Without Objection – Touring the uvm_objection implementation – uses and improvements

Rich Edelman
Siemens EDA, Fremont, CA

***Abstract*- The SystemVerilog[2] UVM[1] implements a class named uvm_objection. An objection is used to guard code that "isn't done yet". For example, an objection can prevent a process from finishing until some other process agrees. uvm_objections are sometimes overused and are always misunderstood. This paper will explain the implementations a bit and share uses and provide some alternative solutions that are easier to understand, simpler to use, and work transparently.**

I. Introduction

The SystemVerilog UVM Objections solve a problem – how to coordinate processes. Two processes P1, and P2 are running. But they can only exit together – they must synchronize their exit.

The UVM is built of processes – the phases from uvm_components, the body task from a uvm_sequence and any threads from these processes. These processes often need to synchronize – for example a request is sent out, and the processing on collecting responses should not exit until the response is received.

The UVM provides phases which are synchronized automatically. Those phases are pre-defined and can be extended. Sticking with the pre-defined phases is the best approach. A phase has an objection which is used to coordinate all the objects that are currently in the current phase. No object can exit the phase when the objection is raised. Each phase that wants to participate in the phase objection must raise an objection and later drop that objection. Synchronization with objections is conceptually simple – raise an objection to "passing the barrier" and drop an objection to "passing the barrier".

But the UVM phasing and objection code quickly become intertwined. There's complexity there that can be problematic to debug and understand. Simple synchronization can be implemented with something much simpler.

In [5] the author proposes a simpler mechanism that end tests – a simple barrier. It has some bells and whistles and is worth consideration for a different way to exit testbenches. In [4] the authors have created a great treatise on OVM and UVM and how to terminate tests. It's much more than just terminating tests, and worth a look. The example code included in this paper is even simpler still – perhaps too simple – with no bells and whistles – with no debug hooks or callbacks. But that's the point – transparent and boringly simple.

II. Background

Synchronization primitives [3] exist in process control theory and operating system design among other places. There are many uses for them.

In the UVM, the run_phase normally raises an objection to exiting. Later the objection will be dropped. At least one objection must be raised – usually in the test. Without the objection, simulation will end immediately. A typical run_phase is below. Most tests have one objection, used in this simple way.

```
task run_phase(uvm_phase phase);
  phase.raise_objection(this);
  `uvm_info(get_type_name(), "...running", UVM_MEDIUM)
  ...
  phase.drop_objection(this);
endtask
```

The UVM offers some debug built-in, but this debug will generate lots of information with even the simplest usage. Using +UVM_OBJECTION_TRACE on the command line for just the simple usage above generates more than 100 lines of output.

III. THE SOURCE CODE – UVM_OBJECTION.SVH AND FRIENDS

The objection has been around since the days of the OVM – before 2009. That code was about 750 lines. And it looks relatively the same as the UVM versions. The UVM versions of the synchronization items hasn't changed much over the years. The number of lines in each file for each major UVM version is listed below.

```
uvm-1.1d uvm 1.2 uvm-1800
   90        90      94      src/base/uvm_event_callback.svh
  361       394     432      src/base/uvm_event.svh
  211       209     204      src/base/uvm_barrier.svh
 1502      1453    1241      src/base/uvm_objection.svh
  338       341     349      src/base/uvm_heartbeat.svh
 1192      1198    1212      src/base/uvm_callback.svh
  300       300     307      src/macros/uvm_callback_defines.svh
 3994      3985    3839      total
```

The file uvm_objection.svh contains 1241 lines in uvm-1800.2_2020_2.0_rc2. It's a thousand lines.

