UVM Error Injection

Using a Two-Phase Slave Sequence

Mona Beimers

Xilinx Inc.

Kanata, ON, Canada

www.xilinx.com

ABSTRACT

Error testing is an important part of the functional verification of any project. It can be quite complex and not trivial to implement. The examples available are very few and very simplistic. Most of them target errors associated with the fields of a packet. I could find no examples of injecting errors at the physical layer. The goal of this paper is to present a possible implementation of error injection, orthogonal to the stimulus generation, and which can be applied at each layer of a protocol. The paper introduces the concept of an error driver connected to its own error sequencer and running an error sequence. The communication between the error driver and the error sequence models a slave device with two phases. The error injection is performed by the error driver, and not by the stimulus driver, which is not polluted with the error injection code.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Error injection and detection is a necessary step in the verification of any project. In general, it consists of injecting stimulus with errors and verifying that the device under test (DUT) is able to handle it, report it and continue operating as documented in the functional specification. Error testing is usually implemented in the later part of the verification process, after the data-path has been verified. Error testing can be quite complex and therefore not easy to implement. A verification engineer could benefit from following a well-known methodology for UVM error testing instead of implementing everything from scratch. There are several possible sources of information: UVM documentation, reuse from previous projects and the internet. The UVM User Guide [1] does not provide a clear template to follow in this case. Previous projects are very useful if they are using the UVM methodology and have the error testing implemented. If they were implemented in other methodologies, they could be used as a starting point, but they might need many changes. Online, there are very few hits on the subject of UVM error testing. One of the best resources is [2]. Most blog posts do not have enough information to actually provide a solution to follow. The few examples available are simplistic and target mostly errors associated with the fields of a packet, like missing or corrupted start or end of packet or the cyclical redundancy check (CRC). The paper will discuss in Section 2 the design challenges for error injection in an Interlaken verification environment and in Section 3 it will propose a UVM error injection method using a two-phase slave sequence and will highlight its advantages over other error injection methods.

2. Design Challenge

This paper will present a solution for injecting errors for protocols that send packets striped over several serial lanes and which on top of the transmission of the data, as packets or frames, need to also inject idles or other protocol-specific physical layer elements for guaranteeing lane alignment, scrambler initialization, clock compensation or data integrity. The solution is customized for Interlaken, but it can be applied in other cases, too, as well as at each layer of a multi-layered protocol.

The typical examples deal mostly with injecting errors on packet data. This is done in the form of adding various flags to the drivers that are individually randomized or adding flags to the packet data itself. Both these methods prove cumbersome. Adding flags to the driver could end up unnecessarily convoluting the driver code and limits re-usability. A solution that decouples the error injection from the driver code and the packet class would enable those components to be more readily re-used on subsequent projects.

Furthermore, these examples do not show how to corrupt the elements that are not related to packets, and which have to be injected regularly by the driver into the traffic flow. These elements are not received from the packet sequencer. The method presented in Section 3 will show how corrupting everything that the driver needs to send can be achieved.

2.1 Interlaken Protocol

The Interlaken protocol [3] is a high-speed narrow protocol used to transfer packets over any number of serial links, called lanes. The packets can be transmitted in non-interleaved mode or in interleaved mode. In the non-interleaved mode, a packet is fully transmitted before the next packet is started. In interleaved mode, parts of a packet, called bursts, can be interleaved with parts of other packets. Packets are broken up into bursts of sizes determined by the BurstMax, BurstMin and BurstShort parameters, and by the protocol rules. The packets are identified by a channel ID which is carried by all bursts of a particular packet. The bursts are data words which are delineated by control words. The basic unit of data is an 8-byte word which is used for 64B/67B encoding.

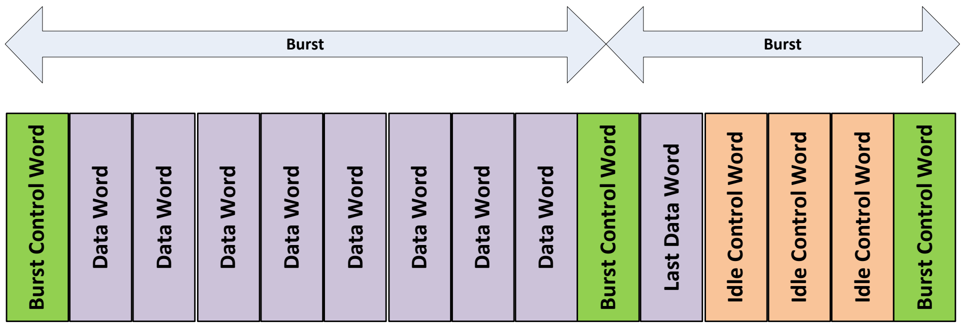

Figure 1 Interlaken bursts (*Copyright © 2006 Cortina Systems, Inc. and Cisco Systems, Inc.)*

The data and control words are striped sequentially over all the lanes of the interface. On each lane these words are encapsulated in metaframes which include the following framing words: a synchronization word, a scrambler-state word, skip words and a diagnostic word. They are used for lane alignment, scrambler initialization, clock compensation and data integrity. Metaframes are orthogonal to the packet transmission and are being injected into the traffic at a programmable regular interval. The Interlaken protocol will be used throughout the rest of the paper to illustrate the concepts of error injection. In particular we'll look to see why the proposed method is preferable for injecting errors, especially in control and framing words.

