Configuring a Date with a Model

A Guide to Configuration Objects and Register Models

Jeff Montesano, Jeff Vance

Verilab, Inc.

Austin, TX, USA

www.verilab.com

ABSTRACT

The topic of register models, configuration objects, and their interaction can be an area of great complexity and sometimes confusion for many verification engineers. Fundamentally, a register model holds the contents of each register in the design for use by the verification environment, while a configuration object holds the configuration for the interface protocol agents, verification components and verification environment. So while one is implementation specific (register model) and one is generic (configuration object), they both hold configuration information and are both required in a given testbench. In this paper we will describe the unique roles of register models and configuration objects, explain why they should both be used in a verification environment, and present a framework in SystemVerilog/UVM for making the testbench and design configurations remain in sync for different types of designs.

Table of Contents

Table of Figures

1. Introduction

The topic of register models, configuration objects, and their interaction, can be an area of great complexity and sometimes confusion for many. On a fundamental level, a register model holds a representation of each register in the design under test (DUT) for use by the verification environment, while configuration objects hold the configuration for the interface protocol agents, UVM verification components (UVCs), and verification environment. Both the DUT and the testbench *must* have the same configuration in order to communicate. In the case of static configurations (e.g. number of channels, bus widths – which may or may not be programmable in DUT registers) synchronizing the DUT and testbench is trivial, as these configurations are fixed to just one value for a given simulation run. When dealing with dynamic configurations (e.g. configurations that are programmed into DUT registers and can change during a simulation), achieving perfect synchronization between the DUT and the testbench can be a real challenge. In terms of roles, it is possible for the register model to play a proactive role with the configuration object playing a reactive role (e.g. a test case randomizes the register model and uses it to program the DUT, the testbench's configuration objects get updated to remain in sync). It is also possible for the roles to be reversed (e.g. a test case randomizes a testbench configuration object and uses it to program the DUT, the register model gets updated to remain in sync). Despite the fact that both structures hold configuration information, any non-trivial verification environment should have both, and in the sections that follow we will show how this should be done.

This paper focuses primarily on keeping the DUT and testbench in sync when dealing with pseudo-static and dynamic configurations within a block-level testbench. It is divided into the following sections: section 2 describes the roles of configuration objects and register models, and looks at their areas of overlap and difference; section 3 discusses how to implement a configuration object that handles designs which do not make immediate use of register settings; and section 4 provides a solution for keeping the configuration object in perfect sync with the DUT when dealing with dynamic configuration fields.

2. Register Models vs. Configuration Objects

Let's start by examining the unique characteristics of register models and configuration objects, and what they can accomplish for us.

A register model's role is:

- to provide a database of each register and register field inside the DUT along with its access type (e.g. read/write, read-only), size, reset value, and current value
- to provide test case writers with an API for doing register accesses using bus transactions (known as front-door access)
- To provide test case writers with an API for doing register accesses without using bus transactions, by directly accessing the RTL signal representing the register in the design (known as back-door access)

The basic structure of a UVM register model is shown in Figure 1.

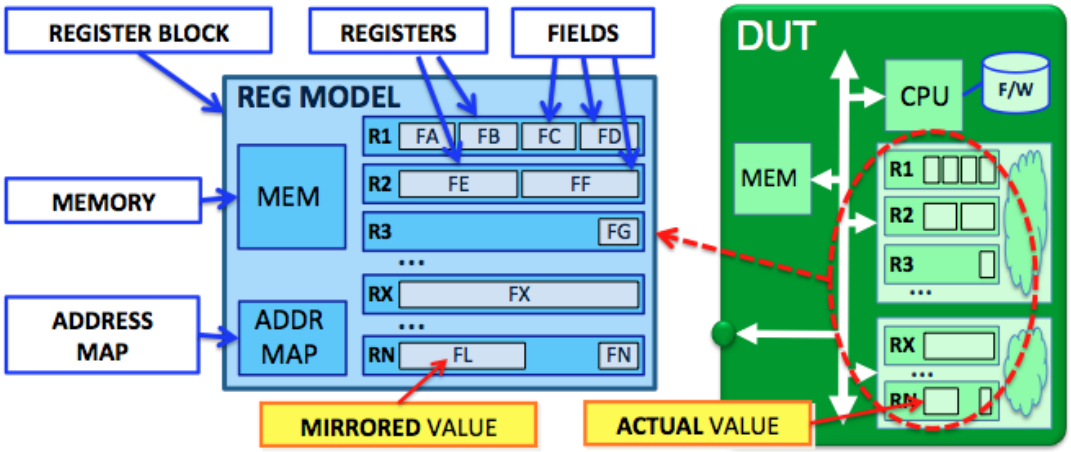

Figure 1. Register model.

A configuration object's role is:

- to encapsulate static (i.e. never changes) or pseudo-static (i.e. changes infrequently) interface protocol configuration information (e.g. number of data lanes, operating speed) as described in a standards-based protocol specification (e.g. USB, I2C) or proprietary block-level specification document
- to encapsulate dynamic configuration information (e.g. design-specific fields such as "start processing", "operation type") as described in a standards-based protocol specification or proprietary block-level design specification document
- to potentially provide constrained-random values for the verification environment to program the DUT with
- To serve as a "hub" to share information between different parts of the testbench

