Customization of RAL Adapters and Predictors in UVM 1.2

Steven K. Sherman

AMD
Boxborough, Massachusetts, USA

www.amd.com

ABSTRACT

This paper reviews lessons learned in customizing RAL adapters and predictors. This includes a brief overview of provided examples and options per UVM reference and user guide. It continues with motives, strategies and customizations used in a real verification environment.

Table of Contents

Table of Figures

1. Introduction

UVM RAL [1] includes a basic adapter (uvm_reg_adapter.svh) and predictor (uvm_reg_predictor.svh). The "UVM 1.1 User's Guide" [2] (version 1.1 as of this writing) includes basic descriptions of the adapter and predictor functions and use, but little about customization though the source code is clearly written with customization in mind. General guidelines for adapter and predictor development are available [3]. This paper begins with the basic UVM adapter and predictor, discusses some general challenges of a real verification environment and demonstrates how the classes can be successfully extended and customized to meet those requirements. This customization includes but goes beyond the relatively minor customizations that may be needed to function in a multi-interface, multi-mode environment [4]. Customization of adapters and predictors are discussed based upon lessons learned in a real verification environment.

2. Why customize?

Why not use the default adapter and predictor provided with UVM or use those provided as part of an imported UVC? After all, they are proven to work in their own environments. You can use these without customization if they work "as is" and you have complete control over all aspects of an environment. This paper assumes you are seldom so fortunate.

The need for customization can be quickly apparent in a real verification environment. In addition to design requirements and restrictions for your environment or design, imported IP may require changes in your verification environment that affect other components, including adapters and predictors. It might not be feasible for you to customize an imported UVC, so you may need to adapt, perhaps with changes to your own components. Otherwise, you may need to customize imported adapters and predictors to meet interface, security, address map or other requirements for your environment.

The default predictor and adapter provided with UVM are intended, by design, to be customized. Both adapter and predictor classes may be extended and virtual functions and tasks overridden. Adapter customization additionally includes option selection and supports use of a special extension object to pass in additional information.

In our verification environment, it is critical that mirrored register values be kept in synchronization with DUT register values with only a few, well-documented exceptions. This is critical because tests, scoreboards and other infrastructure reference the mirrored values to assist with stimulus, checking and coverage. This is a much desired feature over having to access DUT registers directly. The issue becomes especially critical when using a common infrastructure to support a multitude of testbench variants. Effective customization of adapters and predictors is key to satisfying this fundamental requirement.

3. Quick Overview of RAL Adapters and Predictors

As depicted in Figure 1, an adapter serves as a container of functions used to link the register model with the bus. A predictor watches monitor output which it uses to check and update the register model. A scoreboard also watches monitor output. In our environment, the scoreboard typically consults the register model as it checks DUT function.

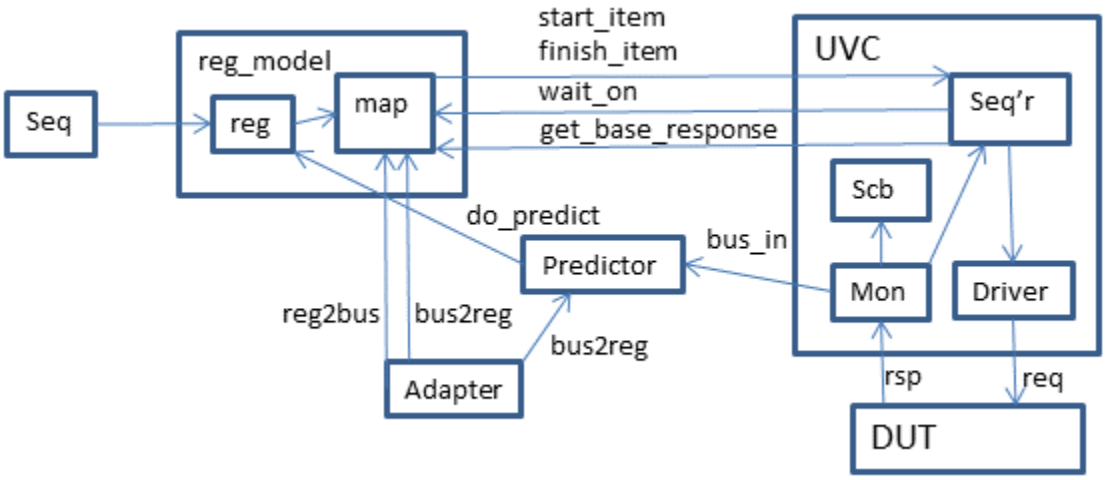

Figure 1 - System with Adapter and Predictor

Use of predictors has been described as "optional" [3]. In our environment, use of a predictor is usually not optional because it is exported [4] as a component that may function passively. When functioning passively, the sequencer is not driven by a sequence, often requiring the predictor alone to perform all register model updates in addition to register value checks.

