Run-Time Phasing in UVM: Ready for the Big Time or Dead in the Water?

John Aynsley
Doulos, Ringwood, UK

***Abstract*- This paper describes and clarifies the run-time phasing mechanism of UVM. The pre-defined UVM Run-Time Phases can be used to ensure that the run-time stimulus generation activities of various verification components are properly ordered when verification IP (VIP) is integrated. User-defined phases can be introduced when the stimulus generation activities do not correspond to any of the pre-defined phases. Phase schedules can be created from scratch or can be populated with the pre-defined phases before adding user-defined phases. Phase jumping permits stimulus to be restarted without having to abort and restart the entire simulation, but needs to be used with great care as there are several pitfalls and no safeguards. The pre-defined pre- and post- phases can be used very effectively as synchronization hooks when integrating VIP, but user-defined phases should be introduced in preference to overriding the phase methods of the pre- and post- phases.**

I. Introduction

The history of the run-time phasing mechanism of UVM can be traced from its antecedents in VMM, through a period of debate within the Accellera Working Group, to the current state of implementation in UVM-1.2. Is the run-time phasing mechanism now Ready for the Big Time, or is it Dead in the Water? If you decide to steer well clear, what are the alternatives?

The UVM run-time phasing mechanism has always caused confusion with its phase domains, synchronization of phases, the possibility of introducing user-defined phases, and the possibility of phase jumping. The goal of this paper is to explain and clarify the run-time phasing mechanism from the point of view of the user, and to answer questions such as: Should you be using the run-time phases, and what is the best way to do so? Which components should be phase-aware, and what does that mean? How do you integrate VIP that makes use of run-time phases, especially if it modifies the standard phase schedule or exploits user-defined phases? What happens if you force a phase jump while a component or sequence is busy?

II. Prerequisites

In order to get the most from this paper you should already be familiar with UVM and in particular with the concept of phases in UVM. The most important phases are the build phase, the connect phase, and the run phase, but there exist several other housekeeping phases (end_of_elaboration, start_of_simulation and so forth). Each UVM component may override the method corresponding to each phase. The method corresponding to a particular phase (e.g. **build_phase**) must execute through to completion for every component before simulation is allowed to advance to the next phase. It is only during the run phase that simulation time is allowed to advance beyond time zero. The **run_phase** method is the only phase method that is a SystemVerilog task, the others being functions.

III. Introducing The Run-Time Phases

UVM defines multiple run-time phases that execute concurrently with the run phase, which was itself a legacy from OVM. The intent of the run-time phases is to structure the run phase into an orderly progression of actions so that multiple verification IP blocks (VIPs) can each reset and configure their portion of the DUT, execute tests, and shut down in a coordinated way. There are actually twelve pre-defined run-time phases (described more fully below), the most important of which are the reset phase, the configure phase, the main phase, and the shutdown phase.

Each run-time phase method has the following general structure:

```
task XXXXX_phase(uvm_phase phase);
  phase.raise_objection(this);
  ... // Consume simulation time
```

```
    phase.drop_objection(this);
  endtask
```

Each run-time phase will end when there are no objections raised. The objection count is set to zero at the start of each run-time phase, which will therefore end immediately unless an objection is raised in the first delta cycle of the phase (i.e. before the first non-blocking assignment region of the scheduler). Each run-time phase method is a task, and thus may consume simulation time. Each run-time phase method may start sequences, and indeed would typically do so in order to generate stimulus.

Each UVM component may override the **run_phase** method, or the new run-time phase methods, or neither, or both. For example:

```
class env extends uvm_env;
  ...
  task reset_phase(uvm_phase phase);
    ...
  task configure_phase(uvm_phase phase);
    ...
  task main_phase(uvm_phase phase);
    ...
  task shutdown_phase(uvm_phase phase);
    ...
endclass

class test extends uvm_test;
  ...
  env m_env1;
  env m_env2;
  function void build_phase(uvm_phase phase);
    m_env1 = env::type_id::create("m_env1", this);
    m_env2 = env::type_id::create("m_env2", this);
    ...
  task run_phase(uvm_phase phase);
    ...
```

By default, the four run-time phases (reset, configure, main, shutdown) of the two instances of the **env** component will be synchronized with one another. The test class in this contrived example is shown with a **run_phase** task merely to illustrate the point that **run_phase** can co-exist with the new run-time phases. The reset phase and the run phase will start at the same time, time zero. The shutdown phase and the run phase will end at the same time, which is not to say that the corresponding methods will necessarily return at the same time, but it does imply that the extract phase will not start until both the shutdown phase and the run phase have ended.

