Mutable Verification environments through Visitor and Dynamic Register map Configuration

Matteo Barbati, STMicroelectronics, Digital Mixed Asic Division, Via Tolomeo 1, Cornaredo (Milano), Italy (matteo.barbati@st.com)

Alberto Allara, STMicroelectronics, Digital Mixed Asic Division, Via Tolomeo 1, Cornaredo (Milano), Italy (*alberto.allara@st.com*)

Abstract— Creating Mutable Verification Environment able to dynamically change its behavior according to the Register Map configuration allowed us to simplify the scenarios generation while creating more and more complex tests. Extension of existing VIP to add Synchronization Mechanism with Register Model and new capabilities to allow dynamic VIP reconfiguration represents the typical solution to this problem. In this paper we propose an approach based on UVM Register Map Dynamic Configuration and UVM Visitor to create powerful mutable verification environment in a simpler way.

I. INTRODUCTION

Creating a Verification Environment that can change its behavior across different projects and, within the same project, dynamically based on the configuration of the registers of a device register map is becoming more and more a key requirement in the design of complex testbenches for the verification of digital and mixed-signal devices. Mutable verification environments indeed greatly simplify the design of new scenario and the development of more complex tests.

There are several approaches for the implementation of mutable verification environments. The most common solution is based on the specialization of Verification IP capable to intercept the updates propagated by the UVM register model triggered by write or read operations and use such information to execute new behaviors adding new capabilities to the VIP.

In [1] we proposed an approach to automatically generate synchronization events when a read and/or write operation is performed on a specific set of fields. To reach this goal we relied on customization of uvm_reg_field callbacks pre_read, pre_write, post_read and post_write to generate a synchronization event, associated to the change of the register map configuration.

In [3] the authors proposed to use UVM visitors supported by UVM 1.2 ([2]) to easily "mutate" the behavior of VIPs inside a Verification environment according to the needs of the project. Visitors are software design patterns typically used in Software Development. A design pattern is a reusable solution that describes a schema to solve a specific software issue. It is not the specific piece of code that can be instantiated "as is" inside the user application, but provides a way to implement software to solve a specific issue. A Visitor provides a way to separate a piece of code implementing a new behavior from an object structure (for instance the UVM components of a VIP) on which it operates. This approach allows to add new functionalities to existing object structures without the need to modify the structure each time.

II. A COMBINED APPROACH

What we propose in this paper is to combine in some way the two approaches with the ultimate goal to simplify the creation of Mutable verification environment capable to adapt to the changes of the design and its configuration both statically and dynamically

In the original solution proposed in [1] we extended the uvm_reg_field class in order to generate uvm_events in pre_read, pre_write, post_read and post_write callbacks. The events generation was managed through uvm_config_db mechanism that allow to turn on/off the trigger generation according to our needs.

In this new implementation, in order to optimize the code and to reduce the data overhead of the original solution we rely on the usage of uvm_reg_cbs. The uvm_reg_cbs class defines a set of callbacks (pre_read, pre_write, post_read, post_write, post_predict, decode and encode) that can be customized and then associated to a set of uvm_reg_field and/or a set of uvm_reg objects. To reproduce the behavior of the original approach we define an extension of the uvm_reg_cbs class that customizes the pre_read, pre_write, post_read and post_write callbacks and we put in place a simplified configuration mechanism based on an external configuration file managed by the verification environment. Such optimization allows to add the trigger generation only on a subset of the uvm_reg_field or uvm_reg of the Register model, reducing in this way the data overhead of the original solution