If default behavior is sufficient, then predictor customization may not be required. This paper will assume that default behavior is generally insufficient in actual usage but very beneficial as guidance for customization. A predictor may also send resulting items (uvm_reg_item) out its own uvm_analysis_port (reg_ap), but this is not explored here.

Technically, use of an adapter is also optional. The map can still function with the register model and UVC. However, the sequence, sequencer and predictor may be severely limited and restricted to what is supported by the base "uvm_reg_item". If your sequence and predictor support items of default type "uvm_tlm_gp", then you may not need to customize the adapter. Otherwise, customization of the adapter is likely required as in our environment.

In this discussion, use and customization of predictors is usually required for maintaining synchronization between the register model and the DUT in both active and passive modes. A customized adapter is also usually required.

Adapters

For adapters, the customization generally includes modification of the "reg2bus" and "bus2reg" functions. It also often includes use of an extension object to pass in additional information.

reg2bus function

As a bridge between the register model and the bus, "reg2bus" should return a sequence object ready to be sent to the bus driver's sequencer. You may well discard all of the default "reg2bus" function and replace it with your own. However, the default function should be used as a guide.

bus2reg function

As a bridge between the bus and the register model, "bus2reg" should return a sequence object ready to be sent to the register model. Again, you may well discard all of the default "bus2reg" function and replace it with your own. However, the default function should be used as a guide.

extension object

Use of an extension object is a suggested means for communicating with the adapter. Use of the extension is optional, depending upon whether or not the "uvm_reg_bus_op" structure passed in has sufficient information. The "uvm_reg_bus_op" structure is defined in "uvm_reg_item.svh".

Predictors

For predictors, the customization mainly includes modification of the "write" function.

write function

The name of the "write" function can be a little confusing. A primary purpose for the predictor is to receive responses from a monitor and use these to check and update mirrored values. Confusion may arise when one considers that the response processed may be the result of a read (which may result in a check against the mirrored value) or a write. Both read and write responses may result in update of the mirrored value. Perhaps it is in that sense that it is a "write." Note also that this is not a function that is called by a sequence. Rather, it is a function connected via "bus_in" from a bus monitor.

4. Adapter Customization

Customization generally starts with an extension of adapter class from either the UVM kit sources or imported UVC. It usually continues with selection and modification via a number of options as described here.

provides_responses

This option is stored in the adapter and accessed by the map for the register. It is deasserted by default.

If asserted, provides_responses indicates that the bus driver (the target of generated requests) provides separate response items for requests. The map for the register is sensitive to this option for both read and write requests. The map then sets up and schedules a task to receive that *response* from the sequence. This response is then passed to the adapter "bus2reg" function.

If deasserted, the map passes the *request* to the adapter "bus2reg" function. This is used to "fill in" the information about the transaction for the register model based on the request itself.

The hazards with this option involve mismatches with the operation of the bus driver. For example, if the bus driver does not provide responses (via "get_base_response" call from the map to the UVC sequencer) and this option is set, then the simulation could hang as it waits for a response that will never come.

If the bus driver provides responses but this option is not set, then the result derived for the register model may be inaccurate, perhaps ignoring critical information about the transaction provided by the bus driver. This may be the more difficult hazard as this malfunction may go undetected much of the time.

