A 360 Degree View of UVM Events – A Case Study

Deepak Kumar E V[1], Sathish Dadi[2], Vikas Billa[3]
[1]Lead Engineer, [2 3]Sr. Verification Engineer
elitePLUS Semiconductor Technologies Pvt. Ltd
2nd Floor, #991 & #992, 5th Main, 7th Sector, HSR Layout
Bengaluru, Karnataka – India 560102

Abstract **- UVM saw a great adoption in the past few years and has been accepted as the most popular methodology choice by the verification community. Since its inception, the community and its users have been constantly adding and enhancing its library, thereby offering rich and flexible data-structure choices to the test-bench developers. An UVM event is one such library class added and has seen tremendous enhancements due to its widespread applications and usage. This paper attempts to give a 360-degree view of the UVM events.**

I. INTRODUCTION

In a verification environment, the test-bench components often communicate in a synchronized manner, to effectively implement the time accurate checks. System Verilog events are dedicated and are widely used data types to achieve the desired synchronized communication between the components. UVM library has a built-in dedicated class using System Verilog events which has broaden the application and usage of event based communication.

The authors have been making the best use of uvm_event class to implement reusable verification environments. This paper is a collective case study of projects executed by the authors, highlighting the difference usage and benefits of uvm_event class over System Verilog events.

II. SYSTEM VERILOG EVENTS

A System Verilog event is a data type which has no storage. An identifier declared with data type *event* is called a named event. As represented in Figure 1, named event can be triggered using the "**->**" operator, and an event triggering occurrence can be captured by using "**@**" operator or inbuilt .triggered method. The named event and event control provides the communication and synchronization between two or more concurrent process.

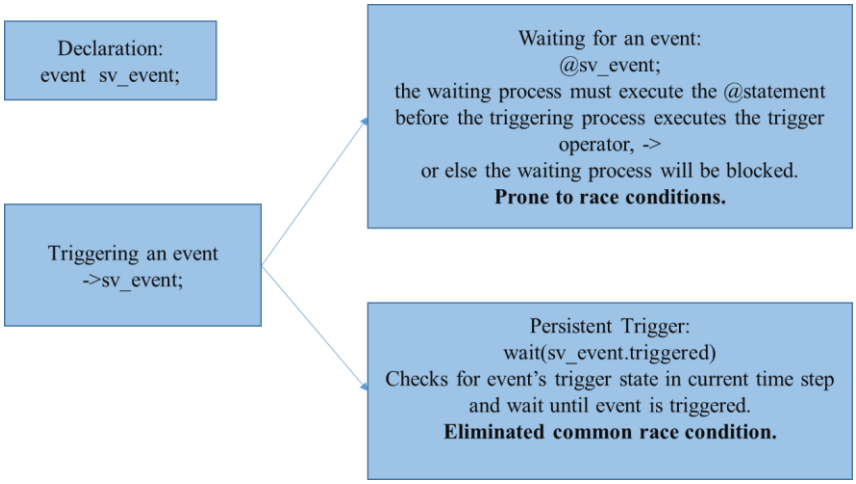

Figure 1 : System Verilog event

III. UVM Events

uvm_event is a parameterized wrapper class created using System Verilog event construct. It provides some additional services such as setting call-backs, data delivery, and maintaining number of waiters, on-off state and timing information.

The uvm_event class is an extension of the abstract uvm_event_base class, Figure 2 shows the hierarchy of uvm_event in UVM class library:

```
class uvm_event#(type T=uvm_object) extends uvm_object;
```

The optional parameter T allows the user to define a data type which can be passed during an event trigger.