The snippet of Figure 1 shows in detail the implemented solution. To simplify the readability, we describe here the flow related to the uvm_reg_field objects. This approach is valid also for uvm_reg classes. First of all, we define an extension of the uvm_reg_cbs class that redefines the pre_read, pre_write, post_read and post_write callbacks, adding the capability to fire synchronization events. We don't use the post_predict callback, even if this callback allows to heavily reduce the code overhead, because this function doesn't allow us to generate events before a read or a write operation. The trigger generation is managed using a list of struct called "field_list" that provides a mechanism to store the configurations related to a specific field regarding the pre_read, pre_write, post_read and post_write trigger generation.

```
typedef struct { bit pre_rd, pre_wr, post_rd, post_wr; }  field_desc;
…
class uvm_reg_trig_cbs extends uvm_reg_cbs;
   `uvm_object_utils(uvm_reg_trig_cbs)
   field_desc fld_list [string];
   uvm_event_pool ep;
   function new(string name = "uvm_reg_trig_cbs");
      super.new(name);
      ep = uvm_event_pool::get_global_pool();
   endfunction
  virtual task pre_read(uvm_reg_item   rw);
     …
  endtask : pre_read
  virtual task pre_write(uvm_reg_item  rw);
     …
  endtask : pre_write
  virtual task post_read(uvm_reg_item   rw);
     …
  endtask : post_read
  virtual task post_write(uvm_reg_item  rw);
      string event_name;
      uvm_event trigger;
      event_data d;
      field_desc fld_d;
      uvm_reg_field  fld;
      $cast(fld, rw.element);
      d = event_data::type_id::create("d");
      fld_d = fld_list[fld.get_name()];
      if(fld_d.post_wr) begin
        event_name = {"post_wr_", fld.get_name()};
        trigger =  ep.get(event_name);
        d.val= fld.get();
        `uvm_info(get_name(), "**** Firing Post Write trigger ****", UVM_NONE)
        trigger.trigger(d);
      end
  endtask : post_write
endclass
```

Figure 1 – Extendend uvm_reg_cbs

In the top level environment we define a mechanism to simplify the mapping between register map fields and callback events as reported in the snippet of Figure 2. We populate the “field_list” data structure starting from a text configuration file as show in the example of Figure 3. In this example file we enabled the pre_write and post_write event generation on field1, pre_read and post_read event generation is enabled on field2 and post_read and post_write event generation is enable on field3.

```
class top_sve extends uvm_env;
…
  virtual function void connect_register_callbacks();
    field_desc tmp;
    uvm_reg_trig_cbs cbs;
    uvm_reg_field f;
     …
    fd = $fopen({"./field_config.dat"}, "r");
    while (! $feof(fd)) begin
      $fscanf(fd,"%s %b %b %b %b\n",fldname, pre_rd, pre_wr, post_rd, post_wr);
      tmp.pre_rd = pre_rd;
      tmp.pre_wr = pre_wr;
      tmp.post_rd = post_rd;
      tmp.post_wr = post_wr;
      field_list[fldname] = tmp;
    end
    $fclose(fd);
    cbs = uvm_reg_trig_cbs::type_id::create("cbs");
    cbs.fld_list = field_list;
    foreach(field_list[key]) begin
      field_found = 1'b0;
      foreach(regmodel[i]) begin
        if(regmodel[i] != null)
          f = regmodel[i].get_field_by_name(key);
        if(f != null) begin
           uvm_callbacks#(uvm_reg_field, uvm_reg_trig_cbs)::add(f, cbs);
          field_found = 1'b1;
        end
      end
      if(field_found == 1'b0) begin
        `uvm_warning(get_full_name(), $psprintf("field %s does not exist", key))
      end
    end
  endfunction : connect_register_callbacks
…
  virtual function void connect_phase(uvm_phase phase);
```

```
      super.connect_phase(phase);
       …
      connect_register_callbacks();
    endfunction : connect_phase
  endclass: top_sve
```

Figure 2 – uvm_reg_cbs connection

```
  field1 0 1 0 1
  field2 1 0 1 0
  field3 0 0 1 1
```

Figure 3 – Example of field_config.dat file

Finally, based on the proposal of [3] we decided to use a UVM visitor class to add synchronization capability to legacy VIPs. More in details, in the original solution of [3], UVM_visitor was used to add debug information to the VIPs of the Verification environment. In our proposal we use the uvm_visitor to add new functionality to the VIPs and in particular we add the capability to intercept the event generated by the Register Model and to update the VIP behavior accordingly with the update done in the Register model

The two main actors are an adapter (called basic_adapter) and an uvm_visitor (called event_manager_visitor). These classes are used to guarantee a correct traverse of the uvm_components present inside the Verification Environment. The adapter provides a way, through accept function, to allow the visitor to visit the components and to add new capabilities.