If it is required that the run-time phases of the two component instances are not fully synchronized with one another, this can be achieved by placing at least one of the component instances in a separate *domain* (to be explained more fully below). This technique is key to integrating VIP where the run-time phases of different components need to be ordered carefully relative to one another.

```
    uvm_domain domain1;
    domain1 = new("domain1");
    m_env1.set_domain(domain1);
  endfunction : build_phase
```

The code fragment above is sufficient to cause the run-time phases of the two component instances **m_env1** and **m_env2** to be entirely decoupled from one another, so that it is possible that the shutdown phase of **m_env1** will have completed before the **reset_phase** of **m_env2** has started, or vice versa. If it is required that there is some partial synchronization between the phases of the two component instances, this can be achieved by making explicit calls to synchronize individual phases. Just for clarity, the following example places **m_env2** in a second named domain:

```
domain2 = new("domain2");
m_env2.set_domain(domain2);

domain1.sync(domain2);     // Sync all the phases of domain1 with the same phases of domain2
domain1.unsync(domain2); // Unsync all the phases of the two domains
domain1.sync(domain2, uvm_main_phase::get()); //  Sync just the main phases
domain1.sync(domain2, uvm_reset_phase::get(), uvm_configure_phase::get());
```

The last line above will sync the reset phase of domain1 with the configure phase of domain2. When two phases are synced, both phases start together and both end together. As a result of the calls above, the timeline of the two domains is as follows

```
domain1:          reset     -> configure -> main -> shutdown
domain2: reset -> configure ->              main -> shutdown
```

In summary, by instantiating new domains, setting UVM components into those domains, and calling the **sync** and **unsync** methods of the domains it is possible to take control over the order in which the run-time phases of components in different domains are executed relative to one another. This mechanism by itself is very useful, but there is a lot more to run-time phasing in UVM, as described below.

IV. Defining Some Terms

Before going any further it will be helpful to define some terms, because some of the terms used in association with the phasing mechanism in the UVM Class Reference can be a little confusing. The UVM Class Reference and the UVM API itself each refer to Common Phases and the UVM Domain, both of which can be rather misleading terms. Furthermore, the distinction between phases, schedules, and domains can be somewhat confusing because each of these is based on the same data type, namely **uvm_phase**.

The *Common Phases* are the nine pre-defined phases that start with the build phase and end with the final phase, namely:

1. build_phase
2. connect_phase
3. end_of_elaboration_phase
4. start_of_simulation_phase
5. run_phase
6. extract_phase
7. check_phase
8. report_phase
9. final_phase

The names given in the list above are the names of the methods of class **uvm_component** that you would override in order to define the behavior of that component in the corresponding phase. Each phase is associated with a class where the class name takes the prefix **uvm_**, e.g. **uvm_build_phase**. Phases can be identified within procedural code by calling the **get** method of the corresponding class, e.g. **uvm_build_phase::get()**, which returns a singleton object (a unique object) corresponding to that particular phase. The argument passed to each phase method is also a reference to an object that represents the phase. Because of a technicality of the UVM library, these may or may not be the same objects, even though they represent the same phase. However, two phase objects can always be compared for equality using their **is** method. For example:

```
function void build_phase(uvm_phase phase);
  assert( phase.is( uvm_build_phase::get() ) );
  ...
```

The Common Phases are executed in the order given in the list above, and unlike the run-time phases, the list cannot be extended or reordered.

The *UVM Run-Time Phases* are the twelve pre-defined phase than run concurrently with the **run_phase**, namely:

1. pre_reset_phase
2. reset_phase
3. post_reset_phase
4. pre_configure_phase
5. configure_phase
6. post_configure_phase
7. pre_main_phase
8. main_phase
9. post_main_phase
10. pre_shutdown_phase
11. shutdown_phase
12. post_shutdown_phase

Again, the UVM Run-time Phases are executed in the order given above, and each phase is associated with a predefined class where the class name takes the prefix **uvm_**. For example, the method **main_phase** is associated with the class **uvm_main_phase** and with the singleton object returned by the call to **uvm_main_phase::get()**.

A *schedule* is a collection of phases that are organized as a single directed acyclic graph (DAG), that is, a set of nodes and vertices where each node correspond to a phase and each vertex corresponds to a phase transition. If two phases share a common predecessor in the DAG they will start together, or if two phases share a common successor in the DAG they will end together. (As was observed above, two phases ending together does not necessarily mean that their phase methods will return at the same time, but it does mean that their successor will not start until both phases methods have returned.) The predefined schedule that incorporates the UVM Run-Time Phases consists of a simple linear sequence of phase transitions in the order given in the list above. An understanding of schedules is

useful in order to be able to synchronize phases, to insert user-defined phases into a predefined schedule, or to create new schedules from scratch.

