2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Obscure face of UVM RAL: To tackle verification of error scenarios

Subhash Pai, Analog Devices, Bengaluru, India (*subhash.pai@analog.com*)

Lavanya Polineni, Analog Devices, Bengaluru, India (*lavanya.polineni@analog.com*)

***Abstract*—The register abstraction layer (RAL) in UVM library allows the users to write tests independent of protocol interface with high degree of reusability. While handling error scenarios in complex designs, verification engineers shy away from using RAL and tend to follow traditional approach of writing transaction based sequences and also writing their own register models to verify DUT. Transaction based sequences when used for register sequences pose code maintenance issues and have severe portability issues. Use of UVM register model for error scenario is further hindered by the fact that whenever DUT response is error status (UVM_NOT_OK), UVM RAL resets the register field values to zero which in turn prevents use of UVM register model for data comparison operation. There is lack of literature discussing handling of error status in UVM RAL. In this paper we discuss a novel method of using callback on UVM register model which allows verification engineers to use UVM RAL model for error conditions without need of modification of the UVM base code. Different ways a design can respond to erroneous register transactions are explored and it is explained how we can configure callback to model these design behaviors. The callback method presented in the paper is highly reusable and can be used across different designs with ease. The paper also describes methods and guidelines for coding error scenarios with RAL APIs. These guidelines will help the user to avoid common pitfalls for transaction sequences. The methods and guidelines discussed in the paper can broadly be applied to error scenarios in communication protocols.**

Keywords—Error Modeling; UVM RAL;Callback.

I. INTRODUCTION

Verification engineers typically use UVM RAL for standard register verification and use transaction based sequences whenever error conditions are to be passed as a part of the transaction. Transaction based approach requires careful editing of sequences for register field changes and also poses severe reusability issues when communication interface is changed while functionality remains largely the same. Use of configuration object to pass error information of the protocol is also quite common but it is difficult to use this approach for more complex scenarios. Sequence item and Configuration object create two different streams for the same data and synchronization between the two streams becomes complicated for complex scenarios like buffered or pipelined transactions. Use of UVM RAL for error scenarios is further hindered by the fact that for error condition where the status is UVM_NOT_OK, the register fields are reset to zero value by UVM register model. Development of custom register modeling for error transactions can lead to increased coding effort and requires thorough review.

II. UVM RAL APPROACH FOR ERROR SCENARIOS

A. Register Verification through UVM RAL

Typical register verification scenarios involve verification environment trying to access DUV (Design under Verification) registers through serial or parallel interface. Serial/Parallel protocol interface will provide the register address and data. Some protocols can also provide additional information like device address, data width, etc. While performing the register operations there can be errors like writing into read only register, accessing non-existent registers, protocol violations like framing error or performing register operations when the DUT is under error state.

B. Passing Error information through UVM RAL

Additional protocol data for the register operation can be passed on to the driver either through configuration object or can be transferred to the driver using the extension mechanism of UVM. Default value for extension argument is null i.e. no additional information is sent. This extension data is processed by the register adapter. It is possible to use the same extension mechanism to transfer the error information for the register operation. We recommend use of extension mechanism for transferring the error information due to following reasons:

1) Configuration objects are typically used for storing static and quasi-static data where as error transactions are dynamic in nature and they are often closely tied to the transaction. Mapping of the error in configuration to the transaction in sequence item can get complicated for complex protocol e.g if we have pipelined or buffered transactions. Also there can be confusion arising out of splitting the data into two streams viz. sequence item and configuration object.

2) In the UVM extension mechanism, error information is sent along with register data and hence they are very tightly coupled. All the information is captured only in the sequence item and hence can be easily used in pipelined and buffered transactions.