In our case the visitor implements the function visit required to all uvm_visitor and a set of tasks. These tasks are started by the visit function and are deputed to intercept the uvm_events triggered by the register model and to apply the proper changes to the existing VIPs.

The snippet of Figure 4 shows the details related to event_manager_visitor and basic_adapter classes. The event_manager_visitor implements tasks that are able to drive interface signals of the visited component (vip_monitor in the example) according to the access done on a specific register map field. In each of this new task we wait for the synchronization event from register map, and according to this trigger we react changing the behavior of the visited component.

```
  class event_manager_visitor extends uvm_visitor;
    virtual function void visit(uvm_component node);
      if (node.get_object_type() == vip_monitor::type_id::get()) begin
        fork
          manage_field1_pre_wr(node);
          manage_field1_post_wr(node);
          …
        join_none
      end
       …
    endfunction
    virtual task manage_field1_pre_wr(uvm_component node);
      …
      vip_monitor mon;
      uvm_reg_field f;
      event_data d;
      uvm_object obj;
      event_name = "pre_wr_field1;
```

```
        $cast(mon, node);
        ep = uvm_event_pool::get_global_pool();
        ev = ep.get(event_name);
        forever begin
          ev.wait_trigger_data(obj);
          $cast(d, obj);
          mon.vif.sig1 = d.val;
         `uvm_info("VISITOR", $psprintf("%s write event received. Value = %d", event_name, d.val), UVM_NONE)
        end
      endtask
      virtual task manage_field1_post_wr(uvm_component node);
        …
      endtask
      function new(string name = "event_manager_visitor");
        super.new(name);
      endfunction
    endclass
    class basic_adapter extends uvm_visitor_adapter;
      virtual function void accept(uvm_component s, uvm_visitor v, uvm_structure_proxy#(uvm_component) p, bit
invoke_begin_end=1'b1);
        if(invoke_begin_end)
          v.begin_v();
        v.visit(s);
        if(invoke_begin_end)
          v.end_v();
      endfunction
      function new (string name = "");
        super.new(name);
      endfunction
    endclass
```

Figure 4 – event_manager_visitor and basic_adapter classes

Finally, in the top level environment we instantiate and connect both the basic adapter and the visitor as reported in Figure 5.

```
    class top_sve extends uvm_env;
     …
      virtual function void start_of_simulation_phase(uvm_phase phase);
       event_manager_visitor event_manager_v;
       uvm_top_down_visitor_adapter adapter;
       uvm_component_proxy proxy;
       event_manager_v = new("event_manager_v");
       adapter         = new("adapter");
       proxy           = new("proxy");
       adapter.accept(this, event_manager_v, proxy);
      endfunction : start_of_simulation_phase
    endclass: top_sve
```

Figure 5 – Visitor instantiation in top level environment

An UML class diagram describing the UVM Visitor extension proposed is reported in Figure 6

