Doing Funny Stuff with the UVM Register Layer: Experiences Using Front Door Sequences, Predictors, and Callbacks

John Aynsley
Doulos, Church Hatch, 22 Market Place, Ringwood, Hampshire, UK. john.aynsley@doulos.com

***Abstract*- This paper focuses on our experiences using three specific aspects of the UVM register layer: front-door sequences, the predictor, and register callbacks. The topic of front-door sequences includes how to define a front-door sequence and how to use that sequence to extend the capabilities of the register layer beyond sending simple request and response transactions to the DUT. The topic of the predictor focuses on understanding the role played by the predictor in updating the register model and how to use the predictor in the presence of a front-door sequence. The topic of register callbacks includes how to associate callbacks with registers and register fields and how to use callbacks to define special register behaviors. Used together, these features provide an important set of mechanisms for extending the capabilities of the register layer in several useful ways.**

I. INTRODUCTION

The UVM register layer is a very broad topic. This paper focuses on our experiences using three specific aspects of the UVM register layer: front-door sequences, the predictor, and register callbacks. These topics can pose particular challenges to practitioners as they move beyond the beginner level because the details and implications of these topics are not fully spelled out in the standard UVM documentation.

When using the UVM register layer, register test sequences make method calls to write or read values to or from registers that are instantiated within the hierarchically organized set of UVM register blocks that form the UVM register layer for a particular verification environment. At a minimum, the caller of these methods only needs a) a reference to the register object, b) whether the operation is a write or a read and c) in the case of a write, the value to be written. For example:

```
task body;
  regmodel.reg0.write(.value(data), .status(status));
  assert( status == UVM_IS_OK );
  regmodel.reg0.read(.value(data), .status(status));
  assert( status == UVM_IS_OK );
  assert( data == expected );
```

Figure 1. Calling write and read.

The caller does not need to be aware of the logical or physical address of the register within any memory map, the details of the protocol used to communicate with the DUT (Design-Under-Test), the endianness of any data sent over the communication interface to the DUT, or the physical location of the register within the DUT. As a consequence of this abstraction, it is possible to create UVM register tests that are independent of the location of the register within the DUT or the means used to access that register.

The UVM register object can then access the value of the actual register within the HDL code that represents the DUT using one of several mechanisms, the choice being under user control. The default operation of the UVM register object (accessed through the variable reg0 in the code fragment above) is to convert each write or read method call into a generic transaction (of type uvm_reg_bus_op) that contains kind (write or read), address, data, and status fields. This generic transaction is passed to a user-defined object of type uvm_reg_adapter that converts it to a protocol-specific transaction. This protocol-specific transaction is then executed on the sequencer within the UVM agent connected to the appropriate interface of the DUT. In the case of a read transaction, data read from the DUT is passed back upstream to the caller of the read method. This mechanism is known as front door access.

The alternative to front door access is back door access in which the read or write method call is converted to a DPI access that uses the HDL hierarchical path name to access the register within the DUT. Back door access is

always simpler and faster than front door access because it bypasses the interface to the DUT and instead uses the fact that the code is running within a simulation environment, rather than an actual electronic system, to access the register contents directly. It is possible to switch to back door access using an argument to the write or read method, for example:

```
regmodel.reg0.write(.value(data), .path(UVM_BACKDOOR), .status(status));
```

Figure 2. UVM_BACKDOOR.

II. User-Defined Front Door Sequences

As described above, the default operation of the UVM register layer is that each call to read or write is translated by the register layer into a single transaction that is executed on the sequencer of a UVM agent. In any situation where a single transaction is not sufficient to implement a read or write operation, the user can replace this default behavior by setting a user-defined front door for the register. A user-defined front door takes the form of a user-defined sequence that is executed whenever the register needs to access the value of the register in the DUT. This sequence runs on the sequencer associated with the address map in the UVM register block (the same sequencer that is used for built-in front door access) and can have any user-defined behavior.

