2014 Design and Verification DVCON Conference and Exhibition INDIA Sept 25-26, 2014 Hotel Park Plaza Bangalore, India

Global Broadcast with UVM Custom Phasing

Jeremy Ridgeway, Avago Technologies, Inc., Fort Collins, CO, USA
(*Jeremy.Ridgeway@avagotech.com*)

Dolly Mehta, Avago Technologies, Inc., Bangalore, India
(*Dolly.Mehta@avagotech.com*)

***Abstract*—Dynamic hard-reset in the middle of hardware simulation is always difficult to achieve. We did precisely this in a recent PCI-Express (PCIe) project using the Universal Verification Methodology (UVM) phase scheduler and custom phasing. With four side-band phases executing in a master-slave relationship alongside our main stimulus generating phase, the UVM main phase, we successfully and always notified *every* project-specific component in the verification environment that a major simulation event was about to occur. Sequencers were given time to pause stimulus generation scenarios for a "quiescent" type event or even end scenarios for a "hard reset" type event that resulted in a UVM phase jump. Additionally, we knew that the requested simulation event *would not* occur until all components were ready and lowered their phase objection. We found that setting up custom phasing was not difficult, but encountered several run-time issues we had to overcome. In the end, our verification environment was able to handle major simulation events gracefully.**

Keywords—Functional Verification, UVM, Phasing, SystemVerilog

I. INTRODUCTION

Dynamic hard reset in the middle of hardware simulation is always difficult to achieve. In a recent PCI-Express (PCIe) project, we had to achieve dynamic and multiple hard reset cycles and device under test (DUT) re-configuration, chosen pseudo-randomly, in a single simulation. Furthermore, our PCIe DUT supported low power modes that required the verification environment to halt traffic and enter a quiescent state. Only under those conditions for a period of time would the DUT transition to low power.

We defined additional requirements specifically on the verification environment. We required early warning to stimulus generating Universal Verification Methodology (UVM) components and checkers that a *major simulation event* was about to occur [1]. Further, the event should only occur **after** the environment was ready. Thus, some form of feedback from components was necessary.

Some approaches use global events or reset-specific handlers to notify the environment of a reset. The event-driven approach in [2] requires instrumentation in verification components to handle a reset assertion. In [3], instrumentation is also required in reset-aware components, but clean-up at reset assertion is handled automatically. These approaches are active in nature and require specific instrumentation in the verification component. Furthermore, the instrumentation cannot be easily encapsulated in a sub-class as multiple inheritance is not supported in SystemVerilog [4]. Neither approach describes how to warn the environment of a pending reset, instead relying on reactive handling post reset assertion. Finally, while these approaches can be expanded to other major simulation events, they are tailored to dynamic hard reset.

The UVM library provides a customizable phase scheduler already tied to **every** UVM component in the simulation [1]. We opted to utilize custom phases for global notification of our major simulation events. Utilizing the UVM scheduler enabled a *passive* notification scheme that every team member implicitly knew how to use. With the scheduler, we accomplished environment preparation for reset, re-configuration, and data quiescence by notifying components with *side-band* custom phases executing in parallel with the UVM main phase. Sequencers were notified in our PCIe-specific component class of an impending UVM phase jump (to rest or configure phases) and forcibly killed or allowed active sequences to "gracefully" exit. Notably, time could elapse between request for hard reset and application of hard reset by components raising an objection if they required more time to prepare. The same sequences were "paused" to simulate stimulus quiescence in the UVM main phase. This passive scheme required no special instrumentation to participate, provided global notification of impending event, and waited for all components to be ready. Importantly, no team member needed training on how to use the scheme because they already understood the logistics.

This paper focuses on four aspects of our approach. First, the custom phases were managed in a master-slave architecture, as described in section II. Second, we describe the framework we devised to implement the custom phasing architecture, in section III, and issues resolved, in section IV. Third, in section V, we present the implementation of custom phases and the phasing master. Finally, we present example usage of custom phases in our environment in section VI.

II. Custom UVM Phases

We followed accepted guidelines on UVM phasing in our PCIe verification environment [4]. The DUT hard reset cycle was performed in the UVM reset phases (reset as well as pre- and post-reset). The DUT was configured in UVM configure phases. All data path traffic was focused in the UVM main phase **only** (no traffic in pre- or post-main). Finally, final checking was performed in UVM shutdown phases.

