Error Injection: When Good Input Goes Bad

Kurt Schwartz
Aletheia Design Services, LLC
1939 Westlake Loop, Newberg, OR 97132
kurt@aletheia-ds.com
503-538-7589

Tim Corcoran.
Willamette HDL, Inc.
6107 SW Murray Blvd. No. 407, Beaverton, OR, 97008
tim@whdl.com
503-590-8499

Abstract

Wouldn't it be nice if we could guarantee that only well-formatted, completely predictable data would arrive at the inputs to our designs? In a simulated world, that would be possible. Unfortunately, the real world in which we live has no such guarantees. Fabrication processes are not perfect, environmental conditions are frequently hostile, and the specter of human error always lurks. In addition to normal functionality, gracefully handling error conditions is an indicator of robustness, which in turn is a key factor in improving total product quality.

We will focus on two aspects of error injection – where is the best place to perform the injection, and how the injection should be specified and implemented. We will examine placing error injection inside the transaction, in a UVM sequence, and in the driver component. After determining the best place to perform the injection, we will evaluate a common technique for specifying and implementing deliberate errors, and propose a more reusable approach.

I. Introduction

Normally, a portion of verification stimulus is devoted to deliberately presenting erroneous stimulus input and evaluating the device's response. The common term for this is called "error injection." Since failures can happen at any point internal or external to the design, it is challenging to devise a uniform error injection scheme that can model the almost unlimited ways errors can occur.

UVM has no preferred methodology for how and where to perform error injection. We will examine some patterns that we have encountered through our consulting and training work, evaluate their benefits and limitations, and highlight an approach to error injection that epitomizes the core values of UVM - flexibility, scalability, and reusability.

II. Error Injection Definition and Placement

Error injection can be performed at any point along the path of stimulus generation – in the transaction code itself, in the sequence that creates transactions, or in the driver that processes transactions.

A. Transaction modification.

Error injection in the transaction can be done through class extension (Figure 1), commonly involving randomization constraints in the extended class or through direct injection. For example, a transaction class has a CRC field and has a function that calculates a correct CRC based on the payload. An error-injecting class that extends from the transaction class could override the CRC function to produce a faulty value. A test could then set a factory override to create instances of the error-injecting transaction wherever the normal transaction is used. This approach is good for scalability and reusability because as new errors are defined, no existing code needs to be modified. A significant limitation of this placement of error injection is that you can only inject errors using information available in the transaction itself. There is no visibility into other transactions in a stream or control over errors that require protocol knowledge residing in the driver/interface.