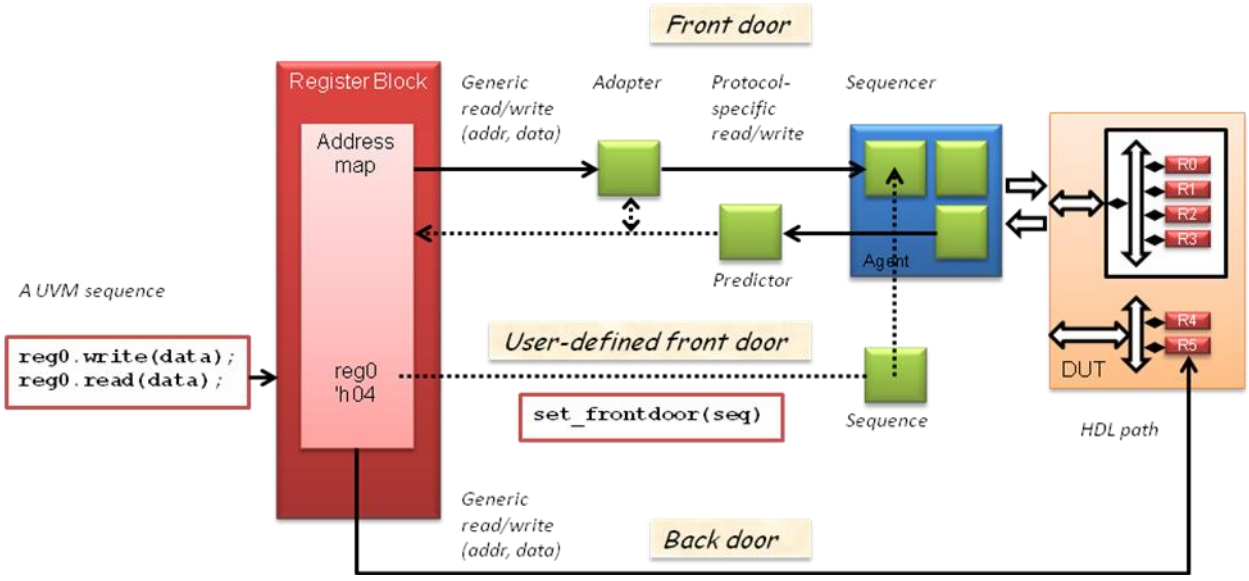

Figure 3. Front door, user-defined front door, and back door.

A user-defined front door is set by creating a new sequence object and passing it as an argument to the set_frontdoor method of the UVM register concerned. This might be done from a UVM env during the build phase after the instantiation of the UVM register model, for example:

```
my_vreg_frontdoor_sequence frontdoor;
frontdoor = my_vreg_frontdoor_sequence::type_id::create("frontdoor");
regmodel.bus.reg0.set_frontdoor(frontdoor);
```

Figure 4. set_frontdoor.

The open-ended nature of the front door sequence mechanism means that it can be used to address a number of different use cases, including unmapped registers, non-linear addressing, and burst mode access. Unmapped registers are registers embedded within the DUT that are not directly accessible using memory-mapped transactions sent through the external pins. Non-linear addressing implies that the registers or register fields to be accessed occupy a non-continuous set of addresses as viewed from the interface to the design-under-test. Burst mode access

implies that access to a single register or register field within the design-under-test requires a burst mode transaction over a memory-mapped bus at the pins of the design-under-test.

Here we will illustrate the technique with an example of a virtual register whose contents are distributed around the DUT in a non-linear way. A single write or read to the virtual register requires two write or read transactions to access two nibbles at non-contiguous addresses on the DUT interface.

