Advanced UVM, Multi-Interface, Reactive Stimulus Techniques

Clifford E. Cummings[1], Stephen DOnofrio[1], Jeff Wilcox[1], Heath Chambers[2]

1 - Paradigm Works
2 - HMC Design Verification

Abstract - UVM reactive stimulus techniques allow sequences to receive feedback from a Design Under Test (DUT) to determine what stimulus should be sent next.

At DVCon 2020, the authors presented fundamental reactive stimulus techniques using a FIFO DUT. In the DVCon 2020 paper it was shown that a master sequence was able to react to FIFO status in order to send appropriate FIFO command transactions (FIFO Writes and Reads). The testbench consisted of an active agent that included a driver that sent request transactions into the FIFO and also sent response transactions from the driver, back through the sequencer and eventually back to the sequence. The single transaction class included both FIFO command and status fields.

This paper details advanced techniques for creating reactive stimulus. First, a **`uvm_tlm_analysis_fifo`** is added to the environment to capture transactions that are broadcast by the monitor. These broadcast transactions are passed to the original sequence to allow the sequence to react to the sampled outputs. Second, the same example is enhanced to run with config objects and a virtual sequencer.

I. Introduction

It is very common for a UVM test to execute a pre-defined set of sequences regardless of the status of the Design Under Test (DUT). An alternate approach is to execute stimulus that reacts to status from the DUT.

At DVCon 2020, the authors presented fundamental reactive stimulus techniques using UVM's built-in request-response paths. The technique used the same sequence-sequencer-driver to send a transaction and retrieve a response. This technique works well if all the required status can be retrieved over the same interface.

There are system-level environments that need to probe System-DUT internal signals to modify the sequences to be driven. This paper addresses how to get a response outside of the sequencer-driver path. A simple example is shown using a **`uvm_tlm_analysis_fifo`** in an environment to pass the output status information back to a sequence, which can then react to the status and modify the behavior of future transactions.

This paper will also show techniques that use the **`uvm_config_db`** to pass status back to the driving sequence.

The terms **`uvm_tlm_analysis_fifo`** and **`tlm_analysis_fifo`** will be used synonymously throughout this paper.

II. Reactive Stimulus Requirements

What are the requirements for reactive stimulus? A test will start a sequence on a sequencer and the driver will get the transactions, one at a time, from the sequencer and drive the stimulus to the DUT inputs. Now the sequence needs to sample (retrieve) the outputs that were generated by the stimulus and send them back to the sequence. The sequence examines the outputs and reactively determines what stimulus to drive next. So reactive stimulus requires DUT outputs to be sampled and sent back to the sequence.

In our DVCon 2020 paper, we used the response-transaction-path from the driver, back through a sequencer queue terminating at the sequence, which would examine the outputs to possibly modify the next transaction that would be sent to the DUT.

In this paper, we ignore the driver-sequencer-sequence response path and find alternate techniques to send the sampled outputs back to the sequence.

The first technique that we demonstrate is a **`uvm_tlm_analysis_fifo`** placed in the environment and allow the sequence to do blocking **`get`**-commands to retrieve the sampled outputs. This is described in the section.

III. RESPONSE TLM ANALYSIS FIFO TECHNIQUE

The first reactive stimulus example implements a simple, flat testbench as shown in Figure 1.

This example is intentionally simple to help the reader understand the basic techniques that are used to accomplish the goal of passing a response transaction back to a sequence without passing the response through the sequencer itself.

The key points to this technique:

(1) An extra **`uvm_tlm_analysis_fifo`** *will be connected to the monitor in the environment and the* **`tlm_analysis_fifo`** *handle will be stored in the* **`uvm_config_db`***.*

(2) The monitor will broadcast sampled transactions to the **`tlm_analysis_fifo`***.*

(3) The base sequence will declare a handle to the **`tlm_analysis_fifo`** *and retrieve the handle from the* **`uvm_config_db`***.*

(4) The sequence will do a **`tlm_analysis_fifo.get()`** *to retrieve the stored transaction and react to the sampled signals.*

All of these points are described in detail in this paper.