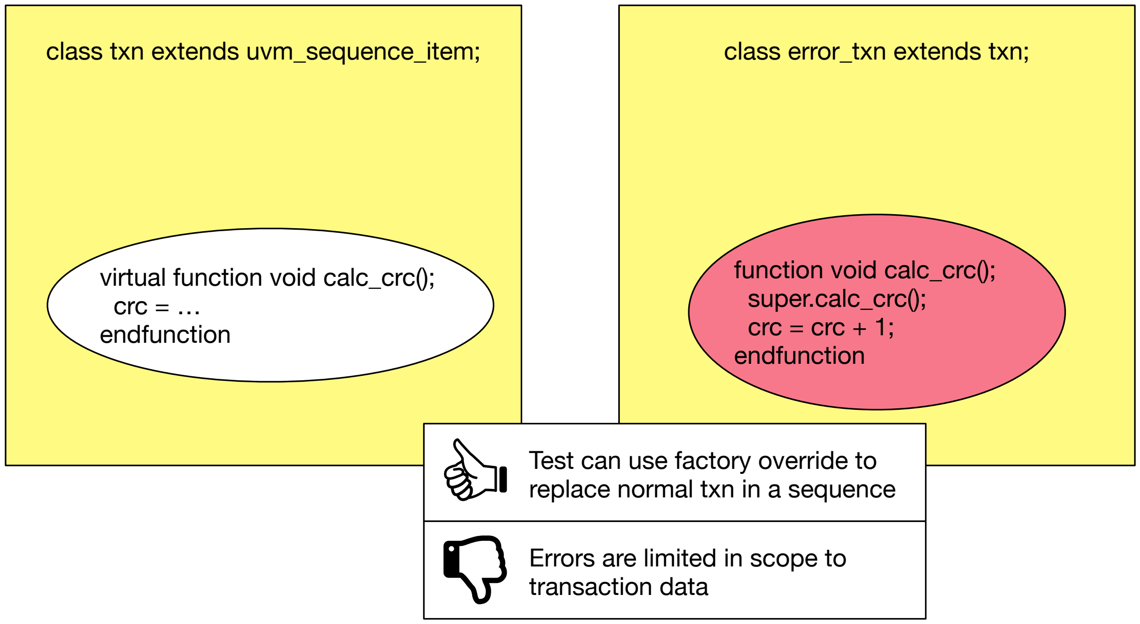

class txn extends uvm_sequence_item;
virtual function void calc_crc();
crc = …
endfunction
class error_txn extends txn;
function void calc_crc();
super.calc_crc();
crc = crc + 1;
endfunction
Test can use factory override to replace normal txn in a sequence
Errors are limited in scope to transaction data

Figure 1. Error injection through transaction class extension.

B. Sequence modification.

Error injection can also be done by writing an error-injecting sequence class. Sequences are the source of transaction selection, so they are naturally the best place to specify which error to inject. In the sequence, transactions are created normally but then modified/corrupted in the sequence code. For example, an error-injecting sequence would create a normal transaction with a correct CRC. Before sending the transaction to the sequencer, the sequence would reach in and modify the CRC field (Figure 2). This approach has better flexibility as it can generate correlated errors across

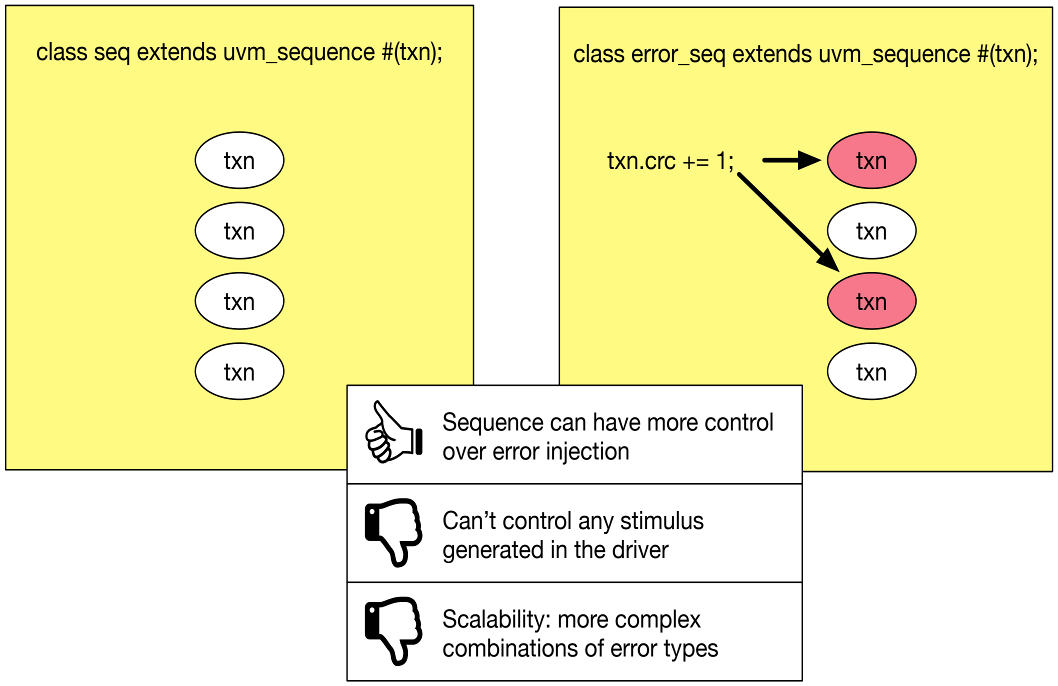
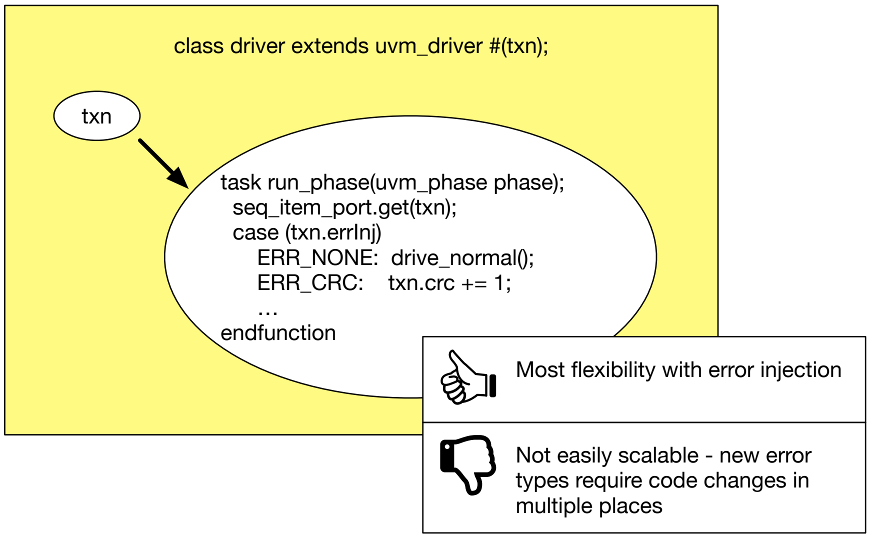