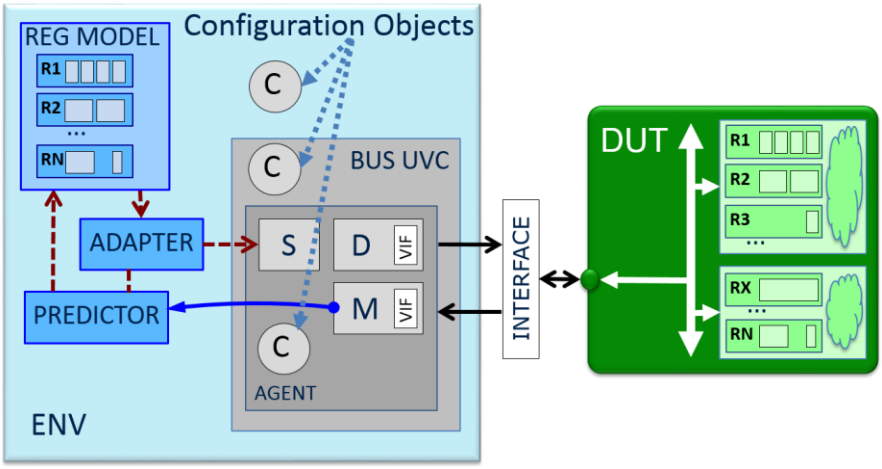

Figure 2. Configuration objects in the verification environment.

Whenever activity is detected on the register bus of the DUT, both the register model and configuration objects at all levels need to be updated (Figure 2). As mentioned in the introduction, when a testbench is verifying a DUT that has only static configurations (which is true in many

projects), synchronization between the DUT and testbench is trivial, and there is no need for any of the synchronization solutions that will be presented later in this paper.

Both the register model and the configuration object exist to serve the needs of the verification environment, and while there is some overlap, there are also important points of divergence.

2.1 Overlap and Divergence Between Register Models and Configuration Objects

In a block-level verification environment, the register model and configuration object may look very similar. Let's take a simple example of an 18-bit counter DUT with 4 configuration fields spread across 5 16-bit registers (see Figure 3).

CTRL1:0x0	15:1	0
	RESERVED	COUNTER_EN
CTRL2:0x2	15:1	0
	RESERVED	PARITY
CTRL3:0x4	15:1	0
	RESERVED	RESET_COUNTER
CTRL4:0x4	15:0	
	MAX_COUNT_L	
CTRL4:0x5	15:2	1:0
	RESERVED	MAX_COUNT_H

Figure 3. An example of 5 control registers in an RTL block.

The UVM code for the register model of such a DUT is shown in Figure 4.

```
class block_reg_model extends uvm_reg_block;
 ctrl1_reg ctrl1;
 …
 ctrl5_reg ctrl5;
…
class ctrl1_reg extends uvm_reg;
 rand uvm_reg_field counter_en;
…
class ctrl2_reg extends uvm_reg;
 rand uvm_reg_field parity;
…
class ctrl3_reg extends uvm_reg;
 rand uvm_reg_field reset_counter;
…
class ctrl4_reg extends uvm_reg;
 rand uvm_reg_field max_count_l;
…
class ctrl5_reg extends uvm_reg;
 rand uvm_reg_field max_count_h;
```

Figure 4. A register model for a design with 5 control registers.

While some more code is required to specify the register addresses, field widths, and access types, we'll leave it at this for now. The code snippet for a possible configuration object for this block is given in Figure 5.

```
class block_config  extends uvm_object;
 rand bit        counter_en;
 rand bit        parity;         // 0=odd, 1=even
 rand bit [17:0] max_count;      // 0x0=1, 0x3FFFF=262144
 rand bit        reset_counter;
 …
endclass
```

Figure 5. Configuration object for a block-level environment.

While functionally usable, the configuration object in Figure 5 suffers from the fact that it is not generic enough - it is too closely tied to the register implementation of the DUT. For example, a protocol specification will usually indicate that a parity bit is required, but it will not specify the encoding (0 could mean odd or even parity – it is up to the design to choose). Recall that our goal as verification engineers is to create a verification environment that will work with *any* design implementing the protocol specification. As for the max_count field, the fact that real hardware encodes a max_count of 1 as 0x0 and a max_count of 262144 as 0x3FFFF is an implementation detail that the configuration object should not be constrained to follow. An improved configuration object is given in Figure 6, which replaces type bit with an enumerated type for the parity field (which is descriptive and not dependent upon encoding), and type bit[17:0] with an integer for the max_count

field (which holds the true maximum count value).

Figure 6. Improved configuration object that is more generic.

```
class block_config extends uvm_object;
 rand bit               counter_en;
 rand my_parity_enum    parity;  //POSITIVE, NEGATIVE
 rand int               max_count;
 rand bit               reset_counter;
 …
 constraint c_max_count {
   max_count >= 1;
   max_count <= 262144;
 }
endclass
```

Having both the register model and configuration object in place, a test writer could randomize the configuration object at the beginning of a test and use the register model's API to program the values into the DUT (Figure 7). Translation from the configuration object's parity and max_count fields into DUT-specific values happens in the test's configuration sequence.

```
class simple_test extends base_test;
 …
 task run_phase(uvm_phase phase);
   …
   env.cfg.randomize();  // randomize the intended configuration
   …
   config_sequence.start(…); // program the configuration to the DUT
  …
endclass

class config_seq extends base_seq;
 task body();
   …
   // translate parity config setting to DUT encoding
   if (env.cfg.parity == ODD)
     env.regmodel.ctrl2.parity.set(0);
   else if (env.cfg.parity == EVEN)
     env.regmodel.ctrl2.parity.set(1);

   // translate max_count config setting to DUT encoding
   translated_max_count = env.cfg.max_count - 1;
   env.regmode.ctrl4.max_count_l.set(translated_max_count[15:0]);
   env.regmode.ctrl5.max_count_h.set(translated_max_count[17:16];

   env.regmodel.ctrl1.update(status);
   …
   env.regmodel.ctrl5.update(status);
```

Figure 7. A test that randomizes the configuration object and writes the DUT registers.