In the sequence where we perform register transactions, we can create a new object that can hold additional protocol and error information. Figure 1 shows example of extension argument to RAL API that can be used to transfer the extra information.

```
//sequence body
virtual task body();
  protocol_add_info_tr add_info=protocol_add_info_tr::type_id::create("add_info_tr");
  add_info.dev_addr=8'h0A;
  //trans_errors carries error information for the transaction
  add_info.trans_errors=FRAME_ERROR;
  //extra protocol information is sent through extension argument in RAL API
  dev_reg.write(status,data,.extension(add_info));
endtask:body
```

Figure 1. Passing error information through extension argument

Example for passing device address and insertion of frame error is shown in Figure 2. In the reg2bus function of the adapter we need to use get_item() function to extract the additional information and then this additional information is mapped to items in the protocol sequence item.

```
//reg2bus function in the adapter of the protocol
virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
  //extra protocol specific info including error type is stored in add_info
  protocol_add_info_tr add_info;
  //protocol bus transaction extended from uvm_sequence_item
  protocol_frame_tr bus_tr = protocol_frame_tr::type_id::create("bus_tr");
  uvm_reg_item item=get_item();
  //check if any protocol specific information is sent
    if(!$cast(add_info,item.extension)) begin
      if(!bus_tr.randomize()
        with { dev_addr == add_info.dev_addr;
        trans_errors == add_info.trans_errors;
        ra == rw.addr; rd == rw.data }) begin
          `uvm_fatal(get_type_name(),"Randomization failed!!")
        end
      end
  return bus_tr;
endfunction : reg2bus
```

Figure 2. Decoding of error transaction in reg2bus

Using a simple enum for the error variable would limit the users to a single kind of error. Verification engineers would often need to create protocol transaction with multiple errors and there can be different priority for different errors e.g. if the packet has CRC error as well address range error then the DUV needs to respond with the behavior for CRC error. We recommend users to create a single variable where each bit would represent one kind of error. Figure 3 shows an example.

```
parameter CRC_ERROR =         (1<<0);
parameter FRAME_ERROR =       (1<<1);
parameter ACCESS_ERROR =      (1<<2);
parameter MORE_DATA_ERROR =   (1<<3);
parameter LESS_DATA_ERROR =   (1<<4);
```

Figure 3. Representation of different errors in the bits

```
class protocol_frame_tr extends uvm_sequence_item;

  rand bit [7:0] addr;
  rand bit [7:0] data;
  //trans_errors stores errors in the transaction, each bit represents one kind of error
  //e.g to have frame and access error in a single frame
  //trans_errors=FRAME_ERROR+ACCESS_ERROR
  rand bit [5:0] trans_errors;
```

Figure 4. Sequence item containing the error variable

C. DUT response for different error scenarios

Design can respond to erroneous register writes/reads with three possibilities:

1) DUT completely ignores the erroneous transaction and no data/responses are sent. This can happen when there are multiple slaves on the same bus and responding to the transaction even with status set to error can lead to slave response corrupting response of correctly addressed slaves on the bus. e.g: CRC error, framing error.

2) DUT accepts and responds to the transaction like a normal transaction but the status is set to error .This can happen when the device is in error state and we would want to perform register access as a part of the debug.

3) DUT ignores the transaction but error status is sent.This can happen when the register transaction tries to write into read only register or tries to access restricted/unimplemented registers.

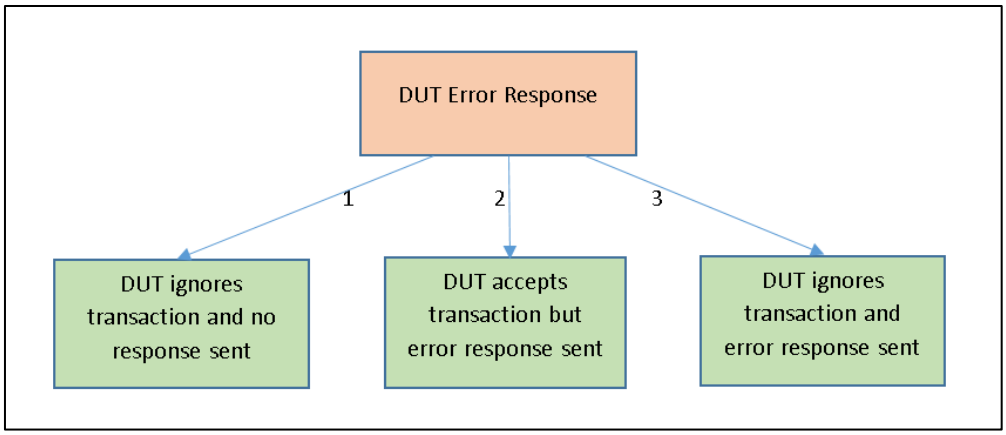

Figure 5. DUT Error Responses

D. UVM RAL Modeling for Error Transaction

Whenever DUT sends error status (UVM_NOT_OK), UVM RAL resets the mirrored value of the register fields to zero. This behavior is highly undesirable since we can no longer rely on the UVM register model to perform data comparison. The desired behavior is, for condition (1) and (3) register model should maintain the old value and for (2) the register model should be updated the DUT value. One solution to this problem would be to change the UVM library code but it is highly undesirable. This UVM issue can be resolved with UVM register callback approach. The solution essentially consists of defining and registering the callback and updating the callback based on transaction response.