The class definition contains 5 tasks, 28 functions and 17 class member variables. Of those class member variables, there are 3 associative arrays indexed by object handle. They are 'm_source_count', 'm_total_count' and 'm_drain_time'. Two arrays of integers and one of time. 'm_source_count' and 'm_total_count' are incremented on a 'raise' and decremented on a 'drop'. 'm_drain_time' is the time for this object to wait once all objections are dropped before calling 'all_dropped' and propagating.

```
Classes
Tasks
    virtual all_dropped
    m_execute_scheduled_forks
    m_forked_drain
    wait_for
    wait_for_total_count
Functions
    virtual clear
    virtual convert2string
    virtual create
    display_objections
    virtual do_copy
    virtual drop_objection
    virtual dropped
    get_drain_time
    get_objection_count
    get_objection_total
    get_objectors
    get_propagate_mode
    get_type
    virtual get_type_name
    m_display_objections
    m_drop
    m_get_parent
    m_init_objections
    m_propagate
    m_raise
    m_report
    m_set_hier_mode
    new
    virtual raise_objection
    virtual raised
    set_drain_time
    set_propagate_mode
    trace_mode
Data
    bit m_cleared
    uvm_objection_context_object m_context_pool
    process m_drain_proc
    memory m_drain_time
    uvm_objection_events m_events
    uvm_objection_context_object m_forked_contexts
    uvm_objection_context_object m_forked_list
    uvm_objection m_objections
    bit m_prop_mode
    bit m_register_cb_uvm_objection_callback
    uvm_objection_context_object m_scheduled_contexts
    uvm_objection_context_object m_scheduled_list
    memory m_source_count
    uvm_root m_top
    bit m_top_all_dropped
    memory m_total_count
    bit m_trace_mode

Design  Library  UVM Testbench  Classes
```

```
uvm_objection.svh - uvm_objection
57   //------------------------------------------------------------------------------
58   // Objections provide a facility for coordinating status information between
59   // two or more participating components, objects, and even module-based IP.
60   //
61   // Tracing of objection activity can be turned on to follow the activity of
62   // the objection mechanism. It may be turned on for a specific objection
63   // instance with <uvm_objection::trace_mode>, or it can be set for all
64   // objections from the command line using the option +UVM_OBJECTION_TRACE.
65   //------------------------------------------------------------------------------
66
67   // @uvm-ieee 1800.2-2020 auto 10.5.1
68   // @uvm-ieee 1800.2-2020 auto 10.5.1.1
69   class uvm_objection extends uvm_report_object;
70     `uvm_register_cb(uvm_objection, uvm_objection_callback)
71
72     protected bit     m_trace_mode;
73     protected int     m_source_count[uvm_object];
74     protected int     m_total_count [uvm_object];
75     protected time    m_drain_time  [uvm_object];
76     protected uvm_objection_events m_events [uvm_object];
77     /*protected*/ bit     m_top_all_dropped;
78
79     protected uvm_root m_top;
80
81     static uvm_objection m_objections[$];
82
83     //// Drain Logic
84
85     // The context pool holds used context objects, so that
86     // they're not constantly being recreated.  The maximum
87     // number of contexts in the pool is equal to the maximum
88     // number of simultaneous drains you could have occuring,
89     // both pre and post forks.
90     //
91     // There's the potential for a programmability within the
92     // library to dictate the largest this pool should be allowed
93     // to grow, but that seems like overkill for the time being.
94     local static uvm_objection_context_object m_context_pool[$];
95
96     // These are the active drain processes, which have been
97     // forked off by the background process.  A raise can
98     // use this array to kill a drain.
99    `ifndef UVM_USE_PROCESS_CONTAINER
100    local process m_drain_proc[uvm_object];
101   `else
102    local process_container_c m_drain_proc[uvm_object];
103   `endif
104
105    // These are the contexts which have been scheduled for
106    // retrieval by the background process, but which the
107    // background process hasn't seen yet.
108    local static uvm_objection_context_object m_scheduled_list[$];
109
110    // Once a context is seen by the background process, it is
111    // removed from the scheduled list, and placed in the forked
112    // list.  At the same time, it is placed in the scheduled
113    // contexts array.  A re-raise can use the scheduled contexts
114    // array to detect (and cancel) the drain.
115    local uvm_objection_context_object m_scheduled_contexts[uvm_object];
116    local uvm_objection_context_object m_forked_list[$];
117
118    // Once the forked drain has actually started (this occurs
119    // ~1 delta AFTER the background process schedules it), the
120    // context is removed from the above array and list, and placed
121    // in the forked_contexts list.
122    local uvm_objection_context_object m_forked_contexts[uvm_object];
123
124    protected bit m_prop_mode = 1;
125    protected bit m_cleared; /* for checking obj count<0 */
126
127
128    // Function -- NODOCS -- new
129    //
130    // Creates a new objection instance. Accesses the command line
131    // argument +UVM_OBJECTION_TRACE to turn tracing on for

t.sv  uvm_objection.svh  uvm_objection.svh
```

