UVM Sequence Item Based Error Injection

Jeffrey Montesano
Mark Litterick

Verilab
Montreal, Canada

www.verilab.com

ABSTRACT

The proper testing of most digital designs requires that error conditions be stimulated to verify that the design either handles them in the expected fashion, or ignores them, but in all cases recovers gracefully. A self-checking constrained-random environment can be put to the test when injecting errors, because unlike the device-under-test (DUT) which can potentially ignore an error, the testbench is required to recognize it, potentially classify it, and determine an appropriate response from the design. In this paper we will present an error injection strategy using UVM that meets all of these requirements. The strategy encompasses both active and reactive components, with code examples provided to illustrate the implementation details.

Table of Contents

Table of Figures

1. Introduction

Introducing error injection into a constrained-random self-checking environment can be an involved and thought-intensive task. While usually left until the end of a project, it can raise issues that challenge decisions made at the very beginning. In this paper we will look at an error injection strategy using the Universal Verification Methodology (UVM) for making an environment able to handle error injection in a straightforward and effective manner. We will examine how to approach error injection in both active and reactive verification components, and propose a methodology that emphasizes consistency across both types of architectures.

To start, let's look at a scenario involving a generic communications protocol that begins with a preamble, and has payload and cyclic redundancy check (CRC) sent between a start-of-frame (SOF) and end-of-frame (EOF) delimiter.

Figure 1 Generic communications protocol frame

Some obvious verification requirements would be to verify that the design handles errors in the preamble SOF, CRC, EOF, and payload. Less obvious however is choosing exactly how to implement such errors. For example an EOF error could take on many forms, including an EOF symbol that arrives at the wrong time within a frame, one that occurs outside a frame, or a frame that has no EOF at all. Ideally all types of EOF errors would be implemented, but in practice choices must be made.

Having made those difficult choices, the next step is the implementation. And then comes the moment of truth, that first simulation provoking the error injection scenario, when all of the carefully-crafted self-checking testbench code attempts to digest a deliberate error without choking on it. Experience has shown that this is difficult to get right on the first try, and as such, an error-free log file should be an indication that something went wrong and the error was in fact not injected properly. Yes, doing error injection in a constrained-random self-checking environment is going to require a lot of debugging, but the benefits it brings are worth the effort.

2. When to Implement Error Injection

The topic of injecting errors into a design can be planned for from the very beginning of a verification project (code hooks, verification plan), though implementation is usually addressed later on, and rightly so. From a practical perspective this is because the task of stimulating non-error cases, and confirming that the design is behaving correctly under normal operating conditions, is already significant. The benefit of doing error injection later in a project is that confidence in the verification environment's non-error cases is high, so when errors are injected, testbench monitors are able to detect them accurately. This is not to say that error injection should be put off until the verification of normal operation is 100% complete – based on experience, error injection can and usually does find bugs in the normal operation flow of the DUT which are extreme-

ly difficult to detect with only legal stimulus. As such, a guideline is to start doing error injection after basic functionality has been verified, but before comprehensive testing has been completed.

3. Types of Error Injection

There are two main categories of errors that should be injected into a design. The first relates to protocol violations, where the protocol (e.g. Ethernet, DigRF, etc.) is deliberately violated by the testbench, as discussed in the introduction. This type of error should never occur in a well-functioning system, but the reality is that today's designs often need to interact with a wide range of external systems and devices, and so this type of error injection is essential. The second category refers to physical phenomena, such as wire latencies between the lanes of a parallel bus, or transmission coding errors over a noisy line. Depending on the real-world operating conditions of the design, these types of errors may or may not be likely to occur in a well-functioning system. As such, only those areas deemed relevant should be exercised using error injection.

4. Types of Error Detection

Unlike the DUT, which can potentially ignore an error, the testbench is required to recognize it, potentially classify it, and determine an appropriate response from the design. The monitors responsible for monitoring the testbench's stimulus must be able to detect and tolerate an error-injecting driver. Upon detecting an error, a monitor should flag it in a published transaction, which can then be used for scoreboarding and coverage.

If only one or two types of errors will be injected by the testbench, it could be possible for a monitor to distinguish between them and set specific flags (e.g. crc_error, latency_error). However, as the number of error injection scenarios increases it becomes increasingly difficult, if not impossible, for a monitor to correctly correlate the observed error with the actual error that was injected. For this reason, typically there cannot be more than one aggregated error flag in the monitor's published transaction. For example, let's return to the generic communications protocol from section 1, and take the case of a frame in which the preamble is one byte longer than it should be. Depending on how the monitor is coded, this will appear to be either an "SOF error" or a "preamble error". There is no way to classify it perfectly, and as a result, there is typically no point in trying. This even applies to errors that appear easy to classify such as CRC errors. Imagine in our generic communications protocol that an error injection scenario randomly flips a payload bit *after* CRC calculation was done, to simulate a physical transmission error – this would likely appear to be a "CRC error" to the testbench monitor, not a "transmission error", and therefore would be wrongly classified. Note that in this example, both the DUT and testbench are properly detecting a corrupted packet, and so it's irrelevant whether the CRC or the payload was the source of the problem.