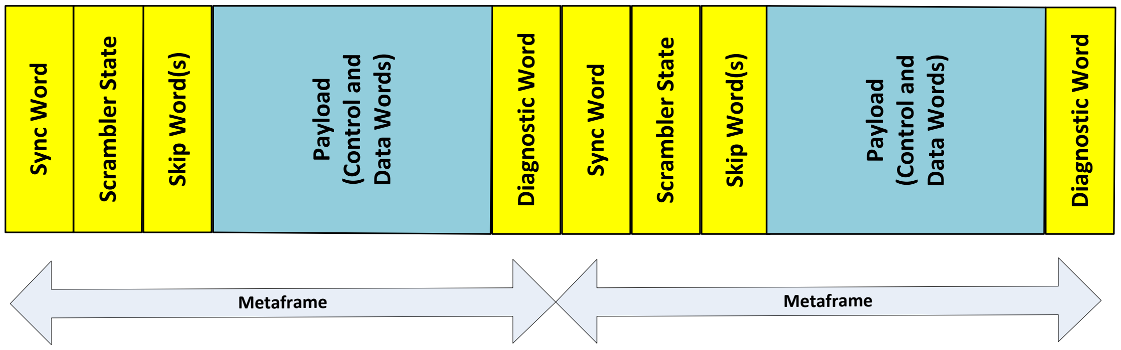

Figure 2 Interlaken metaframes (*Copyright © 2006 Cortina Systems, Inc. and Cisco Systems, Inc.)*

2.2 Interlaken Environment Architecture

Figure 3 shows a simplified diagram of the Interlaken UVM verification environment.

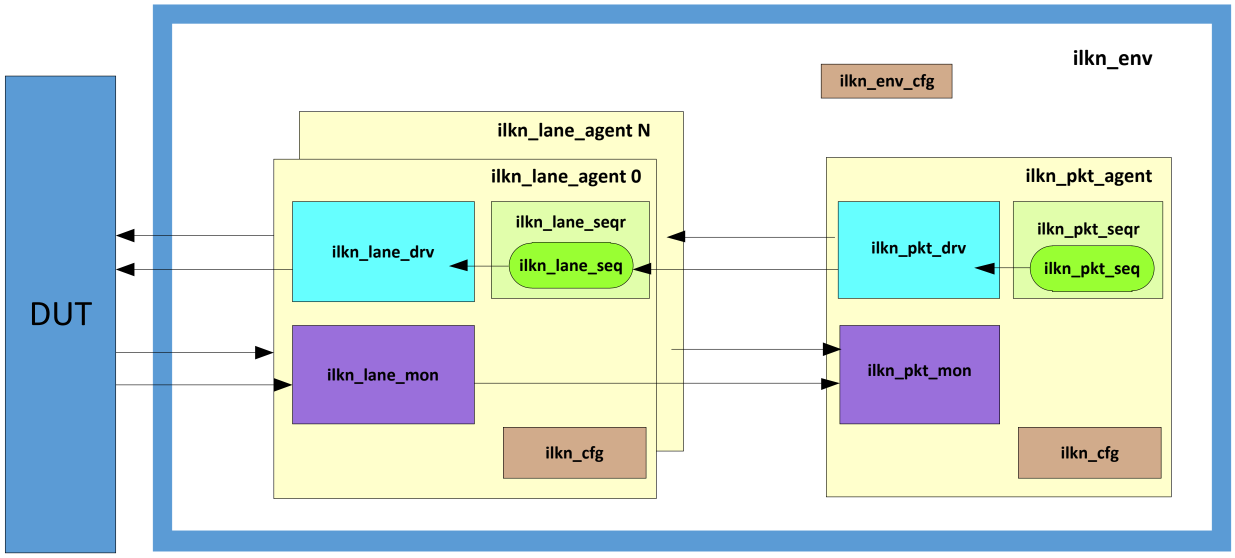

Figure 3 Interlaken verification environment

The Interlaken verification environment consists of a packet agent and several lane agents. As in a standard UVM agent, the packet driver receives the packets from its packet sequencer running a packet sequence that randomizes an Interlaken packet sequence item. The packet driver breaks up the packets into bursts, and each burst into data words. The packet driver has to add idle or burst control words to separate the different bursts. Also, at regular intervals it needs to generate the metaframes by adding the framing words into the normal data stream. In interleaved mode the packet driver services bursts from multiple packets. In this case the packet driver maintains a queue of the packets under processing. Some types of errors to consider at the packet driver level are the corruption of the data, burst, idle and framing words. Changing the intervals of the metaframes is also an error that needs to be injected.

The packet driver needs to stripe the stream of words (data, burst, idle, framing words) over the lower-level lane drivers which will ultimately drive the words onto the physical interfaces and into the DUT. Given this type of architecture, the packet driver has been implemented as a layering driver, as recommended by [4]. The packet driver has handles to the pass-through sequences running on the lower level lane sequencers. The packet driver pushes into each lower level lane sequence the words that the lower level lane driver needs to drive. The lane driver receives from its lane sequencer the word sent by the packet driver, and based on the type of the word, it might calculate CRC32, disparity, scrambler state etc., and update the word based on these calculations, before sending it out on the physical interface. At this level errors on CRC, disparity, scrambler state or clock compensation have to be considered, as well as random corruption of one or multiple bits of any type of word.

2.2.1 Main Interlaken Classes

Several classes will be referenced in the following sections. Snippets of code are presented below to better illustrate the concepts to the reader.

