De-mystifying synchronization between various verification components by employing novel UVM classes

Pushpal Nautiyal & Gaurav Chugh
601-608, KLJ Tower North, Netaji Subhash Place, New Delhi 110034

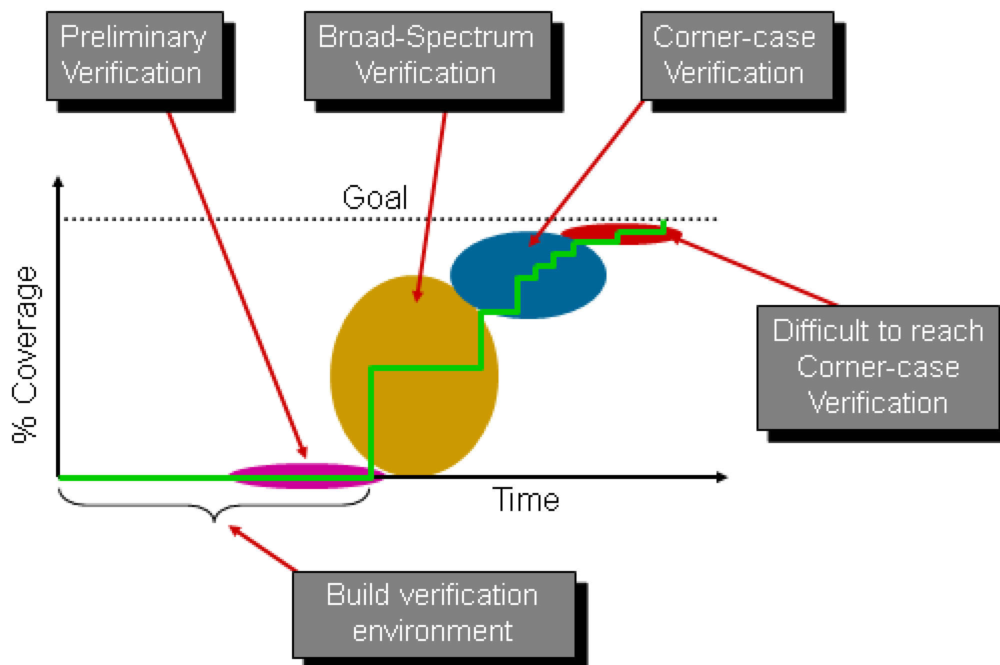
***Abstract*- UVM has been designed to encourage and facilitate modular and layered verification environments, where verification components at all layers can be reused in different environments and facilitate automated Verification .**

With the size of Verification environments booming exponentially one has to constantly brainstorm on how to achieve synchronization between the various verification components , is there any way to reduce the simulation turnaround time during initial Verification phases where chances of setup related issues are quiet high . UVM lends a helping hand in this regard by providing basic structures such as UVM events , barriers , heartbeat etc. which dwindles the Verification engineer's concern and he has few less things to worry about

I. Introduction

UVM class-based test benches have become as complex as the hardware under test [DUV] , and are evolving into large object oriented software designs. However because of such growing complexity of Verification designs one of the key challenge is to how to incorporate "Synchronization" between the Verification components .
System Verilog constructs such as "Events ","Mailboxes" etc were an attempt to overcome such hurdles however robustness and reusability of the environment is degraded . With the advent of UVM , these issues were addressed via structures such as "UVM Events", "UVM Barriers " etc .

UVM Events are an amalgamation of System Verilog Events and adds features of its own which ultimately allows the Verification Engineer to emulate "synchronization" easily and utilize its features .
Below is a Graph highlighting the important phases of Verification :



In addition to this during the Preliminary Verification phase one of the most recurring issues are "setup /configuration" related issues leading to simulation being stuck which in hindsight causes wastage of simulation time .Generally in scenarios where simulation is stuck simulation exits only when "Global/UVM" Timeout happens UVM attempts to resolve such a problem by bringing in UVM heartbeat which monitors the activity in a Verification environment and in case of no activity the simulation will be ceased

However still a lot of mysticism is associated with the synchronization of various verification components.
This is an attempt to lift the veil on sorely underutilized UVM classes such as :

1. Uvm_event ,
2. Uvm_barrier ,
3. Uvm_heartbeat ,
4. Grab/Ungrab [Mechanism to get exclusive access to driver while the sequencer is arbitrating multiple sequences]

II. UVM EVENT /UVM EVENT POOL

The UVM event class congregates the richness of System Verilog/Verilog events and adds a few features of its own, thus achieving leverage.
The main incentive is that it help in setting of the event callbacks thus providing a knob to the user just before or after the activation of uvm_events and can also be employed to return uvm_objects once the respective event is triggered . The latter feature can be highly pragmatic as it can be employed to return back the status of objects as well .