Despite all this, there does exist a subset of detected error conditions that a monitor can perfectly correlate back to the error that was injected. Some examples of this type of error are a timeout, where the testbench is required to respond to the DUT within a given timeframe and intentionally does not, or a differential line, in which the testbench injects an error by making the two signals equal to each other for some period of time. In a DUT interface where these "classifiable"

errors are present, it is recommended to use separate error flags for them in the monitor, alongside an aggregate error flag for the others.

The aggregated error flag approach reduces the granularity of coverage that can be collected by the monitor. Instead of saying "a CRC error" was detected, it can only say "an error was detected". So while coverage is normally captured by passive testbench components, error injection is an example where stimulus coverage is appropriate in order to achieve greater granularity. In addition, a decision must be made as to what the monitor will do once it has detected an error. Will it attempt to continue decoding the frame, or should it abandon it and wait for the next one? The answers to these questions will depend on the type of error detected and application protocol. A monitor should also issue a message to the log file to indicate that error injection was detected. This message should be of severity UVM_WARNING, as it is not a testcase error for the testbench to inject an error, nor is it an ordinary operating scenario worthy of severity UVM_INFO.

5. Planning for Error Injection

There are different implementation strategies available for doing error injection. Some techniques involve the use of callbacks to modify an otherwise error-free sequence item or driver, or the UVM factory to replace an "error-free" driver with an "error-injecting" driver. The approach we will examine in this paper can be implemented in five steps:

1. The sequence item needs to modified to add error fields
2. The sequence library needs to be modified to add an error sequence
3. The driver needs to be modified to be able to inject the error
4. The transaction item needs to be modified to flag the error
5. The monitor needs to be modified to be able to detect the error

After compiling a list of errors to inject, it can be useful to create a table to track the implementation stage of each error so as to ensure that no steps are omitted.

6. A UVM Error Injection Strategy

In this section we will take an in-depth look at a sequence item based approach to error injection using UVM, by looking at two different architectures: the proactive master and the reactive slave. A proactive master is defined as a testbench component that injects whatever stimulus it wants, whenever it wants, while a reactive slave is a testbench component that only injects stimulus in response to requests from the DUT. Performing error injection in a proactive master is the simpler of the two and will be discussed first. Reactive slaves introduce additional complexities and therefore will be looked at second.

Proactive Master Error Injection

Figure 2 shows the standard UVM architecture for a proactive master testbench component. A user executes a sequence on the virtual sequencer (e.g. *send_data_frame_seq()*) from a testcase, which calls the appropriate sequence from the proactive master's sequence library, producing a sequence item. The sequence item is then passed from the proactive master's sequencer to its

driver, which interprets the item and drives signals to the DUT. The proactive master's monitor examines the activity on the DUT interface and publishes it for use by other testbench components.

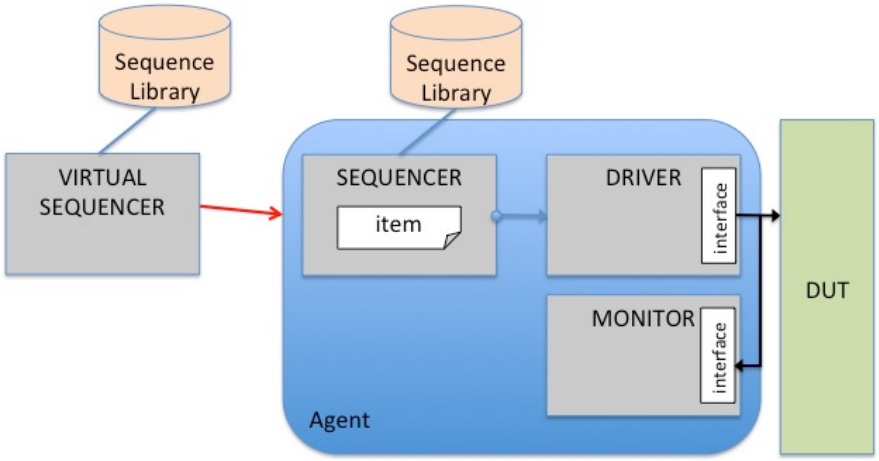

Figure 2 Standard UVM proactive master architecture

In our proposed approach, error injection control fields are added to sequence items (step 1 from section 5). So as to not affect existing non-error sequences (recall that error injection is usually done later on in the project), a constraint in the sequence item disables all error injection by default.

```
class protocol_seq_item extends uvm_seq_item;
  bit [7:0]  payload[];
  bit [15:0] crc;
  ...
  rand bit transmission_err;
  rand bit latency_err;
  rand bit crc_err;
  rand bit eof_err;
  ...
  constraint c_errors {
    transmission_err == 0;
    latency_err      == 0;
    crc_err          == 0;
    eof_err          == 0;
  }
  ...
endclass
```