Though not shown here, an equally valid approach is to randomize the register model, and use the random values obtained to program the DUT. In the case of directed test cases, no randomization of either the configuration object or the register model is needed – instead the test scenario will randomize a sequence that has knobs in it to control the programming of the DUT registers.

2.2 Can We Do Without a Configuration Object?

In the simple example shown in Figure 7 we have left out one very important detail, which is how the configuration object will remain in sync with the DUT if, say for example, a register write is done in a test case without using the configuration object's field. This is shown in Figure 8.

```
env.regmodel.ctrl1.counter_en.set(1);
env.regmodel.ctrl1.update(status);
```

Figure 8. Updating only the register model invalidates the configuration object.

Without some extra code, the configuration object's counter_en field would be whatever it was randomized to earlier (or 0 had it not been randomized at all), while the DUT would have a '1' in the counter_en field, leading to an unacceptable risk of divergence. We'll get into the details of what that extra code entails in section 3, but at this point it begs the following question: do we really need to have a separate configuration object if we already have a register model?

The answer to this question is "yes, we do". The main reason for having a separate configuration object comes down to the concept of encapsulation. A proper verification environment is encapsulated such that it can interoperate with any design that implements the protocol or design specification, regardless of that design's register implementation. In the case of a verification IP (VIP) for a standards-based protocol this is even more crucial – the VIP could potentially be needed for multiple DUTs simultaneously within a given company, each with its own register implementation, each implementing the same protocol. We would not want to have to modify the VIP for each different DUT. While it may seem like a lot of extra time and effort to maintain a separate testbench configuration, according to the authors' and their colleagues' experiences across many projects, the reality is that the investment will pay off in both reduced time and effort in the long-run.

Beyond encapsulation, some other consequences of operating without a configuration object are:

- The verification environment will forever be tied to the version of the register model used at that time (e.g. UVM 1.2 register adaptation layer).
- Any changes to the design's register implementation will require changes to the verification environment - for example if a DUT configuration that initially spanned one physical register were changed to span multiple physical registers.

2.3 Operating With a Configuration Object

To avoid the drawbacks outlined above, a better solution is to operate with a configuration object and to put code in place to keep it in sync with the DUT. The mechanism to keep the configuration object up-to-date is passive monitoring - in other words, the configuration object does not get updated from a testbench driver, but instead gets the updates from register model callbacks, with the register model having received its updates from a testbench monitor[1] (Figure 9). This ensures that the configuration object is properly updated regardless of the source of register traffic (e.g. block-level testbench driver, on-chip CPU, off-chip CPU, system-level testbench driver).

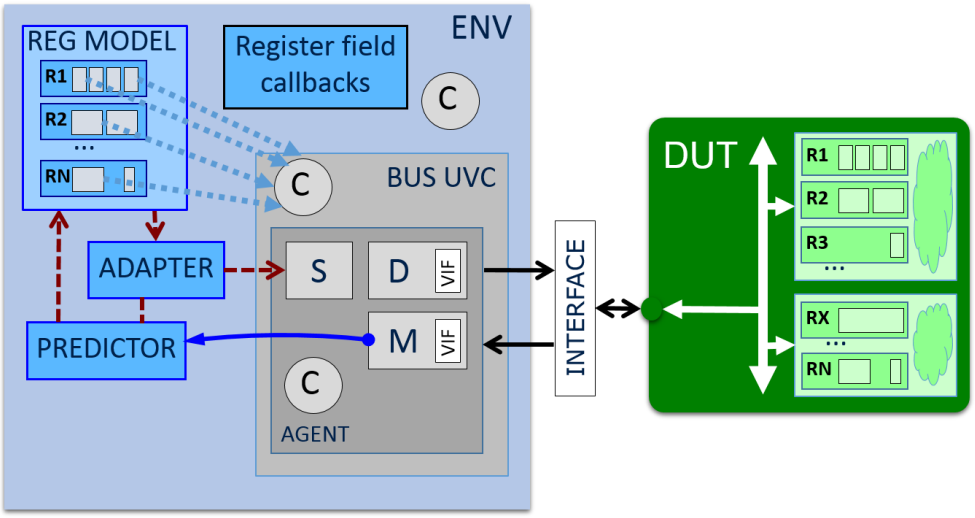

Figure 9. Register model callbacks update the configuration objects.

Note that while most of the configuration object fields have a counterpart in the registers of both the model and the DUT, not all register fields have an equivalent in the configuration object. For example, read-only and interrupt status fields have no counterpart in configuration objects since they do not hold information that is relevant to the configuration of interface protocol UVCs or the enclosing environment. Another important point is that interface-level protocol UVCs should *never* have any reference to a register model (in either the agents or the UVC environment). Doing so would prevent reuse of the UVC with other DUTs implementing the same protocol specification. By contrast, the enclosing testbench environment, which instantiates the UVCs and the register model, will naturally have a reference to the register model. This enclosing testbench environment is responsible for keeping the testbench in sync with the DUT, by making use of the register model. This is discussed in the next section.

3. Keeping the Configuration Object In Sync with the DUT

We now turn our attention to nuts and bolts of what's needed to make the configuration object get updated whenever there is a change to the DUT's configuration. This is accomplished by using register model callbacks that are executed when a register access is detected.