D.1. Define and Register Callback:

Define a callback class based on uvm_reg_cbs. Callback would contain additional data on transaction status (erroneous or normal transaction) and also whether DUV would ignore or process the bus value.Callback method *post_predict* is often used to model quirky registers and it has the advantage that it works for both active and passive prediction[2]. Function *predict* of the *uvm_register_field* is called whenever there is a transaction on the register field and the mirrored value of the register field needs update. For error transactions i.e. status is UVM_NOT_OK the mirrored value for the register field gets reset to zero. The *predict* function also calls *post_predict* function of the callbacks that have been registered with *uvm_reg_field*. In error scenarios there can be cases where we would want to update the fields based on the bus value or maintain previous value of the field. Since for error transactions, *field_value* argument of the *post_predict* function gets reset to zero, callback object stores the bus value of the transaction in a variable called *bus_value*.Value of variables *update_reg*, *error_condition* and *bus_value* are updated in the register adapter.

```
class reg_error_handle_cbs extends uvm_reg_cbs;

  //update_reg controls if the bus value should affect the field value
  bit update_reg=1;
  //error_condition tells if it is an error transaction
  bit error_condition=0;
  //bus_value holds the register value observed on the bus
  uvm_reg_data_t bus_value;

  //post_predict is called by do_predict function of uvm_reg_field
  virtual function void post_predict(input uvm_reg_field  fld,
    input uvm_reg_data_t previous, inout uvm_reg_data_t value,
    input uvm_predict_e kind, input uvm_path_e path,
    input uvm_reg_map    map);

    uvm_reg_data_t field_val;
    if(error_condition ==1) begin
      if(update_reg == 0) begin
        value=previous;
      end else begin
        //bus_value holds the register value, derive field value using bit position and size
        field_val = (bus_value >> fld.get_lsb_pos()) & ((1 << fld.get_n_bits())-1);
        case (kind)
          UVM_PREDICT_WRITE: begin
            value=field_val;
          end
          UVM_PREDICT_READ: begin
            if (path == UVM_FRONTDOOR || path == UVM_PREDICT) begin
              string acc = fld.get_access(map);
              if (acc == "RC" || acc == "WRC" || acc == "WSRC" || acc == "W1SRC" || acc == "W0SRC") begin
                field_val = 0;  // (clear)
              end else if (acc == "RS" || acc == "WRS" || acc == "WCRS" || acc == "W1CRS" || acc == "W0CRS") begin
                field_val = ('b1 << fld.get_n_bits)-1; // all 1's (set)
              end else if (acc == "WO" || acc == "WOC" || acc == "WOS" || acc == "WO1")begin
                return;
              end
            end
            value=field_val;
          end
        endcase
      end
    end
  endfunction
endclass
```

Figure 6. Definition of Callback

Register this callback object for all uvm_reg_fields. We can instantiate the callback in the adapter or the verification environment class. Please note that the callback needs to registered with field and not the registers.

```
class protocol_reg_adapter extends uvm_reg_adapter;
  reg_error_handle_cbs error_handle_cb;

  function new(string name = "protocol_reg_adapter");
      super.new(name);
      error_handle_cb=reg_error_handle_cbs::type_id::create("error_handle_cb") ;
      uvm_reg_field_cb::add (null, error_handle_cb);
   endfunction
```

Figure 7. Instantiation and Registering of Callback

D.2. Update Callback based on Transaction Error

Protocol monitor sends the transaction oberved on the bus lines to the register adapter. Function *bus2reg* of the register adapter processes the bus data and whenever any error has been observed in the transaction, status is updated to UVM_NOT_OK and the *error_condition* bit is set. Value of update_reg would be assigned based on the kind of transaction error . Note that register adapter needs to provide callback with the transaction data received on the bus since UVM register model resets *field_val* to zero instead of bus value for error conditions.