Figure 3 Error fields in a sequence item

Having constrained all error fields to 0, there are two possible ways of setting an error flag: (1) after calling *randomize()*, overwrite the flag values (shown in Figure 4), or (2) disable the constraint before calling randomize using *constraint_mode(0)* and constrain the flag to '1' during randomization (not shown).

```
class error_seq extends uvm_sequence;
  bit [7:0] ss_payload[];
  ...
  bit ss_transmission_err;
  bit ss_latency_err;
  bit ss_crc_err;
  bit ss_eof_err;
  ...

  `uvm_create(s_item)
  if (!randomize(s_item)) `uvm_warning ("", "randomization failed")
  s_item.transmission_err = ss_transmission_err;
  s_item.latency_err      = ss_latency_err;
  s_item.crc_err          = ss_crc_err;
  s_item.eof_err          = ss_eof_err;
  `uvm_send(s_item)

endclass
```

Figure 4 Setting an error flag in an error sequence

Notice in Figure 4 that to overwrite a flag after randomization requires that we split the *`uvm_do* sequence into its three constituent steps (*uvm_create*, *randomize*, *uvm_send*) and call each one individually. This is step 2.

The driver has hooks for reacting to the error flags (step 3):

```
class protocol_driver extends uvm_driver;
  ...
  task run();
    forever begin
      seq_item_port.get(s_item);
      drive(s_item);
    end
  endtask
  ...
  task drive();
  ...
    if (s_item.transmission_error) begin
      // flip a payload bit
      ...
    end
  ...
  endtask
...
endclass
```

Figure 5 Driver injecting an error

Proactive Master Error Detection

In our proposed approach to error injection, a single aggregated error field is added to the transaction item published by the monitor, to accompany the other fields already being filled (step 4).

```
class protocol_trans extends uvm_transaction;
 bit [7:0]  payload[];
 bit [15:0] crc;
 ...
 bit error_detected;
 ...
}
endclass
```

Figure 6 Aggregate error field in monitor's transaction item

The reasons for creating a dedicated transaction item for monitoring, as opposed to making use of the drivers' sequence item, are the following: (1) the sequence item can get very long with all of the different error injection fields, and would result in difficult-to-read published transactions in the log files; (2) the sequence item's fields are mostly of type rand while the monitor's transaction item fields are not, because the monitor is a passive observer and will never randomize these fields; (3) the monitor's transaction item extends *uvm_transaction* instead of *uvm_sequence_item* because no sequences will ever be run by the monitor, so it is clearer to use the non-sequence-enabled base class *uvm_transaction*; (4) unlike the driver's sequence item which has one error flag per error type, the monitor only needs one aggregated error flag. Coverage on this dedicated monitor transaction item will be used to complement the coverage on the driver's sequence item, to ensure all different error types have been injected and detected.

```
class protocol_monitor extends uvm_monitor;
  analysis_port #(protocol_trans) ap;
  protocol_trans trans;
  ...
  task run();
    trans = new();
    bit eof_error, crc_error;
    ...
    if (eof_error | crc_error) `uvm_warning("", "detected error");

    trans.payload         = observed_payload;
    trans.crc             = observed_crc;
    trans.error_detected  = eof_error | crc_error;
    ap.write(trans)
    ...
  endtask

endclass
```

Figure 7 Monitor detecting errors and publishing aggregate flag

In Figure 7, we see an example of a monitor detecting error conditions, issuing warning messages to the log, populating the aggregated error field, and publishing the transaction to an analysis port. This corresponds to steps 5 from section 5.

Reactive Slave Error Injection

Figure 8 shows the standard UVM architecture for a reactive slave testbench component, responsible for generating slave responses to the master transactions coming from the DUT. It works as follows: the driver generates slave responses based on requests from the master, as observed by the monitor. The sequencer generates response sequence items based on the observed request and the current state of the testbench component. The monitor independently decodes all the signals on the DUT interface and publishes observed transactions. Notice that the user never calls anything from the testcase for this to work – it is all completely automated.