As described in [1], the post_predict method of the uvm_regs_cbs callback class can be used to passively monitor changes to DUT registers. The callback is executed by uvm_reg_field::predict after any observed read or write operation, irrespective of the source of the bus traffic, including back-door access (for more details on the use of UVM register model callbacks for passive monitoring see [1]). As such, we can implement the post_predict callback on each of the register fields of the UVM register model that contain configuration information. In the body of the post_predict callback, we assign the value being written to the register field to the corresponding field of the configuration object. This is done using the following steps:

1. Define the callback classes for each DUT configuration register field
2. Create the field callback instances in the environment
3. Assign each field callback to a register field in the register model

The code in Figure 10 corresponds to step 1, and is in the form of a macro so that it can be easily applied to each register field in the DUT. The example in Figure 10 is for the simple case of a 1-bit

configuration field[1]. Step 2 is shown in Figure 11, and step 3 in Figure 12. Recall that macros have global scope in SystemVerilog, and therefore the macro code presented in the following figures could go anywhere in the testbench. However, for ease of readability it is recommended to put it all together within a single file dedicated to register model callbacks.

```
// define macro for adding callback classes to register fields
`define MY_REG_CB_CLASS(fieldname) \
class my_``fieldname``_cb extends uvm_reg_cbs; \
 block_config config_obj;\
 …
 virtual function void post_predict(input uvm_reg_field  fld, \
                                    … );\
   if (kind == UVM_PREDICT_WRITE) begin \
     config_obj.fieldname = value;\
 endfunction: post_predict \
endclass: my_``fieldname``_cb

// define the callback classes for each DUT register field
`MY_REG_CB_CLASS(counter_en)
…
```

Figure 10. Step 1: Define the callback classes for the fields.

```
class block_env extends uvm_env;
 …
 // register callbacks
 my_counter_en_cb counter_en_cb = new();

 …
```

Figure 11. Step 2: Create the field callback instances in the environment.

[1] More complex examples can be found in the sample code download accompanying this paper, dealing with enumerated types and configuration fields that span multiple registers.

```
// define macro for adding callbacks to register model fields
`define MY_ADD_CB(cb, regpath) \
 cb.config_obj = cfg; \
 …\
 uvm_reg_field_cb::add(regmodel.regpath, cb);

class block_env extends uvm_env;

 function void build_phase(uvm_phase phase);
   // Assign each field callback to a register field
   `MY_ADD_CB(counter_en_cb, ctrl1.counter_en)
   …
 endfunction: build_phase
```

Figure 12. Step 3: Assign each field callback to a register field in the register model.

3.1 Designs that Change Pseudo-Static Configurations on Triggers

Recall that the configuration object shown in Figure 6 had one field per DUT configuration field. While this is sufficient for many designs, it does not support those designs that have a delay between the time at which DUT configuration registers are updated with new data, and the time at which the DUT makes use of that new configuration. This delay can result from a design choice or protocol directive to make the previous configuration remain active until all relevant registers have been written. Then, with a single trigger (usually a write to a "trigger" register field), the DUT replaces the old configuration with all of the newly programmed register values. This approach eliminates any strange behavior that would result from having a mix of old and new configurations both active at the same time. This topic is discussed in [1] from the perspective of keeping the register model in sync with the DUT. In this paper we approach the topic from the perspective of keeping the testbench configuration in sync with the DUT. To handle such designs, we add what is called a "pending" field to each field of the configuration object (see Figure 13).

```
class block_config extends uvm_object;

 rand bit  counter_en_pending;
 bit       counter_en;

 rand my_parity_enum    parity_pending;
 my_parity_enum         parity;
 ...
endclass
```

Figure 13. Add pending fields to configuration object for synchronization.

Note that we have done the following to the configuration object:

- Duplicated every field and added the suffix "_pending"

- Made these pending fields “rand” and removed “rand” from the other fields

There are two reasons for using this pending/actual approach: (1) to make it easy for the test writer to change configurations, and (2) to ensure that the testbench doesn’t use the configuration until the proper moment.

```
class simple_test extends base_test;
 …
 task run_phase(uvm_phase phase);
   …
   env.cfg.randomize();       // randomize the pending fields
   config_sequence.start(…); // program the configuration to the DUT
   …
class config_seq extends base_seq;
 task body();

   env.regmodel.ctrl1.counter_en.set(env.cfg.counter_en_pending);
   env.regmodel.ctrl1.update(status);
   …
   env.regmodel.trig_reg.trigger.set(1);
   env.regmodel.trig_reg.update(status); // trigger config change
```

Figure 14. Tests can change configurations and ensure they aren’t used too soon.

When using the active mode style of operation where the register model is randomized prior to configuring the DUT, the pending fields are not needed, but do no harm by being present.

When using the active mode style of operation where the configuration object is randomized prior to configuring the DUT (as shown in Figure 14) the pending fields can be interpreted as “intended configuration” values. The pending fields hold the value that the DUT is programmed to use *but is not yet using*. This goes back to the trigger concept discussed earlier. At the time the trigger is written, the pending fields are copied to the actual fields, keeping the configuration object in sync with the DUT. This can be accomplished with a callback on the trigger field, similar to what was done

with the other configuration fields (see Figure 15).

```
class my_trigger_cb extends uvm_reg_cbs;
 block_config config_obj;

 function void post_predict(input uvm_reg_field  fld,
                            …);
   if (kind == UVM_PREDICT_WRITE) begin
     if (value == 1)
       config_obj.counter_en = config_obj.counter_en_pending;
       config_obj.parity     = config_obj.parity_pending;
       …
   end
 endfunction
endclass
```

Figure 15. Callback on trigger field assigns configuration object pending fields to actual fields.

Figure 16 shows the timing of the different DUT and testbench configuration fields when operating in active mode (e.g. as in Figure 14 where a test makes assignments to the pending fields by calling cfg.randomize()). Figure 17 shows the timing when operating in passive mode, that is, when there is a monitor that takes notice of bus activity going to those registers and executes the post-predict callback to assign to the pending fields.