raise_objection

raise_objection immediately calls m_raise. m_raise sets the counts, and checks to print the debug information. Then it calls the routine 'raised()'. Raised() checks to see if the 'obj' is a component. If so, it calls the component.raised() routine. Then the raised callbacks. Then it fires the raised events. Most components use the default empty 'raised' definition, but the component has the opportunity to define functionality by defining a raised() call. After raised returns back to m_raise routine there's almost 100 lines of code sorting out forks, and fixing up scheduling. And that's the easy side of an objection.

drop_objection

drop_objection immediately calls m_drop. M_drop decrements the counts, and checks to print the debug information. Then it calls the routine 'dropped()'. Dropped() checks to see if the 'obj' is a component. If so it calls the component.dropped() routine. Then the dropped callbacks. Then it fires the dropped events. Most components use the default empty 'dropped' definition, but the component has the opportunity to define functionality by defining a dropped() call. After dropped returns back to m_drop, there's some decisions about forks and propagating.

propagate

When propagate is called if the object is a component, then the objection is propagated to the parent. If the object is a sequence, then the objection is propagated to the sequencer. The comments say parent_sequence first, then the sequencer. This seems like an oversight. Or maybe out-of-date comments. This is the "get_parent()" code in uvm_objection.svh

```
function uvm_object m_get_parent(uvm_object obj);
  uvm_component comp;
  uvm_sequence_base seq;
  if ($cast(comp, obj)) begin
    obj = comp.get_parent();
  end
  else if ($cast(seq, obj)) begin
     obj = seq.get_sequencer();
  end
  else
    obj = m_top;
  if (obj == null)
    obj = m_top;
  return obj;
endfunction
```

Objections are powerful. In the UVM they are intertwined with phasing which makes them more powerful and more exotic and more complicated. Experience and examining the code should lead the reader to use them sparingly – normally just once in a test.

There is some *really* tricky code wrapped up with objections and phasing. There must be something simpler.

IV. SIMPLE TEST

A simple usage of uvm_objections is below. This code is a run_phase. Standard practice is to call raise_objection() at the start of the run_phase, and drop_objection at the end.

This particular test simply creates 4 sequences and runs them on the sequencers in the agents in the environments. Once all sequences have completed, then this test finishes – and calls drop_objection().

```
task run_phase(uvm_phase phase);
  phase.raise_objection(this);
  `uvm_info(get_type_name(), "...running", UVM_MEDIUM)

  pretty_print();

  s1 = seq::type_id::create("s1");
  s2 = seq::type_id::create("s2");
  s3 = seq::type_id::create("s3");
  s4 = seq::type_id::create("s4");
  fork
    s1.start(e1.a.sqr);
    s2.start(e2.a.sqr);
```

```
      s3.start(e3.a.sqr);
      s4.start(e4.a.sqr);
    join

    phase.drop_objection(this);
  endtask
```

The only thing being synchronized is the test itself – "don't exit until the test is done". Limiting the use of objections to this model will keep things easy and simple. If the need arises to synchronize other things besides run_phase and exit, then consider something simpler.