For our major simulation events (hard reset, re-configure, quiescence), we defined four custom phases to execute in parallel to the UVM main phase only. In Figure 1, each PCIe-specific side-band phase operated as a "master" (M) or "slave" (S). This architecture is a *single-master* approach where one master component implements the master PCIe side-band phases, control and control-idle. Additional masters would require coordination that can be avoided by enforcing the single-master approach.

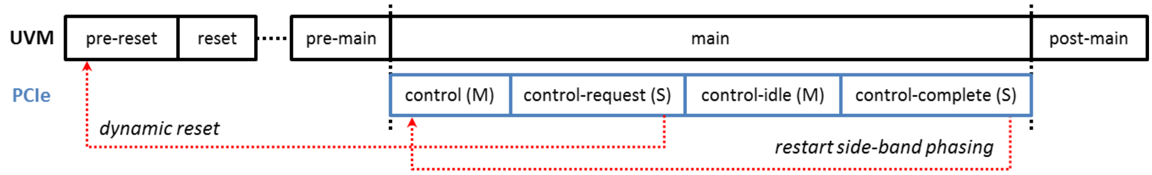

Figure 1: Side-band phase transitions architecture. Each PCIe phase operates as "master" (M) or "slave" (S).

The **pcie_control_phase** is the entry point for our side-band phases. This phase is intended to be implemented in the single master component, as per Figure 1. The master component takes requests from any other component via a UVM analysis implementation port. The control phase raises an objection while waiting for requests. Once a valid request is received, the control phase notes the request in a global structure and lowers its objection, thereby transitioning to the control-request phase. This starts the "passive" notification scheme to all project-specific components. No components are required to actively subscribe to the notification. However, if no valid request is received before UVM main phase is ready to exit, then the master component forces the control phase to exit and disables all side-band phases.

The **pcie_control_request_phase** is implemented in all slave components, as necessary. This phase prepares the simulation for the requested action. For example, if a stimulus generation pause is requested, then each sequencer would notify its active scenarios to stop generating stimulus. As each slave component completes preparation for the requested event, they lower their objections. Once all slave component objections are lowered, the implication is that the environment is ready for the requested event. The side-band phasing either performs a UVM phase jump or transitions to the next side-band phase, control-idle. For example, a hard reset request ends in a jump from UVM main phase to UVM pre-reset phase. A quiescent stimulus generation request stays in the UVM main phase while the side-band phasing transitions to control-idle.

The **pcie_control_idle_phase** is implemented in the master component to hold stimulus generation quiescent. The master raises its objection and waits for some window of time and/or for some notification from a component that the idle time is complete. At that point, the master notifies the environment to continue stimulus generation by transitioning to the control-complete phase.

The **pcie_control_complete_phase** is implemented in all components as an indicator that stimulus generation activities may resume.

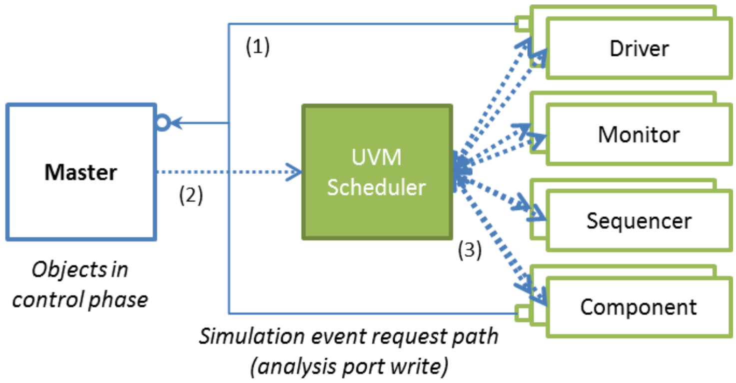

Figure 2: Single-master, many slave component architecture. (1) Slave requests event, (2) master drops objection, and (3) the scheduler broadcasts and waits for environment ready.