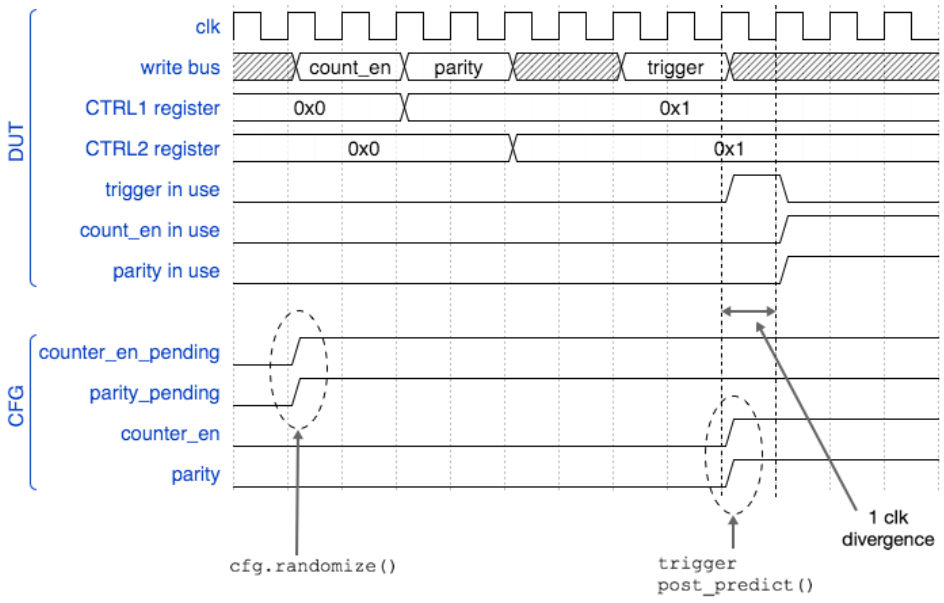

Figure 16. Timing of active mode operation.

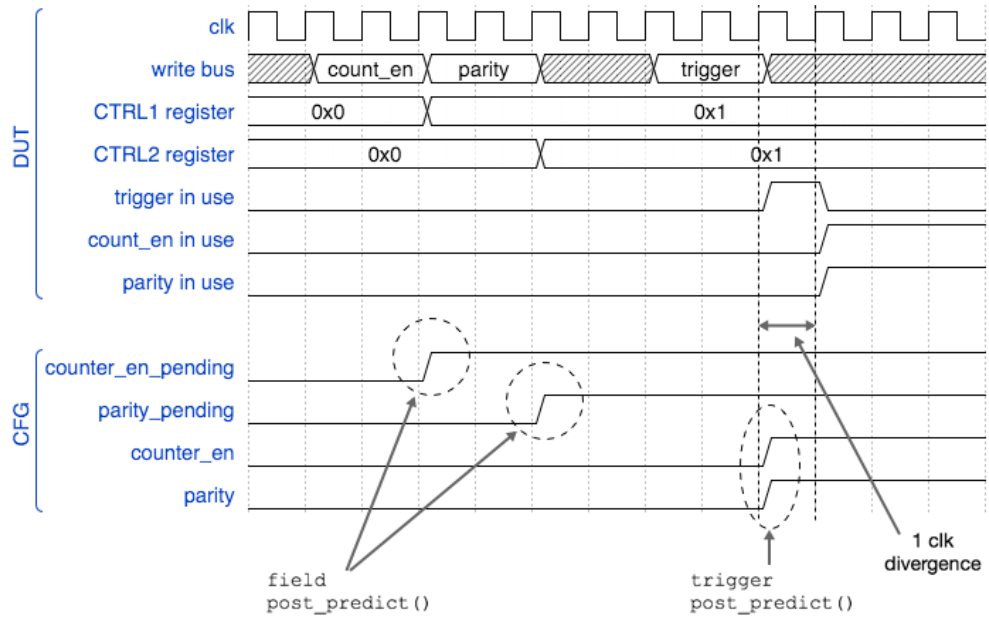

Figure 17. Timing of passive mode operation.

The moment at which the testbench learns about the trigger (via the trigger's post_predict callback shown in Figure 15) is different than the moment at which the DUT makes use of the trigger, leading to one clock cycle of divergence between the DUT and testbench configuration (as highlighted in Figure 16 and Figure 17). Furthermore, in many designs the configuration register signals experience a clock-domain-crossing (CDC) from the register clock domain to the system clock domain (see Figure 18). This leads to an even greater period of divergence between the DUT and register configurations.

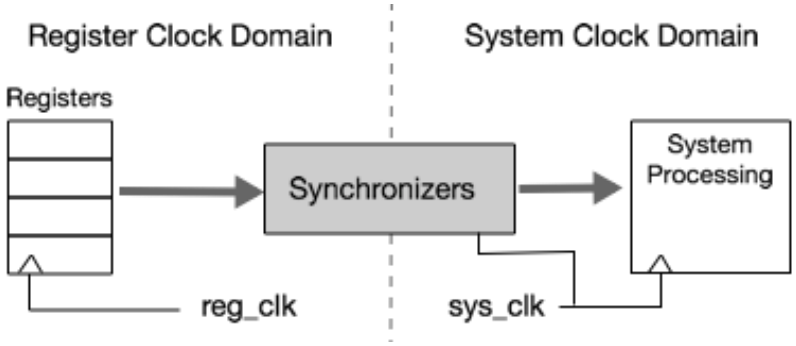

Figure 18. Clock domain crossing from registers to system.