class seq extends uvm_sequence #(txn);
txn
txn
txn
txn
class error_seq extends uvm_sequence #(txn);
txn.crc += 1;
txn
txn
txn
txn
Sequence can have more control over error injection
Can't control any stimulus generated in the driver
Scalability: more complex combinations of error types

Figure 2. Error injection in a sequence.

transactions. A limitation is that the approach lacks the ability to perform errors related to the signal-level protocol. Also, it suffers because as error injection complexity increases, there is little opportunity for reuse of the sequences as the sequences also get more complicated.

C. Driver error injection.

The advantage of placing error injection code in the driver is that it is the only place to inject protocol or timing-related errors. Another advantage is that by the time the transaction gets to the driver, all the transaction information and all the protocol is together in one place. Error injection placed here has access to all aspects of input to the device. In our experience, the most common approach for driver-based error injection is to write the driver with the ability to generate a predefined set of error patterns. The driver will inject the patterns based on an enumeration field in the request transaction (Figure 3). The driver looks at the enumerated field and performs the error injection based on a chained if-else or case statement. A benefit of this strategy is the separation of concerns. The driver performs the error injection but the choice of error injection is done in the sequence. The transaction is just a carrier of the injection instruction. A limitation of this strategy is that it is more difficult to scale and maintain if more kinds of error injection are introduced after the initial code is written. For every new error injection that is introduced, code must be changed both in the definition of an enumeration and in the if/case statement of the driver.


class driver extends uvm_driver #(txn);
txn
Most flexibility with error injection
Not easily scalable - new error types require code changes in multiple places

```
task run_phase(uvm_phase phase);
  seq_item_port.get(txn);
  case (txn.errInj)
     ERR_NONE:  drive_normal();
     ERR_CRC:    txn.crc += 1;
     …
endfunction
```

Figure 3. Error injection in driver.

There is one more thing to take into consideration regarding driver-based error injection. If you use the common agent strategy of placing your protocol and timing code in a BFM interface, error injection in the driver would need to be carefully designed to operate in concert with the interface. Since Systemverilog interfaces are not object-oriented or replaceable by the factory, you would have to write your interface code in a parametric way so that invalid parameters can be passed in, or you would have to write alternate error-injecting API tasks in the interface for the driver to call.

Each of the strategies listed above are, by themselves, insufficient to accomplish a general-purpose, reusable scheme. The transaction error-injecting technique has the most reusability but has the least scope. The driver technique has the best scope but has the least reusability. None of the approaches are scalable. We want to design a better approach that is scalable and reusable. We want to add new error injections with little or no change to existing code. We need the error injections to be flexible enough to handle the wide spectrum of potential errors from simple transaction data errors to protocol and timing errors.

III. Highlighted Approach

The approach that we will highlight promotes reusability by encapsulating error injection details inside "error injector" objects. The base error injector class defines a single virtual function, `inject()`. The inject function takes two inputs. One input provides a data stream in a form that is as close as possible to the format that will be passed to the device. The stream could be a single transaction, a queue of transactions, or a transformed stream of data blocks that have been prepared by the driver. The second input is a handle to the driver so that the error injector has access to any utility or API functions that the driver provides. Derived error injector classes provide an override for the `inject()` function that encapsulates all the details of a particular error behavior. These objects are created in a sequence and attached to transactions as they are sent to the driver.