A *domain* is, in effect, a container for a schedule. Each UVM component is associated with exactly one domain. Multiple domains can co-exist, and the phases that constitute the schedule of each domain may or may not be synchronized with the phases of other domains. An understanding of domains is useful in order to integrate VIP that uses run-time phases.

In terms of the SystemVerilog class hierarchy, phases, schedules, and domains all have the same base class, **uvm_phase**. A schedule is a phase object that happens to have an associated array of phases organized as the nodes of a DAG. A domain is an object of a distinct class **uvm_domain** which extends **uvm_phase** and which has an associated schedule. In other words, every fully-built domain has exactly one schedule, and every fully-built schedule has a collection of one or more phases. A partially built domain or schedule could be empty (i.e. not associated with any schedule or phases, respectively).

The *Common Domain* is a predefined domain that contains the nine Common Phases. It cannot be extended or replaced. Of the nine phases in the Common Domain, only the **run_phase** can run concurrently with any other domain. The concept is very simple and straightforward: it is only the term *Common Domain* that might be confusing.

The *UVM Domain* is a predefined domain that contains the twelve UVM Run-Time Phases listed above. The UVM Domain always executes concurrently with the **run_phase** of the Common Domain. Unlike the Common Domain, the UVM Domain can be extended or replaced, and can execute alongside other user-defined run-time domains. The UVM Run-Time Phases are not confined to the UVM Domain: they can also be added to one or more user-defined domains.

As an example of using the UVM Domain in the API, the following example creates a user-defined domain, puts a component into that domain, and then synchronizes the phases of that domain with the UVM Run-Time Phases of the UVM Domain:

```
uvm_domain domain;
domain = new("domain");                      // New user-defined domain
m_component.set_domain(domain);              // m_component put into domain
domain.sync(uvm_domain::get_uvm_domain());   // domain synced with default domain
```

The constructor for class **uvm_domain** creates a domain that is not initially associated with any schedule. In effect the user-defined domain inherits the UVM Run-Time Phases, but it is the call to **set_domain** that actually achieves this. This mechanism is not obvious, but it is important that it is properly understand. The call to **set_domain** populates that domain with the UVM Run-Time Phases, but those twelve phases are not synced with the identically-named phases of the UVM Domain. The static method **uvm_domain::get_uvm_domain** returns a reference to the object associated with the UVM Domain. The UVM Domain is the default domain into which every UVM component is initially placed, but that domain can be changed by calling **set_domain**.

V. User-Defined Phases

In order to work toward a better understanding of the mechanics of the phasing mechanism we next show how to create a user-defined phase and insert it into a schedule. The pros and cons of user-defined phases will be discussed in the later sections of this paper.

A user-defined phase is defined by a class and is often associated with a phase method that may be overridden in a component (in the same way that any predefined phase method may be overridden), although the latter is not strictly necessary. If a user-defined phase method is required, **training_phase** in this example, the first step is to define a base class for components that are to override that user-defined phase method:

```
class extended_component extends uvm_component;
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction
```

```
  virtual task training_phase(uvm_phase phase); // User-defined phase task
  endtask
endclass
```

Note that the phase method **training_phase** is not declared as a pure virtual task, because that would make it mandatory to override the task.

The next step is to define the new phase class by extending **uvm_task_phase**:

```
class my_training_phase extends uvm_task_phase; // User-defined phase class
  protected function new (string name = "");
    super.new(name);
  endfunction
```

The class should have a static function **get** that returns a singleton instance of the class, that is, the actual object that represents the user-defined phase at run-time. As described in the previous section, this function will be called whenever the phase needs to be identified, e.g. when inserting it into a schedule.

```
  static local my_training_phase m_singleton_inst;
  static function my_training_phase get;
    if (m_singleton_inst == null)
      m_singleton_inst = new("my_training_phase");
    return m_singleton_inst;
  endfunction
```

In programming terms, a phase is a functor (function object) with behavior defined by its **exec_task** method, which is called implicitly whenever the phase is started (which will depend on the schedule into which it is inserted). If a user-defined phase method is required, then **exec_task** must be overridden to call the user-defined phase method:

```
  task exec_task(uvm_component comp, uvm_phase phase);
    extended_component c;
    if ($cast(c, comp))
      c.training_phase(phase);  // Call the overridden user-defined phase task
  endtask
endclass
```

If **exec_task** were not overridden appropriately, the user-defined phase method would not be called, but the user-defined phase could still be used for synchronization purposes. This is why it is not strictly necessary to have a user-defined phase method.