As long as the testbench can hold off sending stimulus during this window of divergence, everything is fine. This is usually the case for pseudo-static configurations. In the case of dynamic

configurations, such momentary divergence could be unacceptable for some designs. Deciding whether or not the divergence is tolerable in a given project is a skill that requires some experience, and is beyond the scope of this paper. In the next section we explore what happens in the case of dynamic configurations where any momentary divergence between the testbench and DUT configurations is unacceptable.

3.2 What About Designs With Dynamic Configurations?

Let's consider a simple counter DUT as shown in Figure 19. The design has two dynamic configuration fields counter_en and reset_counter, and one pseudo-static configuration field max_count. Fields counter_en and max_count experience a CDC from the register clock domain to the system clock domain, while reset_counter does not.

```
module counter (clk,reg_clk, rst_n, count_val);

  …
  always @(posedge clk or negedge rst_n) begin
    if (!rst_n || ctrl3_reg.reset_counter) begin
      count <= 0;
    end
    else if (ctrl1_reg_resync.counter_en &&
             count < max_count) begin
      count <= count + 1;
    end
  end
endmodule: counter
```

Figure 19. RTL code for a simple counter design.

If the project requirements call for cycle-by-cycle comparison between expected and actual counter values, the testbench will need to be in perfect sync with the DUT's dynamic configuration settings[2].

One way of achieving this synchronization (which we are not advocating in this paper however) is to have the testbench attempt to model the CDC of the dynamic configuration signals from the register clock domain to the system clock domain. In our opinion this approach is both time consuming and fragile, and is not recommended. Another approach, which is invalid in the authors' opinions, is to snoop the configuration signals directly from the system clock domain RTL signals themselves. The problem with this approach is that it could mask important bugs in the DUT. For example, imagine if a test were to try programming a 1 into field counter_en, but because of some bug, value 0 actually got written; or if the test did not write a 1 to counter_en, but because of a bug, the DUT started counting anyway. In both of these instances the testbench would just believe whatever the DUT was doing (start counting when the DUT starts, stop counting when the DUT stops) and tests would have falsely passing results. With the snooping approach the verification environment *cannot* achieve its objective of guaranteeing a bug-free DUT.

[2] One DUT encountered by the authors captured incoming data to a memory using dynamic configuration fields "start capture" and "stop capture"; any divergence between the DUT and testbench configurations would result in mismatches between the expected and actual memory contents.

Before getting into the details of the code we propose to handle this situation, we need to first look at the timing of how things work in a testbench that has a configuration object, a register model, and a DUT with dynamic configuration fields experiencing CDCs. In Figure 20 we see that the post_predict callback is executed at the time that the register contents change value in the register clock domain (as indicated by counter_en_pending changing to 1); this is because the register model's passive prediction scheme will invariably be implemented this way. Next we see that some time that elapses before the RTL register value is resynchronized to the system clock domain, where it can be used by the DUT (as indicated by "counter_en in use" changing to 1). So as in the situation with the trigger register shown earlier, the code presented up until this point suffers from a window of divergence between the testbench and DUT configurations for this dynamic configuration field.

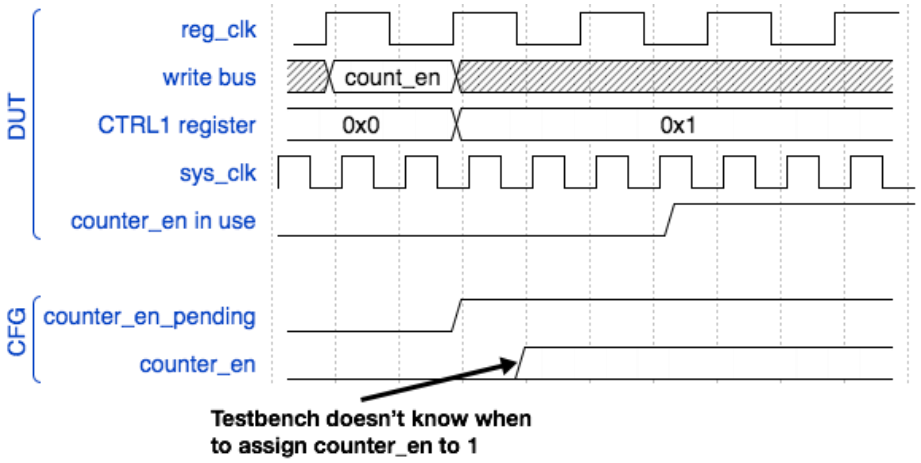

Figure 20. Testbench does not know when a new configuration field takes effect.

In the next section we provide a solution for making the configuration object be in sync with the DUT's dynamic configurations, so that the testbench knows exactly when a new configuration becomes in use.

4. Synchronizing the Testbench with DUT Dynamic Configurations

We now present a detailed solution for keeping the testbench's configuration object in sync with the DUT's dynamic configurations. Note that all of the code presented in this section also requires an environment containing the register model callback code presented in section 3, as well as the pending fields in the testbench's configuration object.

Recall that our goal is for the testbench to make use of the new dynamic configuration value at the exact same moment that the DUT makes use of it. In terms of code, this comes down to making the assignment from the configuration object's pending field to its actual field happen at the same simulation time that the DUT's "in use" signal changes (see Figure 20). To accomplish this, we start by adding a task that monitors changes to the DUT's dynamic configuration field signals in the system clock domain (see Figure 21). This code uses a helper macro SYNCHRONIZE_CONFIG_FIELD shown in Figure 22.

```
class block_config extends uvm_object;
  …
  task keep_config_in_sync_with_dut();
    `SYNCHRONIZE_CONFIG_FIELD(counter_en)
    `SYNCHRONIZE_CONFIG_FIELD(parity)
    …
  endtask
  …
  function new(…)
    …
```

Figure 21. Use a task to monitor when the DUT updates any new value.