In our environment, we explicitly ensure that provides_responses is cleared when a predictor is present and only set when the bus driver provides responses to avoid the above issues.

extension object

Many of the RAL access functions and tasks accept an optional handle to an "extension" object extended from type uvm_obj. This object can be passed via a register task call as shown in the adapter code snippet of Figure 2.

```
// Function: reg2bus
function uvm_sequence_item my_adapter::reg2bus(const ref uvm_reg_bus_op rw);

  my_pkg::my_extension extension;

  // Get the extension.
  uvm_reg_item item = get_item();
  if (item.extension != null) begin
    if (!$cast(extension, item.extension.clone())) begin
      `uvm_error(get_type_name(), "Extension casting failed")
    end
  end
  else begin
     `uvm_error(get_type_name(), "Null extension provided!")
  end
```

Figure 2 - get_item in reg2bus

As shown above, the extension can be made available to the adapter in its "reg2bus" function via the "get_item" function. The extension may contain functions and variables for the adapter beyond the parameters provided in the uvm_reg_bus_op item passed in. This can be easier to develop and maintain than, for example, creating a number of special functions to access the adapter. Instead, the adapter can be configured shortly before use to, for example, pass in burst information as shown in Figure 3.

```
  my_extension.create_burst(data, len, byte_en);
  reg_obj.write(status, data, .path(path), .parent(this),
                .map(map), .extension(my_extension));
```

Figure 3 – create_burst

In our environment it is important to iterate each contained object in a customized "do_copy" function as shown in Figure 4. This is necessary to support the uvm_object "clone" operation (in Figure 2) used to capture the extension data instance in our adapter. Otherwise, the data may be lost.

```
class my_extension extends uvm_sequence_item;
   `uvm_object_utils(my_extension)
  int len = 0;
  int byte_en = -1;

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

  virtual function void do_copy(uvm_object rhs);
    my_extension _rhs;
    super.do_copy(rhs);
    void'($cast(_rhs, rhs));
    // NOTE: Copy all variables here.
    len = _rhs.len;
    byte_en = _rhs.byte_en;
  endfunction : do_copy
```

Figure 4 - Extension Object Class

supports_byte_enable

This option is stored in the adapter and accessed by the map for the register. It is deasserted by default. Registers are accessible by fields. If this option is asserted, it indicates that the register may also be accessed by individual bytes. UVM does not provide example support within the adapter, but the handling of register data by bytes may be critical when, for example, converting to and from a bus protocol within the adapter.

Byte enables need to be supported in the adapter if supports_byte_enable is set, requiring customization. In our environment, we use the extension object to pass byte enable information to the "reg2bus" function of the adapter as shown in Figure 5.

```
logic [63:0]  temp_byten = extension.byte_en;
```

Figure 5 – byte enable in reg2bus

The "bus2reg" function can have byte enable information passed to it from the monitor via the uvm_sequence_item "bus_item" object (cast to a customized object). This can be passed to the uvm_reg_bus_op "rw.byte_en" field as shown in the adapter code snippet of Figure 6.

```
function void my_adapter::bus2reg(uvm_sequence_item bus_item,
                                  ref uvm_reg_bus_op rw);

  my_uvc_pkg::my_data_item my_item;

  if (!$cast(my_item, bus_item)) begin
    `uvm_fatal(get_type_name(),"Provided item type is not as expected")
  end

  if (my_item.req._cmd inside {my_uvc_pkg::WRITE_CMD}) begin
    rw.kind = UVM_WRITE;
    rw.addr = my_item.req._addr;
    rw.byte_en = {my_item.wr_data._byten[1], my_item.wr_data._byten[0]};
    rw.data = {my_item.wr_data._data[1], my_item.wr_data._data[0]};
    rw.status = my_item.status;
  end
```

Figure 6 – byte enable in bus2reg

5. Predictor Customization

Customization generally starts with an extension of predictor class from either the UVM kit sources or imported UVC. It usually continues with selection and modification via a number of options as described here.

auto_predict

This option (stored in a map for the register) controls whether or not the register model will update mirrored values during write requests by invoking a predict. It supports a simplified mechanism for updating the register model via requests without a predictor. It is deasserted by default, implying that prediction is done by other means, usually by predictor.

If asserted, the register model will perform the update via predict as part of processing a frontdoor read or write request. (The register already handles predicts for backdoor read and write requests.) In addition, the register will attempt to access the DUT via backdoor access to check values as shown in Figure 7. This can introduce a problem if, for example, there is significant delay before a write request reaches the DUT register. During read requests, the register will perform the check against the mirrored value.

