SYNAPSE design

UVM Heartbeat: A overlooked gem that stops dead simulations early. I'll show you how to use it.

Mastering UVM

What the user's guide and examples haven't shown you

By David Larson – Director of Verification
verification@synapse-da.com

Mastering UVM

What the user's guide and examples haven't shown you

From time to time we will post papers on topics that relate to UVM, OOP and verification in general. These topics are designed to help your team meet the ever increasing demanding schedules. Some are informative; some are innovative; sometimes controversial, but always worth a read.

Today's topic: UVM Heartbeat. What is it? How do you use it, and why you should care.

UVM Heartbeat

Introduction

The UVM Heartbeat monitor is very useful, yet sorely underused. It is just another name for a watchdog timer, but it is more powerful and flexible than the ones you may have seen before. It watches for activity in the test bench and if it finds that there isn't the right amount of activity, in the specified amount of time, it'll issue a fatal message and end the simulation. This can catch a simulation lock-up early on – before the global timeout kicks in, which can be *very late*. This is a godsend for tight schedules, saving you tons of wasted simulation time.

Lock ups can (and do) occur for many reasons:

- When there are missing connections between blocks.
- When there is a race condition, such as when a receiver starts listening for a response after the response was already given. Now both the transmitter and receiver are waiting for each other.
- An error occurred, but the error handler doesn't clean up properly.
- When a transmitter doesn't send enough data.
- Internal FSM problems,
- And so on…

However, UVM's heartbeat monitor doesn't work right out of the box; there is some assembly required. I will give you the best tools for putting it together and walk you through how to use it.

The Usage Model

The uvm_heartbeat is designed to watch for activity on a *single, special objection object*. The heartbeat monitor thinks your test bench is alive when a registered component raises or lowers objections on that objection object.

The usage model expects you to create the special objection object (of type *uvm_callbacks_objection* which is a child class of uvm_objection), probably in your test case or TB and pass that object reference around to each one of your components. When a component is active it raises or lowers the objection. If

the heartbeat monitor doesn't see activity within a specific period of time, then it'll issue a fatal HBFAIL message, ending the simulation.

Coincidentally, for most of us, the heartbeat objection will be raised and lowered at the same time the run phase's objection is raised and lowered. Also, you may already have a lot of components in your test bench and you don't want to retrofit all those components just to give a special heartbeat.

The good news is that you can put the heartbeat monitor's finger right on the pulse of the pre-existing run-phase's objector. No need to create a special pulse objector. No need to retrofit all your components.

This is how you do it…

The Constructor

To get started, let's work on constructing the heartbeat object. uvm_heartbeat's constructor looks like this:

```
function new(string name,
             uvm_component cntxt,
             uvm_callbacks_objection objection)
```

The `name` argument is easy; just give it a super slick name like "activity_heartbeat" or "watchdog".

The context argument (`cntxt`) specifies the parent scope of where the objections will be monitored and also specifies the context of the heartbeat messages. You can usually just pass in a "this" reference since you'll want to instantiate this object either in your base test class or test bench class. You can also leave it as "null" if you want to use the root context.

The `objection` argument is where things start to get interesting. If you want to tap into the objections that you are already using, which is normally what you want, then you'll need a reference to the run_phase's objection object. You can get it by calling phase.get_objection() on the run_phase's phase object. But don't do it quite yet. You'll notice that get_objection() returns an object of type uvm_objection, while we need an object of type uvm_callbacks_objection, which is a *child* of uvm_objection. And if you $cast() the objection to uvm_callbacks_objection, it'll fail. Have no fear; you are closer to heartbeat bliss than you think.

The reason it fails is because the run_phase's internal objection object is constructed from a uvm_objection class. But you can change it to be constructed as a uvm_callbacks_objection (which is what we really want) if you add this define:

UVM_USE_CALLBACKS_OBJECTION_FOR_TEST_DONE

It is probably best to add this define to the command line so you know that it is defined before the UVM libraries are compiled. The switch you use is simulator dependent, but you probably already knew that.

The Mode

Before starting the heartbeat monitor, you'll need to specify the passing condition of the heartbeat:

```
heartbeat.set_mode(uvm_heartbeat_mode mode);
```

Should the monitor be happy only if ALL of the components show activity during the heartbeat period (UVM_ALL_ACTIVE)? Or if *ONLY ONE* component shows activity (UVM_ONE_ACTIVE)? Strange but possible. Most of us will want it to pass if ANY component shows activity (UVM_ANY_ACTIVE):

```
heartbeat.set_mode(UVM_ANY_ACTIVE);
```

Ready… Set …

The last step is to start the heartbeat monitor. You can do this by calling the set_heartbeat function. But we aren't ready to call it quite yet. It takes some arguments that need some preparation.

```
function void set_heartbeat (uvm_event e, ref uvm_component comps[$]);
```

The `uvm_event` argument determines the *heartbeat window*. You must trigger this event on a periodic basis. If the heartbeat monitor doesn't see the expected amount of activity within that period of time, then it becomes unhappy and has a fatal message tantrum.