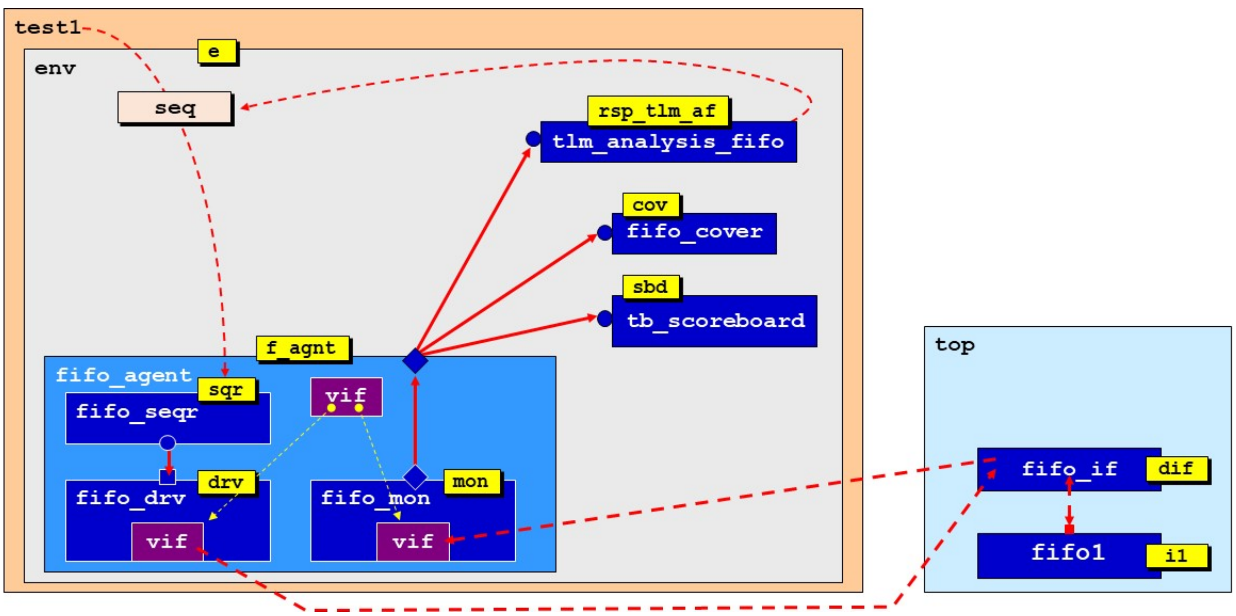

Figure 1 - Simple example - reactive stimulus passed back to the sequence through a tlm_analysis_fifo

IV. Environment Class Code Details

This technique uses a **uvm_tlm_analysis_fifo** (shortened to **tlm_analysis_fifo**) placed in the environment and connected to the analysis port of the **fifo_agent**. The environment class used for this simple example is shown in Figure 2, and highlights of the environment class are described below.

In the **env** class:

- A **tlm_analysis_fifo** with handle name **rsp_tlm_af** is declared on line 8.
- The **rsp_tlm_af** component is **new()**-constructed on line 20.
- The **rsp_tlm_af** is then stored in the **uvm_config_db** as a **uvm_tlm_analysis_fifo#(fifo_trans)** type on line 21.
- Finally the **rsp_tlm_af** is connected to the analysis port (**ap**) of the **fifo_agent** (**agnt**) on line 28. The **uvm_analysis_imp** port on the **tlm_analysis_fifo** is named **analysis_export**, so the agent connects its analysis port to the **rsp_tlm_af.analysis_export** on line 28.

```
 1 class env extends uvm_env;
 2   `uvm_component_utils(env)
 3
 4   fifo_agent    agnt;
 5   tb_scoreboard sbd;
 6   fifo_cover    cov;
 7
 8   uvm_tlm_analysis_fifo #(fifo_trans) rsp_tlm_af;
 9
10   function new (string name, uvm_component parent);
11     super.new(name, parent);
12   endfunction
13
14   function void build_phase(uvm_phase phase);
15     super.build_phase(phase);
16     agnt =     fifo_agent::type_id::create("agnt", this);
17     sbd  = tb_scoreboard::type_id::create("sbd",  this);
18     cov  =     fifo_cover::type_id::create("cov",  this);
19
20     rsp_tlm_af = new("rsp_tlm_af", this);
21     uvm_config_db#(uvm_tlm_analysis_fifo#(fifo_trans))::set(
                                         null, "", "rsp_tlm_af", rsp_tlm_af);
22   endfunction
23
24   function void connect_phase(uvm_phase phase);
25     super.connect_phase(phase);
26     agnt.ap.connect(sbd.axp);
27     agnt.ap.connect(cov.analysis_export);
28     agnt.ap.connect(rsp_tlm_af.analysis_export);
29   endfunction
30 endclass
```

Figure 2 - Response TLM Analysis FIFO - env.sv

In the block diagram of Figure 1, there are three components connected to the analysis port of the agent, a scoreboard, a coverage collector and a **tlm_analysis_fifo**, but it appears that the **tlm_analysis_fifo** is just a dangling component that does not add value to the environment. This is not true!

First recognize that the **tlm_analysis_fifo** will queue-up the transactions that were sampled and broadcast by the monitor. These queued transactions contain the sampled outputs from the DUT. It is these sampled signals that will be passed to the active sequence so that the sequence can examine the outputs and perhaps react (modify) the next driven transaction. Just how that happens is described in the **fifo_seq_base** class description, starting in Section VI.

V. Monitor class code details

The **fifo_mon** class is very common monitor code and is shown in Figure 3. The monitor is sampling inputs (shown on lines 28-32), synchronizing to the next posedge clk, using the clocking block notation **@vif.cb1** (shown on line 34), then resampling the asynchronous reset to determine if the reset has become active by the end of the cycle (shown on line 35), then sampling the outputs **#1step** before the **@vif.cb1** synchronizing clock (shown on lines 37-41). It is the sampled outputs that are required by the reactive sequence.

The sampled inputs and outputs are reassembled into the sampled transaction and broadcast to other components using the **ap.write(tr)** command shown on line 22.