The key to creating front door sequences is to understand the use of the rw_info object, which is of type uvm_reg_item and is inherited from class uvm_reg_frontdoor. This object holds information about the UVM register being written or read, including whether the access is a write or a read (rw_info.kind), the data being written or read (rw_info.value), the status of the access (rw_info.status), a reference to the register object itself (rw_info.element), and several other properties (refer to the documentation for class uvm_reg_item for further details). The address of the register within the address map of the containing register block can be found by calling the get_offset method of the register object, having first obtained the register object from rw_info.element. The rw_info.value field is actually a dynamic array of type uvm_reg_data_t, so element rw_info.value[0] is used for data values that fit within the uvm_reg_data_t type.

```
class my_vreg_frontdoor_sequence extends uvm_reg_frontdoor;
  ...
  task body;
    uvm_reg        the_reg;
    uvm_reg_addr_t reg_addr;
    bit            cmd;
    uvm_reg_data_t data;

    $cast(the_reg, rw_info.element); // Find the original uvm_reg object
    reg_addr = the_reg.get_offset(); // Find the address of the register

    cmd  = (rw_info.kind == UVM_WRITE);
    data = rw_info.value[0][3:0];    // Bottom nibble

    one_transaction( .cmd(cmd), .addr(reg_addr+1), .data(data) );

    if (cmd == 0)                    // Read command
      rw_info.value[0][3:0] = data;

    data = rw_info.value[0][7:4];    // Top nibble

    one_transaction( .cmd(cmd), .addr(reg_addr+5), .data(data) );

    if (cmd == 0)                    // Read command
      rw_info.value[0][7:4] = data;

    rw_info.status = UVM_IS_OK;
  endtask
```

Figure 5. uvm_reg_frontdoor.

Task one_transaction is very straightforward: it sends a single write or read transaction through the UVM driver to the DUT. In the case of a read transaction, it copies the data from the response object received from the driver into its data argument so that the data can be passed back to the register layer. Note the types of the address and data arguments, which are as they appear in the generic register transaction.

```
  task one_transaction(bit cmd, uvm_reg_addr_t addr, ref uvm_reg_data_t data);
    bus_tx req;
    bus_tx rsp;
    uvm_sequence_item item;

    req = bus_tx::type_id::create("req");
    start_item(req);

    req.cmd  = cmd;
    req.addr = addr;
```

```
    req.data = data;

    finish_item(req);

    get_response(item);
    $cast(rsp, item);
    assert(rsp != null);

    if (cmd == 0)
      data = rsp.data;
  endtask
endclass
```

Figure 6. one_transaction.

There are a couple of subtleties here. The first concerns addressing. Many aspects of the UVM register layer hinge around the address map of each UVM register block. A register block may have multiple address maps, where each address map is associated with an adapter, a predictor, and the sequencer and monitor of a particular agent. The original UVM register regmodel.bus.reg0 will have been placed at a specific offset when it was added to the address map of the enclosing register block, for example:

```
bus_map.add_reg( reg0, 'h0, "RW");
```

Figure 7. add_reg.

Ultimately, all address maps have to be added as sub-maps of a so-called root map, which is found in the top-level UVM register block, for example:

```
root_bus_map.add_submap(bus.bus_map, 'h0);
```

Figure 8. add_submap.

The front door sequence can find the address map using the rw_info object, for example:

```
uvm_reg regs[$];
rw_info.local_map.get_root_map().get_registers(regs);
foreach (regs[i])
  $display("[REGS] %s", regs[i].get_full_name()); // Print all the regs in the map
```

Figure 9. rw_info.

The address returned by the call the_reg.get_offset() in the front door sequence will be the address within the address map through which the register is being accessed, which by default will be the default_map of the containing register block. But in our example this nominal offset within the address map is not the actual address of the register(s) within the DUT. The addresses of the top and bottom nibbles are calculated from the original register address by the body task of the front door sequence, for example:

```
one_transaction( .cmd(cmd), .addr(reg_addr+5), .data(data) ); // Top nibble
```

Figure 10. one_transaction.