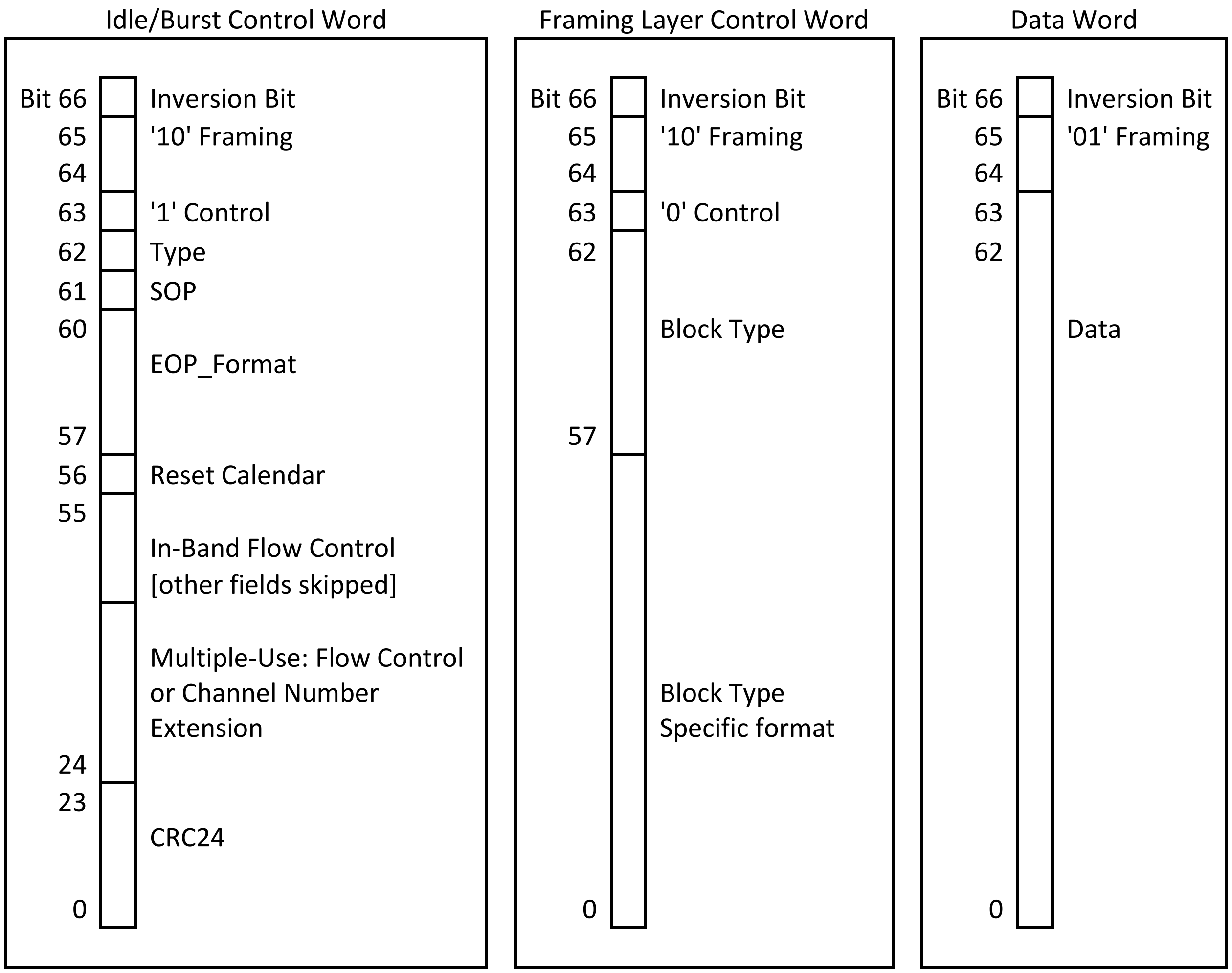

Figure 4 Interlaken word format (Copyright © 2006 Cortina Systems, Inc. and Cisco Systems, Inc.)

The Interlaken word is used to model the data, burst, idle and framing layer words as 8-byte units which are used for 64B/67B encoding, as shown in Figure 4. Get/set functions are associated with each field of the word.

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

  bit [66:0]          ilkn_word_data;
  ilkn_word_type_e    word_type;      // DATA_WORD, BURST_WORD, IDLE_WORD,
                                      // SYNC_WORD, SCRAM_WORD, SKIP_WORD,
                                      // DIAG_WORD, UNKNOWN_WORD
…
  extern virtual function void set_inversion_bit(bit inversion);
  extern virtual function bit  get_inversion_bit();
…
```

The Interlaken packet is the main class used for stimulus generation. The packet is randomized by the packet sequence and given to the packet driver to split into an array of data words that make up the bursts of the packet. The data words are processed by the packet driver which will also add, according to the protocol, burst or idle words to delineate the bursts, and framing control words when it is time to send the metaframes.

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

  rand int          pkt_length;
  rand int unsigned chan;
  rand byte         data_bytes[];
  rand int unsigned burstmax;
  rand int unsigned burstshort;
  rand int unsigned burstmin;

  ilkn_word_c  ilkn_word_data[$];   // data words making up packet
…
  constraint pkt_length_cons { pkt_length inside { MIN_LEN, MAX_LEN }; }
…
```

The Interlaken packet sequence randomizes the Interlaken packet by using the constraints from the packet class itself (for example, for pkt_length and chan), as well as the configuration programmed into the DUT (burstmax, burstshort, burstmin).

```
class ilkn_pkt_seq_c extends uvm_sequence #(ilkn_pkt_c);
  `uvm_object_utils(ilkn_pkt_seq_c)
  `uvm_declare_p_sequencer(ilkn_pkt_seqr_c)
…
  virtual task body();
…
    if( !ilkn_pkt.randomize() with {
                           burstmax   == p_sequencer.ilkn_cfg.burstmax;
                           burstmin   == p_sequencer.ilkn_cfg.burstmin;
                           burstshort == p_sequencer.ilkn_cfg.burstshort;
    …
                           })
```

2.3 Error Injection in Interlaken

A project can use several mechanisms for error injection. Different classes of errors might be easier to target by using a specific type of approach. There can be advantages and disadvantages with each mechanism used and it might be difficult to find a mechanism that could be used in all cases.

2.3.1 Error Injection on Packet Fields

Two possible methods that can be used to corrupt the fields of a packet data type could include extending the packet or extending the packet sequence.

As mentioned above, one method to implement error injection might be extending the packet data type and overwriting some of the constraints from the base class to allow illegal values on some of the packet fields, for example, for the packet length, to be outside of the minimum and maximum values supported by the DUT. In the corresponding test, use set_type_override to replace the packet data with the extended errored one. The packet driver will break up the packet into bursts and words without knowing that it is an illegal-size packet.