This broadcast transaction will be captured by the scoreboard, the coverage collector and by the semi-dangling **uvm_tlm_analysis_fifo**. Each transaction broadcast by the monitor will be queued up into the semi-dangling **uvm_tlm_analysis_fifo**, so now the reactive sequence needs a mechanism to **get** (retrieve) the queued transactions. That will be accomplished in the **fifo_seq_base** class, which is described starting in Section VI.

```
 1 class fifo_mon extends uvm_monitor;
 2   `uvm_component_utils(fifo_mon)
 3
 4   virtual fifo_if vif;
 5
 6   uvm_analysis_port #(fifo_trans) ap;
 7
 8   function new (string name, uvm_component parent);
 9     super.new(name, parent);
10   endfunction
11
12   function void build_phase(uvm_phase phase);
13     super.build_phase(phase);
14     ap = new("ap", this);
15   endfunction
16
17   task run_phase(uvm_phase phase);
18     fifo_trans tr;
19     //-------------------------------------
20     forever begin
21       sample_dut(tr);
22       ap.write(tr);
23     end
24   endtask
25
26   task sample_dut (output fifo_trans tr);
27     fifo_trans t;
28     t = fifo_trans::type_id::create("t");
29     t.din   = vif.din;
30     t.write = vif.write;
31     t.read  = vif.read;
32     t.rst_n = vif.rst_n;
33
34     @vif.cb1;
35     if (!vif.rst_n) t.rst_n = '0;
36
37     t.full   = vif.cb1.full;
```

```
    t.af     = vif.cb1.af;
    t.empty  = vif.cb1.empty;
    t.ae     = vif.cb1.ae;
    t.dout   = vif.cb1.dout;

    tr      = t;
    `uvm_info("sample_dut", tr.convert2string(), UVM_FULL)
  endtask
endclass
```

Figure 3 - Response TLM Analysis FIFO - fifo_mon.sv

VI. FIFO_SEQ_BASE CLASS CODE DETAILS

The first part of the **fifo_seq_base** class code is described starting in this section, then the remaining command-task portions of the **fifo_seq_base** class code are described in Section VII.

Understanding the **fifo_seq_base** class is key to understanding this **uvm_tlm_analysis_fifo** reactive sequence technique. The detailed description of the **fifo_seq_base** class is shown below.

A. RANDOMIZE_FAIL message macro

Many of the **fifo_seq_base** command tasks randomize the transaction data fields and it is important that the randomization be tested to ensure that the constraints are met. Since this randomization is a common activity, we included a **RANDOMIZE_FAIL** macro definition to print a consistent "RANDOMIZE FAIL" message as shown on lines 1-3 of the **fifo_seq_base** class code in Figure 4.

Each call to **tr.randomize()** in the **reset()**, **write()**, **read()**, **write_read()** and **do_item()**, command tasks, call the common **RANDOMIZE_FAIL** macro.

```
1 `ifndef RANDOMIZE_FAIL
2   `define RANDOMIZE_FAIL
            `uvm_fatal("TR_S", "fifo_seq_base randomization failed")
3 `endif
```

Figure 4 - Common RANDOMIZE_FAIL macro - fifo_seq_base.sv

B. Reactive Sequence Base class declarations and pre_start() method

The Reactive Sequence Base Class (abbreviated *RSBC*) includes the declaration of a **uvm_tlm_analysis_fifo** that should point to the **tlm_analysis_fifo** in the testbench environment. It also includes synchronization capabilities so that the reactive sequence can drive stimulus and sample outputs to calculate the next stimulus to be driven. How these features are implemented and why they work are described below.

The *RSBC* of Figure 5, includes the following declarations:

- A **uvm_tlm_analysis_fifo** handle declaration on line 11.
- An event declared as **rsp_tlm_af_event** is listed on line 12.
- A **uvm_config_db#(uvm_tlm_analysis_fifo(fifo_trans))::get(…)** command to retrieve the **tlm_analysis_fifo** handle that was stored by the environment. Shown on lines 20-21.

The *RSBC* of Figure 5, also includes a **pre_start()** method, **forever** loop and the following synchronization code:

- A **pre_start()** method that executes a **fork-join_none forever** loop (an autonomously running process) shown on lines 18-28.
- The **forever** loop synchronizes to the output transaction by calling **rsp_tlm_af.get()** shown on line 24. This causes the **forever** loop to pause (block) until the monitor has queued up the next sampled output transaction. So this is the first of a 3-step synchronizing action with the reactive sequence.
- After getting a transaction, the *RSBC* triggers an event, the second of the synchronizing actions, by triggering the **->rsp_tlm_af_event** shown on line 25 in the **forever** loop.
- Each of the command tasks will drive a transaction and then wait for a response. The command tasks will wait for the triggered event by pausing (blocking) the command until the triggered event is observed using the **@rsp_tlm_af_event**. This is the third of the 3-step synchronizing actions used by the reactive sequence.

```
 5 class fifo_seq_base extends uvm_sequence #(fifo_trans);
 6   `uvm_object_utils(fifo_seq_base)
 7
 8   fifo_trans tr  = fifo_trans::type_id::create("tr");
 9   fifo_trans rsp;
10
11   uvm_tlm_analysis_fifo #(fifo_trans) rsp_tlm_af;
12   event  rsp_tlm_af_event;
13
14   function new (string name = "fifo_seq_base");
15     super.new(name);
16   endfunction
17
18   virtual task pre_start();
19     super.pre_start();
20     if (!uvm_config_db#(uvm_tlm_analysis_fifo#(fifo_trans))::get(
                                         null, "", "rsp_tlm_af", rsp_tlm_af))
21         `uvm_fatal(get_type_name(),
                               "The response uvm_tlm_analysis_fifo must be set!")
22     fork
23       forever begin
24         rsp_tlm_af.get(rsp);
25         ->rsp_tlm_af_event;
26       end
27     join_none
28   endtask
```

Figure 5 - Response TLM Analysis FIFO - fifo_seq_base.sv - Part #1

The **pre_start()** method in the **fifo_seq_base** is used to start the synchornization **forever** loop running before the **body()** task of all sequences extended from the **fifo_seq_base** class begin to execute.

pre_start() versus pre_body()

Why use the **pre_start()** method in the **fifo_seq_base** class? Why not use the **pre_body()** method?