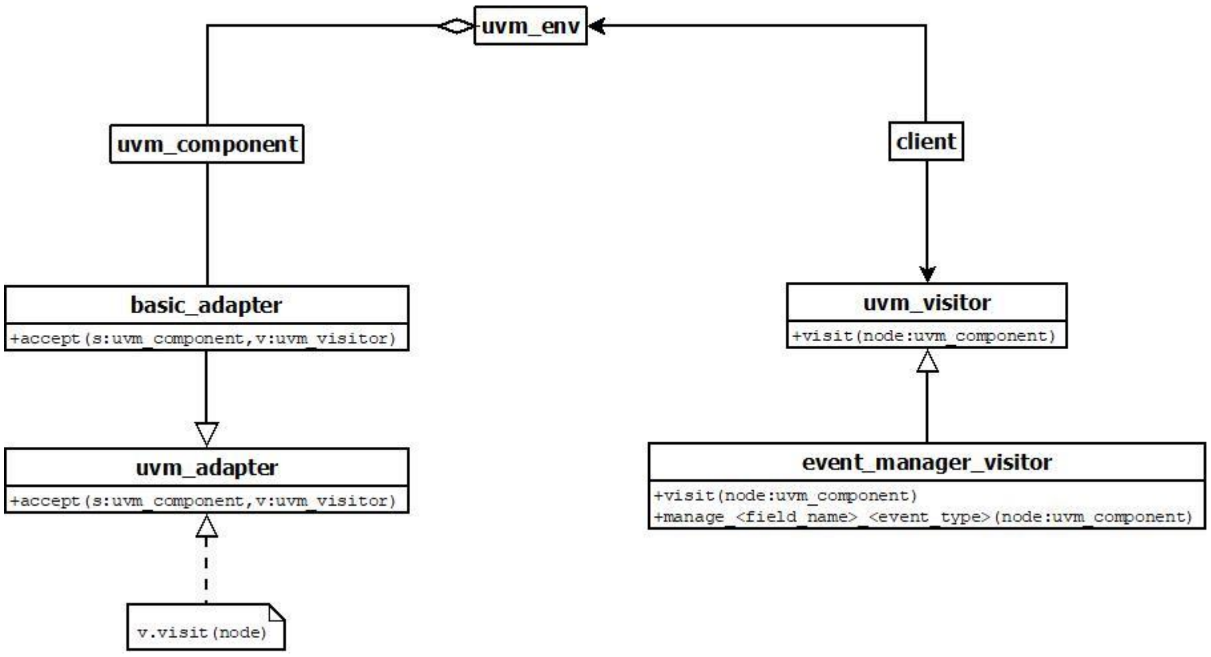
uvm_env
uvm_component
client
basic_adapter
+accept(s:uvm_component,v:uvm_visitor)
uvm_visitor
+visit(node:uvm_component)
uvm_adapter
+accept(s:uvm_component,v:uvm_visitor)
event_manager_visitor
+visit(node:uvm_component)
+manage_<field_name>_<event_type>(node:uvm_component)
v.visit(node)

Figure 6 - UML Class Diagram

An UML Sequence Diagram describing the interactions between Register Model events and Visitor is reported in Figure 7. The interations between the Trigger_cbs and the Event_manager_visitor/VIP represent the event generation related to field. Once an event is fired by the Trigger_cbs it triggers the Event_manager_visitor that updates the VIP behavior according to the performed operation.