V. A LESS SIMPLE, BUT STILL SIMPLE TEST

In the example test above, those environments (e1, e2, e3 and e4) may have an interesting run_phase that should be synchronized. They each call raise_objection() and drop_objection(). This is an example that uses a raise and a drop in many "env" classes.

```
class env extends uvm_component;
  `uvm_component_utils(env)

  agent a;

  function new(string name = "env", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    a = agent::type_id::create("a", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    #100000;
    `uvm_info(get_type_name(), "...Objection Testing - 'env' ready to exit", UVM_MEDIUM)
    phase.drop_objection(this);
  endtask
endclass
```

For example, the environment run_phase might simply wait until #100000 ticks have passed, then exit. Producing the following at the end of the log

```
UVM_INFO sequence.svh(19) @ 6476: uvm_test_top.e3.a.sqr@@s3 [transaction] ...started
UVM_INFO driver.svh(20) @ 6476: uvm_test_top.e3.a.d [driver] Executing: id: 7 READ(48, 5) #3999
UVM_INFO sequence.svh(23) @ 6484: uvm_test_top.e3.a.sqr@@s3 [transaction] ...finished
UVM_INFO test.svh(46) @ 6484: uvm_test_top [test] ...Objection Testing - 'test' ready to exit
UVM_INFO env.svh(18) @ 100000: uvm_test_top.e4 [env] ...Objection Testing - 'env' ready to exit
UVM_INFO env.svh(18) @ 100000: uvm_test_top.e3 [env] ...Objection Testing - 'env' ready to exit
UVM_INFO env.svh(18) @ 100000: uvm_test_top.e2 [env] ...Objection Testing - 'env' ready to exit
UVM_INFO env.svh(18) @ 100000: uvm_test_top.e1 [env] ...Objection Testing - 'env' ready to exit
```

The original – regular test – was ready to end at time 6484, but the envs didn't want to exit until much later. This kind of behavior is not a replacement for the normal uvm_objection drain_time. It's just 4 more run_phase synchronizations. This example is headed in the wrong direction. The recommendation is to use as few objections as possible.

VI. MY_OBJECTION

What about a simpler objection? Something like a Barrier? A barrier prevents passage. The UVM has a barrier implemented in uvm_barrier.svh. It's 240 lines of code with debug and comments. Probably a worthwhile function, certainly simpler than a uvm_objection but even simpler yet than uvm_barrier – a simple barrier defined as

```
class barrier;
  int count;

  task raise();
    #0;
    count = count + 1;
   endtask
```

```
  task drop();
    #0;
    if (count == 0)
      return;
    count = count - 1;
    wait (count == 0);
  endtask
endclass
```

The code is straightforward if not overly simple. The barrier has a count. In order to “pass” the barrier, cooperating processes need to cause the count to go to zero by all calling the ‘drop’ routine. Originally they each called the ‘raise’ routine. To be sure, there are many more improved implementations. The #0 is a kind of escapism which allows a zero time example to work properly. (Not a realististic testbench – just an exercise in supporting all timing modes).

This code is simple enough to try many different implementations – re-code it to improve it. Be careful to test thoroughly lest the processes are starved or blocked. The diligent reader should improve the code, adding comments, traceability and debugging hooks.

VII. USING MY_OBJECTION BY-NAME

Communicating processes are normally organized or coordinated by an “overseer” process – a guardian. The guardian may start the processes and elect the communication. For example, three processes that are coordinated by a guardian are told which objection or barrier to use. Two processes which share a critical region might have a common mutex which a guardian process has initialized and assigned.

Another kind of coordination could be agreeing on “synchronization names” – such as “start”, “middle” and “end”. Or “compress” and “uncompress”.

A simple class which uses an associative array indexed by strings will provide just this functionality.

Instead of using a barrier as b.raise_objection and b.drop_objection, the call would change to ua.raise(“start”) and ua.raise(“middle”) and ua.raise(“end”). The barrier is “hidden” in the associative array – only known my its name – “start” or “middle” or “end”.