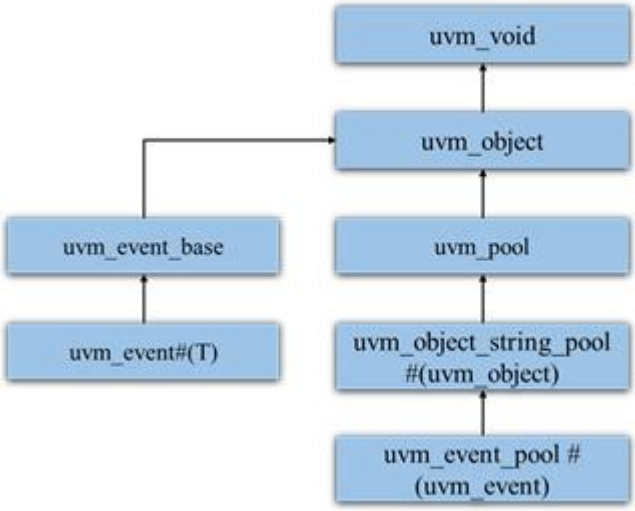

Figure 2: uvm_event class hierarchy [3][4]

Frequently used uvm_event methods are:

S. No	Method	Description
1	trigger(T data = null)	Trigger an event. You can pass data as parameter to trigger function which is waiting to be triggered.
2	Wait_trigger()	Waiting for an event. A system Verilog equivalent of @ operator.
3	wait_ptrigger()	Waiting for a persistent trigger. A system Verilog equivalent of .triggered. Avoids any race condition.
4	Get_trigger_data()	Gets data if any provided by the last call to trigger.
5	wait_trigger_data(output T data)	Call and returns get_trigger_data method value followed by wait_trigger.
6.	wait_trigger_pdata(output T data)	Call and returns get_trigger_data method value followed by wait_ptrigger.

Table 1 uvm_event methods

III. A CASE STUDY

1. RESET aware testbench

Handling on-the-fly reset is one of the challenges in testbench design. The UVM methodology does not define how an on-the-fly reset must be handled. The reset is a major disruptive event which can occur at any point of time, it's very important to ensure that the chip exists out of reset and resumes normal operation without any issue.