The second subtlety concerns the sequencer. The front door sequence will execute on the same sequencer as regular front door transactions, that is, the sequencer referred to from the address map. Once again this could be found using the rw_info object, for example:

```
assert( rw_info.local_map.get_root_map().get_sequencer() == this.get_sequencer());
```

Figure 11. get_sequencer.

In the example above, the front door sequence generates transactions on the same sequencer that the regular front door transaction would run on, but there is no obligation for it to do so. A front door sequence can do anything! It

could perform further register reads or writes, execute transactions on multiple agents, use the DPI, access external files, or perform calculations. This is the beauty of the user-defined front door.

Another case where a front door sequence can prove useful is in passing response transaction objects back up to the register test sequence. If a driver passes an explicit response back upstream to the sequencer, this response object would be available to a regular UVM sequence (as shown in task one_transaction above) but is not made available to the register sequence that made the original read or write method call. It is possible to make the response available to the register sequence by utilizing the extension argument to the read and write methods of the register layer, but doing so requires the use of a front door sequence because the extension argument is not always accessible from the register adapter of the register layer. The extension argument is available within the uvm_reg_item class as returned by the get_item method of class uvm_reg_adapter, but this is only available for the bus2reg method, not the reg2bus method, so the response cannot be passed back upstream through the register adapter. This means that when using the built-in front door with the UVM register adapter, it makes sense to use the extension argument to the write method call but not to the read method call.

In order for the register test sequence to get the response transaction back from the driver, the front door sequence must create a suitable response object and pass it as the extension argument to the write or read method. The key point here is that the object must be created first and passed into the method as opposed to being created by the method:

```
bus_tx rsp1;
rsp1 = bus_tx::type_id::create("rsp1");

regmodel.reg0.write(.value('hab), .status(status), .extension(rsp1));
regmodel.reg0.read (.value(data), .status(status), .extension(rsp1));

assert(status == UVM_IS_OK);
assert(data == 'hab);
assert(rsp1.data == 'hab);
```

Figure 12. extension.

Of course, passing the data in the response object in this way is pointless since the data is available anyway. The point is that additional information could be made available using the response object. The interesting work is done by the front door sequence, which copies the contents of the response transaction from the driver back into the extension argument using the rw_info object:

```
task body;
  ...
  finish_item(req);

  get_response(item);            // Get response object from driver
  $cast(rsp, item);
  assert(rsp != null);
  rw_info.extension.copy(rsp);   // Copy response into the extension argument

  if (cmd == 0)
    rw_info.value[0] = rsp.data; // Copy data into the value argument
  ...
```

Figure 13. rw_info.extension.

III. User-Defined Back Doors

Although it is possible, it is not particularly useful to associate a user-defined front door sequence with a memory in the register layer because back door access is generally recommended for memories. However, it is possible to set a user-defined back door for a memory (or for a register, for that matter), which would allow the user to deal with any kind of irregular mapping between the memory in the register layer and the memory in the design-under-test.

A back door is an object of a class that extends uvm_reg_backdoor, for example:

```
class my_mem_backdoor extends uvm_reg_backdoor;
  `uvm_object_utils(my_mem_backdoor)
```

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

  virtual task write(uvm_reg_item rw);
    bit ok;
    int n = rw.value.size();
    for (int i = 0; i < n; i++)
    begin
      ok = uvm_hdl_deposit(
           $sformatf("top_tb.th.uut.mem[%0d]", rw.offset + i), rw.value[i]);
      assert(ok);
    end
    rw.status = UVM_IS_OK;
  endtask

  virtual task read(uvm_reg_item rw);
    bit ok;
    int n = rw.value.size();
    for (int i = 0; i < n; i++)
    begin
      ok = uvm_hdl_read(
           $sformatf("top_tb.th.uut.mem[%0d]", rw.offset + i), rw.value[i]);
      assert(ok);
    end
    rw.status = UVM_IS_OK;
  endtask

endclass
```

Figure 14. uvm_reg_backdoor.

The user-defined back door then needs to be set for the appropriate register or memory objects:

```
...
my_mem_backdoor backdoor;
backdoor = my_mem_backdoor::type_id::create("backdoor");
regmodel.bus.mem.set_backdoor(backdoor)
```

Figure 15. set_backdoor.

The example above is for illustrative purposes only because it implements straightforward access to the contents of the memory using an HDL path name, which is pretty much what the built-in back door would do anyway. Note that the overridden write and read methods are passed an object of type uvm_reg_item with the fields value, offset, and status, the same technique as was used in the user-defined front door shown previously. Also note the use of the UVM HDL back door access support routines uvm_hdl_deposit and uvm_hdl_read. These calls are of interest in that they provide the user with access to the same DPI code that is used to implement built-in back door access.

Here is an example of performing a burst write operation through such a memory back door:

```
task body;
  uvm_reg_data_t burst_data[];
  burst_data = new[4];
  for (int i = 0; i < 4; i++)
    burst_data[i] = 'h10 + i;

  regmodel.mem.burst_write(.status(status), .offset('h10), .value(burst_data),
                           .path(UVM_BACKDOOR), .parent(this));
  assert(status == UVM_IS_OK);
```

Figure 16. UVM_BACKDOOR.

IV. THE PREDICTOR

Each UVM register contains a mirror value that is meant to mirror the current value of the actual register within the DUT. The mirror value allows the user to optimize the number of reads and writes to registers within the DUT by reading the mirror value instead of the actual value when the mirror is known to be up-to-date, and by only writing the actual value to the DUT when the value being written (the desired value) is different from the mirror value, for example:

```
value = regmodel.reg0.get_mirrored_value();  // Return mirror without accessing DUT
regmodel.reg0.set( .value(`hab) );           // Set the desired value
regmodel.reg0.update( .status(status) );     // Write desired value to register
regmodel.update( .status(status) );          // Only write if desired != mirrored
```

Figure 17. get_mirrored_value.

There are two ways in which the mirror values in the UVM register layer can be kept in step with the actual register values in the DUT: auto-prediction and explicit prediction. With auto-prediction, the mirror values are refreshed whenever a read or write call is performed through the register layer. With explicit prediction, the analysis port of a UVM monitor component is connected to a UVM predictor object, which in turn is connected to the address map and thus to the registers within that address map. Explicit prediction is usually preferred because it allows the mirror values in the register layer to be refreshed to reflect all of the traffic visible at the level of the DUT interface, not just the read and write calls through the register layer.

To use explicit prediction, the register adapter must be instantiated and connected to the address map and the sequencer, as usual. The predictor must be instantiated and connected to the address map, the adapter, and the monitor analysis port. For example:

```
reg2bus_adapter             m_reg2bus_adapter;
uvm_reg_predictor #(bus_tx) m_bus2reg_predictor;
...

// Build phase
m_reg2bus_adapter   = reg2bus_adapter::type_id::create("m_reg2bus_adapter", this);
m_bus2reg_predictor = uvm_reg_predictor #(bus_tx)::type_id::create(
                                            "m_bus2reg_predictor", this);
...

// Connect phase
regmodel.bus_map.set_sequencer(m_bus_agent.m_sequencer, m_reg2bus_adapter);
regmodel.bus_map.set_auto_predict(0);

m_bus2reg_predictor.map     = regmodel.bus_map;
m_bus2reg_predictor.adapter = m_reg2bus_adapter;

m_bus_agent.analysis_port.connect(m_bus2reg_predictor.bus_in);
```

Figure 18. uvm_reg_predictor.

Explicit prediction can interact with read and write operations in unexpected ways because the predictor will refresh the mirror values in the register layer independently from the values that are passed though read and write method calls. With explicit prediction, there are two paths back from the UVM agent to the register layer, one path through the predictor and another path that uses the response transaction sent back through the adapter.

In the case of a user-defined frontdoor sequence, the front door sequence can and should return a value from the read method by assigning rw_info.value = data_being_read, even in the presence of an explicit predictor. The explicit predictor will refresh the mirror value correctly regardless of whether or not the user-defined front door sequence returns the correct value. On the other hand, the path from the analysis port of the monitor back to the explicit predictor is always necessary in order to refresh the mirror value correctly, even though the front door sequence returns the correct value. If the monitor is not connected to the predictor, the mirror value will not be refreshed.

In the presence of an explicit predictor, methods of the UVM register object have the following behavior:

```
regmodel.reg0.read(.value(value)); // Return the value from the front or back door
value = regmodel.reg0.get_mirrored_value(); // Return the value from the predictor
value = regmodel.reg0.get();                // Return the desired value
```

Figure 19. read.

V. Register Callbacks

The mirror value within each register in the register layer should accurately reflect the value of the actual register within the design-under-test, even in the presence of quirky registers for which read and write operations have some side-effect that goes beyond a straightforward read or write to a single register or field. The register layer supports a range of built-in access policies to model such effects, for example the policy WSRC, in which a write sets all the bits of the register and a read clears all the bits of the register. For example:

```
class fancy_reg extends uvm_reg;
   `uvm_object_utils(fancy_reg)

   rand uvm_reg_field F1;

   function new(string name = "");
      super.new(name, 16, UVM_NO_COVERAGE);
   endfunction

   virtual function void build();
      F1 = uvm_reg_field::type_id::create("F1");
      F1.configure(this, 16, 0, "WSRC", 1, 16'h0000, 1, 1, 0);
   endfunction
endclass
```

Figure 20. WSRC.

It is important to understand that by selecting a quirky access policy such as WSRC for the UVM register, you are merely ensuring that the mirrored value in the UVM register will be updated to reflect the value of the actual register. It is still necessary to model the actual behavior of the quirky register within the HDL code that represents the DUT itself. For example:

```
// Fancy register
regmodel.reg2.write(status, .value('h0001), .parent(this));

expected = 'hffff;
data = regmodel.reg2.get_mirrored_value(); // All the bits should have been set
assert(data == expected);

regmodel.reg2.read(status, .value(data), .parent(this));
assert(data == expected);

expected = 'h0000;
data = regmodel.reg2.get_mirrored_value(); // All the bits should have been cleared
assert(data == expected);
```

Figure 21. get_mirrored_value.

To model more complex behaviors, the user can associate callbacks with registers, register fields, memories, and backdoor access in the register layer, and these callbacks can be used to predict the mirror values stored in the register layer. The pre_read and post_read callbacks are called before and after each call to the read method, while the pre_write and post_write callbacks are called before and after each call to the write method. Here is an example that uses callbacks to set the mirror value of a quirky register where a write toggles the contents of the register and a read clears the register. First we need to define a callback class:

```
class my_reg_callbacks extends uvm_reg_cbs;

  task post_read(uvm_reg_item rw);
    bit ok;
    uvm_reg the_reg;
    assert(rw.element_kind == UVM_REG);
    $cast(the_reg, rw.element);

    // Predict the side-effect of the register read
    ok = the_reg.predict(0, -1, UVM_PREDICT_DIRECT);
    assert(ok);
  endtask

  task post_write(uvm_reg_item rw);
    bit ok;
    uvm_reg the_reg;
    assert(rw.element_kind == UVM_REG);
    $cast(the_reg, rw.element);

    // Predict the side-effect of the register write
    ok = the_reg.predict(~rw.value[0], -1, UVM_PREDICT_DIRECT);
    assert(ok);
  endtask

endclass
```

Figure 22. uvm_reg_cbs.

Then we instantiate a callback object and add it to the register object. This might be done in the build phase of a UVM env. In general a callback could be added either to a register or to a field provided the callback methods have been overridden appropriately. This paper shows one example of each.