The next step is to override the user-defined phase method in one or more components that extends the new component base class above:

```
class env extends extended_component;
  ...
  task training_phase(uvm_phase phase);
    phase.raise_objection(this);
```

```
    // Consume time
    phase.drop_objection(this);
  endtask
  ...
```

As with any task-based phase, it is necessary to raise an objection to prevent the phase from ending immediately. The phase will end when the number of objections returns to zero.

The final step is to insert the user-defined phase into a schedule. The simplest approach is to insert the phase into the UVM Domain from the test or top-level env class:

```
function void build_phase(uvm_phase phase);
  uvm_phase schedule;
  m_env1 = env::type_id::create("m_env1", this);
  schedule = uvm_domain::get_uvm_schedule();
  schedule.add(my_training_phase::get(), .after_phase(uvm_configure_phase::get()),
                                         .before_phase(uvm_main_phase::get()));
```

The static method **uvm_domain::get_uvm_schedule** returns a reference to the predefined schedule of the UVM Domain. The **add** method inserts the user-defined phase into the schedule of the UVM Domain, which thus makes the user-defined phase available to all components in the default UVM Domain. But note that defining a **training_phase** task in a component of the UVM Domain is not by itself sufficient to get the task called, even though the training phase is now linked into the schedule of the UVM Domain. It is also necessary to extend the component from the base class referred to by **exec_task**, which is **extended_component** in this example (otherwise the dynamic cast within **exec_task** would fail).

The **add** method inserts the user-defined phase into the predefined schedule. Note that phases are identified by calling the static **get** method of the phase class, and that the **.after_phase** and **.before_phase** arguments are being used to insert the new phase at the appropriate point of the schedule.

```
configure -> | post_configure -> pre_main | -> main
             |         training           |
```

The only domain being used in this example is the UVM Domain. This implies that all components are in the same domain, and there is no flexibility to sync or unsync phases across components. Of course, it is also possible to add user-defined phases to user-defined schedules and domains, which will be described below.

VI. Creating a Schedule from Scratch

We next show how to create a schedule from scratch. This will help clarify some of the concepts involved and also provide a further example of using the API. Again, a discussion of the methodological issues will be deferred until later in the paper.

```
uvm_domain domain1, domain2, common;
uvm_phase  schedule;

schedule = new("uvm_sched", UVM_PHASE_SCHEDULE); // Make a brand new schedule
```

Remember that phases and schedules are both of type **uvm_phase**. The constructor for class **uvm_phase** takes an argument that indicates the *phase type* of the object, which can be a node (phase), schedule, domain, or one of two internal types concerned with the implementation of the DAG.

A new phase schedule can be built entirely from user-defined phases, but it is often useful to start from the predefined run-time phases. The following call, which is entirely optional, populates the newly created schedule with the predefined run-time phases:

```
uvm_domain::add_uvm_phases(schedule);
```

Next we insert a user-defined phase, as per the previous example. It would be possible to define and insert any number of user-defined phases, or the schedule could be composed exclusively of user-defined phases:

```
schedule.add(my_training_phase::get(), .after_phase(uvm_configure_phase::get()),
                                       .before_phase(uvm_main_phase::get()));
```

Then we add the newly created schedule to a pristine user-defined domain:

```
domain1 = new("domain1");
domain1.add(schedule);
```

The rules of UVM require that we make the any run-time domain run concurrently with the **run_phase** of the common domain. Because we are creating a new schedule from scratch, this will not happen by default as it did in the previous examples:

```
common = uvm_domain::get_common_domain();
common.add(domain1, .with_phase(uvm_run_phase::get()));
```

Note that the above call exploits the fact that domains, schedules, and phases all share the same base class, **uvm_phase**. A domain can be inserted into (the schedule of) another domain as if it were itself a phase.

Finally, we can place components into the newly created domain by calling their **set_domain** method. This time, unlike the previous examples, **set_domain** does *not* populate the domain with the predefined run-time phases solely because domain1 already has a schedule attached.

```
m_env1.set_domain(domain1);
```

When building a schedule, phases can be inserted either in series or in parallel. This is shown in the following example, where a schedule is built from scratch without pre-populating the schedule with the run-time phases:

```
schedule = new("uvm_sched", UVM_PHASE_SCHEDULE);
```

Add a first phase:

```
schedule.add(uvm_reset_phase::get());
```

Insert a second phase directly after the first using the **.after_phase** argument:

```
schedule.add(uvm_main_phase::get(), .after_phase(uvm_reset_phase::get()));
```

```
    reset -> main
```