Below is an example of code in which the stimulus traffic is made up of packets of legal and illegal sizes, depending on how the pkt_length is randomized. The corresponding error test uses the extended packet data type.

```
class ilkn_pkt_extended_length_c extends ilkn_pkt_c;
  `uvm_object_utils(ilkn_pkt_extended_length_c)

  // overwrite constraint with same name from base class
  //   legal   values: MIN_LEN..MAX_LEN
  //   illegal values: 0..MIN_LEN or > MAX_LEN
  constraint pkt_length_cons {pkt_length inside {[0:MAX_LEN+100]};}

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

endclass // ilkn_pkt_extended_length_c


class test_ilkn_err_illegal_size_pkts_c extends test_base_c;
  `uvm_component_utils(test_ilkn_err_illegal_size_pkts_c)

  function new(string name, uvm_component parent=null);
    super.new(name, parent);
  endfunction // new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ilkn_pkt_c::type_id::set_type_override(
                         ilkn_pkt_extended_length_c::get_type());
  endfunction // build_phase

endclass // test_ilkn_err_illegal_size_pkts_c
```

Another method to inject packet field errors might be extending the packet sequence and providing different constraints for the randomization of the packet, for example, to use incorrect values for BurstMax, BurstMin and BurstShort parameters. The packet sequence has access to these values from its sequencer's configuration. In the corresponding test, use set_type_override to replace the packet sequence with the extended errored one. The packet driver will break up the packet into bursts based on the incorrect values of the BurstMax, BurstMin and BurstShort parameters and will generate the corresponding words without knowing that the bursts generated are of illegal sizes.

Below is an example of an extended packet sequence which uses an incorrect value of BurstMax for 10% of the packets, and the corresponding error test that uses the modified packet sequence.

```
class ilkn_pkt_burstmax_violation_seq_c extends ilkn_pkt_seq_c;
  `uvm_object_utils(ilkn_pkt_burstmax_violation_seq_c)
  `uvm_declare_p_sequencer(ilkn_pkt_seqr_c)

  rand   int burstmax_add_val;  // 90% correct, 10% incorrect
  constraint burstmax_add_val_cons {
    burstmax_add_val dist { 0 := 90, [1:10] :/ 10 };
  }

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

  virtual task body();
    ...
    if(!randomize())
      `uvm_error(get_name(), $sformatf("can't randomize seq"));

    if(!ilkn_pkt.randomize() with {
      burstmax   == p_sequencer.ilkn_cfg.burstmax + burstmax_add_val;
      burstmin   == p_sequencer.ilkn_cfg.burstmin;
      burstshort == p_sequencer.ilkn_cfg.burstshort;
    ...
                               })
    ...
endclass // ilkn_pkt_burstmax_violation_seq_c

class test_ilkn_err_burstmax_violation_c extends test_base_c;
  `uvm_component_utils(test_ilkn_err_burstmax_violation_c)

  function new(string name, uvm_component parent=null);
    super.new(name, parent);
  endfunction // new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ilkn_pkt_seq_c::type_id::set_type_override(
                    ilkn_pkt_burstmax_violation_seq_c::get_type());
  endfunction // build_phase

endclass // test_ilkn_err_burstmax_violation_c
```

2.3.2 Error Injection on Configurations

Similar to error injection on packet fields, to inject errors on configuration fields, the configuration class can be extended with a custom error config class. For proper operation, the DUT and driver have to be configured with the same values. In an error case the driver could be configured with a different value than the one programmed into the DUT. For example, the driver's config could be extended to use a different value for the metaframe length than the one programmed into the DUT and not allow synchronization to be achieved. In this type of error test, the driver's config will be replaced with the extended config.

2.3.3 Error Injection Associated with Flags inside the Packet Class

Online examples and [2] suggest adding error flags, as extra fields inside the packet data class, and have the randomized packet indicate to the packet driver what kind of errors to inject. This method can be used in some cases, for example, to corrupt the start-of-packet or end-of-packet where there is an easy mapping of one error flag to a clear type of corruption. If, on the other hand, a data word should be corrupted by flipping a bit in it, then how many more flags will the packet driver need to receive to indicate the index of the data word to corrupt and which bit in it? What if it would make sense to corrupt more than one data word in the packet, and more than one burst control word? The number of flags keeps on growing and won't be able to provide the proper control for error injection in the presented verification environment. Also, there is a pollution of the packet data class with a large number of flags to be used only for error injection. A solution to the pollution of the packet data class would be to hide all the error flags in an error object attached to the packet object. This approach still suffers from the problem of too many flags and not enough control for all the corruptions.

Shown below is an example of the packet class polluted with some error flags and a constraint to have them turned off.