The overall master/slave architecture is presented in Figure 2. It is the phasing master component that controls the timing of phasing requests as well as phase jumping. Any slave component may request a simulation event, Figure 2-(1), through an analysis port write. In practice, only a very few components require this connection. The master validates the request and initiates the control loop by dropping its objection in the control phase (2). The UVM scheduler, therefore, acts as the scheme's global servicing agent by notifying each component of a request, then waiting for all components to be ready before continuing, (3). Finally, the master controls timing and target of phase jump. For hard jump (reset and reconfiguration), the master jumps to a UVM phase prior to main. Otherwise, after a data quiescence ends, the master restarts the loop by jumping back to control phase.

III. FRAMEWORK

A framework was devised during our implementation to properly execute custom phases in all PCIe-specific components. A project-specific base class library is required to execute custom phases in components. The custom phases themselves as well as a phase execution scheme is necessary, notably when using parameterized classes.

A. Project-specific Base Component Class Library

The UVM component class, of course, does not contain the custom phase methods and, therefore, will not execute them [1]. Instead, all project-specific component types should extend from a project-specific base class library rather than from UVM. Each base class need only define (empty) custom phase tasks (and a phase execution proxy as in section III.C). Each component within the verification environment inherits the custom phases when extended from the project-specific base class.

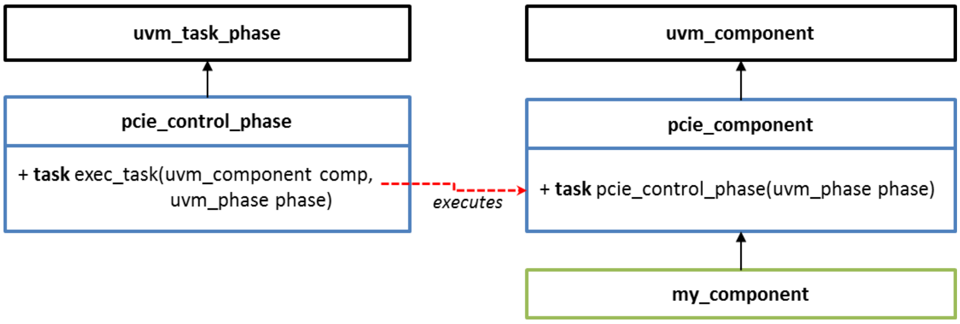

Figure 3: Custom task phase executes a task of similar name in custom component. Extensions to the custom component inherit custom phase methods.

A project-specific base component class library is a requirement for general custom phase execution in a UVM-based verification environment. A shortcut extends only those component types actually used in the environment. Indeed, in our PCIe verification environment, we do not use all UVM component types and therefore have extended only a subset into a project-specific base class library.

B. *Custom Phases*

UVM phases are simply static classes with a reference in the UVM scheduler. Two kinds of UVM base phases exist:

1. zero-time phase, uvm_topdown_phase or uvm_bottom_up phase, and
2. time-consuming phase, uvm_task_phase.

Once the phase is ready to execute in the UVM scheduler, the zero-time phase's exec_func method or the task phase's exec_task method is called. This method will, in turn, call a similar-named function or task in the project-specific component (as per guideline in [1]). In Figure 3, pcie_control_phase is a time-consuming phase. It's exec_task method calls the pcie_control_phase task in the PCIe-specific base component type.

Custom phases must differentiate a reference to a UVM component extension from a reference to a project-specific custom component. For most classes, a simple cast will suffice, as in Listing 1.

```
task pcie_control_phase::exec_task(uvm_component comp, uvm_phase phase);
  pcie_component c;
  if($cast(c, comp))
  begin // Found PCIe-specific component base class
    c.pcie_control_phase(phase);
  end   // But missed all the parameterized classes
endtask
```

Listing 1: Identifying project-specific components with simple casting works for non-parameterized classes only.

However, some UVM component classes are parameterized (e.g., uvm_driver). This leads to a difficulty of clear identification because the dynamic cast depends on the parameter type [5]. One option is to *test all* potential parameter types and execute the task if one matches. This option quickly becomes untenable. A second option is to *maintain a reference* to all instantiated project-specific components in a SystemVerilog associative array [3].

C. *Phase Proxy*

We implemented a phase proxy in each PCIe-specific base component class. With a class member initializer, the proxy is always constructed alongside the component. In Figure 4, the local class function m_get_phase_proxy both instantiates the proxy and registers it in a singleton associative array.