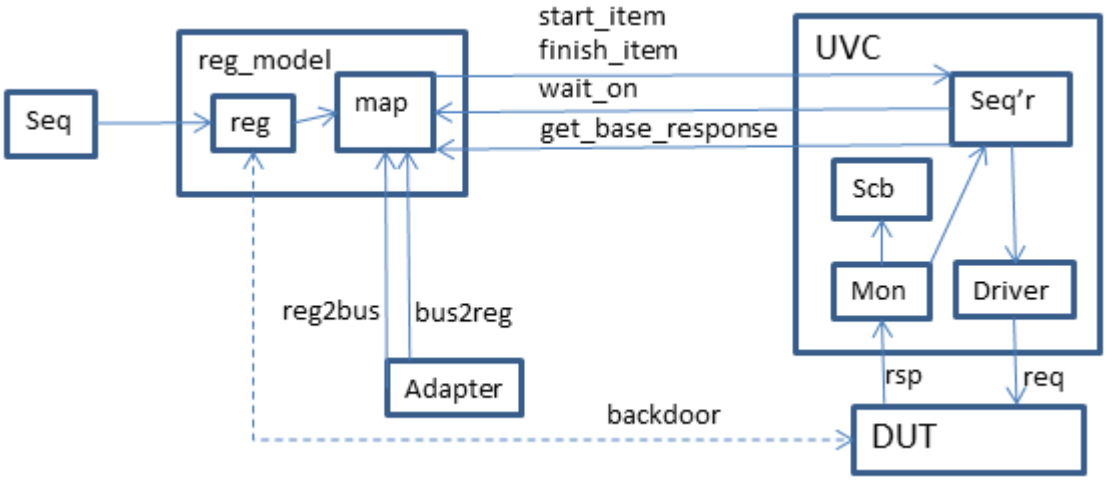

Figure 7 - Stimulus with auto_predict, No Predictor

Stimulus may be generated and checked against values in the register model. This can work so long as this is the only mechanism by which the DUT registers are updated. Without a predictor, loss of synchronization may result with auto_predict if another component changes the DUT even though that activity may be visible on the bus. Because the register model relies on backdoor access for checks, there is also the possibility of inaccuracies due to delays in updating the DUT. For example, a write request may be delayed significantly, causing a mismatch when the register attempts a check against it via backdoor.

If auto_predict is not asserted and there is no predictor maintaining synchronization with the register model, then there may be nothing automatically checking the DUT against mirrored values. This can introduce problems as the mirrored values may become out of sync with the DUT and mismatches may go undetected. To avoid this problem, we explicitly ensure that auto_predict is set for most of our variants without predictors to maintain synchronization.

supports_byte_enable

This adapter option may also become important for the predictor which may be required to process data on a byte-by-byte basis using byte masks.

```
// Limit check per rw.byte_en.

exp_val = rg.get_mirrored_value(); // start with mirrored bytes

for(int bi = 0; bi < `MY_BYTENABLE_WIDTH; bi++) begin
  if(!rw.byte_en[bi]) begin
    // Set this expected byte value to reg_item.value[0]
    exp_val[(bi*8)+7 -: 8] = reg_item.value[0][(bi*8)+7 -: 8];
  end
end

void'(rg.do_check(exp_val, reg_item.value[0], local_map));
```

Figure 8 - Byte Enable Checking

One technique for handling the predictor byte enable check is to start by setting the check value to the mirrored value as shown in Figure 8. Based on the byte masking, the expected value is updated per byte with the response value byte. When the check is called in the predictor, there should then be mismatches detected only for the enabled bytes.

A similar predictor issue arises with the current uvm_reg_field predict function, byte enables and predicts. Specifically, the "do_predict" function in uvm_reg_field receives a byte enable for a field. But, it assumes that the bytes are either all enabled or not enabled and tests the lowest bit. If, for example, a register field is 4 bytes wide and the byte enable provided only enables the top byte, the mirror value for the field will not be updated. This function is called by the register "do_predict" function that is called by the predictor. With supports_byte_enable set, this can result in unexpected failure of updates to some register values by the predictor.