The driver processes the transaction normally. Before sending the data to the device, it will check to see if the transaction has an attached error injector object. If so, it will call the `inject()` function. The error injector will then corrupt the data stream or change BFM parameters just before the data is transmitted to the device (Figure 4). This technique has the benefit that the driver does not need to know about error injection types or enumerations, so new error injections can be defined without changing the driver code. The driver processes transactions normally and only needs to know about the error injector base class. It makes the call to the `inject()` function polymorphically.

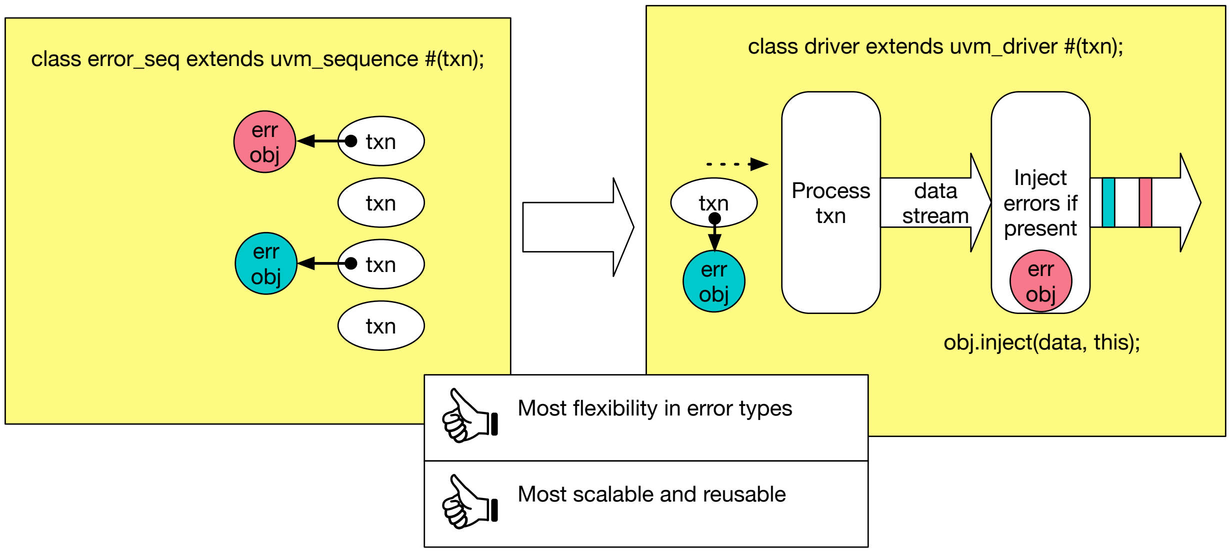

class error_seq extends uvm_sequence #(txn);
err obj
txn
txn
err obj
txn
txn
class driver extends uvm_driver #(txn);
txn
err obj
Process txn
data stream
Inject errors if present
err obj
obj.inject(data, this);
Most flexibility in error types
Most scalable and reusable

Figure 4. Error injection with error objects.

Through its two inputs, the `inject()` function has access to the full set of data being driven, and access to all the driver functionality and parameters via the driver handle. The driver handle allows the error object to access driver utility functions, and allows the error object to change/corrupt timing parameters that the driver or BFM interface uses.

IV. Example of Highlighted Approach

We will now apply this technique to a test environment that generates a data stream used in high speed networking equipment. IEEE spec 802.3ba defines a 40 Gb/s intra-chip interface called XLGMII (XLG = 40G, MII = Media-Independent Interface). An XLGMII interface is made of two streams of octets (8-bit values). One stream (TXC) is one octet wide and holds a control value that describes the content of the data (TXD) stream, which is eight octets (64 bits) wide. Each set of 8 octets in the TXD stream is called a "record."

In this example, the overall block of data is divided into a packet structure (Figure 5), where the first "preamble" record (SOP) contains a particular set of octet values. Following the SOP, there are up to 128 data records containing the main payload, broken into 8-octet chunks. After the data records, there is a special record signifying the end of the packet (EOP). The EOP contains a 4-octet checksum (FCS) of all the data records, a special termination octet value and 3 “gap” octets. After the EOP there are one or two more “gap” records before another packet starts.