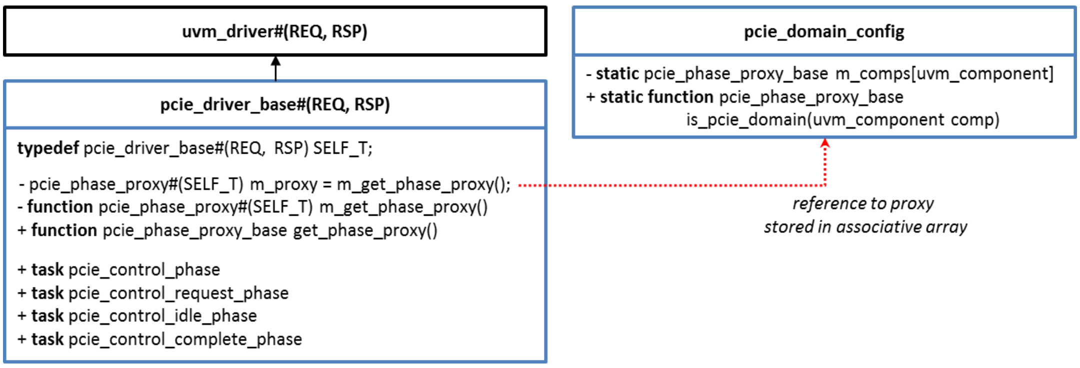

Figure 4: PCIe-specific proxy phasing class.

Now, the phase class exec_task method need not cast to determine a PCIe-specific component. Instead, it simply queries the PCIe-domain configuration which looks up the component reference in its array. If the component is not a PCIe-specific component then it will not be in the array. However, this is only true when the proxy is part of **all** project-specific component class instances. As described in section III.A, custom phasing already requires project-specific base component classes. Adding the phase proxy to those classes is straightforward.

Notice, in Figure 4, that the associative array connects a component reference to its proxy. Even though project-specific components are clearly identified, the phase class still cannot access a project-specific component reference. Parameterized classes hinder any kind of casting. Therefore, the proxy itself maintains a typed reference to its component. In Figure 5, the phase proxy is instantiated with a reference to its typed parent and stored locally in m_parent. Now, the phase class executes the proxy's phase method which, in turn, executes the component's project-specific phase method via the parent reference.

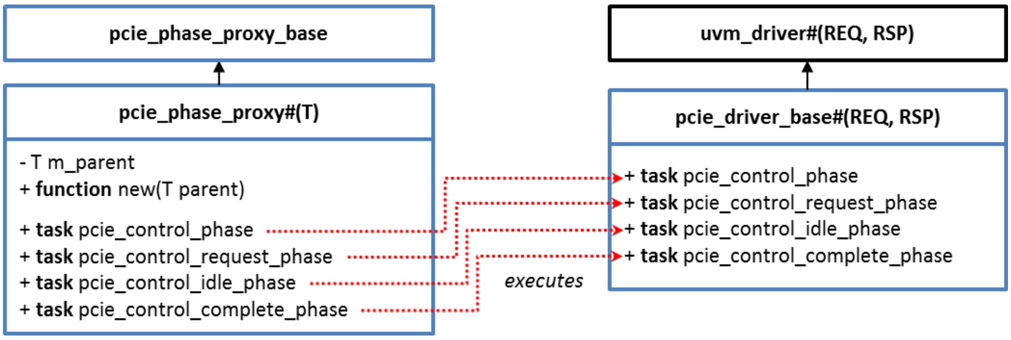

Figure 5: The phase proxy implements the complement of custom phases. It uses its typed class reference to call the parent component's phase task.

IV. PHASING GOTCHAS

We encountered several issues during implementation of the PCIe side-band phasing. First, jumping out of two simultaneous schedules (UVM main phase and PCIe side-band phasing) results in dual executions of the destination phase. Second, determining when the UVM main phase was ready to end was not straightforward. Finally, determining when the PCIe control-complete phase was ready to end was similarly hindered.