Quoting from the UVM 1.2 Class Reference, Section 20.2:

Executing sequences via start:

> A sequence's **start** method has a *parent_sequence* argument that controls whether **pre_do**, **mid_do**, and **post_do** are called in the parent sequence. It also has a *call_pre_post* argument that controls whether its **pre_body** and **post_body** methods are called. In all cases, its **pre_start** and **post_start** methods are always called.

Executing sub-sequences via `uvm_do macros:

> A sequence can also be indirectly started as a child in the **body** of a parent sequence. The child sequence's **start** method is called indirectly by invoking any of the **`uvm_do** macros. In these cases, **start** is called with *call_pre_post* set to 0, preventing the started sequence's **pre_body** and **post_body** methods from being called. …

Since there are some possible situations when **pre_body()** is not executed, we chose to use the **pre_start()** method, which is always executed.

VII. FIFO_SEQ_BASE CLASS COMMAND TASKS

Many of the FIFO **fifo_seq_base** command tasks are very similar to, or the same as the command tasks that were used in the DVCon 2020 Reactive Sequence paper [1]. For this paper, the command tasks were moved to the **fifo_seq_base** class, which is then extended to create fifo sequences. The inclusion of these tasks in a base class greatly simplifies the development of the other fifo sequences.

The command descriptions will indicate if the specified tasks match the DVCon 2020 paper or if there are important differences. In general, the DVCon 2020 paper required **get_response(rsp)** commands to send a response transaction back to the reactive sequence through the sequencer. These have been replaced by transactions that are now passing through a **uvm_tlm_analysis_fifo** and corresponding 3-step synchronization actions as described in Section VI.

A. reset() task

The **reset()** task does randomization with **tr.rst_n** asserted as shown in Figure 6. In the DVCon 2020 paper, there was a **get_response(rsp)** command on line 35. That line has been replaced by a **do-while** loop (lines 35-36) that first waits for a **rsp_tlm_af_event** (line 35) and only exits the **do-while** loop when **rst_n** is low (line 36), which is the expected state of the **rst_n** signal during a reset operation.

```
30   virtual task reset (fifo_trans tr);
31     `uvm_info("reset", "executing", UVM_FULL)
32     start_item(tr);
33     if (!(tr.randomize() with {tr.rst_n=='0;})) `RANDOMIZE_FAIL
34     finish_item(tr);
35     do @rsp_tlm_af_event;
36     while (rsp.rst_n);
37     `uvm_info("reset", tr.convert2string(), UVM_FULL)
38   endtask
```

Figure 6 - reset() command task - fifo_seq_base.sv

B. FIFO write commands

The FIFO write commands are built from a common-base **write()** command (shown in Figure 7) and additional targeted write commands that test status signals and conditionally call the common-base **write()** command.

The common-base **write()** executes the **start_item(tr)** command, followed by transaction randomization with inline constraint that sets the **tr.write** bit, clears the **tr.read** bit and disables the **tr.rst_n** input. Then the **write()** command completes by calling the **finish_item(tr)** command (lines 41-43).

In the DVCon 2020 paper, there was a **get_response(rsp)** command on line 44. That line has been replaced by a **do-while** loop (lines 44-45) that first waits for a **rsp_tlm_af_event** (line 44) and only exits the **do-while** loop when **write** is high (line 45), which is the expected state of the **write** signal during any write operation.

```
40   virtual task write(fifo_trans tr);
41     start_item(tr);
42     if (!(tr.randomize() with {tr.write=='1; tr.read=='0;
                                  tr.rst_n=='1;})) `RANDOMIZE_FAIL
43     finish_item(tr);
44     do @rsp_tlm_af_event;
45     while (!rsp.write);
46     `uvm_info("FLAGS", sample_flags(rsp), UVM_HIGH)
47   endtask
```

Figure 7 - FIFO write() command task - fifo_seq_base.sv

Three additional reactive write commands call this common-base **write()** command:

write_until_full(fifo_trans1 tr) (shown in Figure 8) uses a **while (!rsp.full)** loop (line 51) to continue writing until **rsp.full** is detected in the response. This task also prints the message "**starting write_until_full**" with leading and trailing blank lines when the runtime **+UVM_VERBOSITY=HIGH** command switch is enabled. The **HIGH** verbosity message can be helpful during test and sequence development.

There is no difference in this command compared to the DVCon 2020 version of this command. The difference occurs in the common-base **write()** command called by this command.

```
49   virtual task write_until_full(fifo_trans tr);
50     `uvm_info("body", "\n\nstarting write_until_full\n", UVM_HIGH)
51     while (!rsp.full) write(tr);
52   endtask
```

Figure 8 - FIFO write_until_full() command task - fifo_seq_base.sv

write_until_AF(trans1 tr) (shown in Figure 9) uses a **while (!rsp.af)** (while not Almost-Full) loop (line 56) to continue writing until **rsp.af** is detected in the response. This task also prints the message "**starting write_until_AF**" with leading and trailing blank lines when the runtime **+UVM_VERBOSITY=HIGH** command switch is enabled.