The class wrapper creates the barrier class if needed and interested processes simply need access to the allocated ‘user objections’ class. That’s their coordinated handle. They share the handle and the various synchronization points – by string name.

```
class user_objections;
  barrier barriers[string];

  function barrier get(string name);
    barrier b;
    if (barriers.exists(name)) begin
      b = barriers[name];
    end
    else begin
      b = new();
      barriers[name] = b;
    end
    return b;
  endfunction

  task raise(string name);
    barrier b;
    b = get(name);
    b.raise();
    $display("...Raised %s", name);
  endtask

  task drop(string name);
    barrier b;
    b = get(name);
    b.drop();
    $display("...Dropped %s", name);
```

```
  endtask
endclass
```

This simple wrapper has a raise, and drop and a get. Easy to see and understand.

VIII. SMALL EXAMPLE

The code snippets below are a class and a module. In the full example, there are two class instances – Class1 and Class2, and 3 verilog modules – A, B and C. There are three “regions” or “phases” that are synchronized across the modules and class instances – “start”, “middle” and “end”. Each object does “work” – it just spins in each region/phase for a random amount of time, and then drops its objection. Once all have dropped, then processing moves forward.

This small example has 5 main threads running – 3 always blocks from the modules and two forever loops from the class objects. Each has a “start”, a “middle” and an “end”.

```
user_objections ua = new();

class Class1;
  task body();
    forever begin
      int n;
      n = $urandom_range(10, 1);
      ua.raise("start", thread_name);
      for (int i = 0; i < n; i++) begin
        #n;
      end
      ua.drop("start", thread_name);
       $display("@%t: %m done done done with %s", $time, "start");

      ua.raise("middle", thread_name);
      n = $urandom_range(10, 1);
      for (int i = 0; i < n; i++) begin
        #n;
      end
      ua.drop("middle", thread_name);
       $display("@%t: %m done done done with %s", $time, "middle");

      ua.raise("end", thread_name);
      n = $urandom_range(10, 1);
      for (int i = 0; i < n; i++) begin
        #n;
      end
      ua.drop("end", thread_name);
       $display("@%t: %m done done done with %s", $time, "end");
    end
  endtask
endclass

module a();
  initial begin
    forever begin
      int n;
      n = $urandom_range(10, 1);
      ua.raise("start", thread_name);
      for (int i = 0; i < n; i++) begin
        #n;
      end
      ua.drop("start", thread_name);
       $display("@%t: %m done done done with %s", $time, "start");

      ua.raise("middle", thread_name);
      n = $urandom_range(10, 1);
      for (int i = 0; i < n; i++) begin
        #n;
      end
      ua.drop("middle", thread_name);
       $display("@%t: %m done done done with %s", $time, "middle");
```

```
        ua.raise("end", thread_name);
        n = $urandom_range(10, 1);
        for (int i = 0; i < n; i++) begin
          #n;
        end
        ua.drop("end", thread_name);
         $display("@%t: %m done done done with %s", $time, "end");
      end
    end
  endmodule
```

The top level

```
module top();
  a A();
  b B();
  c C();
  Class1 c1;
  Class2 c2;

  initial begin
    setup();
    c1 = new();
    c2 = new();
    fork
      c1.body();
      c2.body();
    join_none
    repeat(1000)
      #10;
    $finish(2);
  end
endmodule
```

Upon running the simple example with the alternative "objection" mechanism, we can see that each object reports after its call to "drop" has returned. The $display called after each drop call. Each object is ready to move forward after that.

```
# @                  81: t_sv_unit.Class1.body done done done with start
# @                  81: t_sv_unit.Class2.body done done done with start
# @                  81: top.A done done done with start
# @                  81: top.B done done done with start
# @                  81: top.C done done done with start

# @                 145: top.C done done done with middle
# @                 145: top.B done done done with middle
# @                 145: t_sv_unit.Class1.body done done done with middle
# @                 145: top.A done done done with middle
# @                 145: t_sv_unit.Class2.body done done done with middle

# @                 209: top.A done done done with end
# @                 209: t_sv_unit.Class2.body done done done with end
# @                 209: top.C done done done with end
# @                 209: top.B done done done with end
# @                 209: t_sv_unit.Class1.body done done done with end

# @                 290: top.C done done done with start
# @                 290: top.B done done done with start
# @                 290: t_sv_unit.Class2.body done done done with start
# @                 290: top.A done done done with start
# @                 290: t_sv_unit.Class1.body done done done with start
```