The XLGMII spec provides 4 parallel 8-bit control and 64-bit data busses (TXC0-3 and TXD0-3), onto which the BFM will apply the record stream, 4 records at a time per clock cycle.

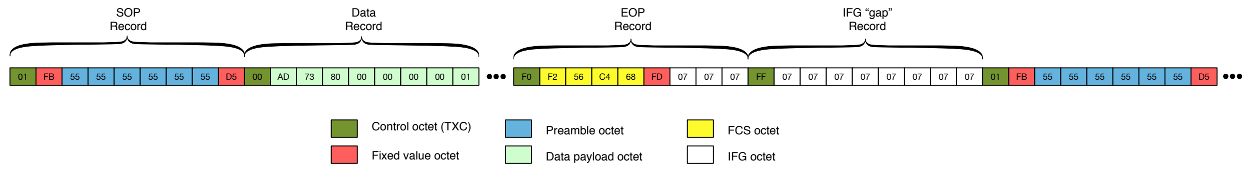

SOP Record
Data Record
EOP Record
IFG “gap” Record
01 FB 55 55 55 55 55 55 D5 00 AD 73 80 00 00 00 00 01 ••• F0 F2 56 C4 68 FD 07 07 07 FF 07 07 07 07 07 07 07 07 01 FB 55 55 55 55 55 55 D5 •••
Control octet (TXC)
Fixed value octet
Preamble octet
Data payload octet
FCS octet
IFG octet

Figure 5. Octet stream format.

Strategy

The overall stimulus generation strategy will be to abstract the stimulus into record descriptors generated by sequences which will be processed by the driver to create actual record octets. The descriptors can have one or more error objects attached to them in an error object queue. The rest of this section will walk through the stimulus generation flow. You can refer to the following (Figure 6) as a “map” while you read the example code.