Insert the training phase in parallel with the reset phase using the **.with_phase** argument, which both starts and ends the new phase at the same time as the phase passed as an argument:

```
schedule.add(my_training_phase::get(), .with_phase(uvm_reset_phase::get()));
```

```
    | reset    | -> main
    | training |
```

Insert the configure phase after the reset phase using the **.after_phase** argument.

```
schedule.add(uvm_configure_phase::get(), .after_phase(uvm_reset_phase::get()));
```

```
    | reset -> configure | -> main
    |     training       |
```

Note that the configure phase ends at the same time as the training phase. Had we instead used the **.before_phase** argument to insert the configure phase before the main phase, the result would have been that the configure phase would have been inserted after the training phase instead of ending at the same time as the training phase:

```
schedule.add(uvm_configure_phase::get(), .before_phase(uvm_main_phase::get()));
```

```
    | reset    | -> configure -> main
    | training |
```

The choice between using **.before_phase** or **.after_phase** really matters when the DAG already contains parallel phases. When considering a split or a merge in the schedule graph, **.after_phase** means immediately after a phase whereas **.before_phase** means immediately before a phase. Any of **.before_phase**, **.after_phase**, or **.with_phase** permit phases to be inserted in parallel with existing phases, but it is important to add the phases in the right order to get the desired result.

Finally, insert the shutdown phase in parallel with the main phase:

```
schedule.add(uvm_shutdown_phase::get(), .with_phase(uvm_main_phase::get()));
```

```
    | reset    | -> configure -> | main     |
    | training |                 | shutdown |
```

VII. OVERRIDING DEFINE_DOMAIN

The previous section described a mechanism for associating a new schedule with a component, that is, a call to add the schedule to the domain followed by a call to **set_domain** to set the domain for the component. UVM offers an alternative mechanism that can be more convenient in that it avoids the need to explicitly create a schedule and add it to a domain, although this alternative mechanism has a pitfall (described below) and requires some careful thought.

The **uvm_component** class has a method **define_domain** which can be overridden to define or modify the schedule of the domain in which the component is placed, that is, the domain passed as an argument to **set_domain**.

```
function void define_domain(uvm_domain domain);
```

set_domain always calls **define_domain**, which may or may not be overridden in the user-defined component class. **define_domain** has the ability to modify the schedule of the domain passed as an argument (the domain passed to **set_domain**). Since it is possible to pass the same domain object to multiple calls to **set_domain**, it is critical that **define_domain** should check whether the domain has already been defined before attempting to add or modify the schedule. Without this check, each call to **set_domain/define_domain** could add further copies of the same phases to the schedule! The check can be done as follows:

```
  if (domain.get_parent() == null)
  begin
```

The default implementation **uvm_component::define_domain** creates a new schedule for the domain passed as an argument and adds the UVM Run-Time Phases to the domain, as described in the previous sections. A call to **super.define_domain** here would add the pre-defined run-time phases to the domain, but this is entirely optional:

```
    super.define_domain(domain);
    domain.add(my_training_phase::get(), .after_phase(uvm_configure_phase::get()),
                                         .before_phase(uvm_main_phase::get()));
  end
endfunction
```

A call to **set_domain** with a pristine domain causes **define_domain** to build the schedule for that domain:

```
function void build_phase(uvm_phase phase);
  ...
  domain1  = new("domain1");
  m_env1.set_domain(domain1);

  domain2  = new("domain2");
  m_env2.set_domain(domain2);
```

Both domain1 and domain2 will be populated (by the call to **super.define_domain**) with the pre-defined run-time phases and will each have a single copy of the training_phase. As before, it is possible to synchronize phases across domains, or not:

```
  domain1.sync(domain2, my_training_phase::get());
```

VIII. The Start and End of Each Phase

As a prelude to a discussion of phase jumping, this section will describe what happens at the start and end of each phase from the point-of-view of the component.

UVM provides three callbacks, each a method of the class **uvm_component**, which can be overridden to intercept the start and the end of each phase. The **phase_started** method is called immediately before each common and each run-time phase, the **phase_ended** method is called immediately after each common and each run-time phase, and the **phase_ready_to_end** method is called when the objection count of a run-time phase drops to zero.