With employment of uvm_event_pool the uvm_events can be globally accessed .
For example in verification environment two components such as monitor and driver may depend on single event say "reset_event" and hence can easily be shared .

Advantages of UVM events:

1) Provides a knob to the user just before or after the activation of uvm_events via "trigger_callbacks"
2) Can be used to return status of uvm_objects on triggering of event

3) Ease of accessibility [Event objects can be passed around the Test bench using "uvm_config_db " mechanism]

The most prevalent API's for uvm_event class which adds to its richness

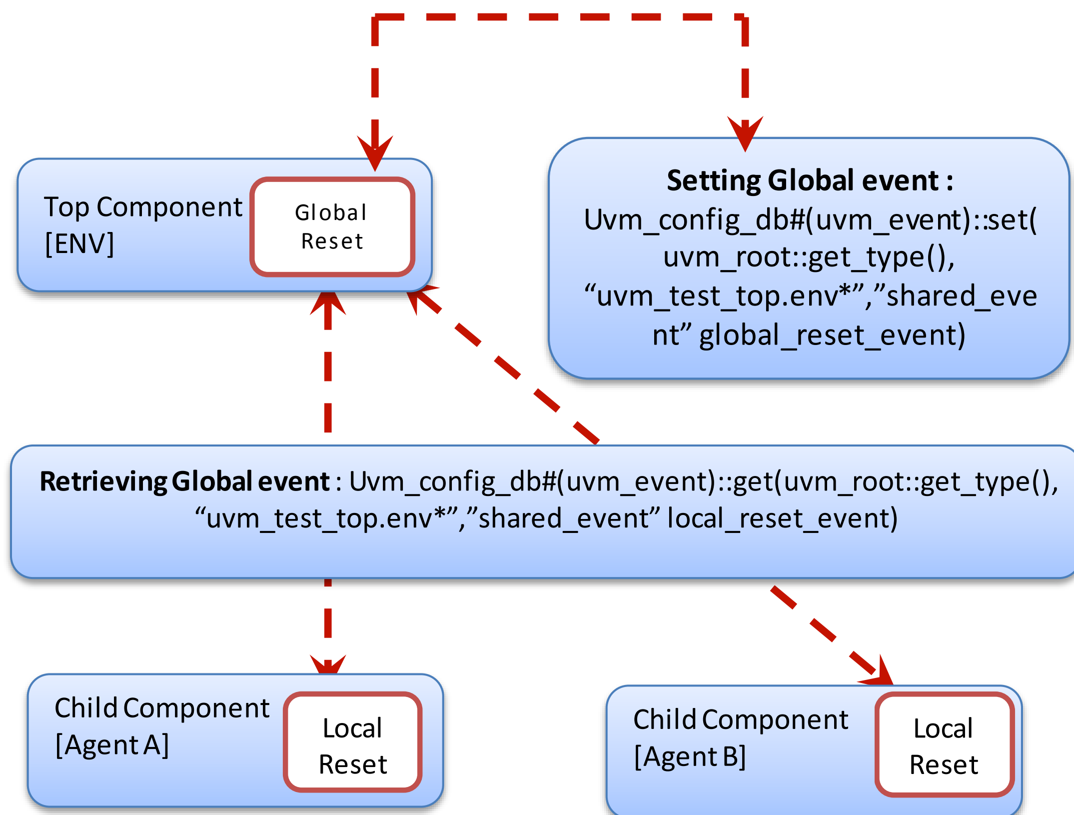
1) wait_trigger: Checks for triggering of the Event
2) wait_ptrigger : Checks for triggering of the Event over the time slice including delta cycles , help to remove the Race around conditions
3) get_trigger_time: Retrieves the Trigger time of the Event , in case event is not triggered , it will return 0
4) get_trigger_data : Retrieves the data if provided during invocation of "trigger"
5) is_on/is_off: Indicates if the event is triggered or not .