One technique for handling this predictor is similar to what is suggested for handling the "do_check" call, as shown in Figure 9. This works by calculating a byte-accurate value for a register, then setting all byte enable bits before calling the "do_predict".

```
// Limit predict per rw.byte_en.

exp_val = rg.get_mirrored_value(); // start with mirrored bytes

for(int bi = 0; bi < `MY_BYTENABLE_WIDTH; bi++) begin
  if(rw.byte_en[bi]) begin
    // Set this expected byte value to reg_item.value[0]
    exp_val[(bi*8)+7 -: 8] = reg_item.value[0][(bi*8)+7 -: 8];
  end
end

// Modify reg_item.value[0] and enable all bytes for do_predict.

reg_item.value[0] = exp_val;
rw.byte_en = -1;

rg.do_predict(reg_item, predict_kind, rw.byte_en);
```

Figure 9 – Byte Enable Predicting

With supports_byte_enable off, the default tends to be to enable all bytes. This simplifies checking and predicting since all bytes will be checked and predicted by the predictor.

With supports_byte_enable on, byte enables may vary. Complications in byte enable calculation may result in byte enable errors. For example, some protocols may use byte enables on writes but not on reads. If byte enables are not handled correctly, byte enable errors may be introduced.

Byte enable errors can be particularly problematic for a predictor. If byte enables are errantly on, the predictor may detect this as byte miscompares during a check. This is the easier byte enable error to detect. If byte enables are errantly off, these bytes may not be checked and may not be updated by the predictor. This may result in byte miscompares going undetected or errant byte miscompares. These can be very difficult to detect and debug. Thus, it may be prudent to visually inspect and perhaps add other checks to ensure that byte enables are accurate when supports_byte_enable is on.

"get" vs "get_mirrored_value"

An additional, subtle choice involves whether or not testbench components will access register model "desired" values or "mirrored" values, depicted in Figure 10. This choice can have dramatic impact upon predictor customization.

Unlike mirrored values, desired values are immediately set (via the register "set" function) and immediately available (via the register "get" function). Usage of desired values may be critical if, for example, a scoreboard must be sensitive to the value set but not yet updated in the DUT for a control register. However, this value may differ from what is actually in the DUT. Worse, the desired value may even be changed elsewhere and never actually appear in the DUT.

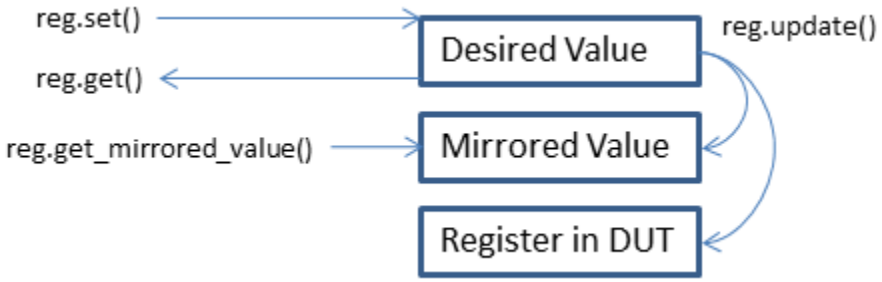

Figure 10 – Desired, Mirrored and DUT Values

Mirrored values require "update" before they are available. An update derives a mirrored value from its desired value. Mirrored values are available via the register "get_mirrored_value" function. (This function is not mentioned or depicted in the "UVM 1.1 User's Guide"[2] but will likely be found in a later release.). A primary advantage of mirrored values over desired values is that mirrored values are kept synchronized with DUT register values.

Use of the "get" function should be avoided in favor of the "get_mirrored_value" function with only a few, well-documented exceptions. Both are functions supported by a register object.

They both return the value of the register, useful in tests, scoreboards and so forth. The difference is that "get" returns the desired value of a register, which may be subject to change and may differ from the mirrored value until an update is performed. The update takes into account access restrictions that may affect the final value for the mirrored value. Using the "get" bypasses those controls and the result may be out of sync with DUT value.

While "get" should generally be avoided, it can be used with care if, for example, a value has been scheduled and must be immediately used by a scoreboard before DUT update can be confirmed. This should be well-documented, noting the caveat that the value is not in sync nor subject to access restrictions.