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

  rand int          pkt_length;
  rand int unsigned chan;
  ...
  rand bit          corrupt_sop;
  rand bit          corrupt_eop;
  rand bit          corrupt_crc;
  ...
  constraint errors_off_cons {
    corrupt_sop == 0;
    corrupt_eop == 0;
    corrupt_crc == 0;
  }
```

2.3.4 Error Injection on Control, Data and Framing Words

As mentioned in the previous sections, the packet driver needs to split a packet into bursts and send the packet payload surrounded by burst or idle control words. It also needs to regularly inject framing control words. The question is how to corrupt all these extra words that are not part of the packet received by the driver from its sequencer, but rather generated by the driver itself. The previous methods were relying on the packet itself to instruct the packet driver what to corrupt. None of these approaches would work effectively in this case.

A callback is a solution that could be applied in this case. The callback could be called to potentially corrupt each word before its transmission. The callback code could randomize an error object which encapsulates the flags for the different types of errors, and based on their values (on or off), corrupt the word received from the driver. The next section proposes a callback-like solution mapped onto a UVM agent. Compared to a conventional callback, it will be shown how this method offers more versatility in terms of generating error tests by modifying an error sequence or an error data object.

3. Error Injection Mechanism Using a Two-Phase Slave Sequence

This section explains the proposed error injection mechanism using a two-phase slave sequence in the context of the Interlaken verification environment, provides an example of the handshake used and highlights some of the advantages of this approach.

3.1 Error Injection Interlaken Environment Architecture

In this implementation the error generation is orthogonal to the stimulus generation. The packet driver receives the packet to drive from its packet sequencer. After breaking up the packet into lower level words and before sending them to the lower level lane sequencers, or after deciding which protocol-specific words it needs to inject itself, the packet driver will call a task inside an error driver to which it has a handle. The error driver will be responsible for potentially corrupting the word received from the packet driver before returning it to the packet driver for transmission. A similar role is played by the lane error drivers connected to the lane drivers to implement the corruption at that level.

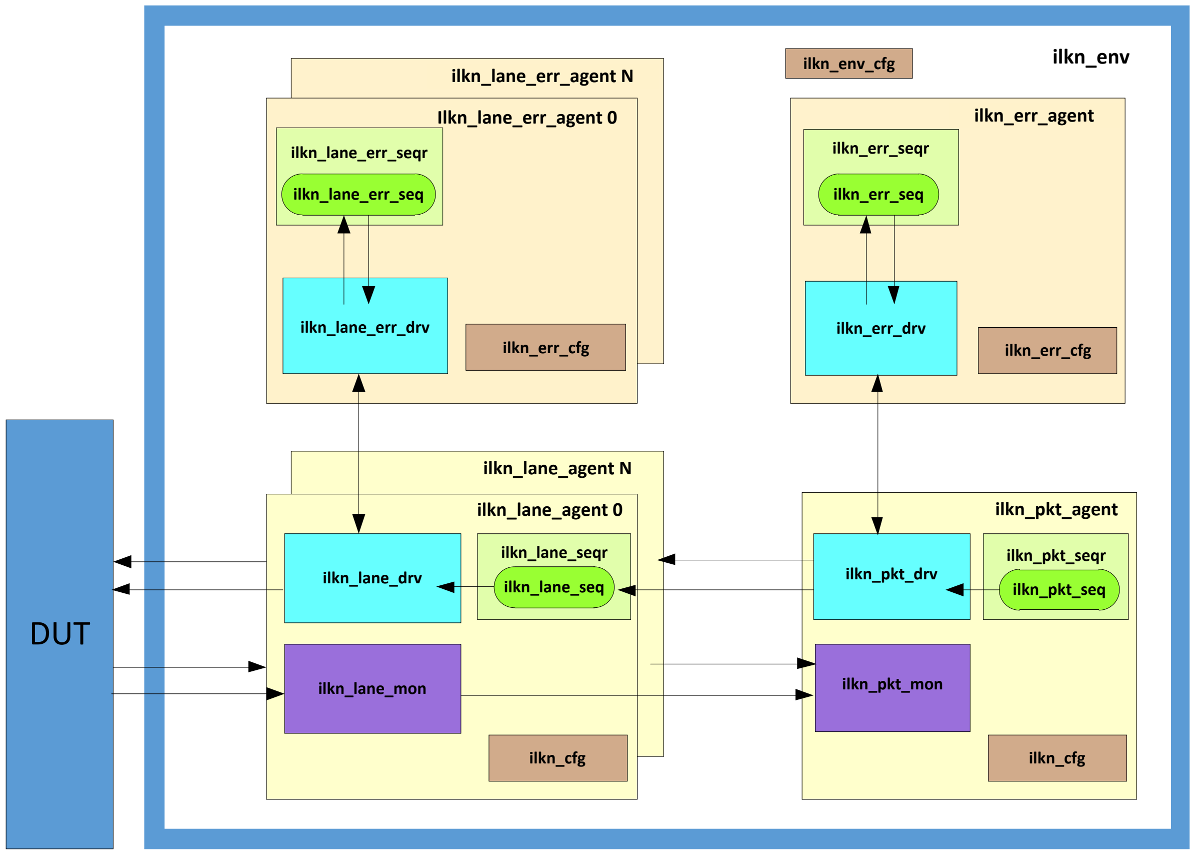

Figure 5 Interlaken verification environment with error agents

Figure 5 shows the environment used to implement the proposed solution. The error driver is connected to its own error sequencer which runs an error sequence. The error sequence randomizes an error data sequence item which has flags for all the errors that can be enabled. All the error flags are members of the error data class, and not of the packet class, which remains unpolluted. The error driver is not an active component. It reacts to requests initiated by the packet driver (through task calls), and that is why its sequencer runs a slave sequence forever. The mechanism used to communicate between the error driver and the error sequence models a slave device with two phases: in the first phase the sequence receives a request from the error driver on what category of errors to generate, and in the second phase it provides the response to the error driver, namely the randomized errors, based on the category specified in the request.

The role of a sequence is to randomize a sequence item for a driver and to provide it to the driver when requested. Unlike an active master sequence, a slave sequence needs to react to events. The slave sequence is not aware of the type of word the error driver is processing. In a one-phase approach, the sequence would provide to the error driver a randomized error data with some error flags set. This would not guarantee that the randomized error corruption could be applied on the current word being processed by the error driver.

To make sure that the randomized error would fit a category of errors (for example, control word errors, data word errors or framing errors), and that it could be applied on the word to be corrupted (for example, burst or idle word in the case of control words), a two-phase approach is being used. Figure 6 shows a side-by-side temporal representation of the error sequence and the error driver code. The handshake between the two is detailed below, with numbers matching code snippets with explanations. Please note that the error sequence is generic and can be used from different tasks inside the error driver. It is being called from the `ilkn_err_drv::corrupt_control_word` task in the example below, but it is also used in other tasks (not shown) to inject other categories of errors.

The error driver uses two sequence items for the two phases which can be seen as a request and a response. Each sequence item is an error data object with flags for all errors to inject.

```
ilkn_error_data_c req;
ilkn_error_data_c rsp;
```

1) In the first phase, the error sequence has a sequence item (req) ready for the error driver.

2) The error driver receives the first sequence item (req) from its sequencer.