The **phase_started** method has no obvious purpose aside from diagnostics or housekeeping: anything that might be done in this method could just as well be done in the regular phase method. Once a phase has started an objection must be raised within one delta cycle, otherwise the phase will end immediately.

```
function void phase_started(uvm_phase phase);
  `uvm_info("", {phase.get_full_name(), " started"}, UVM_MEDIUM)
endfunction
```

The **phase_ready_to_end** method is called for a run-time phase when the objection count for that phase drops to zero. This is "last chance saloon" for a run-time phase: the phase will end unless another objection is raised within one delta cycle, but raising an objection will have the effect of prolonging the phase. There is, however, a fixed limit of 20 iterations on this mechanism which helps catch infinite loops but also limits its usefulness. It is debatable whether it is a good idea to use of this mechanism at all: arguably it is better to keep objections raised until all activity is finished rather than prolonging the phase after activity is nominally complete.

```
function void phase_ready_to_end(uvm_phase phase);
  if ( phase.is(uvm_main_phase::get()) )
  begin
    static bit first = 1;
    if (first)
    fork
      begin
        `uvm_info("", {phase.get_full_name(), " being prolonged"}, UVM_MEDIUM)
        first = 0;
        phase.raise_objection(this);
        #(m_delay);
        phase.drop_objection(this);
      end
    join_none
  end
endfunction
```

Since sequences are not components, sequences do not have access to the **phase_ready_to_end** callback. Instead, a background sequence could call the **uvm_phase::wait_for_state** method if it needed to prolong a phase. For example:

```
task body;
  phase = get_starting_phase();
  fork
    if (phase != null)
```

```
    begin
      phase.wait_for_state(UVM_PHASE_READY_TO_END);
      phase.raise_objection(this);
      #100;  // Prolong the phase
      phase.drop_objection(this);
    end
  join_none
  ...
```

The **phase_ended** method is called immediately after each common and each run-time phase regardless of how the phase may have ended. This is "past the point of no return" for a run-time phase. Unlike **phase_ready_to_end**, which is not called in the event of a phase jump, **phase_ended** *can* be used to tidy up after a phase has been aborted, for example following a phase jump (see below).

IX. Phase Jumping

The UVM phasing mechanism permits jumps between phases. In other words, instead of a phase being allowed to run to completion, the phase is forced to end prematurely and the domain jumps to some phase elsewhere in its schedule. An obvious application of phase jumping would be to jump back to the start of the run phase in order to execute another test without the need to start a new simulation. The phase jumping API in UVM 1.2 has been tidied up to the point where it is usable and its effects predictable, although it is very important to understand its limitations because there are very few safeguards built-in. Phase jumping should only be used with great care.

At its simplest, a phase jump can be achieved by calling the **jump** method of class **uvm_domain**:

```
// Test or env
task run_phase(uvm_phase phase);
  #(...) // Wait until middle of main_phase
  domain1.jump(uvm_reset_phase::get());

  #(...) // Wait until middle of main_phase second time around
  domain1.jump(uvm_extract_phase::get());
```

The call to **domain1.jump** abandons whatever phase is currently being executed by the components associated with domain1, assumed here to be the main phase, and jumps to the phase passed as an argument. The call to **domain1.jump** only affects the execution of components in domain1. The code shown above would typically be running in some other domain, although it could itself be in domain1! Jumps can be forward or backward in the phase sequence of a domain. Backward jumps should be restricted to run-time phases, since UVM is not designed to support jumps to the Common Phases that precede **run_phase**. Forward jumps should be restricted to the Common Phases that follow the **run_phase**, since each run-time phase generally assumes the successful completion of the preceding phases. It is generally unsafe to jump over run-time phases or to jump into one strand of a number of parallel phases, and the phase jumping mechanism will not detect mistakes. In other words, phase jumping should be restricted to either restarting from the beginning, perhaps going back to the reset or configure phase, or aborting and jumping to the end, perhaps to the extract phase.

A phase jump in a given domain will not cause a jump in a second domain that happens to be synced to the first. If a domain jumps backward, a synced domain would, in effect, wait for the first domain to catch up. If a domain jumps forward, a synced domain would not also jump forward but would be allowed to proceed forward at its own pace until it caches up with the first domain. This happens because the next phase transition of each domain is still synced with the corresponding phase transition of the other, even though the first domain has jumped elsewhere in its schedule.

The topic of hard versus soft jumps has been much discussed in the Accellera UVM Working Group. A hard jump is an immediate, unconditional jump, where the component that is the target of the jump has no choice but to jump and jump immediately. A soft jump would be a jump request where the component that is the target of the jump may get the chance to prolong the current phase before the jump is executed or may even reject the jump request entirely. UVM 1.2 only supports hard jumps.