Upon any change to the RTL signal of the configuration field in the system clock domain, task keep_config_in_sync_with_dut assigns the configuration object's pending field to the actual field (see Figure 22). It then checks the RTL value against the configuration object's field value. This check can save a lot of time when trying to diagnose the cause of a failing simulation that resulted from a divergence between the testbench and DUT configurations. Note that the code in Figure 22 assumes a virtual interface exists in the environment (named "vif") with paths to the RTL register signals in the system clock domain.

```
`define SYNCHRONIZE_CONFIG_FIELD(field)\
fork \
 forever begin \
   @(vif.field); \
   field = field``_pending; \
   if (field != vif.field) \
     if (check_register_values) \
       `uvm_error("", $sformatf("Field field in config_obj has value
'h%0h but RTL signal has value 'h%0h", field, vif.field)) \
 end \
join_none
```

Figure 22. Any change in the DUT field prompts the testbench to start using the pending value.

We stress that while this solution involves probing inside the DUT to the RTL signals, it does *not* compromise the integrity of the verification environment because it is only looking at changes in the register contents, not their values. As such, bugs like the one described earlier (test case programs "x", DUT erroneously uses value "y") will be caught. Notice the "fork/join_none" multi-threaded approach, which allows task keep_config_in_sync_with_dut to fork off each thread simultaneously on each DUT dynamic configuration field at the beginning of a simulation.

We have now solved the problem of keeping the configuration object in sync with DUT dynamic configurations fields that experience CDCs (see Figure 23). The sections that follow describe how to handle particular situations that will likely arise when implementing this in a real testbench.

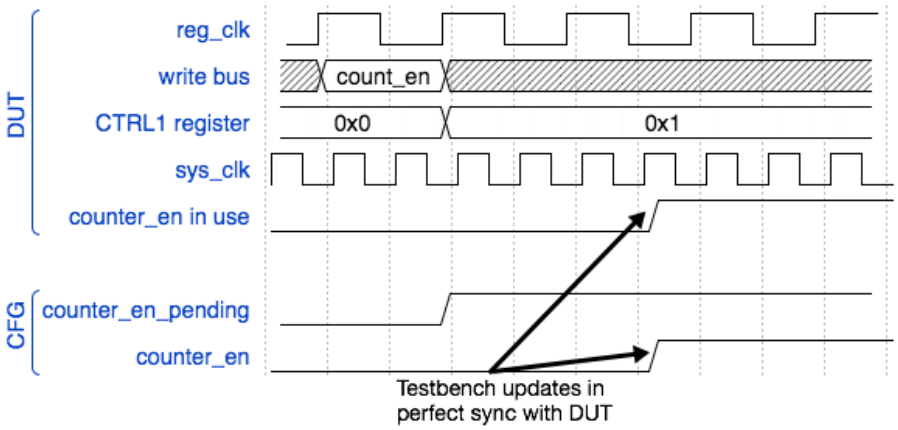

Figure 23. Testbench dynamic configuration field updates at same time as DUT.

4.1 Handling X's at Time Zero

The transition of DUT signals from X to logic 1 or 0 that frequently occurs at the beginning of a simulation would cause the "@vif.field" in Figure 22 to unblock, resulting in unwanted behavior. To handle this, we add a statement that waits until values other than X appear on the RTL signal of the dynamic configuration field in the system clock domain (see Figure 24).

```
`define SYNCHRONIZE_CONFIG_FIELD(field)\
fork \
 forever begin \
   wait (!$isunknown(vif.field));\
   @(vif.field); \
   field = field``_pending; \
   …
```

Figure 24. Avoid issues from X at time zero.

4.2 Handling Signals that Have No Resynchronization

There is an additional challenge if the DUT has dynamic configuration fields that are not resynchronized to the system clock domain. This will cause the register value to be used by the DUT as soon as it changes in the register clock domain. The synchronizing task we defined previously cannot handle this scenario since there would be a race between the post_predict callback updating the pending value and the assignment of the pending value to the actual field by configuration object method keep_config_in_sync_with_dut (both will happen at the same simulation time). Such a race would lead to unpredictable results and potential divergence between the testbench and DUT configurations.

To handle this we use events. The configuration object is modified to add one event field per dynamic configuration field, that triggers at the moment the pending field is updated by the post-predict callback (see Figure 25). By waiting on that event, we can ensure that post_predict has updated the pending field before it gets assigned to the actual field by the SYNCHRONIZE_CONFIG_FIELD macro (see Figure 26 and Figure 27).

```
class block_config extends uvm_object;

 rand bit  counter_en_pending;
 bit       counter_en;
 uvm_event counter_en_changed_evt;
 ...
endclass

`define MY_REG_CB_CLASS(fieldname) \
class my_``fieldname``_cb extends uvm_reg_cbs; \
 block_config config_obj;\
 …
 virtual function void post_predict(input uvm_reg_field  fld, \
                                    … );\
   if (kind == UVM_PREDICT_WRITE) begin \
     config_obj.fieldname``_pending = value;\
      if (config_obj.fieldname !== value)\
        config_obj.fieldname_changed_evt.trigger(); \
   …
 endfunction: post_predict \
endclass: my_``fieldname``_cb
```

Figure 25. Use events to avoid races in signals that are not resynchronized.

```
`define SYNCHRONIZE_CONFIG_FIELD(field)\
fork \
 forever begin \
   wait (!$isunknown(vif.field));\
   @(vif.field); \
   field``_changed_evt.wait_on(); \
   field``_changed_evt.reset();\
   field = field``_pending; \
   if (field != vif.field) \
     if (check_register_values) \
       `uvm_error("", $sformatf("Field field in config_obj has value
'h%0h but RTL signal has value 'h%0h", field, vif.field)) \
 end \
join_none
```

Figure 26. Use events to avoid race conditions between updates and use of pending values.