Class1 and class2 are SystemVerilog class handles. A, B and C are Verilog module instances. The "payload" or "work" performed by each of the objects in each phase is a simple model – random delays in a loop. But each object is free to do as little or as much work as desired – the other phases can't get ahead of the next barrier.

```
        ua.raise("start", thread_name);
        for (int i = 0; i < n; i++)
          #n;
        ua.drop("start", thread_name);
        $display("@%t: %m done done done with %s", $time, "start");
```

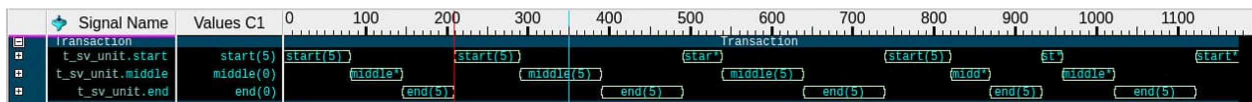
```
        ua.raise("middle", thread_name);
        n = $urandom_range(10, 1);
        for (int i = 0; i < n; i++)
          #n;
        ua.drop("middle", thread_name);
        $display("@%t: %m done done done with %s", $time, "middle");
```

```
        ua.raise("end", thread_name);
        n = $urandom_range(10, 1);
        for (int i = 0; i < n; i++)
          #n;
        ua.drop("end", thread_name);
        $display("@%t: %m done done done with %s", $time, "end");
```

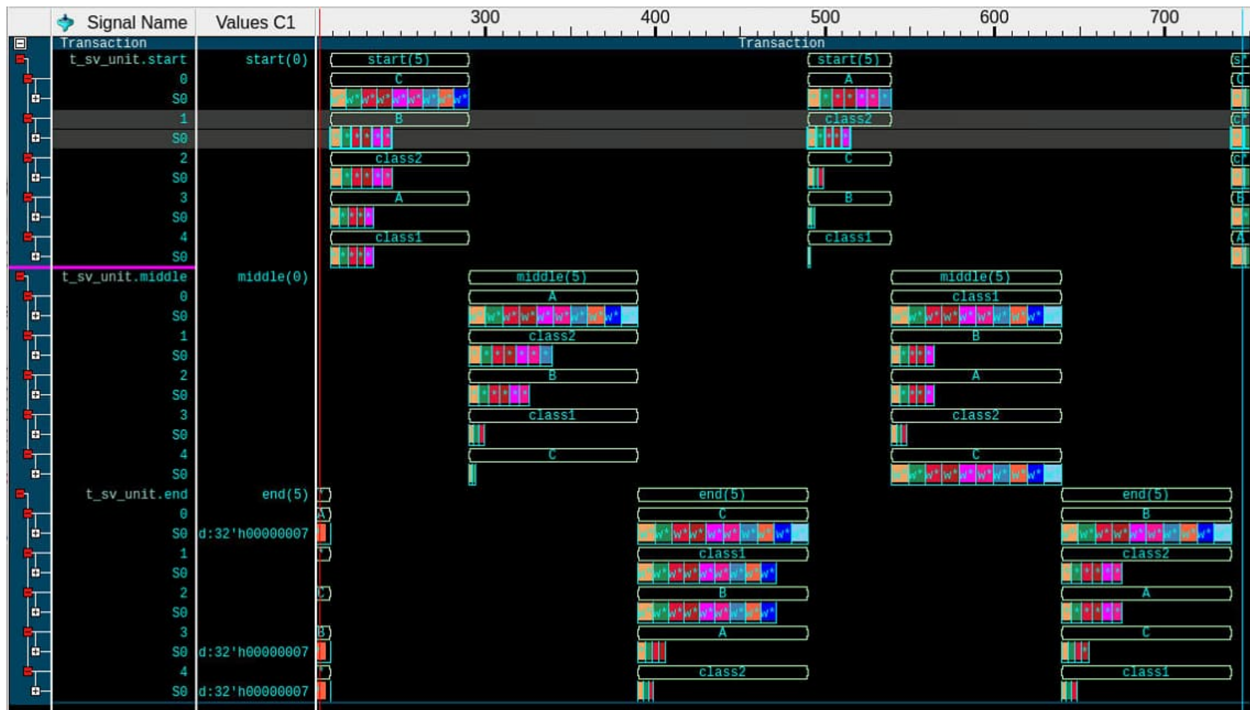
Each phase is synchronized with the barrier / objection.