The challenge with implementing a hard jump is to restore the target components to a safe state following the jump, but UVM offers no safeguards here. On a jump out of a given phase, the phase method and all its child processes are killed immediately, which will include any sequences that were started from the phase method. Any objects that were only referenced from the killed processes will be swept up by SystemVerilog's garbage collection mechanism. Moreover, the objection counts associate with the phase will automatically be reset to zero. This is all well and good with respect to processes and objections, but it is no guarantee that the target components will have been left in a consistent state or that allocated resources will have been freed. However, it is possible to use the **phase_ended** method of the target components to tidy up after a phase jump:

```
function void phase_ended(uvm_phase phase);
  if (phase.get_objection() != null)
    assert( phase.get_objection_count() == 0); // Objections will have been dropped
  if (m_busy)
    ... // Tidy up the component following a phase jump
  m_busy = 0;
endfunction
```

X. Integrating Phase-Aware Components

When authoring UVM components it is important to distinguish between the use of run-time phases and the use of state. Phase-aware code is code that behaves differently according to which of the UVM Run-Time Phases or which user-defined phase is currently executing. The only code that should be phase aware is code to generate stimulus, that is, tests, envs, and sequences. On the other hand, monitors, drivers, agents, subscribers, and scoreboards, although they may be state-dependent, should not be phase aware. This principle seems to be well-established in the UVM community. For example, a driver may set and then test a flag that indicates that it is currently in a certain mode, but it should not explicitly test which phase is running. By this definition, a sequence could use the **starting_phase** variable to raise and drop objections without necessarily being phase-aware. A test writer can force a sequence *not* to be phase aware by not setting its **starting_phase** variable.

Components can generate stimulus without relying on the run-time phases. It would be entirely possible to define separate sequences for each of the "modes" of a device, i.e. reset, configuration, and so forth, and then start these sequences in the appropriate order from the **run_phase** method of a test or env. In fact, even when using run-time phases, it is necessary to define separate sequences to be started in each distinct phase (a reset sequence, a configure sequence, and so forth). So the question that naturally arises is whether it is worth using run-time phases at all. Should stimulus be generated by simply starting the right sequences in the right order?

When authoring and distributing verification components (VIP), maximum flexibility can be achieved by providing a set of sequences for reset, configuration, shutdown, and so forth that are not tied to specific run-time phases. The VIP integrator can then choose between starting those sequences from the appropriate run-time phase method of their env (or test), or starting them in order from the **run_phase** method. So there is no particular advantage in distributing VIP that uses the UVM Run-Time Phases. On the contrary, it could be argued that the VIP should be kept phase-agnostic. The VIP integrator can then start the appropriate sequences from their top-level env (or test) in the appropriate order according to the rules laid down by the VIP creators.

The challenge comes when envs (or other UVM components) that start sequences are themselves reused as part of a larger verification environment. Ultimately, sequences have to be started from phase methods. If the envs being integrated start their sequences from their **run_phase** method, then it may be impossible to get the sequences started from different envs in the correct order, because each env has "hard wired" the order in which sequences are started within the procedural code of its **run_phase** method. Of course, you can always rip up the existing code and start again! But with the goal of reuse in mind, the better approach is for the VIP integrator to use the UVM Run-Time Phase methods to start sequences. Then, when integrating envs that each use the Run-Time Phase methods, the

super-integrator can take advantage of the run-time phasing mechanisms discussed in this paper to orchestrate the sequencing of multiple envs. It is very straightforward to place each component that overrides run-time phase methods into the appropriate domain (just a call to **set_domain**), and then to sync phases across domains as required. For example:

```
function void top_env::build_phase(uvm_phase phase);
  ...
  m_serial_env = serial_env::type_id::create("m_serial_env", this);
  m_bus1_env   = bus1_env  ::type_id::create("m_bus1_env", this);
  ...
  domain2 = new("domain2");
  m_bus1_env.set_domain(domain2);
  domain2.sync(uvm_domain::get_uvm_domain(), uvm_reset_phase::get(),
                                             uvm_configure_phase::get());
endfunction : build_phase
```

XI. Pre-defined versus User-defined Phases

The UVM Run-Time Phases provide a possible starting point for run-time phasing, but may be insufficient. If particular VIP requires further phases beyond the predefined reset-configure-main-shutdown phases, user-defined phases may be introduced as described in this paper. Exactly the same mechanisms can be used to synchronize pre-defined and user-defined phases. For example:

```
  domain2.sync(uvm_domain::get_uvm_domain(), my_training_phase::get(),
                                             uvm_configure_phase::get());
```

When integrating VIP that uses the UVM Run-Time Phases, it is always necessary to understand exactly how the VIP creator has used and interpreted each of the pre-defined run-time phases. The UVM 1.2 Class Reference now tries to describe the intent of each of the twelve predefined run-time phases, but there is still plenty of room for ambiguity, particularly with respect to the eight pre- and post- phases. For example, **uvm_post_configure_phase** and **uvm_pre_main_phase** have very similar descriptions. In order to avoid making unwarranted assumptions about the meaning of the pre-defined run-time phase, it has been argued [2] that they should not be used at all and that user-defined phases should be used exclusively in order to force the VIP integrator to pay attention. Certainly it would be unwise to use the pre-defined reset phase for any purpose other than reset, and better to introduce a user-defined phase instead.