A. Multiple Executions of Jump Target Phase

We found in test that jumping out of simultaneous schedules results in multiple executions of the target phase (equal number of executions as simultaneous schedules). Multiple schedules means multiple threads are executing in parallel. On a uvm_domain::jump_all function call, both threads jump and execute the target phase. At the end of the target phase, the threads join and the subsequent phase executes normally. While issues in UVM phase jumping have been reported, such as [7, 8], we did not encounter those specifically.

Referring back to Figure 1, a dynamic reset request resulted in two concurrent executions of the pre-reset phase. This was solved by introducing a “dummy” phase into the schedule.

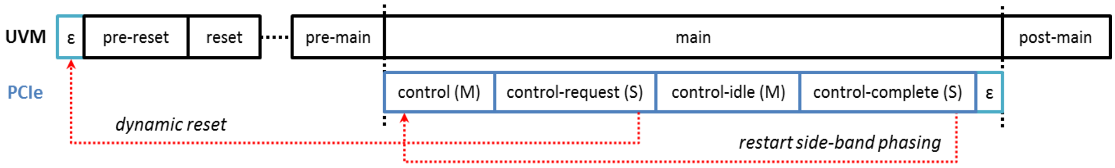

Figure 6: Final custom phasing schedule with an empty target phase (noted with ε).

A dummy task phase executes no method, its exec_task is empty (a dummy zero-time phase would have an empty exec_func). Its sole requirement is to be a target phase in which simultaneous threads may merge. When the dummy phase is executed in the schedule, it takes no time, executes no task, and returns immediately. In Figure 6, the dummy phase is an empty phase within the schedule to handle jumping gracefully.

B. Ending UVM Main Phase

In the PCIe side-band phasing, any component operating in the UVM main phase may issue side-band phasing requests to the phasing master component. As such, the phasing master raises an objection in control phase, see Figure 7. This prevents the control phase from exiting, but it also prevents the UVM main phase from exiting and, ultimately, the simulation will not exit normally.

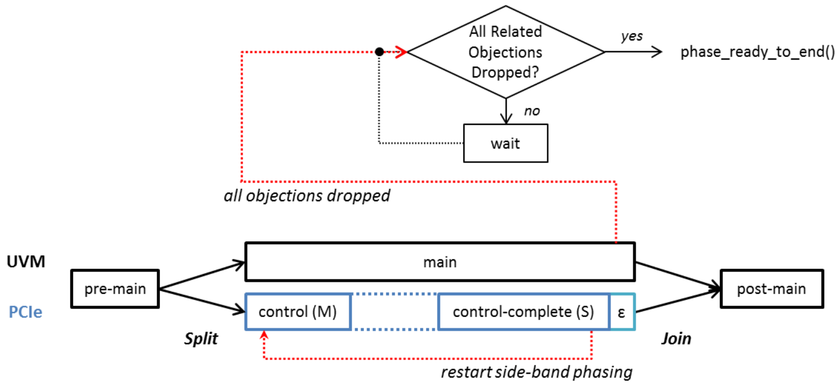

Figure 7: Simultaneous phase schedules join only after the last phase in both are ready to end.

In Figure 7, after UVM main phase drops all objections it waits until the related phases drop their objections, too. Related phases are either child or phases in a simultaneous schedule. The PCIe-side-band phasing, however, never exits. It has two control paths: from control-request to an earlier phase (see Figure 6) or from control-complete to control. An additional "gotcha" in simultaneous custom phasing is relying on the two UVM library function notifications for phase ending: phase_ready_to_end and phase_ended. Neither function will execute until all objections in all simultaneous phase schedules are dropped.

Our phasing master component required implementation of its UVM main phase. The master monitors objections and, when all objections have been dropped, it disables all side-band phases and requests the control phase to end. Now, the side-band phases do not hinder normal simulation exit.

C. Ending PCIe Control Complete Phase

The PCIe control-complete phase suffers from the same exit issue as the main phase in section IV.B. However, it differs in that the PCIe control complete does not exit normally. Instead, at control complete phase end, the phasing master unconditionally jumps to control phase. Therefore, instead of the solution for main exit, we simply appended a dummy terminal phase after control complete. Now, control complete can end because there is a subsequent phase that UVM main phase can synchronize with. This dummy terminal phase executes no task and takes no time

V. Phase and Master Implementation

After the custom phasing framework is complete (refer to section III), the phases themselves and the phasing master component can be implemented.