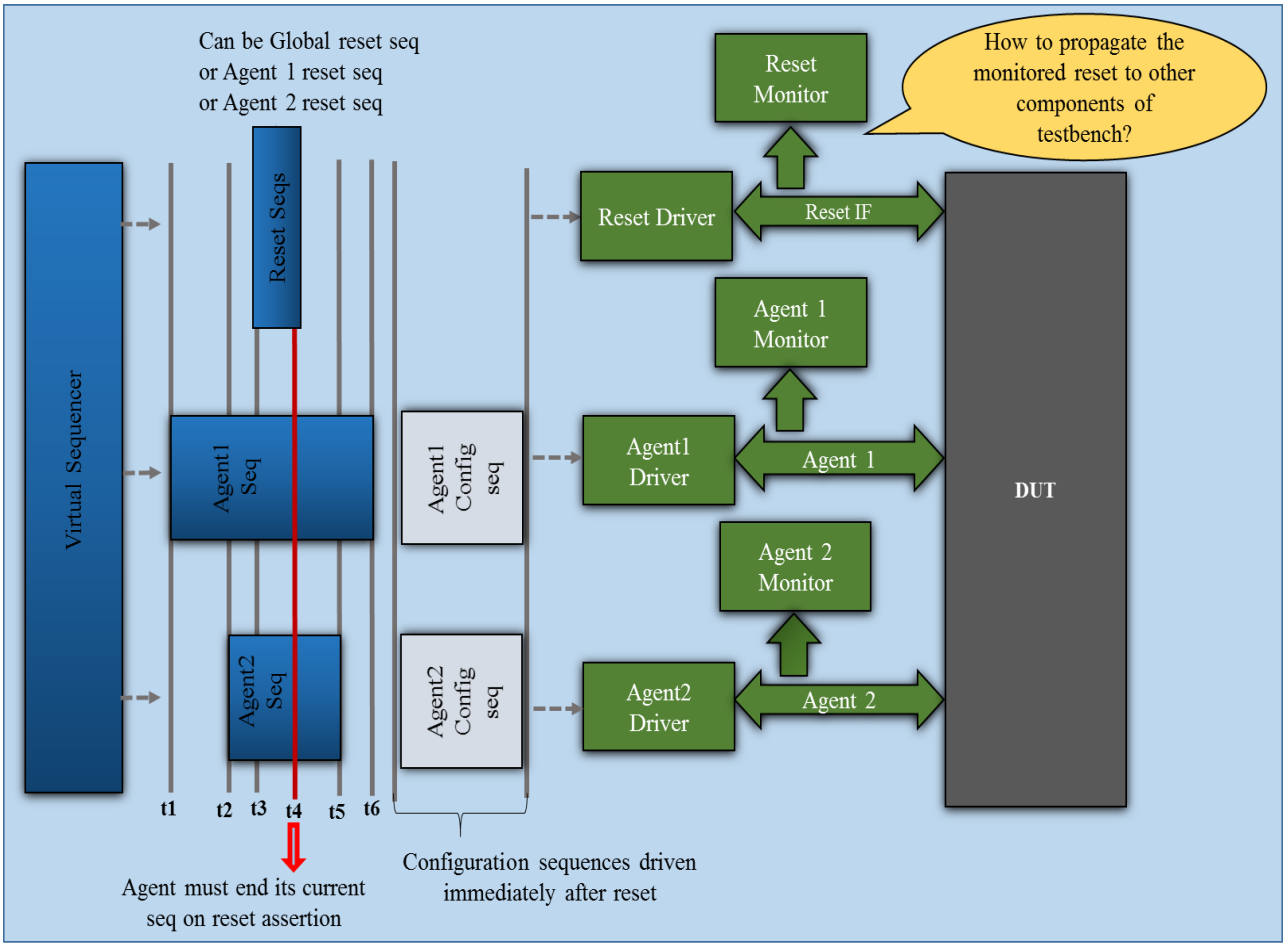

Figure 3: Reset aware testbench

Typically, we will have multiple resets in any given SoC which adds complexity to reset verification as the tester need to ensure all the IP/modules in the chip reacts only to the desired resets and ignore the rest. On-the-fly reset must be taken into account in the test-bench and any house-keeping must be made to handle the resetting scheme carefully.

Figure 3 is a representation of stimulus life-span and flow of reset aware test-bench. Apart from reset agent, the environment has two other agents which can be reset individually by applying agent1 reset or agent2 reset respectively or simultaneously by applying a global reset. Reset agent from Figure 3 will be continuously monitoring the reset interface and triggers the reset event on successful capture of any of the

above mentioned resets. All other components in the test bench will be waiting for a respective reset event to trigger.

Here is how the 4 major components Drive, Monitor, Scoreboard and Sequences must behave on capturing the reset event:

Driver:

- Should not drive data under reset and wait until reset is removed. (t4 from Figure 3)
- Should stop driving the bus and send item_done on reset application. (t4 from Figure 3)
- Must complete the transaction if there was no reset while the transaction is in progress. (t5 and t6 from Figure 3)

Monitor:

- Should be monitoring the bus and trigger report error for conditions on bus which are not expected under reset. (t4 from Figure 3)
- Should treat the bus data as invalid if a reset is applied in between a transaction. (t4 from Figure 3)

Scoreboard:

- On application of the reset, all FIFO's must be flushed. (t4 from figure 3)

Sequences:

- Immediately after reset, the configuration sequence should be driven before driving any other sequence. (t6 from figure 3)

Reset is an important state of an IP and hence the test-bench needs to be designed to accommodate and handle this state. Here are simple steps with code samples to make your test-bench is reset-aware:

Triggering interrupts from sequences: As part of stimulus

Triggering global reset event:

```
global_reset_ev = cfg.event_pool.get("global_reset");
global_reset_ev.trigger();
```

Triggering an Agent1 reset event:

```
agent1_reset_ev = cfg.event_pool.get("agent1_reset");
agent1_reset_ev.trigger();
```