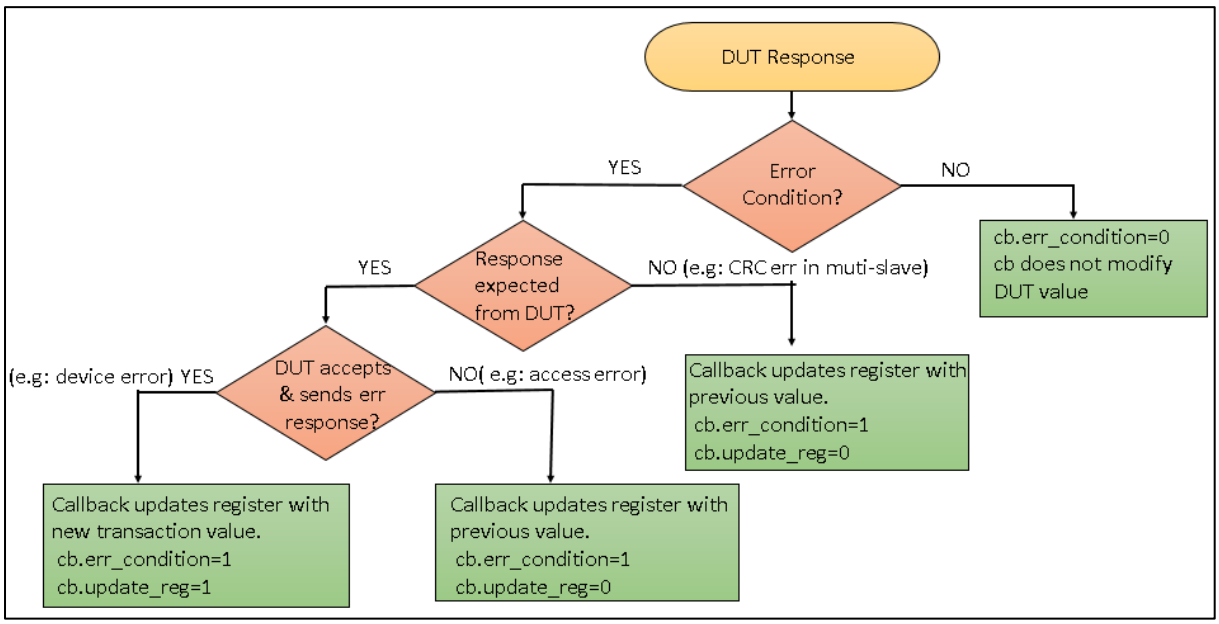

Figure 8. Handling of different DUT responses for error scenarios

```
virtual function void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
  protocol_frame_tr bus_tr;
  $cast(bus_tr, bus_item)
  //trans_errors would be non zero for error transactions
  if(bus_tr.trans_errors != 0) begin
    //transaction has error, setting the status to UVM not ok
    rw.status=UVM_NOT_OK;
    if((bus_tr.trans_errors & CRC_ERROR ) > 0) begin
      //DUT does not send any response to the transaction, register value is not modified
      error_handle_cb.error_condition=1; error_handle_cb.update_reg=0;
    end else if((bus_tr.trans_errors & DEVICE_ERROR) > 0) begin
      //DUT responds to the transaction, it is just that device is in error state
      error_handle_cb.error_condition=1;error_handle_cb.update_reg=1;
      error_handle_cb.bus_value=bus_tr.data;
    end else if((bus_tr.trans_errors & ACCESS_ERROR) > 0) begin
      //DUT responds to the error transaction with error status, register is not modified
      error_handle_cb.error_condition=1;error_handle_cb.update_reg=0;
    end
  end
endfunction : bus2reg
```

Figure 9. Handling DUT Error response in bus2reg

III. CONCLUSIONS

With callback method and UVM extension mechanism we could easily use UVM RAL model for error transactions. This greatly reduces the verification effort required to verify error scenarios for all the interfaces used in the design. Further this code is generic in nature and can be reused for any communication protocol in future projects. We recommend verification engineers to use UVM RAL even for register error scenarios for improved reusability and ease of code maintenance.

REFERENCES

[1] UVM User guide, Accelera

[2] M. Litterick, M. Harnisch, “Advanced UVM Register Modeling – There’s More Than One Way to Skin a Reg,” DVCon 2014.