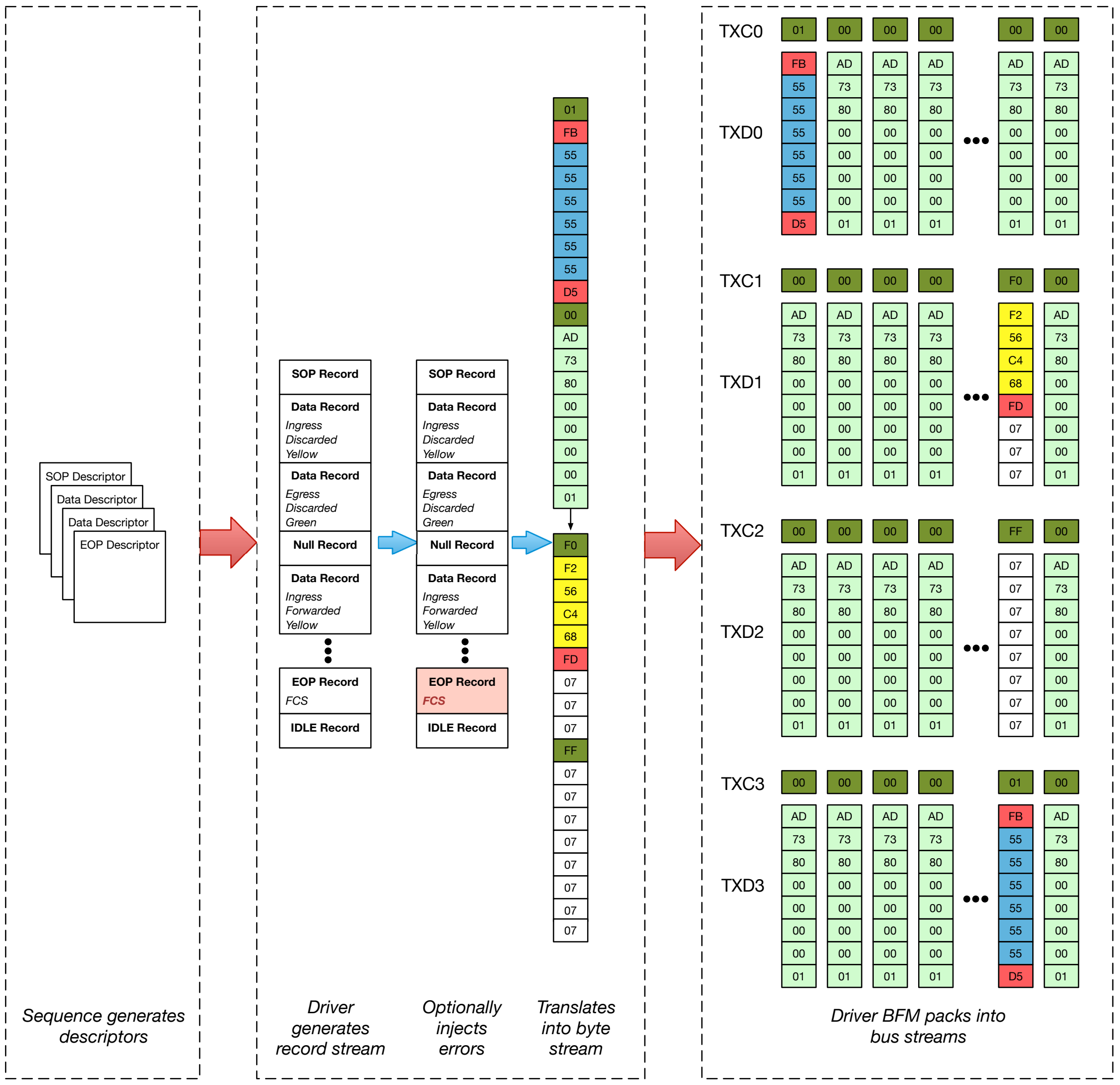


Figure 6. Overview of object-based error injection strategy.

Record Descriptor

The XLGMII record descriptor (Snippet 1) is a UVM sequence item that specifies a high-level description of the record type and the parameters necessary for the driver to create a record octet stream. Additionally, it contains a queue of error injection objects, and acts as a "carrier" for those objects to the driver.

```
class xlgmii_txn extends uvm_sequence_item;
`uvm_object_utils(xlgmii_txn)

  xlgmii_txn_t txn_type;  // e.g. SOP, Data, EOP, Gap
  distribution_table distribution; // Controls random weights for data

  rand xlgmii_txn_ipg_t ipg_style;
  error_injector_base error_injectors[$]; // Queue of error injectors

// Other data and methods not shown

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

endclass
```

Snippet 1. Record descriptor class

Error Injector Base Class

The error injector base class (Snippet 2) defines the virtual function inject(). This function takes two inputs. The first is a block stream, which is a queue of XLGMII records containing the actual values of the TXC and TXD octets filled in by the driver. The second input is a handle to the driver.

```
class error_injector_base extends uvm_object;

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

   virtual function void inject(ref blockStream    blocks,
                                    xlgmii_driver   driver);
   endfunction

endclass
```

Snippet 2. Error injector base class

Error Injector Implementation Class

Each kind of error injection is implemented in a class that extends the error injector base class (Snippet 3) and provides a concrete implementation of the inject() function. Note that the error injecting function has access to the entire record stream of the packet, as well as a handle to the driver, which can provide API utility functions and access to timing parameters.

```
class ei_FCS extends error_injector_base;
   `uvm_object_utils(ei_FCS)

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

   function void inject(ref blockStream    blocks,
                            xlgmii_driver  driver);
      bit[31:0] crcVal;
      sif_eop_txn fcsBlock;

      foreach (blocks[i]) begin  // Find the EOP record by its TXC value
         if (blocks[i].control == 8'hF0) begin
            $cast(fcsBlock, blocks[i]);
            break;
         end
      end

      // Corrupt the FCS
      fcsBlock.fcs[7:0] = ~fcsBlock.fcs[7:0];

   endfunction

endclass
```

Snippet 3. Error injection implementation class.

Error Injecting Sequence

The error injection sequence (Snippet 4) creates record descriptor objects (transactions) that the driver will use to generate the record octet stream. In this example, the sequence calls a task called send_sop() which is a task in the base class that creates and sends a SOP descriptor. It has an optional argument that is a queue of error injection objects which it will assign to the descriptor, then send the descriptor to the driver.

```
class xlgmii_err_fcs_seq extends xlgmii_seq_base;
`uvm_object_utils(xlgmii_err_fcs_seq)

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

  task body();
    record_counter_key_t ckey_gen;
    xlgmii_pkg::ei_FCS einj;

    `uvm_info(get_name(), "Starting Sequence", UVM_MEDIUM)
```