Below are visualizations - the "middle" phase and the work load for each object. Each "phase" – here the middle phase – displays each object – "class1", "B", "A", "class2" and "C", in order, along with the payload for each object.

Many phase begins and ends – all synchronized. It's easy to debug the regions/phases – no overlap.



An expanded view – each phase details of the objects and payloads



IX. Recommendations

There's no reason to avoid objections when considering the phasing of UVM. Objections and phasing are inextricably linked. To change that relationship is certain to be a hardship. A lot of hard work went into getting it working in the beginning. Use objections with phasing. Do limit phasing usage. Stick with the build, connect, run phases – the run-time phases are not for the faint hearted (`uvm_pre_reset_phase`, `uvm_reset_phase`, `uvm_post_reset_phase`, `uvm_pre_configure_phase`, `uvm_configure_phase`, `uvm_post_configure_phase`, `uvm_pre_main_phase`, `uvm_main_phase`, `uvm_post_main_phase`, `uvm_pre_shutdown_phase`, `uvm_shutdown_phase` and `uvm_post_shutdown_phase`). The recommendation is to avoid them. Creating your own phasing is also not recommended.

But a way to create phasing – is to use a barrier. Use the uvm_barrier – or something much simpler – the objection-by-name in the example above. Or write your own. Writing a fit-for-purpose widget is a good idea to understand the problem you are trying to solve. Synchronization is a simple problem. It should have a simple solution.

X. Warnings

Some users put raise_objection() and drop_objection() pairs in code sections that are executed frequently – for example on a positive edge of a clock. Or in the start_item/finish_item of a sequence. Be careful. Raise_objection and drop_objection() can be very slow.

Merging OVM phasing and UVM phasing is possible but will require quite an effort. A few users have done it successfully, but the code has many race conditions due to thread execution order – all LRM compliant, but terrible to debug. While not a true objection related issue, the synchronization of phasing is a hard thing to do – marrying OVM and UVM makes it harder.

XI. Conclusion

The uvm_objection is a powerful tool. It is used widely – in most UVM testbenches. But be careful. With the right set of calls it can hang itself without any testbench or DUT.

The simple alternative code offered here can also hang itself if not written and used correctly. Experimenting with this code or other synchronization constructs is instructive in synchronization but also in how SystemVerilog spoofs a real parallel machine.

Use uvm_objections wisely. Source code is available by request from the author.

XII. REFERENCES

[1] UVM - 1800.2-2020 - IEEE Standard for Universal Verification Methodology Language Reference Manual, *https://ieeexplore.ieee.org/document/9195920* , source code: *https://www.accellera.org/downloads/standards/uvm*

[2] SystemVerilog - 800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, *https://ieeexplore.ieee.org/document/8299595*

[3] Synchronization primitives in Wikipedia, *https://en.wikipedia.org/wiki/Synchronization_(computer_science)#Barriers*

[4] "OVM & UVM Techniques for Terminating Tests", Clifford E. Cummings, Sunburst Design, Inc. and Tom Fitzpatrick, Mentor Graphics, DVCON US 2011 (58 pages), *http://www.sunburst-design.com/papers/CummingsDVCon2011_UVM_TerminationTechniques.pdf*

[5] "Shutdown with Agreements in a UVM Testbench", Mark Glasser, NVIDIA Corporation, SNUG 2017 (16 pages), contact author.