Generating the heartbeat events is easy enough. You can just trigger the event in an infinite loop with a configurable delay and then throw it in a background thread:

```
int m_heartbeat_timeout;
uvm_event e = new("e");
// …
heartbeat.set_heartbeat(e, ...);
// …
fork begin
   forever begin
      #m_heartbeat_timeout e.trigger();
   end
end
join_none
```

Now, on to the last argument:

```
function void set_heartbeat (uvm_event e, ref uvm_component comps[$]);
```

comps is a queue that needs to contain a reference to every component in your hierarchy that you want to monitor, which is usually all of them. You could do it manually, like this:

```
uvm_component comps[$];
comps.push_back(env.agent1.driver);
comps.push_back(env.agent1.monitor);
comps.push_back(env.scoreboard);
```

```
/// … etc. etc.
```

But that is tedious, error prone (esp. for system-level sims), and really boring. A better approach is to automatically collect a list of all the components in the hierarchy. Fortunately, UVM also provides a function in uvm_root that does just that:

```
function void find_all (string comp_match,
                        ref uvm_component comps[$],
                        input uvm_component comp=null);
```

The first argument is a regular expression. If it matches a component's name then it is included. The comps queue contains all the components found. The last argument specifies where the search should begin. Only components from that level down will be found.

But since we want everything, then we can call it like this:

```
uvm_top.find_all("*", comps, this);
```

By the way, if you want to remove some items from the list, then you can always call the heartbeat's remove function:

```
heartbeat.remove(env.agent1.noisy_monitor);
```

Go!

Now you are ready to go. Below is the fully assembled heartbeat monitor, plus a few trimmings:

- ✓ When the heartbeat monitor gives up, we need to know who is causing the traffic jam so there is a heartbeat message catcher that prints the outstanding objections before the fatal message is issued. Super useful for zeroing in on the component that is the real stick in the mud.
- ✓ The heartbeat is placed in an intermediate base test class (which is creatively called: "base_test"). This is a great place to put the heartbeat in your test bench as well. If you don't have a base test, then you should add one.
- ✓ This heartbeat monitor is actually started just *before* the run phase. I didn't want to put it in the run phase because most UVM tests do not call super.run_phase(phase) (most components don't either for that matter).
- ✓ If the m_heartbeat_timeout setting isn't set in the test case, then the monitor isn't started. This is just a personal preference. You may want to have a default heartbeat timeout value set so it is turned on unless the test case writer explicitly turns it off.

You now have everything you need to fully leverage this very useful and under-used UVM feature. No more hour-long simulations that show nothing but dead wires.

Feel free to play around with it and love it.

```
class timeout_catcher extends uvm_report_catcher;
```

```
   uvm_phase m_run_phase;

   `uvm_object_utils(timeout_catcher)

   function action_e catch();
      if(get_severity() == UVM_FATAL && get_id() == "HBFAIL") begin
         uvm_objection obj = m_run_phase.get_objection();
         `uvm_error("HBFAIL", $psprintf("Heartbeat failure! Objections:"));
         obj.display_objections();
      end
      return THROW;
   endfunction
endclass

class base_test extends uvm_test;
   int m_heartbeat_timeout;

   function void start_of_simulation_phase(uvm_phase phase);
      super.start_of_simulation_phase(phase);
      if (m_heartbeat_timeout)
         heartbeat(phase);
   endfunction

   function void heartbeat(uvm_phase phase);
      uvm_callbacks_objection cb;
      uvm_heartbeat hb;
      uvm_event e = new("e");
      uvm_component comps[$];
      timeout_catcher catcher;
      uvm_phase run_phase = phase.find_by_name("run", 0);

      catcher = timeout_catcher::type_id::create("catcher", this);
      catcher.m_run_phase = run_phase;
      uvm_report_cb::add(null, catcher);

      assert ($cast(cb, run_phase.get_objection()))
      else
         `uvm_fatal("run_phase", $psprintf("run phase objection type isn't of
type uvm_callbacks_objection. You need to define
UVM_USE_CALLBACKS_OBJECTION_FOR_TEST_DONE!"));

      hb = new("activity_heartbeat", this, cb);
      uvm_top.find_all("*", comps, this);

      hb.set_mode(UVM_ANY_ACTIVE);
      hb.set_heartbeat(e, comps);

      fork begin
         forever begin
```

```
            // The heartbeat must see activity within the event
            // window.
            #m_heartbeat_timeout e.trigger();
         end
      end
      join_none
   endfunction
endclass
```