```
my_reg_callbacks cb;
cb = new;
uvm_reg_cb::add(regmodel.bus.reg0, cb);
```

Figure 23. add.

Register callbacks have a number of use cases, including the modeling of quirky registers described above and the modeling of aliased registers, where a single physical register appears at several different places in an address map. By associating user-defined callbacks with a set of aliased registers in the register layer, it is possible to keep the mirror values synchronized across the full set of aliased registers.

Here is an example that uses register callbacks to implement a register model for a set of aliased registers. In the register model, registers regi[j], for j = 0 to 15, are each placed at a different address. regi[n], regi[n+4], regi[n+8], and regi[n+12], for n = 0 to 3, are aliases for the same physical register. The post_write callback method predicts the side-effect of the register write by modifying the mirror value of the aliased registers.

```
class my_reg_field_callback extends uvm_reg_cbs;

  uvm_reg_field m_alias_regs[];

  function new(uvm_reg_field alias_regs[]);
    m_alias_regs = alias_regs;
  endfunction

  virtual task post_write(uvm_reg_item rw);
    for (int i = 0; i < m_alias_regs.size(); i++)
    begin
      bit ok = m_alias_regs[i].predict( rw.value[0] );
      assert(ok);
    end
  endtask
```

```
endclass
```

Figure 24. post_write.

The callbacks are instantiated and added to the appropriate register objects in the build phase:

```
my_reg_field_callback cb;

uvm_reg_field m_alias_regs[] = new[3];

// Loop through the full set of registers
for (int i = 0; i < 16; i++)
begin
  int n = 0;

  // For each register, loop through the aliased addresses
  for (int j = i % 4; j < 16; j += 4)
  begin
    // Build an array of aliased registers
    if (j != i)
      m_alias_regs[n++] = regmodel.bus.regi[j].F;
  end
  assert(n == 3);

  // Create a unique callback for each register with the appropriate set of aliases
  cb = new( m_alias_regs );
  uvm_reg_field_cb::add( regmodel.bus.regi[i].F, cb);
end
```

Figure 25. uvm_reg_field_cb.

Register callbacks have a number of pitfalls. UVM provides four callbacks pre_write, post_write, pre_read, and post_read that apply to registers and register fields, and a fifth callback post_predict that only applies to register fields. In the presence of an explicit predictor, the post_predict callback is only called when the predictor receives an incoming analysis transaction from the UVM agent it is connected to, and should be used with caution.

The mirror value can be refreshed by calling the predict method of a register or a register field, as shown in the two examples above. Care must be taken not to attempt to modify the mirror value of a register from the post_predict callback of the register itself, because this can result in a recursive call to post_predict. Similarly, register callbacks should not initiate operations that would result in mutually recursive callbacks across sets of registers. It is better to call the predict method from one or more of the pre/post_write/read callbacks, as shown above, and to avoid calling predict from the post_predict callback.

The predict method, called to refresh the mirror value, has a kind argument that is set to one of three values: UVM_PREDICT_DIRECT, UVM_PREDICT_READ, or UVM_PREDICT_WRITE. The default value UVM_PREDICT_DIRECT simply overwrites the mirror with the given value. In the other two cases, the value is modified according to the register field access policy before the mirror is overwritten in order to model quirky register effects.

VI. Conclusion

User-defined front door sequences and register callbacks provide considerable flexibility when modeling the behavior of DUT registers in the UVM register layer. Front door sequences are useful whenever there is no simple one-to-one correspondence between register read and write method calls and transactions executed by UVM agents. Register callbacks are useful whenever a register read or write method call has side effects that go beyond simply reading or writing the value of the given register or register field. Both mechanisms interact with explicit prediction, and the details of these interactions, as described in this paper, need to be understood and considered when using front door sequences and register callbacks.

REFERENCES

[1] UVM 1.2 Class Reference, http://www.accellera.org/downloads/standards/uvm, June 2014.