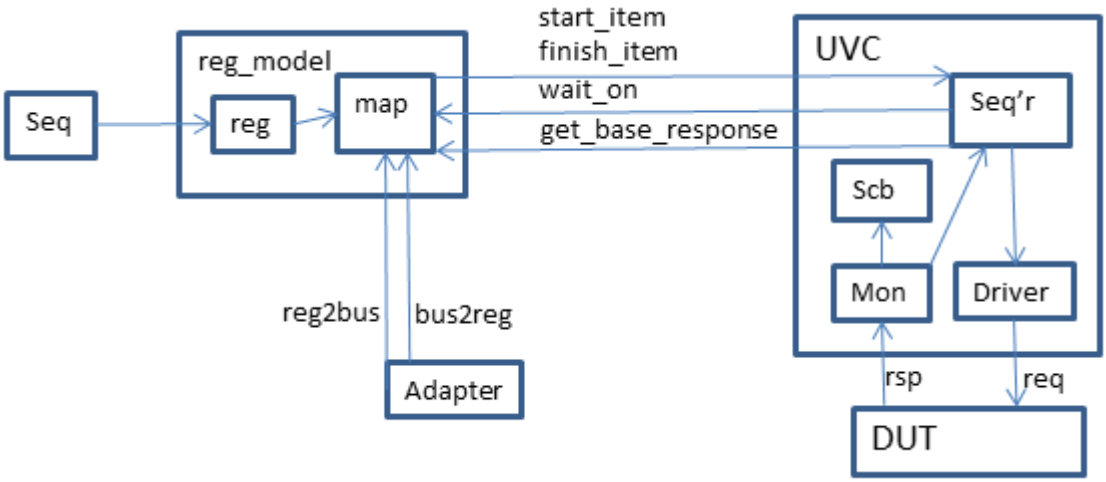

Figure 11 - Ignore auto_predict, No Predictor

Users of some testbench variants may be tempted to ignore auto_predict and discard predictors in favor of simplified use of desired values as shown in Figure 11. In this configuration, there may be no automated checking of the desired or mirrored values against the DUT. Similarly, desired values may differ from mirrored, also undetected. Thus, synchronization can be lost and undetected as components update direct values or the DUT. This can yield unreliable tests with failures that can be difficult to root-cause.

"post_write" vs "post_predict" Callbacks

Portions of our environment, such as scoreboards, depend upon mirrored values including changes to those values. When these values change, we depend upon callbacks to update components such as scoreboards. Callbacks are handles to routines contained in these components that can be called when there are changes to a register. When a sequence does a write via register model, the register invokes post_write callbacks. We use these callbacks to efficiently keep scoreboards in sync with the register model.

This use of callbacks introduces a problem for predictors. A predictor may see a register change and then call the register "do_predict" function to update the mirrored value. This will result in a call to the field object "predict" function. And, that results in "post_predict" callbacks. But, in our environment, we choose to rely upon "post_write" callbacks, not "post_predict" callbacks. Thus, a mirrored value may be updated by the predictor but a scoreboard, for example, may not know because it is only sensitive to register "post_write" callbacks.

```
// Do register callbacks.
//
function void do_callbacks(uvm_reg reg_obj,
                           uvm_reg_item rw);

  // get the callback iterator
  uvm_reg_cb_iter cbs = new(reg_obj);

  // If no callbacks, then return.
  if(cbs.first() == null) begin
    return;
  end

  fork: cb_thread_001
    begin
      automatic uvm_reg_item rwf = new;
      automatic uvm_reg reg_objf;
      automatic uvm_reg_cb_iter cbsf;

      // preserve current variables
      reg_objf = reg_obj;
      cbsf = new(reg_objf);
      rwf.copy(rw);

      // process callbacks
      for (uvm_reg_cbs cb=cbsf.first();
           cb!=null; cb=cbsf.next()) begin
        if(cb != null) begin
          cb.post_write(rwf);
        end
      end

      reg_objf.post_write(rwf);
    end
  join_none

endfunction : do_callbacks
```

One solution is to ensure that the predictor calls a function to do the post_write callbacks for the register as shown in Figure 12. In this way, register post_write callbacks are called whenever either the register or the predictor update the mirrored value. This feature of the predictor is especially valuable when operating in passive mode where the register model is dormant yet the scoreboard must maintain sensitivity to register model changes.

Our sequences support direct update of a register mirrored value. This is valuable when, for verification purposes, we create a register object that has no counterpart in the DUT but to which scoreboards must still be sensitive. That is, we allow a mirrored value to be updated by a sequence through use of the register "predict" function. This does not automatically result in a "post_write" callback. So, we also ensure that "post_write" callbacks are performed when sequences call register "predict" functions.