A. Implementing the Framework

A singleton PCIe phase domain configuration class implemented the associative array, mapping component references to their phase proxies. It also implemented the enable/disable bit for side-band phasing. If the sideband phases are disabled, then they will not execute their exec_task method (see Figure 3).

Four singleton PCIe phase classes implemented the connection between the UVM scheduler and the component proxies (and by proxy the components themselves). Listing 2 presents the PCIe control phase. All PCIe side-band phases are implemented similarly. A singleton PCIe dummy phase class implements an empty phase that does not execute any component method.

```
class pcie_control_phase extends uvm_main;
  pcie_control_phase m_self = get();
  function pcie_control_phase get();
    if(m_self == null)
      m_self = new;
    return m_self;
  endfunction
  task exec_task(uvm_component comp, uvm_phase phase);
    if(pcie_domain_cfg::get_sideband_phases_enabled())
    begin
      pcie_phase_proxy_base proxy =
                     pcie_domain_config::is_pcie_domain(phase);
      if(proxy != null)
        proxy.control_phase(phase);
    end
  endtask
endclass
```

Listing 2: Implementation of the control phase. All custom phases are implemented similar to this example.

To register the side-band phasing with the UVM scheduler, a singleton PCIe phase domain class is employed, as in Listing 3. At time zero, this class instantiates all phase singletons, sets the side-band schedule, and registers the schedule with UVM scheduler to execute with the main phase.

```
class pcie_domain extends uvm_domain;
  static function pcie_domain pcie_domain::get_pcie_domain();
    if(m_self == null) // only once
    begin
      uvm_domain uvm_sched = uvm_domain::get_uvm_domain();
      uvm_phase main_sched;
      m_self = new("pcie");

      // Add the sync phases FIRST
      uvm_sched.add(.phase(pcie_dummy_phase::get()),
                    .before_phase(uvm_reset_phase::get()));

      // Add the main schedule
      main_sched.add(pcie_control_phase::get());
      main_sched.add(pcie_control_request_phase::get());
      main_sched.add(pcie_control_idle_phase::get());
      main_sched.add(pcie_control_complete_phase::get());
      main_sched.add(pcie_control_terminal_phase::get());
      uvm_sched.add(.phase(main_sched),
                    .with_phase(uvm_main_phase::get()));
    end
  endfunction
endclass
```

Listing 3: Registering the PCIe side-band phasing schedule.

B. *Implementing the Master Component*

Custom phasing requires the phases, project-specific components, and phase framework to operate. The phasing master ties everything together to enable phase jumping as requested. The phase master has the following functions:

1. maintain the side-band request queue and analysis implementation port,
2. implement the PCIe control loop,
3. implement phase jumps, and

4. monitor for end of main phase.

Structurally, the phasing master maintains a queue of side-band phasing requests, see Figure 8. The queue is written to via a UVM analysis implementation port from any environment component operating in the main phase, including the master, itself. Procedurally, the phasing master processes and initiates phase requests. At UVM main phase exit indication, it requests side-band phasing exit.

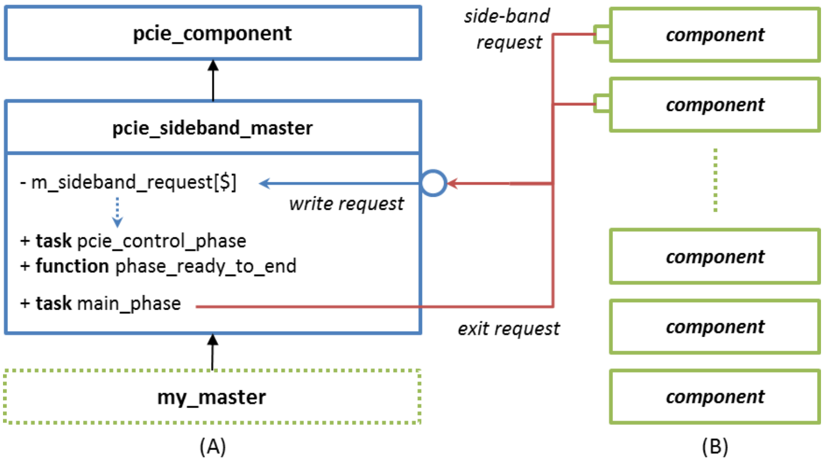

Figure 8: The phasing master (A) processes and initiates side-band requests. It requests control phase exit at UVM main ready to end. Slaves (B) implement slave phases as necessary and can make side-band requests.