Snippet 4. Error injecting sequence

```
    // Send 50 packets
    repeat (50) begin

      // 30% chance of packet having CRC error
      randcase
      7: send_sop(); // Normal descriptor. No error injectors
      3: begin
        einj = xlgmii_pkg::ei_FCS::type_id::create("ei_FCS");
        send_sop(.error_injectors( '{einj} )); // Create queue of 1 item
      end
      endcase

      void'(randomize(ckey_gen));

      send_record_block(.ckey_gen(ckey_gen) );

      send_eop();

    end // repeat

    `uvm_info(get_name(), "Finished sequence", UVM_MEDIUM)
  endtask

endclass
```

Snippet 4 (continued). Error injecting sequence

Driver

The driver (Snippet 5) receives the record descriptors, and uses the values in the fields to generate a stream (queue) of record transaction objects. These record objects each represent an 8-octet group of data. There are four kinds of record objects: sop_txn, record_txn, eop_txn, and idle_txn. A normal packet would be made up of a sequence of 130 of these objects ((preamble + 128 data + fcs), followed by one or two idle (gap) transactions. The driver fills in the data fields for these transactions based on the instructions in the record descriptors.

Once the record stream is created, it is passed to an error injection manager, and the error manager has a chance to inject errors into the stream of transactions, if any error injection objects exist. After the error manager has processed the queue of transactions, they are converted into their respective group of 8 bytes. They are then appended to a byte queue which is sent out to the BFM interface. The BFM then arranges them onto the four TXD and TXC bus streams.

```
class xlgmii_driver extends uvm_driver #(xlgmii_txn);
`uvm_component_utils(xlgmii_driver)

  ErrorInjectionManager ErrManager;

  virtual xlgmii_driver_bfm v_bfm;
  xlgmii_configuration m_config;
```

Snippet 5. Driver

```
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    v_bfm = m_config.v_driver_bfm;
    ErrManager = ErrorInjectionManager::type_id::create("ErrManager");
  endfunction

  task run_phase(uvm_phase phase);
    xlgmii_txn txn;
    int numGaps;
    bytestream pkt;
    blockStream blkStream;
    int thandle;

    forever begin
      seq_item_port.get_next_item(txn); // get descriptor
      case (txn.txn_type)
        SOP: begin
          ErrManager.reset(); // Start a new packet in a fresh state
          if (txn.error_injectors.size() > 0) begin
            // Install the error injectors, if any
            ErrManager.add_error_injectors(txn.error_injectors);
          end
          blkStream = {};
          blkStream = {blkStream, generatePreamble()};
        end
        REC:
          blkStream = {blkStream, generateRecords(txn)};
        IDLE: begin
          `uvm_warning(get_name(), "IDLE not supported")
        end

        EOP: begin
          blkStream = {blkStream, generateFCS()};
          blkStream = {blkStream, generateIPG(txn)};

          ErrManager.blocks = blkStream;
          ErrManager.perform_injections(this);
          // Send the data out the bus
          pkt = streamBytesFromBlockStream(ErrManager.blocks);
          v_bfm.send_packet(pkt);
        end
      endcase
      seq_item_port.item_done();
    end

  endtask

// Other utility tasks not shown

endclass
```

Snippet 5 (continued). Driver

Error Manager

Error injection is done by modifying the stream of record transactions generated by the driver. The error manager (Snippet 6), which is part of the driver, takes the generated transaction queue and applies error injectors to it by calling inject() on each injector in the queue. If there are no injectors installed, then no injection occurs. The inject() function takes the record queue as input by reference so that it can modify the queue and/or its contents as it sees fit. Error injectors also have access to the driver through a handle so that they can call upon driver API functions as needed (e.g. CRC calculation).

```
class ErrorInjectionManager extends uvm_object;
   `uvm_object_utils(ErrorInjectionManager)

   blockStream blocks;
   error_injector_base injectors[$];

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

   function void reset();
      blocks = '{};
      injectors = '{};
   endfunction

   function void add_error_injectors(error_injector_base _injectors[$]);
      injectors = {injectors, _injectors};
   endfunction


   function void perform_injections(xlgmii_driver driver);
      foreach (injectors[i]) begin
         injectors[i].inject(blocks, driver);
      end
   endfunction

endclass
```

Snippet 6. Error Manager

V. Scoreboarding with Error Injection

When the driver injects errors, it will normally induce the device into an error response. Typical error responses include any combination of flags in status register fields, error code registers, interrupts, or sometimes silent dropping of input and maintaining a count register of dropped values. This response is usually different from normal operating behavior, so any checkers, monitors, and scoreboards need to be made aware of this fact and expect the error response behavior. This means that there needs to be some kind of sideband communication between the driver and the analysis components, which can be handled nicely with UVM analysis ports, exports and fifos. This kind of communication is common, even in normal operating conditions.