Figure 12 - post_write Callbacks

Customized get_rg_by_offset

The register model may support a multitude of different predictors. To avoid having to support redundant functions for these predictors, our environment uses a common "get_rg_by_offset" function which replaces the uvm_map "get_reg_by_offset" function in the base predictor code. In support of this customized function, each predictor must present the same variables consistently, regardless of bus interface (as shown in Figure 13 and called in Figure 15).

```
function uvm_reg my_ral::get_rg_by_offset( input     uvm_reg_map map,
                                           ref uvm_reg_addr_t offset,
                                           input bit special_detected);
```

Figure 13 - Customized get_rg_by_offset

Address Filtering

Values provided in the transaction item from the monitor to the predictor require some conversion by the adapter, including the address. This means that any original address information may be lost due to this conversion. So, any special addresses may need to be filtered and tracked before the adapter call. Because we lose the original address, we need to filter it and save detection of a special address for later processing. In Figure 14, the "special_detected" variable is calculated before the adapter call and passed to "get_rg_by_offset" in Figure 15.

```
if((tr.req._addr[47:0] >= 'hFFF1_2340_0000) &&
   (tr.req._addr[47:0] <= 'hFFF1_2340_FFFF)) begin
  special_detected = 1;
end
else begin
  special_detected = 0;
end

adapter.bus2reg(tr,rw);
```

Figure 14 - Address Filtering

Indirect Registers

Our environment supports a multitude of complex, indirect register access mechanisms. The indirect register support provided with UVM RAL seemed insufficient for our purposes. To simplify prediction, the customized "get_reg_by_offset" function handles all indirect register lookup and returns the target of the indirection. Thus, the predictor will always continue with the target register after the "get_rg_by_offset" call.

Special checks via "my_check" are required to discern valid register access as shown in the predictor code snippet of Figure 15.

```
adapter.bus2reg(tr,rw);

checked = my_ral.my_check(rw.addr);

if(checked) begin
  int old_addr = rw.addr;

  // rg = map.get_reg_by_offset(rw.addr, (rw.kind == UVM_READ));
  rg = my_ral.get_rg_by_offset(map, rw.addr, special_detected);

  if(old_offset != rw.addr) begin
      checked = my_ral.my_check(rw.addr);
```

Figure 15 - Special Checks

In our environment, "my_check" is required to validate register access (shortly after the "bus2reg" function call). It is also used to validate target register access (shortly after the "get_rg_by_offset" call) as shown in Figure 15. Specifically, the old register offset may be for an indirect register which passed the first check. The custom "get_rg_by_offset" function may then return the target register being accessed indirectly. That register would be at a different offset, which can be detected and result in check to validate access to that register as well.

6. Conclusions

Our verification environment supports a handful of critical, large development projects supported by dozens of testbenches at various levels of design abstraction. In support of these, we customize nearly a half-dozen sets of adapters and predictors to support our register model.

Driven by requirements that include keeping mirrored values in sync with the DUT, adapters and predictors are carefully customized as described in this paper. Our customizations include modifications of specific options including auto_predict, provides_responses and supports_byte_enable. We also use extension objects to pass additional information to adapters. Customization is also affected by choices with respect to desired versus mirrored values and callbacks. Additional customization is required in support of address filtering and indirect register access.

Our experience is that with such customizations it is possible to maintain synchronization between the register model and the DUT. Further, these customizations enable reuse of our register model and related infrastructure across a wide variety of testbench variants.

7. References

[1] VIP Technical Committee. *Universal Verification Methodology (UVM) 1.2 Class Reference.* Accellera, June 2014.

[2] VIP Technical Committee. *Universal Verification Methodology (UVM) 1.1 User's Guide.* Accellera. May 2012 <http://www.accellera.org/downloads/standards/uvm/uvm_users_guide_1.1.pdf>.

[3] Smith, Doug. "Easier RAL: All You Need to Know To Use the UVM Register Abstraction Layer", SNUG San Jose, 2012.

[4] Sherman, Steven K. "Taming UVM 1.1d RAL in a Multi-Interface, Multi-Mode Environment", SNUG Boston, 2014.