```
seq_item_port.get_next_item(req);
```

3) The error driver does not look at any of the fields inside the req object, but rather uses the object to set up some fields that will be used by the error sequence during the second phase to guide the randomization of the error data. The driver sets the relevant fields for the category of errors to generate, and the type of word being processed. For example:

```
req.ilkn_err_group = CW_ERROR;  // category of errors (control word/…)
req.word_type      = word_type; // type of word (idle/burst/data/…)
```

4) The driver unblocks the sequence:

```
seq_item_port.item_done();
```

```
class ilkn_error_drv_c extends
uvm_driver #(ilkn_error_data_c);
  `uvm_component_utils(ilkn_error_drv_c)


task ilkn_error_drv_c::
      corrupt_control_word(
// or corrupt_data_word(
  ...
  ilkn_word_type_e word_type,
  ref ilkn_word_c  ilkn_word,
      bit[23:0]    crc24_error_mask
);

  ilkn_error_data_c req;
  ilkn_error_data_c rsp;



  if(ok_to_inject_errors())
    begin
      // first phase

2)    seq_item_port.get_next_item(req);

3)    req.ilkn_err_group = CW_ERROR;
// or req.ilkn_err_group = DW_ERROR;
      req.word_type      = word_type;

4)    seq_item_port.item_done();

      // second phase

6)    seq_item_port.get_next_item(rsp);









7)    // executes the corruption code

      case(rsp.error_type)
        // error flag: corruption code
        // error flag: corruption code
      endcase // case (rsp.error_type)

8)    seq_item_port.item_done();
      …
    end // if (ok_to_inject_errors())
endtask // corrupt_control_word
```

```
class ilkn_error_seq_c extends
   uvm_sequence #(ilkn_error_data_c);
  `uvm_object_utils(ilkn_error_seq_c)
`uvm_declare_p_sequencer(ilkn_error_seqr_c
)
  function new(string name="");
    super.new(name);
  endfunction // new







  virtual task body();
    ilkn_error_data_c  req;
    ilkn_error_data_c  rsp;

    […] // create req, rsp

    forever
      begin
       // first phase
1)     start_item(req);

       finish_item(req);





       // second phase
5)     start_item(rsp);
       rsp.copy(req);

       // randomization of errors
       // based on req fields)
       if( !rsp.randomize() with
         {    ilkn_err_group ==
          req.ilkn_err_group;
                           })
        `uvm_error(...);

        finish_item(rsp);








    end // forever begin
  endtask // body
endclass // ilkn_error_seq_c
```

Figure 6 Two-phase error sequence

5) In the second phase, the sequence uses the fields set by the error driver inside the req to control the randomization of the response that it will send back to the error driver, next time when the error driver asks for a new sequence item.

```
start_item(rsp);
rsp.copy(req);

// Sequence does randomization of errors inside rsp
// (based on req fields)

if( !rsp.randomize() with {ilkn_err_group == req.ilkn_err_group;}) …

finish_item(rsp);
```

6) The error driver requests this second object that has been randomized according to the fields set in the first phase.

```
seq_item_port.get_next_item(rsp); // rsp has flags for errors
                                  // to inject
```

7) The error driver looks at the rsp fields (the error flags) and decides if an error should be generated or not, and if so, which type. It corrupts the word received from the packet driver, as indicated by the error sequence rsp. The word (corrupted or not) is returned to the packet driver which will send it to the lower-level lane driver. The lower-level lane driver employs a similar mechanism to implement the error injection at its level.
8) The error driver is done processing the rsp with the error flags.

3.2 Application

This section will provide a more detailed view of how the proposed error injection mechanism works. First, suggestions on how to create and group error flags will be shown. Afterwards, the mechanism to communicate between the packet driver and the error driver will be explained step-by-step.

3.2.1 Error Flags and Groups of Errors

For each project it is important to decide the errors necessary to inject and if they will be controlled by a flag to enable them. The error flags can be thought of as knobs that turn on or off a type of error injection. All the error flags can be added to the same error data class which is an extension of an uvm_sequence_item.

A word can be corrupted by randomly flipping a bit in it, but that does not guarantee that each field of a word has been indeed corrupted and that the DUT has been stressed enough to handle all types of corrupted incoming words. A better approach is to associate an error flag with each field.

Figure 4 on page 6 represents some of the fields of the Interlaken words. Error flags should be created for each field inside the word if it is considered that the DUT would need to detect each such error type. The flags can be members of an enum called ilkn_error_type_t. Examples of flags are:

- For the inversion bit (bit[66]): a flag can be added to invert the correctly calculated inversion bit. The flag could be named ILKN_INVERSION_BIT_ERR.
- For the framing bits (bits[65:64]): two cases need to be considered, namely when the framing bits are corrupted and replaced with a legal value (but incorrect) or with an illegal value. For

example, a control or framing word with framing bits 2'b10 can be corrupted to use framing bits 2'b01 and will be decoded as a data word. If however, one of 2'b00 or 2'b11 illegal combinations are used, then the word won't be able to be decoded. The two flags could be named: ILKN_WRONG_FRAMING_BITS_ERR and ILKN_ILLEGAL_FRAMING_BITS_ERR.
- For the SOP bit (bit[61]): if it is 1, it indicates that the following burst is the start of a new packet; otherwise it is an intermediate or last burst of a packet. Corrupting this field could translate into two cases that will be handled by the DUT differently. They can be modelled by separate error flags, like: ILKN_ERR_MISSING_SOP_ERR and ILKN_ERR_BOGUS_SOP_ERR.

Once all the error flags have been decided, it is obvious that some of them could be applied to more than one type of word, while other ones are specific to only one. For example, the missing SOP has meaning only in a burst control word, while the inversion bit corruption can be applied to all types

```
typedef enum { ILKN_NO_ERR, ILKN_INVERSION_BIT_ERR,
               ILKN_WRONG_FRAMING_BITS_ERR, ILKN_ILLEGAL_FRAMING_BITS_ERR,
                ...} ilkn_err_type_e;

typedef enum { CW_ERROR, DW_ERROR, LANE_ERROR } ilkn_err_group_e;

class ilkn_error_data_c extends uvm_sequence_item_c;
  `uvm_object_utils(ilkn_error_data_c)

  rand ilkn_err_group_e ilkn_err_group;  // CW_ERROR, DW_ERROR, LANE_ERR
       ilkn_word_type_e word_type;       // set from inside the sequence
  rand ilkn_err_type_e  error_type;      // errors to be injected
  ...