    1. "is_on"returns 1 if the event is triggered

    2. "is_off" returns 1 if the event is not triggered

6) Trigger : Triggers the event ensuring unblocking of "wait_*trigger*' calls .
   One can also the provide "data" type argument to feed trigger related information

   Below is sample usage case manifesting the synchronization of "reset event" from top level component onto "child level " component



<u>Psuedo Code for top level component "env":</u>

*/** Handle of child component class */*
child_component child_inst;
………………….
*/** Instance of UVM Event */*
uvm_event global_reset_event   = new ("global_reset_event");

*/** Instance of UVM Event Pool */*
uvm_event_pool global_reset_pool   = new ("global_reset_ pool");

*/** Addition of the reset event into the pool */*
global_reset_pool.add("child_inst", global_reset_event);

*/** Configuring the instance of uvm_event "global_reset_event" inside the "uvm_test_top.env" hierarchy */*
uvm_config_db#(uvm_event)::set(uvm_root::get_type(), "uvm_test_top.env","shared_event" global_reset_event)*

Psuedo Code for child component:
*/** Local Reset event synchronized with Global reset event */*
uvm_event local_reset_event;
………….

Retrieving the "Global reset event " :
uvm_config_db#(uvm_event)::get(uvm_root::get_type(), "uvm_test_top.env","shared_event" local_reset_event)*
…………………………

The sharing of "uvm_events " across UVM hierarchies augments its advantages .

UVM_EVENT_CALLBACKS:

The "uvm_event_callbacks" "pre_trigger" and "post_trigger" allows the user a great deal of flexibility before/after triggering of UVM_EVENT .

For Example : The "pre_trigger" callbacks can be used to replace tasks which wait for triggering of events etc

“**Pre_trigger**” event_callback : Can replace tasks that wait on triggering of event

Event Trigger : Ensure unblocking of “wait_*trigger*’ calls such as
1) wait_trigger: Checks for triggering of the Event
2) wait_ptrigger : Checks for triggering of the Event over the time slice etc …
One can also the provide “data” type argument to feed trigger related information

<loc_event>.trigger(data); //”data “ will have trigger specific info
get_trigger_data : Retrieves the data if provided during invocation of “trigger

“**Post_trigger**” event_callback: Invoked after triggering of the Event

III. UVM BARRIER / UVM BARRIER POOL

The UVM barrier class provides an efficient and effective way to achieve synchronization between various processes.
Its bane come to the fore when a user wishes to block desired number of processes until a threshold/synchronization point is achieved.
The uvm_barrier_pool classes make it easier to manage barriers between components that share the same barriers as they also can be accessed globally just as uvm_events .
In a nut shell from functionality perspective UVM Barriers are just like a semaphore although in reverse direction .

One of the most active areas where it can be easily be employed is when multiple sequences are getting executed in parallel and one needs to wait for completion of all the sequences to go forward .
The most prevalent API’s for uvm_barrier class are :

1) wait_for : Will block until completion of processes to reach the barrier
2) set_threshold : Determines numbers of processes who would wait on the barrier before they resume
3) get_num_waiters: Queries the number of processes currently waiting at the barrier

This is quite helpful in scenarios where multiple sequences are executed in parallel and we need to wait for all sequences to be completed before resumption .