Triggering an Agent2 reset event:

```
agent2_reset_ev = cfg.event_pool.get("agent2_reset");
agent2_reset_ev.trigger();
```

In reset aware components: Components will have to wait for respective interrupts and implement their reset behavior upon successful reception of the interrupts.

```
forever begin : Reset_service
  fork   : capture_reset
    begin
      agentX_reset_ev.wait_ptrigger;
    end
   begin
      global_reset_ev.wait_ptrigger;
    end
  join
  reset_procedure();
end
```

On the fly resets can be made better controlled by using a customized uvm_phases. The discussions in this paper are restricted to uvm_event. [2]

2. **RAL :**

Register Abstraction Layer is one of the popular features in UVM. UVM_RAL provides some standard callback methods like pre_write, pre_read, post_write, post_read, post_predict, encode and decode methods. By using these callback methods the desired functionality can be added at the required position during register access.

uvm_event can be used for communication from register callback class to other components like scoreboard, reference model, sequences etc. Adopting the uvm_event for RAL to TB communication requires minimal testbench code changes and also the data delivery through uvm_event helps a great deal especially in score-boarding and reference modeling.

Creating a register callback class:

```
class trans_capture_cb extends uvm_reg_cbs;
uvm_event#(uvm_reg_item) wr_event;
uvm_event#(uvm_reg_item) rd_event;
  …..
  function new( …);
   …..
  endfunction

  virtual function void post_write(uvm_reg_item rw);
    wr_event = cfg.event_pool.get("reg_write_event");
    // Triggering write event
    wr_event.trigger(rw);
 endfunction
 virtual function void post_read(uvm_reg_item rw);
   rd_event = cfg.event_pool.get("reg_read_event");
   // Triggering write event
   rd_event.trigger(rw);
 endfunction
endclass : trans_capture_cb
```

The user defined trans_capture_cb callback class is created by extending the uvm_reg_cbs class. The post_write and post_read methods are overridden with developer's method. In overridden methods wr_event and rd_events are taken from the event pool and triggered as shown in above code.

Adding register callback with registers [1]:

```
trans_capture_cb trans_cap_cb;
trans_cap_cb = new(…);
uvm_reg_cb::add(env.reg_model.reg*, trans_cap_cb);
```

The trans_capture_cb class is created and added to the uvm_reg_cb queue. When an read or write operation to a register is performed on to registers the callbacks objects are pricked from the queue and based on the type of access the post_write or post_read function is called.

Capturing the events from RAL:

```
class ref_model extends uvm_component;
  // Declaring uvm events for write and read
  uvm_event#(uvm_reg_item) wr_event;
  uvm_event#(uvm_reg_item) rd_event;
  uvm_reg_item reg_wr;
  uvm_reg_item reg_rd;
  uvm_object reg_obj;
  …..
  function new (….);
  ……
   virtual function void build_phase(uvm_phase);
     wr_event = cfg.event_pool.get("reg_write_event");
      rd_event = cfg.event_pool.get("reg_read_event");
   endfunction : build_phase

    task run_phase()
      fork
        forever begin
          wr_event.wait_ptrigger_data(reg_obj);
          if(!$cast(reg_wr, reg_obj)) begin `uvm_fatal(…) end
          process_wr_pkt(reg_wr);
        end
        forever begin
          rd_event.wait_ptrigger_data(reg_obj);
          if(!$cast(reg_rd, reg_obj)) begin `uvm_fatal(…) end
          process_rd_pkt(reg_rd);
        end
    endtask
endclass
```

In build_phase the events reg_write_event and reg_read_event are taken from event pool and assigned to the local wr_event and rd_event respectively. In run_phase the threads will be spawned to wait for the

events using the method .wait_ptrigger_data and once the event is triggered by the register callback class, it is captured by the waiting threads. To avoid race conditions .wait_ptrigger_data method which is a persistent trigger is used instead of normal trigger. On capturing of an uvm_event - the return type is always of type uvm_object which is casted to desired uvm_reg_item object type. Similarly, the same event can be used at other components of testbench as long as they have access to the related uvm_pool.