The control phase in the master is the **only** implementation of control phase in the environment. Upon entering, the control phase raises an objection and waits for a sideband request. When a valid request is received, the control phase writes the request type to a global accessible location (not pictured in Figure 8) and drops its objection, initiating the side-band phases.

The phasing master monitors phases as they are ready to end. All slave components implement pcie_control_request_phase to prepare for the side-band phasing event (e.g., hard reset). Slaves can take some time to prepare by objecting in the control-request phase. As all objections are dropped, the phasing master is notified that the environment is ready for the event.

When the pcie_control_request_phase is ready to end, and if the current request requires a hard jump out of UVM main phase, then the master initiates the jump via:

```
pcie_domain::jump_all(pcie_dummy_phase::get()).
```

This (prematurely) ends UVM main phase and both threads jump to the preceding "dummy" phase, merge, and enter UVM pre-reset phase, refer to Figure 6. When the pcie_control_complete_phase is ready to end (implying the request did not require a jump out of main), then the master initiates a local jump via:

```
phase.jump(pcie_control_phase::get).
```

This restarts the side-band phasing loop with no effect on UVM main phase. Note that the uvm_phase reference passed to phase_ready_to_end is **not** the actual phase singleton. Instead, this is a placeholder in the UVM scheduler. The master retrieves a reference to the placeholder's "implementation" phase via `uvm_phase imp = phase.get_imp()`, then compares against the singleton phases.

The phasing master also monitors the UVM main phase objections. A reference to the phase's objection is retrieved from the phase placeholder reference passed to the UVM main phase task. Then, the master waits for all objections to be dropped using uvm_root as the component reference, in Listing 4.

```
task pcie_sideband_phasing_master::main_phase(uvm_phase phase);
  uvm_root top = uvm_root::get();
  uvm_objection main_done = phase.get_objection();
  if(!main_done.m_top_all_dropped)
      main_done.wait_for(UVM_ALL_DROPPED, top);
  pcie_domain_cfg::set_sideband_phases_disabled();
  request_exit_control_phase();
        // internal write to implementation port
endtask
```

Listing 4: Phasing master monitors UVM main phase for all objections dropped. It then disables and ends the PCIe side-band control loop.

The uvm_root is the top-most component in the UVM environment. Therefore, when objections are dropped all the way up to the root, as in Listing 4, then all objections have been dropped and the phase is ready to end.

VI. USAGE

We had several major testing requirements for our PCI-Express controller device under test (DUT). First, the DUT supports low-power modes after a period of zero data traffic. Second, the DUT must be pseudo-randomly hard reset and/or re-configure during simulation. We tackle hard reset in this section, but reconfiguration only differs in the target phase to jump to (UVM pre-configure versus pre-reset).

A. Low-Power Testing

To achieve low-power in the DUT, the verification environment must halt all data traffic generation for a period of time. This was handled by issuing a pause request to the phasing master in the UVM main phase, refer to Listing 5.

```
class sim_change_comp extends pcie_component;
  task main_phase(uvm_phase phase);
    // Decide to issue PAUSE request
    // Wait for appropriate time to make request
    pcie_sideband_phase_event ev = new(PCIE_CONTROL_REQ_PAUSE);

    // Write to phasing master via analysis port
    ap.write(ev);
  endtask
endclass
```

Listing 5: Pause request notification from slave component to master phasing component.

In our data traffic generation sequencers, we implemented local class variables to flag when data traffic should block. Sequences started on this sequencer would query the flag prior to generating the next data packet, in Listing 6.

```
class pcie_packet_sequencer extends pcie_sequencer;
  task pcie_control_request_phase(uvm_phase phase);
     if(get_request_type() == PCIE_CONTROL_REQ_PAUSE)
     begin
       // notify active sequences to pause generation
       pause_incoming_traffic = 1;
     end
  endtask
endclass
```

Listing 6: Packet sequencer blocks packet generation when PAUSE is requested.