Pseudo Code :

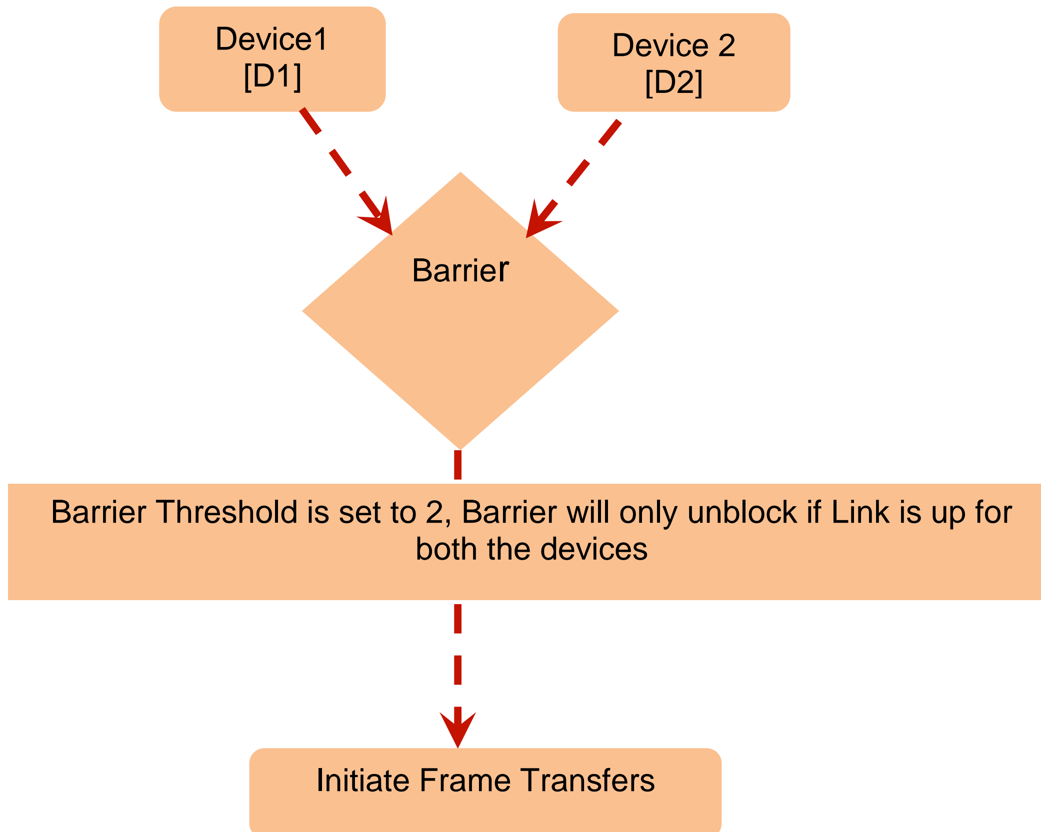
Snippet from the Testbench :

```
/** Instance of UVM Barrier */

Uvm_barrier barrier = new("barrier", 3); // Configuring the Barrier threshold as "3"

/** Initiation of 3 Parallel Frame Tranfers */

fork
/** Number of concurrent processes required to unblock the barrier is "3" */
initiate_frame_transfers(1);
initiate_frame_transfers(2);
initiate_frame_transfers(3);
join
```

Definition of the task "initiate_frame_transfers"

```
/** Instance of Barrier object is also passed to the subroutine */
task initiate_frame_transfers(uvm_barrier local_barrier , int count);
case (count)

/** Assumption : Sequence sequence_a, sequence_b, sequence_c are running on the same sequencer */
1: sequence_a.start(<sequencer_path>, null) ;
2: sequence_b.start(<sequencer_path>, null) ;
3: sequence_c.start(<sequencer_path>, null) ;
………….
endcase

 /** Will wait for threshold number of processes to be completed and then only unblock */
this.local_barrier.wait_for();
endtask
```

This functionality of Uvm barrier can also be used in a scenarios where one has multiple devices and the Verification Engineer needs to check for "Activation of each device/Link being up " before initiation of frames / Valid data between them



III UVM HEARTBEAT

The UVM Heartbeat behaves as a watchdog timer, but it is more powerful. It watches for activity in the test bench components and if it finds that there isn't the right amount of activity, in that window , it'll issue a fatal message and end the simulation. This can catch a simulation lock-up early on – even before the global timeout kicks in, hence saving one's precious amount of time .

Here are few of the scenarios where we might be interested in ceasing the simulation rather than UVM_TIMEOUT being called out :

1) Absent Connections between the Verification Components
2) When State Machines are stuck etc.