There is no difference in this command compared to the DVCon 2020 version of this command. The difference occurs in the common-base **write()** command called by this command.

```
54  virtual task write_until_AF(fifo_trans tr);
55    `uvm_info("body", "\n\nstarting write_until_AF\n", UVM_HIGH)
56    while (!rsp.af) write(tr);
57  endtask
```

Figure 9 - FIFO write_until_AF() command task - fifo_seq_base.sv

`write_until_not_AE(trans1 tr)` (shown in Figure 10) uses a `while (rsp.ae)` (while Almost-Empty) loop (line 61) to continue writing while `rsp.ae` is still true in the response. This task also prints the message "`starting write_until_not_AE`" with leading and trailing blank lines when the runtime `+UVM_VERBOSITY=HIGH` command switch is enabled.

This command is used after resetting the FIFO to continue writing until the Almost Empty flag is cleared, which allows data values to partially fill the FIFO buffer right after releasing reset.

There is no difference in this command compared to the DVCon 2020 version of this command. The difference occurs in the common-base `write()` command called by this command.

```
59  virtual task write_until_not_AE(fifo_trans tr);
60    `uvm_info("body", "\n\nstarting write_until_not_AE\n", UVM_HIGH)
61    while (rsp.ae) write(tr);
62  endtask
```

Figure 10 - FIFO write_until_not_AE() command task - fifo_seq_base.sv

C. FIFO read commands

The FIFO read commands are built from a common-base `read()` command (shown in Figure 11) and additional targeted read commands that test status signals and conditionally call the common-base `read()` command.

The common-base `read()` executes the `start_item(tr)` command, followed by transaction randomization with inline constraint that clears the `tr.write` bit, sets the `tr.read` bit and disables the `tr.rst_n` input. Then the `read()` command completes by calling the `finish_item(tr)` command (lines 65-67).

In the DVCon 2020 paper, there was a `get_response(rsp)` command on line 68. That line has been replaced by a `do-while` loop (lines 68-69) that first waits for a `rsp_tlm_af_event` (line 68) and only exits the `do-while` loop when `read` is high (line 69), which is the expected state of the `read` signal during any read operation.

```
64  virtual task read(fifo_trans tr);
65    start_item(tr);
66    if (!(tr.randomize() with {tr.write=='0; tr.read=='1;
                                  tr.rst_n=='1;})) `RANDOMIZE_FAIL
67    finish_item(tr);
68    do @rsp_tlm_af_event;
69    while (!rsp.read);
70    `uvm_info("FLAGS", sample_flags(rsp), UVM_HIGH)
71  endtask
```

Figure 11 - FIFO read() command task - fifo_seq_base.sv

Two additional reactive read commands call this common-base `read()` command:

`read_until_empty(trans1 tr)` (shown in Figure 12) uses a `while (!rsp.empty)` loop (line 75) to continue reading until `rsp.empty` is detected in the response. This task also prints the message "`starting`

read_until_empty" with leading and trailing blank lines when the runtime **+UVM_VERBOSITY=HIGH** command switch is enabled.

There is no difference in this command compared to the DVCon 2020 version of this command. The difference occurs in the common-base **read()** command called by this command.

```
virtual task read_until_empty(fifo_trans tr);
  `uvm_info("body", "\n\nstarting read_until_empty\n", UVM_HIGH)
  while (!rsp.empty) read(tr);
endtask
```

Figure 12 - FIFO read_until_empty() command task - fifo_seq_base.sv

read_until_AE(trans1 tr) (shown in Figure 13) uses a **while (!rsp.ae)**, (while not Almost-Empty), loop (line 80) to continue reading until **rsp.ae** is detected in the response. This task also prints the message "**starting read_until_AE**" with leading and trailing blank lines when the runtime **+UVM_VERBOSITY=HIGH** command switch is enabled.

There is no difference in this command compared to the DVCon 2020 version of this command. The difference occurs in the common-base **read()** command called by this command.

```
78  virtual task read_until_AE(fifo_trans tr);
79    `uvm_info("body", "\n\nstarting read_until_AE\n", UVM_HIGH)
80    while (!rsp.ae) read(tr);
81  endtask
```

Figure 13 - FIFO read_until_AE() command task - fifo_seq_base.sv

D. FIFO write_read command

The FIFO **write_read()** command is a new command that performs simultaneous write and read operations. This command was not included in the DVCon 2020 paper.

The **write_read()** executes the **start_item(tr)** command, followed by transaction randomization with inline constraint that sets the **tr.write** bit, sets the **tr.read** bit and disables the **tr.rst_n** input. Then the **write_read()** command completes by calling the **finish_item(tr)** command (lines 84-86).

This command also includes a **do-while** loop (lines 87-88) that first waits for a **rsp_tlm_af_event** (line 87) and only exits the **do-while** loop when both **write** and **read** control signals are high (line 88).

```
83  virtual task write_read(fifo_trans tr);
84    start_item(tr);
85    if (!(tr.randomize() with {tr.write=='1; tr.read=='1;
                                  tr.rst_n=='1;})) `RANDOMIZE_FAIL
86    finish_item(tr);
87    do @rsp_tlm_af_event;
88    while (rsp.write && rsp.read);
89    `uvm_info("FLAGS", sample_flags(rsp), UVM_HIGH)
90  endtask
```

Figure 14 - write_read() command task - fifo_seq_base.sv

E. do_item() task

There is a general-purpose testing task called **do_item()**. The **do_item()** task does randomization with **tr.rst_n** disabled (line 95), as shown in Figure 15. The randomly generated **write** and **read** signals are concatenated and tested in a **case** statement that will execute idle (do nothing), **read()**, **write()** and

write_read() commands (lines 96-101). The **do_item()** task also takes a **cnt** input to do repeated, randomly generated commands. If the **cnt** input is not specified when **do_item()** is called, the default **cnt** value is set to 1 (line 92).

