysis. Achieving 100% CC for all safety instances will reduce the risk for FuSa verification. There are 100+ safety instances in the design and a total of 200+ I/Os for fault injection and reporting considering all the safety instances. This is contributing around 10,000+ elements in the toggle coverage only because of signals widths. 100% CC easily achieved by adding distribution constraints in the FuSa vector UVC transaction randomization.

- The RTM work product necessitates a verification result that illustrates a 'PASS' for ISO 26262 ASIL (A-D) certification. Transaction items from FuSa Vector UVC monitor are utilized for sampling TSR related cover groups by differentiating between internal and external fault injection. This FC (vPlan using vManager) is directly mapped to the Jama Connect tool (Example of RTM) via REST (Representational State of Resource) APIs. Ultimately, achieving 100% FC results demonstrates a 'PASS' for each RTM verification result connected to the TSRs.
- This UVC is used as a passive component at the top-level testbench (TB), which facilitates the identification of unexpected assertions related to module errors or fault signals. These issues often arise when a module receives incorrect input values from other modules or TB interfaces. Furthermore, the passive component is instrumental in fault logging and fault injection path verification.
- During fault injection testing, whether internally or externally simulated, the safety item instances may experience fault propagation leading to another type of fault detection. However, this can impact the testbench's smooth operation. Fortunately, the UVC's monitored transaction item allows the testbench to take the necessary actions upon detecting fault injection or propagation, ensuring the system's robustness.

IV. CONCLUSION

The FuSa Vector UVC significantly simplifies the development of testbenches and reduces verification time for RTL designs targeting Functional Safety (FuSa) compliance. Due to its reusability and seamless integration into both existing and new testbenches, we highly recommend this UVC. It can be applied across various levels, including module, sub-system, and SOC-level testbenches, within any UVM testbench environment. The UVC facilitates the swift identification of design bugs and offers easy synchronization techniques following fault injection, making it a more viable option for efficient verification. Notably, this UVC is completely independent of the DUT, indicating that the testbench itself should be adaptable for each FuSa verification test case.

REFERENCES

[1] ISO 26262. Road vehicles - Functional safety. (2018-12). International Organization for Standardization.

[2] IEC 61508 Ed. 2.0 2010-04 Functional safety of electrical/electronic/programmable electronic safety-related systems

[3] IEC 61709 Edition 3.0 2017 Electric components – Reliability – Reference conditions for failure rates and stress models for conversion

[4] ISO 26262. Road vehicles - Functional safety. Part 11: Guidelines on Application of ISO 26262 to Semiconductors. Paragraph 5.1.9. (2018-12). International Organization for Standardization.

[5] ISO 26262. Road vehicles - Functional safety. Part 5: Product Development at the Hardware Level. Paragraph 7.4.4. (2018-12). International Organization for Standardization.

[6] ISO 26262. Road vehicles - Functional safety. Part 8: Supporting Processes. Paragraphs (9.4.1.1.a. , 9.4.1.1.b. , 9.4.1.1.c. , 9.4.1.1.d. , 9.4.1.1.e. , 9.4.1.1.f. , 9.4.1.1.g. , 9.4.1.1.h. , 9.4.1.1.i. , 9.4.1.2.a. , 9.4.1.2.b. , 9.4.1.2.d.) (2018-12). International Organization for Standardization.

[7] ISO 26262. Road vehicles - Functional safety. Part 8: Supporting Processes. Paragraphs (9.4.2.1.b. , 9.4.2.1.c , 9.4.2.2.a. , 9.4.2.2.b. , 9.4.2.2.c. , 9.4.2.2.d. , 9.4.2.2.e. , 9.4.2.2.f. , 9.4.2.2.g. , 9.4.2.3.a. , 9.4.2.4.) (2018-12). International Organization for Standardization.