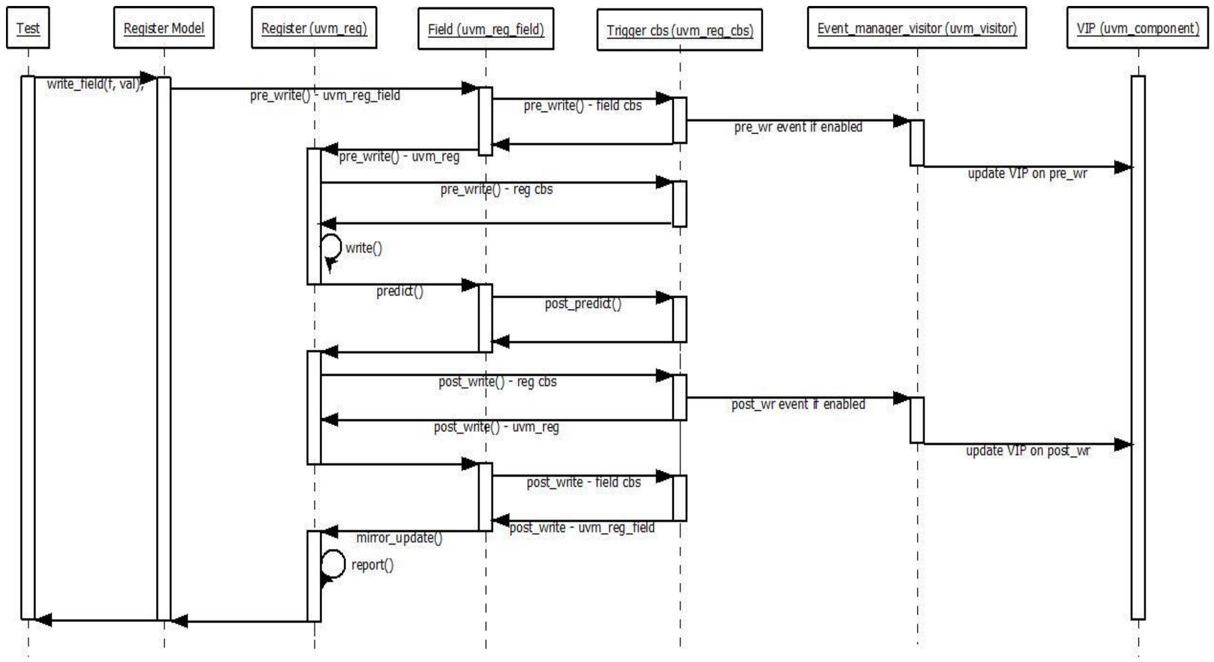

Figure 7 – UML Sequence Register Model/Visitor Interactions

III. USE CASE

In STMicroelectronics the proposed approach has been used inside our verification environments to verify SerDes IPs. We applied the approach for instance with a Verification component that allow to estimate the frequency error of a TX or RX line expressed in PPM or Part Per Million. The VIP configured as “Passive UVC”, have to be turned on/off according to a specific Register Map configuration that enable a functionality inside our DUT. Originally the tests, using this VIP, required to explicitly turn on/off the VIP capabilities putting at 1 or 0 on “en” signal defined inside the VIP Interface based on the assertion/de-assertion of a register field of the DUT. The VIP must start after the DUT feature is enabled and must stop before the DUT feature is turned off. The Figure 8 reports the portion of code implementing such behavior

```
regmap_write_field("en_count_tx",1'b1);
 // Enable PPM estimator
force tb_top.ppm_tx.en = 1'b1;
#(100us);
// Disable PPM estimator
force tb_top.ppm_tx.en = 1'b0;
regmap_write_field("en_count_tx",1'b0);
#(1us);
current_ppm = tb_top.ppm_tx.actual_ppm;
```

Figure 8: explicit VIP enable/disable

The implementation of our approach required to enable pre_write and post_write trigger generation on register field “en_count_tx” using the optimized implementation of [1] based on uvm_reg_cbs. Then we defined a Visitor that add the automatic enable/disable of the PPM computation when the pre_write or post_write event associated to “en_count_tx” is fired. In order to guarantee the correct timing between the DUT feature en_count_tx and the VIP enable we assert the PPM capability when a post_write event is detected and the VIP is in a “disabled” state. On the contrary, we turn off the PPM capability when a pre_write event is detected and the VIP is in “enabled” state.

The code of the test of Figure 8 once the proposed approach is in place can be simplified as shown in Figure 9.

```
// Automatically Enable PPM estimator with write at 1 on en_count_tx
regmap_write_field("en_count_tx1",1'b1);
#(100us);
// Automatically Disable PPM estimator with write at 0 on en_count_tx
regmap_write_field("en_count_tx",1'b0);
#(1us);
current_ppm = tb_top.ppm_tx.actual_ppm;
```

Figure 9: Automatic enable/disable of VIP PPM measures

IV. CONCLUSIONS

Mutable environments are a must of modern verification environments. They allow to create more complex scenario in an easiest way. In this paper we have proposed an approach that combines mechanisms of dynamic register map configuration and Visitor design pattern.

We also reached the goal to reduce the data overhead related to the original proposal of dynamic register map configuration [1], adding extra variables and events only on the required fields and registers. In addition, with this optimization we are able also to reduce the number of generated events, thanks to the capability to fire synchronization events also on selected uvm registers.

Regarding the UVM visitor, this Design Pattern allow us to easily extend the behavior of 3-rd party and/or legacy VIPs adding the Register Map Synchronization feature with few extra SystemVerilog lines, reducing in this way the risk to inject errors in pretty stable VIPs.

Acknowledgment

We would like to thank and acknowledge the verification engineers of our team: Andrea Masi, Rinaldo Franco, Massimo Vincenzi and Matteo Ferranti for their contributions to create a positive and collaborative context where we developed our proposal.

References

[1] M. Barbati and A. Allara "UVM Register Map Dynamic Configuration," DVCon Europe 2018

[2] Accellera, "UVM Class Reference Manual, v1.2" , https://www.accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf

[3] D. M. Tomušilović, "Extending functionality of UVM components by using Visitor design pattern," DVCon Europe 2018