If a TLM/Mailbox was used instead of uvm_event; It would have caused changes at multiple levels of testbench hierarchy and if an object was to be broadcasted to multiple components, the individual connection to each components must be made and only structural components can get connected through TLM's and this requires a different mode of communication for sequences and sequence_item.

3. **Interrupt**

Interrupt handling is a must have feature of any verification environment. Interrupt is an event that is triggered by a Design or an IP block once certain conditions are fulfilled; the CPU has to service these events. Actions that are to be taken care by CPU while servicing the interrupts are collectively called as - Interrupt Service Routines (ISR).

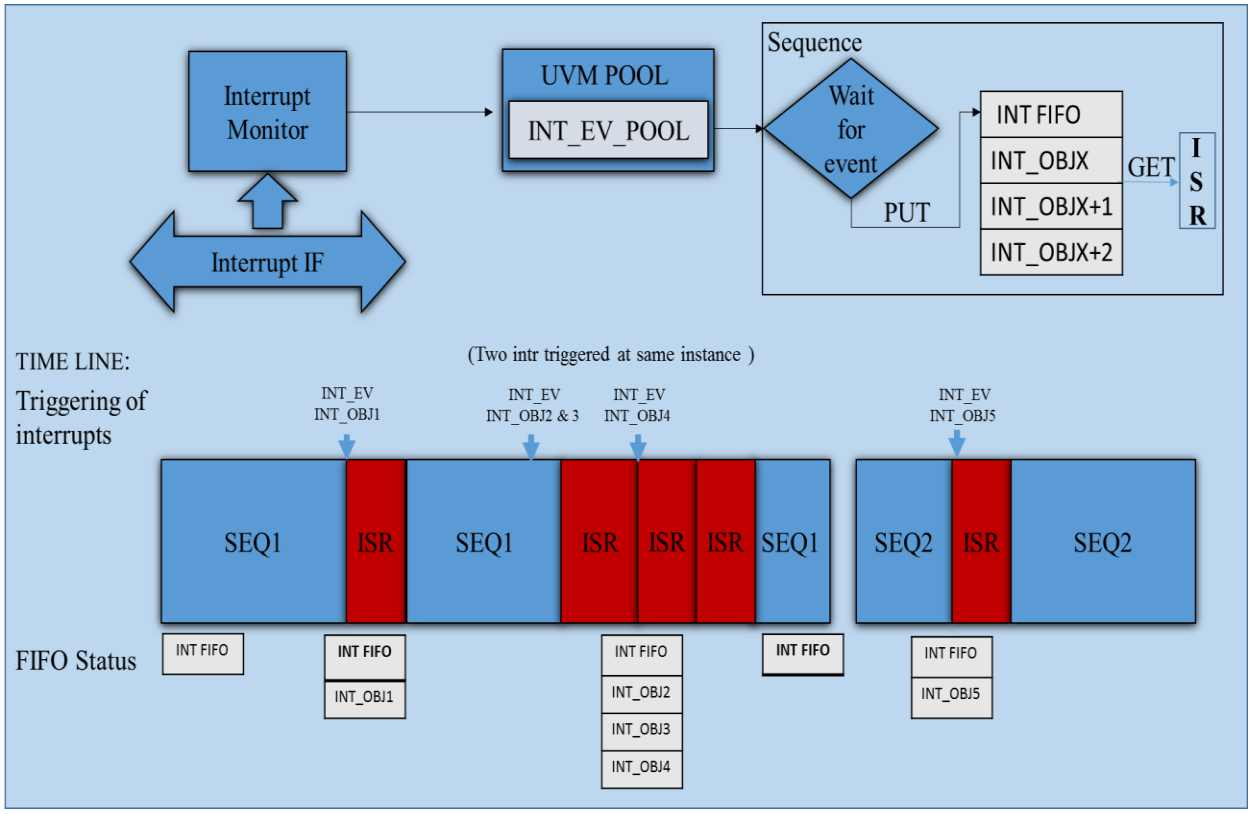

Figure 5: Interrupt Mechanism using uvm_events

Now, let's focus on how uvm_events can be used in this context; the interrupt monitor in Figure 5 monitors the interrupt signal on the interrupt interface. Whenever an interrupt is detected it will trigger an event, this event is registered with the global uvm_event_pool declared in the configuration class. The base virtual sequence will be continuously waiting for the occurrence of interrupt event. Once the event is captured the delivered data along with the event is push to a FIFO. The ISR method in a sequence pops in virtual sequence which is running forever loop pops interrupt information for the FIFO and uses grab()

method to get the exclusive access of the current sequencer and executes accordingly. It will use the ungrab() method before the sequence completes so the previous/next sequence continues execution.
As shown in figure 5 while SEQ1 was being executed, there was an interrupt INT_EV1. Once this event is captured by a sequence, interrupt FIFO is filled with data delivered (INT_OBJ1) along with the event, the ISR sequence pops interrupt information INT_OBJ1 uses grab() method to get the exclusive access of the sequencer which stops the SEQ1 and handles the control to ISR. On the completion of ISR the sequencer is released by ungrab() and SEQ1 resumes from where it was halted.

The following pseudo code explains how uvm_event is used in the interrupt mechanism.