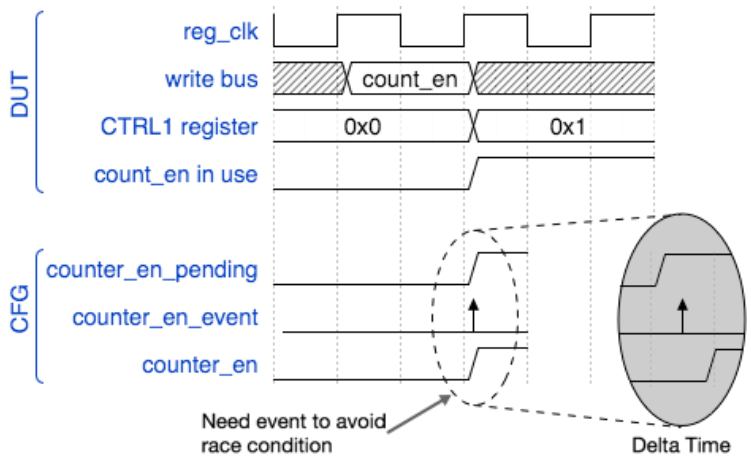

Figure 27. Delta time expansion showing how events prevent race condition on pending field.

4.3 Handling Resets

When a hard reset occurs, all registers in the DUT should return to their default values. We can keep the register model in sync with the DUT by calling regmodel.reset(), which will return all of the register model field values to their defaults. Keeping the configuration object in sync will require some extra consideration though. Recall that the post_predict callback only executes when there is a bus transaction to the register field, and therefore does not execute upon reset. However, on *any* change to the actual RTL field, the configuration object method keep_config_in_sync_with_dut will perform a comparison with the last pending value and the RTL signal of that register. With the code we have presented thus far, such a comparison would fail because the pending field would not have the reset value in it. The solution is therefore to make the configuration object's pending fields get the reset values at the time of the reset, so that this check will pass.

To accomplish this, we write a separate "reset" method for assigning reset values to all pending fields of the configuration object. The environment can call the configuration object's reset method at the time a reset is noticed, and the configuration object and the DUT will be in sync because the pending fields will match the actual RTL values at the time of reset. A good way to do this is to encapsulate the resetting of the register model and configuration object into the run phase of the enclosing UVM environment, so that it will execute in both active and passive operation of the environment (Figure 28).

```
class my_env extends uvm_env;
  ...
  task run_phase(uvm_phase phase);
    super.run_phase(phase);
    fork
      forever begin
        @(negedge tb_vif.rst_n);
        regmodel.reset(); // reset register model
        cfg.reset();      // reset configuration object
      end
    join_none
  endtask: run_phase
```

Figure 28. Use passive monitoring to encapsulate all reset operations.

As a final note, recall the field``_changed_evt we introduced in section 4.2 . In the case of a reset, post_predict does not execute, and therefore this event is never triggered. So the synchronization code needs to be modified to *not* wait for that event in the case of a reset (Figure 29).

```
`define SYNCHRONIZE_CONFIG_FIELD(field)\
fork\
  forever begin\
    wait(!$isunknown(vif.field));\
    @(vif.field);\
    if ( !(!vif.rst_n)) begin\
      field``_changed_evt.wait_on();\
      field``_changed_evt.reset();\
    end\
    field = field``_pending;\
    …
```

Figure 29. Synchronization code must not wait on events on reset.

4.4 Handling Auto-clear Registers

It is common for some register fields to auto-clear to 0 one clock cycle after they are set to 1 by a

write access. When the RTL signal of an auto-clear dynamic configuration register field transitions to 0, the synchronization code cannot wait for the "field changed" event introduced in section 4.2 because the transition to 0 is done by hardware, not by a register access (hence no post_predict callback is executed). Also, when the RTL signal transitions to 1, the pending field of the configuration object needs to be updated to be 0, because that is its expected future value. With this code in place (**Error! Reference source not found.**), the checks on pending values versus actual RTL values continue to work for auto-clear fields.

```
`define SYNCHRONIZE_CONFIG_FIELD(field)\
fork\
  forever begin\
    localparam bit AUTOCLEAR = (`"field`" == "reset_counter");\
    @(vif.field);\
    if ( !((!vif.rst_n) ||\
       (AUTOCLEAR && field == 0))) begin\
      field``_changed_evt.wait_on();\
      field``_changed_evt.reset();\
    end\
    field = field``_pending;\
    …
    if (AUTOCLEAR && field == 1)\
      field``_pending = 0;\
    …
join_none
```

Figure 30. Handling auto-clear registers.

5. Conclusion

This paper has demonstrated the roles of configuration objects and register models, and how they can work together within a verification environment to keep the DUT and testbench in sync. Having a configuration object that is separate from the register model is important for achieving proper encapsulation of the verification environment. Developing a register model, configuration object, and implementing the ideas in this paper to achieve synchronization between the testbench and DUT is hard work, but from the authors' experience, it will save both time and effort over the course of a project.

All code examples in this paper were simulated using VCS 2015.09-SP2 and UVM1.2. Working sample code can be download from http://www.verilab.com/resources/source-code. The sample code demonstrates how to keep the dynamic configuration register fields of a simple counter RTL design (as described in section 3.2) in sync with a UVM testbench, using the register model callback and configuration object synchronization techniques discussed in sections 3 and 4 of this paper. See the README.TXT file under the doc directory of the download for instructions.

6. References

[1] Mark Litterick, Advanced UVM Register Modeling – There's More Than One Way to Skin a Reg, DVCon 2014