This paper takes a middle line. The four principal pre-defined phases, reset-configure-main-shutdown, provide a general and comprehensible framework that should be used as a starting point for VIP integration where their obvious meanings can be assumed, but user-defined phases should be introduced beyond that. In other words, use the **reset_phase**, **configure_phase**, **main_phase**, and **shutdown_phase** for reset, configuration, main function, and shutdown respectively, and add user-defined phases for anything else. This does not excuse the VIP integrator from considering the ordering of the phases across components, but it does provide a framework around which more exotic requirements can be introduced as required.

XII. Pre- and Post- Phases

This paper recommends that you do *not* override the eight pre- and post- phases of the twelve UVM Run-Time Phases. In the opinion of the author, their definitions are too vague to be relied upon. It is better to introduce new user-defined phases with descriptive names, e.g. protocol_negotiation_phase, training_phase.

There is a further reason for *not* overriding the pre- and post- phases. When two phases are synchronized using the **sync** method, the phases both start together and end together. The UVM API does not provide a direct way to constrain a phase in one domain to start as a phase in another domain ends, but this is a typical requirement when ordering phases across multiple domains to integrate VIP. This can be achieved, however, by using the pre- and post- phases exclusively as synchronization phases. For example:

```
domain2.sync(uvm_domain::get_uvm_domain(), uvm_configure_phase::get(),
                                           uvm_post_configure_phase::get());
domain2.sync(uvm_domain::get_uvm_domain(), uvm_shutdown_phase::get(),
                                           uvm_pre_shutdown_phase::get());
```

```
uvm domain: reset -> configure ->(post_configure)-> main ->(pre_shutdown)-> shutdown
domain2   : reset ->           -> configure      -> main -> shutdown
```

The same technique can be applied when synchronizing a user-defined phase in the VIP being integrated:

```
domain2.sync(uvm_domain::get_uvm_domain(), my_training_phase::get(),
                                           uvm_pre_main_phase::get());
```

Reserving the pre- and post- phases for synchronization hooks turns out to be really convenient. It would even be possible to define pre- and post- phases for user-defined phases just for this purpose alone:

```
class my_pre_training_phase extends uvm_task_phase; // User-defined phase class
  ...
  // task exec_task(uvm_component comp, uvm_phase phase);
  //   deliberately omitted because pre_training_phase will not be overridden
endclass

function void bus1_env::define_domain(uvm_domain domain);
  if (domain.get_parent() == null) // Beware multiple calls!
  begin
    super.define_domain(domain);
    domain.add(my_pre_training_phase ::get(),
               .after_phase(uvm_configure_phase::get()));
    domain.add(my_training_phase::get(),
               .after_phase(my_pre_training_phase::get()));
    domain.add(my_post_training_phase::get(),
               .after_phase(my_training_phase::get()));
  end
endfunction

domain2.sync(uvm_domain::get_uvm_domain(), my_post_training_phase::get(),
                                           uvm_pre_main_phase::get());
```

XIII. Summary of Recommendations

1. Do define sequences for the specific purposes of reset, configuration, and so forth
2. Do not make sequences phase-aware
3. Do not use the UVM Run-Time Phase methods other than to start sequences in components that perform stimulus generation
4. Do override the pre-defined reset, configure, main, and shutdown phase methods, but only when the behavior is appropriate to the name
5. Do introduce user-defined phases when none of the above pre-defined phases are appropriate
6. Do not override the **run_phase** method to start sequences that could meaningfully be started from the reset, configure, main, or shutdown phases.
7. Do not override the eight pre- and post- phases of the UVM Run-Time Phases
8. Do introduce new domains and do sync phases between domains when integrating VIP
9. Do use the pre- and post- phases for synchronization where appropriate
10. Do not use phase jumping casually. There are no built-in safeguards
11. Do plan any phase jumps carefully to ensure components are left in a consistent state

References

[1] UVM 1.2 Class Reference, http://www.accellera.org/downloads/standards/uvm, June 2014.
[2] Justin Refice, *Run-Time Phasing in the UVM: The Long Lost User's Guide*, internal Accellera document, March 2013.