a. Monitor: The Interrupt monitor checks for the interrupt signal and triggers an event.

```
class isr_monitor extends uvm_monitor;
  trans_c trans;
 // isr event declaration
 uvm_event #(trans_c) isr_event;

function buid_phase(uvm_phase phase);
  isr_event=cfg.event_pool.get("interrupt_event")
endfunction

task run_phase(uvm_phase phase);
 forever begin
 // Capturing and triggering the event
  @(posedge isr_if.int_n);
   isr_event.trigger(trans);
endtask
endclass: isr_monitor
```

b. Configuration: The configuration class has a global uvm_event pool.

```
class env_cfg extends uvm_object;
//event pool declaration
uvm_event_pool event_pool;

endclass: env_cfg
```

c. Test: The test has an event_pool declaration and it is created in build_phase and is assigned to the configuration event pool.

```
class env_base_test extends uvm_test;
uvm_event_pool evt_pool;

function buid_phase(uvm_phase phase);
  evt_pool=uvm_event_pool::get_global_pool();
  env_cfg.event_pool=evt_pool;
endfunction

endclass: env_base_test
```

d. Sequence: The isr sequence waits for the interrupt event, grabs the current sequencer and will service the interrupt.

```
class isr_seq extends uvm_sequence;
trans_c trans;
// isr event declaration
 uvm_event #(trans_c) isr_event;

task body();
    // Waiting for the isr event to trigger
    isr_event.wait_ptrigger_data(trans);
    // Store the event in queue.
    //grab the sequencer
     m_sequencer.grab(this);
    // isr scenario
    m_sequencer.ungrab(this);
endtask
endclass: isr_seq
```

4. Callbacks

Callback is a piece of executable code that is passed as an argument to other code, which is expected to callback (execute) the argument at some convenient time. UVM callbacks are used to add the new capabilities, without creating a huge OOP hierarchy.

uvm_event has built in methods to add or delete the callbacks. The uvm_event_callback class is an abstract class that is used to create a callback object which is attached to uvm_event#(T) as shown in below code.

```
virtual class uvm_event_callback#(    type  T  =  uvm_object) extends uvm_object;
```

The uvm_event_callback class has two empty virtual methods. The user has to create a callback class which extends from uvm_event_callback and has to override the virtual methods to implement the required functionality.