Notice in the sequence, in Listing 7, the body's loop will generate a number of packets. The pause can take effect at the end of the current packet transmission. In this case, the environment need not raise an objection because the timing is not critical. When jumping out of UVM main phase, however, timing is critical.

```
class pcie_packet_sequence extends pcie_sequence;
  task body();
    while(...)
    begin
      if(packet_type == incoming && parent.pause_incoming_traffic)
      begin
          // Block until pause flag is cleared
      end
      // Do packet generation and pass to driver
    end
  endtask
endclass
```

Listing 7: Packet generation blocks when PAUSE is requested.

When the quiescent state is concluded, then the sequencer's control-complete phase executes and it lowers the pause flags, as in Listing 8. This releases the sequences to continue data traffic generation.

```
class pcie_packet_sequencer extends pcie_sequencer;
  task pcie_control_complete_phase(uvm_phase phase);
     if(get_request_type() == PCIE_CONTROL_REQ_PAUSE)
     begin
       // Notify active sequences to continue
       pause_incoming_traffic = 0;
     end
  endtask
endclass
```

Listing 8: Packet generation resumes in control-complete phase.

B. Hard Reset

In some testing scenarios, one or more hard resets must occur during simulation. The decision to hard reset was random based on constraints. It was not uncommon to issue two or more hard resets in one simulation, as in Listing 9.

```
class sim_change_comp extends pcie_component;
  task main_phase(uvm_phase phase);
    // Decide to issue HARD RESET request
    // Wait for appropriate time to make request
    pcie_sideband_phase_event ev = new(PCIE_CONTROL_REQ_HARD_RESET);

    // Write to phasing master via analysis port
    ap.write(ev);
  endtask
endclass
```

Listing 9: Hard reset request notification from slave component to master phasing component.

Timing is critical to the hard reset request as a jump out of UVM main phase will occur. All active sequences must be killed or exited prior to the jump. As in the code below, once the master has lowered its objection in control phase, the control request phase executes. Here, the packet sequencers must block to allow the sequences to exit or get killed. In Listing 10, the UVM sequencer stop_sequences function is called. This is a zero time function. In practice, we may allow a packet to finish transmission prior to killing the sequence.

```
class pcie_packet_sequencer extends pcie_sequencer;
  task pcie_control_request_phase(uvm_phase phase);
     phase.raise_objection(this, "Waiting for sequences")
     if(get_request_type() == PCIE_CONTROL_HARD_RESET)
     begin
         // Forcibly kill stimulus generation when applicable
         stop_sequences();
     end
     phase.drop_objection(this);
  endtask
endclass
```

Listing 10: Packet generation blocked when PAUSE is requested.

VII. Conclusion

While some setup is required for custom side-band phasing to enable global communication of major simulation events, the actual implementation effort is relatively minor. The custom class extensions themselves were minimal and the PCIe-specific base classes were already a project requirement. The code presented in this paper is nearly a complete implementation. An advantage of this scheme is that *all* project components are automatically notified in every instance when a major simulation event is about to occur. Each component has the time to prepare for the event while objecting in a custom phase. The requested event *cannot* occur in simulation *until* the environment is prepared.

References

[1] Accellera, "Universal Verification Methodology (UVM) 1.1d Class Reference," 2013.

[2] M. Ramalingaiah and B. Anatharaman, "OVM & UVM Techniques for On-the-fly Reset," in *Design & Verification Converence*, San Jose, 2012.

[3] C. Schmitt, P. Huynh, S. McInnis and U. Simm, "Design & Verification Conference," San Jose, 2014.

[4] IEEE Computer Society, "SystemVerilog--Unified Hardware Design, Specification, and Verification Language (1800-2012)," New York, 2013.

[5] Accellera, "Universal Verification Methodology (UVM) 1.1 User's Guide," 2011.

[6] D. Rich and A. Erickson, "Using Parameterized CLasses and Factories: The Yin and Yang of Object-Oriented Verification," in *Design & Verification Conference*, San Jose, 2012.

[7] J. Refice and M. Strickland, *Issue 4714: Jump from on branch of parallel branches does not work as intended*, Accellera Mantis UVM Database, 2013.

[8] J. Refice, *Issue 4716: Clean up "jump" logic,* Accellera Mantis UVM Database, 2013.