This command is a significant modification of the DVCon 2020 **do_item()** command.

```
92   virtual task do_item (fifo_trans tr, int cnt=1);
93     `uvm_info("do_item", $sformatf(
                 "\n\nstarting do_item (loop cnt=%0d)\n", cnt), UVM_HIGH)
94     repeat (cnt) begin
95       if (!(tr.randomize() with {tr.rst_n=='1;})) `RANDOMIZE_FAIL
96       case ({tr.write,tr.read})
97         2'b00: ;               // IDLE
98         2'b01: read(tr);       // FIFO READ
99         2'b10: write(tr);      // FIFO WRITE
100        2'b11: write_read(tr); // FIFO WRITE & READ
101      endcase
102    end
103    `uvm_info("do_item", tr.convert2string(), UVM_FULL)
104  endtask
```

Figure 15 - do_item() command task - fifo_seq_base.sv

F. sample_flags() method

The **read()**, **write()** and **write_read()** commands, which are also called by the other **write**-variation, **read**-variation, and the **do_item()** commands, call the **sample_flags()** method shown in Figure 16 to display the **full** / **af** / **ae** / **empty** flags when run-time simulation verbosity is increased to **UVM_HIGH.**

```
106  virtual function string sample_flags(fifo_trans rsp);
107    return($sformatf("full=%b / af=%b / ae=%b / empty=%b",
108                     rsp.full, rsp.af, rsp.ae, rsp.empty));
109  endfunction
```

Figure 16 - sample_flags() function - fifo_seq_base.sv

Printing these status flags is useful when debugging the FIFO design or testbench.

VIII. FIFO_SEQUENCE

For this paper, the command tasks were moved to the sequence base class and all fifo sequences, including this **fifo_sequence** class, extend the **fifo_seq_base** class. This greatly simplifies the development of fifo sequences. The **body()** task of the **fifo_sequence** is shown below in Figure 17 and the sequence executes the following stimulus actions:

Line 9 - The stimulus first resets the FIFO for two clock cycles.
Line 10 - Then completely fills the FIFO.
Line 11 - Later, after the FIFO is detected to be full, the stimulus reads the FIFO until it is empty.
Line 12 - The FIFO is written until it is past the Almost Empty mark.
Line 13 - Then 6 random read/write commands are issued.
Line 14 - The FIFO is then written until it is Almost Full.
Line 15 - Then 10 random read/write commands are issued.
Line 16 - The FIFO is written until full.
Line 17 - An attempt is made to randomly do 4-8 additional write commands, which should not change anything in the FIFO.
Line 18 - Read until the FIFO is Almost Empty.
Line 19 - Write until the FIFO is full.
Line 20 - Read until the FIFO is empty.
Line 21 - An attempt is made to randomly do 5-9 additional read commands, which should not change anything in the FIFO.
Line 22 - Write until the FIFO is Almost Full.
Line 23 - Do 100 random read/write commands. And finish this sequence.

```
 1 class fifo_sequence extends fifo_seq_base;
 2   `uvm_object_utils(fifo_sequence)
 3
 4   function new (string name = "fifo_sequence");
 5     super.new(name);
 6   endfunction
 7
 8   task body;
 9     repeat(2) reset(tr);
10     write_until_full(tr);
11     read_until_empty(tr);
12     write_until_not_AE(tr);
13     do_item(tr, 6);
14     write_until_AF(tr);
15     do_item(tr, 10);
16     write_until_full(tr);
17     repeat($urandom_range(4,8)) write(tr);
18     read_until_AE(tr);
19     write_until_full(tr);
20     read_until_empty(tr);
21     repeat($urandom_range(5,9)) read(tr);
22     write_until_AF(tr);
23     do_item(tr, 100);
24   endtask
25 endclass
```

Figure 17 - Response TLM Analysis FIFO - fifo_seq_base.sv

IX. Multi-Interface Reactive Stimulus

The simple reactive stimulus environment using a uvm_tlm_analysis_fifo can be easily extended to use multiple agents connected to a DUT.

The second agent could be an active or passive agent that uses a different transaction that samples DUT status signals, which might not be present in the first transaction.

Just as was shown in FIG1, the second agent could also be connected to a uvm_tlm_analysis_fifo and the common sequence base class could declare a handle to the second uvm_tlm_analysis_fifo and retrieve that handle from the **uvm_config_db**. Now the reactive sequence could get sampled transactions from different interface through a second uvm_tlm_analysis_fifo and use signals from the second transaction type to determine how to modify the stimulus being driven through the primary agent.

Using this technique, a sequence could query any number of different status signals from multiple interfaces, agents, and transaction types how to modify the stimulus being driven through the primary agent.

X. Summary & Conclusions

The simple reactive stimulus example used a **uvm_tlm_analysis_fifo** connected to the monitor in the environment. This **tlm_analysis_fifo** appeared to be a dangling component but it was capturing the broadcast transactions, with sampled outputs, from the monitor.