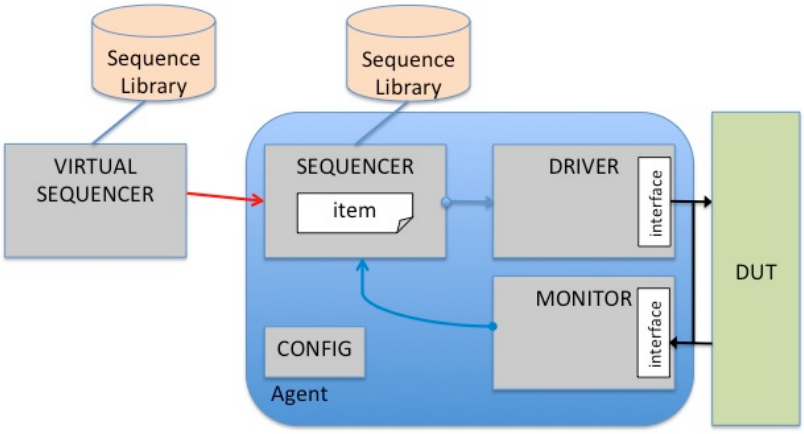

Figure 8 Standard UVM reactive slave architecture

One way to implement this type of architecture in UVM is to have a single "slave response" sequence running infinitely on the agent's sequencer, which blocks until it learns of a DUT request from the monitor. Injecting errors into this type of architecture can be a challenge, as it's a bit like trying to change a part in a running engine: the reactive slave component must be ready to respond at any time, yet the user needs to exert some control over what is returned. A callback mechanism would be the most appropriate way to handle this type of situation, but in the interest of consistency, we will take a similar approach to that taken with the proactive master.

Looking at Figure 8, we see the same components as we did in the proactive master, but this time we will need to bring both the configuration object and virtual sequencer into the discussion. It is assumed that all components have a reference to the configuration object, and the virtual sequencer has a reference to the agent's sequencer (this is explicitly shown with an arrow). The configuration object holds counters for each type of error to be injected (see Figure 9).

```
class agent_config extends uvm_object;
  ...
  int latency_err_cnt;
  int eof_err_cnt;
  ...

endclass
```

Figure 9 Reactive slave configuration

To increment these error counters, a sequence is added to the virtual sequencer:

```
class incr_err_seq extends uvm_sequence;
  ...
  bit incr_latency_err;
  bit incr_eof_err;
  ...

  task body();
    if (incr_latency_err)
      p_sequencer.cfg.latency_err_cnt++;
    if (incr_eof_err)
      p_sequencer.cfg.eof_err_cnt++;
    ...
  endtask
...
endclass
```

Figure 10 Virtual sequencer's error increment sequence

Next, an "error sequence" is added to the agent's sequence library, which causes the slave-response mechanism to use the configuration's error counters, and send the driver sequence items with error flags enabled. This is shown in Figure 11. Note that the sequence item being sent to the driver is identical to that shown for the proactive master (see Figure 3).

```
class agent_err_seq extends uvm_sequence;
 ...
  task body();
    forever begin
      // wait for monitor to detect request
      p_sequencer.peek_port.peek(trans)

      // create the sequence-item
      `uvm_create(s_item)

      // randomize the sequence-item
      if (!randomize(s_item))
       `uvm_warning ("","randomization failed")

      // set the error flags after randomizing
      if (p_sequencer.cfg.latency_err_cnt > 0) begin
        s_item.latency_err = 1;
        p_sequencer.cfg.latency_err_cnt--;
      end
      if (p_sequencer.cfg.eof_err_cnt > 0) begin
        s_item.eof_err = 1;
        p_sequencer.cfg.eof_err_cnt--;
      end
      if (...
      ...
      `uvm_send(s_item)
    end
  endtask
...
endclass
```

Figure 11 Reactive slave's error sequence

With all of this in place, all that's left is for the user to specify that the reactive slave component will use the error sequence as its default response sequence. This can be done using a factory override in a testcase's build phase, in those testcases in which error injection is desired.

```
class error_test extends uvm_test;
  ...
  task build();
    ...
    set_config_string("agent.sequencer","default_sequence","agent_err_seq");
    ...
  endtask
...
endclass
```

Figure 12 Setting reactive slave's default sequence

7. Conclusion

Injecting errors into a design not only puts both the verification environment and DUT to the test, it also opens up meaningful discussion as to how the design should behave under abnormal operating conditions. Verification environments that fail to explore the topic of error conditions are incomplete, and increase the possibility of a design bug going unnoticed. The UVM provides all the necessary tools for doing error injection, and in this paper we have shown one possible sequence item based approach for both proactive and reactive verification components.