The error analysis behavior can implement reusability by factory overrides set in the error injecting test class but you can also use the same reusable approach that we have shown in the driver. The error injection base class object could define one or more additional virtual functions that take the analysis component state and the component handle as input (Snippet 7). The derived error injection object would then provide a concrete implementation that uses the handle and state values to tell the component about the expected error behavior. The analysis component would then flag an error if the expected behavior is not witnessed (Snippet 8). The analysis component would call the base class function, and polymorphically be updated to expect the error condition (Snippet 9).

One significant benefit of this strategy compared to component factory overrides is that all the error behavior, from stimulus generation to expected changes in normal response, is encapsulated together in one place – the error object.

```
class error_injector_base extends uvm_object;

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

   virtual function void inject(ref blockStream   blocks,
                                    xlgmii_driver driver);
   endfunction

   virtual function void prepare_for_completion_error(
                                    completion_checker complChecker);
   endfunction

endclass
```

Snippet 7. Error injector base class with analysis function.

```
class ei_FCS extends error_injector_base;
   // Same as above...

   function void prepare_for_completion_error(
                                    completion_checker complChecker);
      complChecker.expected_error_state = 1; // This injection will
                                             // cause STATUSREG.ERR = 1
      complChecker.expected_error_code = 8'h05;  // FCS Error code
   endfunction

endclass
```

Snippet 8. Error injector implementation with analysis function

```
class completion_checker extends checker_base;

   `uvm_component_utils(completion_checker)

   uvm_analysys_export #(xlgmii_txn)      expected;
   uvm_analysys_export #(completion_txn) observed;
   // The above exports will be connected to these fifos:
   uvm_analysis_fifo #(xlgmii_txn)      expected_fifo;
   uvm_analysis_fifo #(completion_txn) completion_fifo;

   bit      expected_error_state;
   bit[7:0] expected_error_code;
```

Snippet 9. Completion-checking analysis component

```
   function new(string name = "compl_checker", uvm_component parent);
      super.new(name, parent);
   endfunction

   task run_phase(uvm_phase phase);
      xlgmii_txn expTtxn;
      completion_txn complTxn;
      error_injector_base errInj;
      bit ok;

      forever begin
         expected_fifo.get(expTxn);
         completion_fifo.get(complTxn); // Indicates completion of a
                                        // packet
         expect_no_error();
         if (expTxn.error_injectors.size() > 0) begin
            foreach (expTxn.error_injectors[i]) begin
               errInj = expTxn.error_injectors[i];
               errInj.prepare_for_completion_error(this);
            end
         end // if error injector present
         ok = check_error_status();
      end
   endtask

   function void expect_no_error();
      expected_error_state = 0;
      expected_error_code  = 0;
   endfunction

   function bit check_error_status();
      uvm_status_e status;
      uvm_reg_data_t rdval, errcode;

      // Update register model
      regModel.STATUSREG.read(status, rdval, UVM_BACKDOOR);
      regModel.ERRCODE.read(status, errcode, UVM_BACKDOOR);
      // Check the ERR field of STATUSREG
      if (regModel.STATUSREG.ERR.get() != expected_error_state) begin
         `ovm_error(get_name(),
                  "STATUSREG ERR field does not match expected value")
         return 0;
      end
      // Check the ERROCODE value
      if (errcode[7:0] != expected_error_code) begin
         `ovm_error(get_name(),
             $sformatf(
                  "ERRCODE value %x does not match expected value %x",
                   errcode, expected_error_code))
         return 0;
      end
      return 1;
   endfunction

endclass
```

Snippet 9 (continued). Completion-checking analysis component

VI. Summary

Errors can occur in an infinite variety, and injecting error stimulus and expected response is a required part of test development for robust devices that have specified behavior to error conditions. By choosing a methodology that integrates error injection by encapsulating the details of the error outside of the test environment, and polymorphically executing those details, error scenarios can be quickly added and evaluated. This approach remains true to the UVM tenets of flexibility, reusability, and scalability.