The *R*eactive *S*equence *B*ase *C*lass (*RSBC*) included:
– A **uvm_tlm_analysis_fifo** handle declaration.
– An event declared as **rsp_tlm_af_event**.
– A **uvm_config_db::get** command to get the **tlm_analysis_fifo** handle from the **env**.
– **pre_start()** method that executes a **fork-join_none forever** loop.
– **forever** loop that synchronizes to output transactions by calling **rsp_tlm_af.get()**.
– After getting a transaction, RSBC triggers the **->rsp_tlm_af_event**.

– Command tasks that are called by extended sequences.
– A **RANDOMIZE_FAIL** macro for common randomization error reporting.
– Command tasks drive stimulus.
– Command tasks wait for output transactions by waiting for the **@rsp_tlm_af_event**.
– Command tasks examine outputs to re-calculate the next input stimulus.

This technique can be extended to multiple interface with multiple agents and multiple transaction types, each of which can be examined by a reactive stimulus base sequence.

An additional section (Section XI) follows this Summary Section showing some of the code that we used in a more advanced version of the simple test that we have previously described in this paper. Sharing this code for the more advanced features might help engineers to look at different way to use the reactive sequence technique described in this paper.

References

[1] Clifford E. Cummings, Stephen Donofrio, Heath Chambers, "UVM Reactive Stimulus Techniques," DVCon 2020, San Jose, CA. Also available at www.sunburst-design.com/papers
[2] Universal Verification Methodology (UVM) 1.2 Class Reference - June 2014

XI. More Advanced Example using tlm_analysis_fifo

In the more advanced example shown in Figure 18, the environment includes an **env_cfg** and a virtual sequencer. The **fifo_agent** is now part of a UVM Verification Component (UVC), and the agent also has a **fifo_cfg**.

The agent has a **fifo_seq_main** base class and a **fifo_sequence**, which are controlled through the virtual sequencer using a virtual sequence base class and virtual sequence.

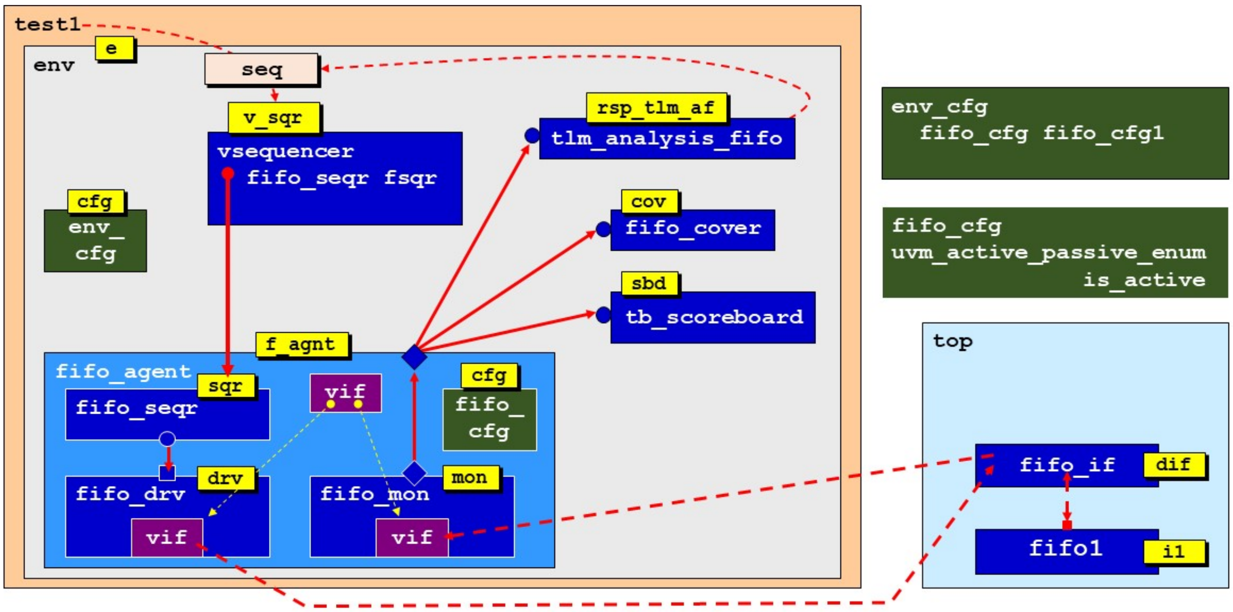

Figure 18 - Advanced example - reactive stimulus passed back to the sequence through a tlm_analysis_fifo

The most significant differences using this advanced version of the example are described below.

The advanced example environment now declares and uses a **vsequencer**, an **env_cfg** and a **fifo_cfg**.

```
class env extends uvm_env;
  `uvm_component_utils(env)

  fifo_agent                         f_agnt;
  tb_scoreboard                      sbd;
  fifo_cover                         cov;
  vsequencer                         v_sqr;

  env_cfg                            cfg;
  fifo_cfg                           f_cfg;

  uvm_tlm_analysis_fifo #(fifo_trans) rsp_tlm_af;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    f_agnt =    fifo_agent::type_id::create("f_agnt", this);
    sbd    = tb_scoreboard::type_id::create("sbd",    this);
```

```
    cov    =    fifo_cover::type_id::create("cov",    this);
    v_sqr  =    vsequencer::type_id::create("v_sqr",  this);

    cfg    =       env_cfg::type_id::create(  "cfg");

    uvm_config_db#(env_cfg)::set(this, "v_sqr",  "cfg", cfg);
    cfg.fifo_cfg1.is_active = UVM_ACTIVE;

    uvm_config_db#(fifo_cfg)::set(this, "f_agnt", "cfg", cfg.fifo_cfg1);

    rsp_tlm_af = new("rsp_tlm_af", this);
    uvm_config_db#(uvm_tlm_analysis_fifo#(fifo_trans))::set(
                                         null, "", "rsp_tlm_af", rsp_tlm_af);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    f_agnt.ap.connect(sbd.axp);
    f_agnt.ap.connect(cov.analysis_export);
    f_agnt.ap.connect(rsp_tlm_af.analysis_export);

    v_sqr.fsqr = f_agnt.sqr;
  endfunction
endclass
```