  constraint cw_errors_cons;      // errors injected on control words
                                  // passing through the ilkn_pkt_drv
  constraint dw_errors_cons;      // errors injected on data words
                                  // passing through the ilkn_pkt_drv
  constraint lane_errors_cons;    // errors injected on all words passing
                                  // through the ilkn_lane_drv

constraint ilkn_error_data_c::cw_errors_cons
{
  if(ilkn_err_group == CW_ERROR)  // control word errors
  {
    if(word_type == BURST_WORD)
      {
        error_type inside { ILKN_NO_ERR,
                            ILKN_TYPE_ERR,
                            ILKN_BOGUS_SOP_ERR,
                            ILKN_MISSING_SOP_ERR ... };
      }
    else if(word_type == IDLE_WORD)
      {
        error_type inside { ILKN_NO_ERR,
                            ILKN_TYPE_ERR,
                            ILKN_BOGUS_EOP_ERR,
                            ILKN_MISSING_EOP_ERR ... };
      }
    …
  }
}
```

of words. In this application there can be three groups of errors considered: control word errors, data word errors and lane-level error (applied on all words). Inside the error data class, constraints for each group of errors allows the selection of the relevant flags based on word type.

3.2.2 Error Injection Handshake

The sequence of steps used for error injection is shown in Figure 7 and explained in more detail in the section below.

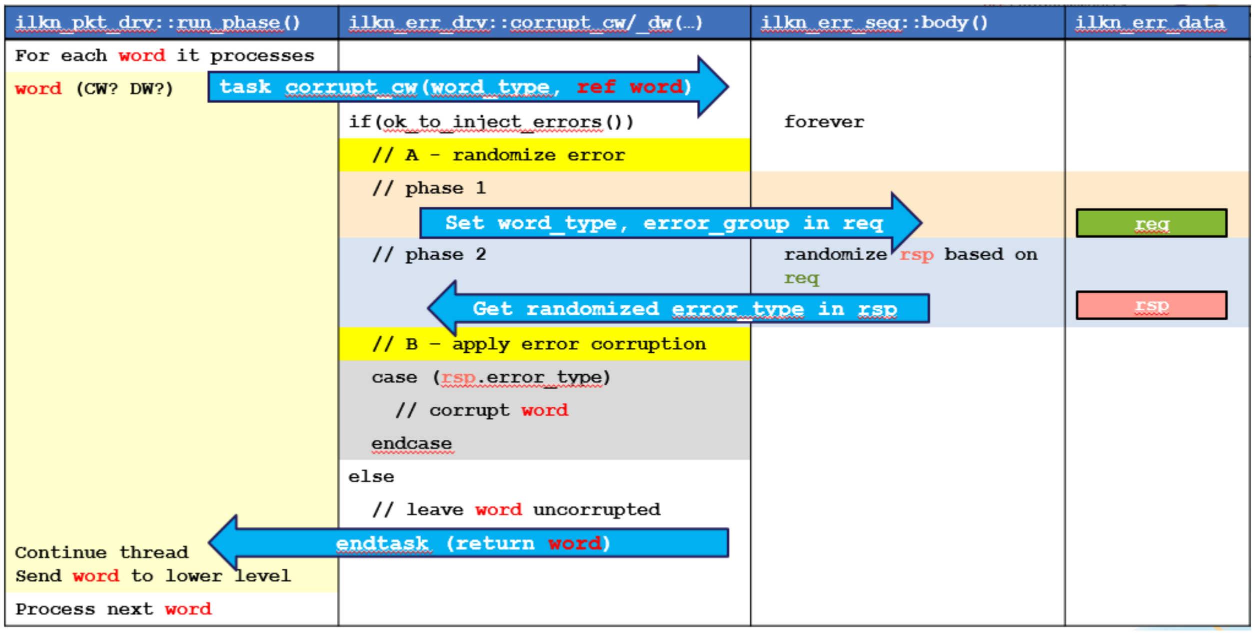
ilkn_pkt_drv::run_phase()
ilkn_err_drv::corrupt_cw/_dw(…)
ilkn_err_seq::body()
ilkn_err_data
For each word it processes
word (CW? DW?)
task corrupt_cw(word_type, ref word)
if(ok_to_inject_errors())
// A - randomize error
// phase 1
Set word_type, error_group in req
req
// phase 2
forever
randomize rsp based on req
Get randomized error_type in rsp
rsp
// B - apply error corruption
case (rsp.error_type)
// corrupt word
endcase
else
// leave word uncorrupted
endtask (return word)
Continue thread
Send word to lower level
Process next word

Figure 7 Error injection handshake

1. The packet driver is done processing a word, and before sending it to the lane driver, it will send it to the error driver to be potentially corrupted. Depending on the type of word, the packet driver will call a different task inside the error driver.
   In the case of a control word, the packet driver calls the following task inside the error driver: `ilkn_error_drv_c::corrupt_control_word()`, to potentially corrupt the control word

```
// task declaration
task ilkn_error_drv_c::corrupt_control_word(
  […]
  ilkn_word_type_e word_type,           // type of word to corrupt
  ilkn_word_type_e preceding_word_type, // type of previous word
  ref ilkn_word_c  ilkn_word,           // word to be corrupted
      bit[23:0]    crc24_error_mask     // corruption mask to return
                                        // to pkt_drv
);

// task call from ilkn_pkt_drv
ilkn_error_drv.corrupt_control_word(word_type, preceding_word_type,
ilkn_word, crc24_error_mask, …);
```

(for data words, it would call the `ilkn_error_drv_c::corrupt_data_word()` task). The task has several arguments for passing information between the packet driver and the error driver, as seen in the task declaration.
Conceptually, these tasks used to communicate between the packet driver and the error driver can be thought of as something similar to a callback call which allows the callback to modify the data sent to it, in this case, to corrupt or not the word. As seen above, some of the arguments are of type "ref", like the Interlaken word being processed.

2. The error driver executes the code inside the `corrupt_control_word()` task and checks to see if it is OK to try to inject errors based on some user-defined criteria. For example, in the case when error injection is not enabled at that time, or if the desired number of errors have already been injected, or if the time between consecutive errors has not been reached yet, then the task returns without corrupting the control word. The packet driver will continue its thread on the uncorrupted control word and continue with step 7.
3. If the error driver is allowed to inject errors, it communicates with the error sequence in a two-phase process, as explained earlier. The code is shown in Figure 6.
4. The error driver looks at the item received from the error sequence in the second phase (an ilkn_error_data object) and checks the error flags. Based on the randomizations inside the constraints of the ilkn_error_data, there is a chance that no error will be injected.
5. If an error flag is set, the error driver corrupts the control word accordingly. For some error cases, there are some other conditions that might have to be checked inside the error driver in order to decide if the randomized error can be applied. In the example below, creating the bogus SOP requires the control word to have been a burst control word and to have the SOP bit set to 0, i.e. to not have indicated start-of-packet already.