The callbacks support is one of the main feature of any VIP. The authors have been using the uvm_event as place holders for callbacks in their VIP's to reduce the overhead of adding separate callbacks. Here is how it can be achieved:

a) Triggering events at strategic positions in VIP components:

```
class vip_component extends uvm_driver;
// Declaring uvm event of type transaction class
uvm_event #(trans) ev1;
uvm_event #(trans) ev2;
uvm_event #(trans) ev3;

function new(...);
   super.new(…);
endfunction

virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  ev1 = uvm_event_pool::get_global("ev1");
  ev2 = uvm_event_pool::get_global("ev2");
  ev3 = uvm_event_pool::get_global("ev3");
endfunction

task run_phase ( uvm_phase phase);
   vip_pkt pkt;
   start(pkt)
    ->ev1.trigger(pkt);
    process(pkt);
    ->ev2.trigger(pkt);
    end(pkt);
    ->ev3.trigger(pkt);
endtask
```

In all the VIP components the events which are registered with the uvm_pool are triggered at right positions. In the above code ev1, ev2 and ev3 are registered with global pool with key "ev1", "ev2" and "ev3". While registering with event pool the get_global method checks if the element exists in the event pool array if not the event is created and assigned with the key.

b) To add event at predefined points which are marked by placement of uvm_events use Event callback class [5]:

In the following code a user_event_callback class is extended from the uvm_event_callback class. The uvm_event_callback class has built in registers pre_trigger which is executed before the event is triggered and post_tirgger callback class which is triggered after the event is triggered. The functionality which has to be performed before event trigger can be added in pre_trigger method and functionality which has to be performed after event trigger can be added in post_trigger method.

```
class user_event_callback extends uvm_event_call_back #(trans);
  `uvm_object_utils(user_event_callback)

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

 virtual function bit pre_trigger(uvm_event ev, trans pkt);
     //update logic-we can change trans/data before triggering event
 endfunction

 virtual function bit post_trigger(uvm_event ev, trans pkt);
  //update logic-we can change data/trans after triggering event
 endfunction

endclass
```

Adding the callback to the events:

```
class my_test extends uvm_test;
 `uvm_component_utils(my_test)

  uvm_event ev1;
  uvm_event_pool evt_pool = uvm_event_pool :: get_global_pool();
  user_event_callback cb_event;

  function new(string name = "my_env", uvm_component parent=null);
    super.new(name,parent);
     ev1 = evt_pool.get("ev1");
  endfunction

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    //to add call backs
    ev1.add_callback(cb_event);
   ……
    //to delete callbacks
    ev2.delete_callback(cb_event);
  endfunction
```

Once the callback class is coded an instance of user_uvm_callback class, it is created with name cb_event. The cb_event is added to the event class by using the uvm_event built in method add_callback and to remove a callback delete_callback method is used. In the above code the authors are interested to process the packet only at event ev1. Similarly the callbacks can be introduced at the locations ev2 and ev3.

CONCLUSION

The uvm_event and event pool provides synchronization between multiple threads or concurrent processes in the verification environment. It is observed that by using uvm_event we can achieve better synchronization without making major changes to the exiting test-bench. The uvm_event wrapper class around the traditional system Verilog event with added methods makes uvm_event to be applicable to a broader and complex application than just synchronization.

ACKNOWLEDGMENT

We profoundly thank colleagues and management team at elitePLUS Semiconductor Technologies Pvt Ltd, Bangalore for their valuable guidance and thought provoking discussions.

REFERENCES

[1] Verilab - Advanced UVM Register Modeling – "There's More Than One Way to Skin A Reg" – Mark Litterick and Marcus Harnisch.

[2] DVCON INDIA 2014 - Global Broadcast with UVM Custom Phasing Jeremy Ridgeway, Dolly Mehta - Avago Tech.

[3] Accellera, "UVM User Guide, v1.1d", www.uvmworld.org

[4] Accellera, "UVM Reference Guide, v1.1d", www.uvmworld.org

[5] Dialog Semiconductor – "The UVM Register Layer Introduction and Experiences" - Steve Holloway