Figure 19 - Advanced Example - Environment

The advanced example **env_cfg** now declares a **fifo_cfg** handle and when the **env_cfg new()** constructor is called, it will also factory-create the **fifo_cfg**. This means that both configs are created back-to-back when the **env_cfg** is factory created.

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

   fifo_cfg fifo_cfg1;

   function new(string name="env_cfg");
      super.new(name);
      fifo_cfg1 = fifo_cfg::type_id::create("fifo_cfg1");
   endfunction
endclassclass
```

Figure 20 - Advanced Example - env_cfg

The advanced example **vsequencer** is a typical virtual sequencer that is little more than a wrapper for subsequencer handles. This **vsequencer** also has a handle that points back to the **env_cfg**, although it is not being used in this example.

```
class vsequencer extends uvm_sequencer;
   `uvm_component_utils(vsequencer)

   env_cfg   cfg;

   fifo_seqr fsqr;

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

```
   endfunction

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      if (!uvm_config_db#(env_cfg)::get(this, "", "cfg", cfg))
         `uvm_fatal("NOCFG",
     {"(env_cfg configuration object required for: ",get_full_name(),".cfg"});
   endfunction
endclass
```

Figure 21 - Advanced Example - vsequencer

The advanced example **fifo_agent** is pretty typical agent code, but it does include a **fifo_cfg** config object and retrieves its **is_active** flag from this **fifo_cfg**.

```
class fifo_agent extends uvm_agent;
  `uvm_component_utils(fifo_agent)

  virtual fifo_if vif;

  uvm_analysis_port #(fifo_trans) ap;
  fifo_drv  drv;
  fifo_mon  mon;
  fifo_seqr sqr;

  fifo_cfg  cfg;

  uvm_active_passive_enum is_active;

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

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    if (!uvm_config_db#(fifo_cfg)::get(this, "", "cfg", cfg))
        `uvm_fatal("NOCFG",
    {"fifo_cfg configuration object required for: ", get_full_name(),".cfg"});

    is_active = cfg.is_active;

    if (is_active == UVM_ACTIVE) begin
      drv =  fifo_drv::type_id::create("drv", this);
      sqr = fifo_seqr::type_id::create("sqr", this);
    end
    mon    =  fifo_mon::type_id::create("mon", this);
    ap     = new("ap", this);
    get_vif();
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if (is_active == UVM_ACTIVE) begin
      drv.seq_item_port.connect(sqr.seq_item_export);
      drv.vif = vif;
    end
```

```
    mon.ap.connect(ap);
    mon.vif = vif;
  endfunction

  function void get_vif;
    if(!uvm_config_db#(virtual fifo_if)::get(this,"","vif",vif))
        `uvm_fatal("NOVIF",{"virtual interface must be set for:",
                            get_full_name(),".vif"});
  endfunction
endclass
```

Figure 22 - Advanced Example - fifo_agent

The advanced example **fifo_cfg** class has storage for the **is_active** flag used by the **fifo_agent**.

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

  uvm_active_passive_enum is_active;

  function new(string name="fifo_cfg");
    super.new(name);
  endfunction
endclass
```

Figure 23 - Advanced Example - fifo_cfg

The advanced example vseq_base class executes very common actions:

- Calls the **`uvm_declare_p_sequencer** macro.
- Declares a **fifo_seqr** handle named **fsqr**.
- Copies the **p_sequencer.fsqr** handle to the local **fsqr** handle.

```
class vseq_base extends uvm_sequence;
  `uvm_object_utils(vseq_base)

  `uvm_declare_p_sequencer(vsequencer)

  fifo_seqr fsqr;

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

  virtual task body();
    `uvm_info("VSEQ_BASE DBG","vseq_base body() starting", UVM_FULL)
    fsqr = p_sequencer.fsqr;
  endtask
endclass
```

Figure 24 - Advanced Example - Virtual Sequence Base class vseq_base

The advanced example **vseq1** virtual sequence class is just setup to extend the vseq_base class and start the virtual sequence on the virtual sequencer.

```
class vseq1 extends vseq_base;
  `uvm_object_utils(vseq1)

```

```
  function new (string name = "vseq1");
    super.new(name);
  endfunction

  virtual task body();
    fifo_sequence1 vseq = fifo_sequence1::type_id::create("vseq");
    `uvm_info("VSEQ1 DBG","vseq1 body() starting", UVM_FULL)
    super.body();
    vseq.start(fsqr);
    `uvm_info("VSEQ1 DBG","vseq1 body() complete", UVM_FULL)
  endtask
endclass
```

Figure 25 - Advanced Example - Virtual Sequence vseq1

For the advanced example, all of the **`fifo_agent`** and subcomponents were put into a separate UVM Verification Component (UVC) directory and package. We were able to run this version of the example with the same results and the simple example.