```
case(rsp.error_type)

  ILKN_BOGUS_SOP_ERR:
     begin
        if(ilkn_word.get_ilkn_word_type() == BURST_WORD &&
           ilkn_word.get_sop_bit() == 0)  // BURST with SOP=0
          begin
             ilkn_word.set_sop_bit(1'b1);  // change it to BURST with SOP=1
             […]
          end
     end // case: ILKN_BOGUS_SOP_ERR
 …
```

6. As the `corrupt_control_word()` task completes, the packet driver receives the corrupted word and continues its thread of execution.
7. Eventually the packet driver sends the word to the lower-level lane driver.
8. The packet driver moves on to the next word to process and goes back to step 1.

3.3 Advantages

The proposed implementation offers several advantages:

- It is a scalable solution and in a layered architecture it can be applied at each layer to inject errors on the data types processed at that layer.

- The error injection can be turned on or off in each error driver, by using an enable switch in its config. For example, only one of the lane drivers could inject errors, while the other ones could send normal traffic.
- The error injection flags don't have to pollute the main stimulus data types; they can be isolated into the error data sequence item classes. In fact, one error data class could include the code for all error flags, grouped per categories of errors.
- The corruption code, which can be quite complex, can be mainly implemented inside the error driver thus keeping the stimulus driver clean and simpler.
    - The error driver can be configured for how many errors to inject, when to start and stop error injection, and to keep track of the desired spacing between consecutive errors.
- The use of a task interface between the stimulus driver and the error driver has several advantages:
    - In cases when some code inside the stimulus driver needs to be bypassed to avoid overwriting the corruption introduced by the error driver, a flag inside the task call can indicate to skip processing of that particular code inside the stimulus driver (similar to a callback). For example, if disparity has been corrupted by the lane error driver, then the lane driver should not execute the code to calculate the correct disparity.
    - The task can return some parameters that could be used by the stimulus driver to perform some corruption, in the cases when the error driver could not perform the corruption itself. For example, if the CRC is calculated inside the stimulus driver over the data sent in a burst or a metaframe, the task can return a CRC mask which can be used to corrupt the CRC inside the stimulus driver at a later time.
    - There is no limit to the number of task calls that can be added where error injection inside the stimulus driver is required; each task can have a different interface and pass different arguments between the stimulus driver and the error driver.
- The two-phase communication between the error driver and the error sequence guarantees that the randomized error can be applied on the type of word under processing. This is due to the first phase specifying the error category and the word type to be corrupted. Without this information being communicated to the slave sequence it is possible that the error generated could have made no sense for the current word.
- Most error tests can easily be created by using an extended error data class in which the constraints have been modified to narrow down the types of errors enabled.
- More complicated error tests can be created by extending the two-phase slave sequence to perform the same operation as before (req and rsp for error driver), but using also some of its own random members in the randomization of the error class (for rsp) to create interesting scenarios. For example, the 64/67 word boundary lock is lost in Interlaken if 16 Sync words are corrupted within a 64 Sync word window. The extended slave sequence could keep track of how many Sync words have been corrupted within the afore mentioned window. If the test checks that the DUT is supposed to lose sync, then the error sequence could make sure that more than 16 sync words are corrupted. If the test checks that the DUT is supposed not to lose sync, the error sequence could make sure that in any 64 Sync window, at most 16 Sync words are corrupted, by returning in this case ILKN_NO_ERR error flag to the driver to avoid exceeding 16 Sync word corruptions. Errors that are not atomic in nature and have dependencies over several cycles can be handled, as shown above, by extending the two-phase slave sequence.

4. Future Work

The error injection mechanism through the error agent presented in this paper acts like a callback to the stimulus driver. The chosen method of hooking up the stimulus driver and the error driver has been a handle, which unlike a real callback, ties the two components together. A better approach would have been to move the error agent into a callback registered with the stimulus driver. The stimulus driver would have used callback calls instead of calling directly the `corrupt_*_word()` tasks inside the error driver. The use of the error agent provides the encapsulation of the error code and the familiar UVM approach for the error sequence randomization and communication with the error driver. The connection of the orthogonal error agent through a callback mechanism decouples it from the packet agent and adheres to object-oriented principles.

5. Conclusions

This paper has presented a possible mechanism for UVM error injection by connecting a stimulus driver to an error driver responsible for injecting errors based on running a two-phase slave sequence on its own sequencer. There are several advantages to using this solution. This strategy can be applied at any layer of a protocol. It avoids the pollution of the stimulus data and stimulus driver with flags and code related to error injection. Targeted error tests can be easily created. The solution presented can be easily applied to various verification environments.

6. References

[1] Accellera. “Universal Verification Methodology (UVM) 1.2 User’s Guide”. 2015. http://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[2] Montesano, J., & Litterick, M. “Sequence item Based Error Injection”. Paper presented at SNUG 2012. http://www.verilab.com/files/seq_Item_error_injection_snug_final_paper.pdf.

[3] Cortina Systems and Cisco Systems. “Interlaken Protocol Definition. Rev. 1.2”. Available at: www.interlakenalliance.com/Interlaken_Protocol_Definition_v1.2.pdf.

[4] Bergeron, J., Delguste, F., Knoeck, S., McMaster, S., Pratt, A., & Sharma A. “Beyond UVM: Creating Truly Reusable Protocol Layering.” Paper presented at DVCon 2013.