To employ the UVM heart beat we need to associate a specific objection object , and the heartbeat object must raise (or drop) the synchronizing objection during the heartbeat window

The most prevalent API's for uvm_hearbeat class are :

1. set_mode: Sets or retrieves the heartbeat mode:

The heartbeat can be configured so that all components (UVM_ALL_ACTIVE), exactly one (UVM_ONE_ACTIVE), or any component (UVM_ANY_ACTIVE) must trigger the objection in order to satisfy the heartbeat condition.

2. Add/remove: Add/Removes a single component to the set of components to be monitored
3. set_heartbeat: Sets the target list of components that are required to be monitored and setting of the heartbeat event. Soon after invocation of this monitoring is initiated

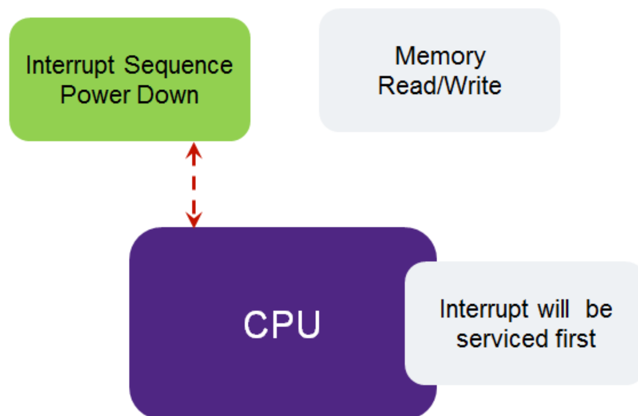
<u>Psuedo Code :</u>

```
/** Local variable to periodically trigger heartbeat event */
int timeout ;
/** Instance of uvm_heartbeat */
uvm_hearbeat heartbeat;

/** Instance of uvm_heartbeat event */
uvm_event heartbeat_event;

/** Queue of uvm_components whose "heartbeat" will be monitored */
uvm_component comp_queue[$];

/** Instance of callback objection required while creating an object of uvm_heartbeat */
uvm_callbacks_objection callback_objection;

/** Casting the run/main_phase objection to type "uvm_callbacks_objection" */
assert ($cast(callback_objection , run_phase.get_objection()))
else
`uvm_fatal("run_phase", $sformatf ("Objection of phase isn't of type uvm_callbacks_objection. Please define
UVM_USE_CALLBACKS_OBJECTION_FOR_TEST_DONE!"));

/** Instantiation of UVM HEARTBEAT */
heartbeat = new("heartbeat", this, callback_objection);

/** Storing all the components required for monitoring */
uvm_top.find_all("*",comp_queue, this);

/** Configuring the Mode of the heartbeat object to UVM_ANY_ACTIVE */
/** Setting it to UVM_ANY_ACTIVE will ensure that any component can trigger the
objection in order to satisfy the heartbeat condition */

heartbeat.set_mode(UVM_ANY_ACTIVE);

/** Sets up the heartbeat event and assigns a list of objects to watch */
/** Initiation of heartbeat monitoring */
heartbeat.set_heartbeat(heartbeat_event, comp_queue);

/** The heartbeat event must be periodically triggered */
fork begin
forever begin
#timeout heartbeat_event.trigger();
end
end
join_none
```

IV GRAB /UNGRAB

This mechanism provides the sequence with exclusive access to the driver and will allow a sequence to complete its operation without any other sequence operations in between them . grab() method requests a lock on the specified sequencer. A grab() request is put in front of the arbitration queue. It will be arbitrated before any other requests

This mechanism is highly required in scenarios when virtual sequencer requires full control over its subsequencers for a limited time and then let the original sequences continue working.
From an implementation perspective it will be highly useful in generating INTERUPT Sequences where disabling of sub sequencers is required and highest priority needs to be given to specific sequences .



Psuedo Code :

```
virtual task body();
// Grab the cpu sequencer if not virtual.
if (sequencer.cpu_seqr != null)
p_sequencer.cpu_seqr.grab(this);
// Execute a sequence.
`uvm_do_on(intrpt_seq,sequencer.cpu_seqr)
// Ungrab.
if (sequencer.cpu_seqr != null)
sequencer.cpu_seqr.ungrab(this);
```

REFERENCES

[1] UVM Class Reference 1.2
[2] UVM User Guide