UVM 1.0 Errata Documentation

This errata document details the Natural Doc (API) changes found in the latest Base Class Library (BCL) release relative to the officially approved Accellera UVM 1.0 spec. The intention of this document is to aid developers utilizing this version of the release so that a very clear set of changes are described.

The UVM committee within Accellera provides four documents for the community.

1. An officially sanctioned and Accellera approved standards document, also known as our API Reference Guide which describes the UVM feature by feature in API format. This is considered a specification document for the UVM and anyone can use it to create their own implementation (should they choose).
2. A BCL implementation of the UVM. This is implemented in SystemVerilog and is a set of base classes and utilities put together to enable the creation of test environments.
3. A User's Guide. This details an overview of the UVM, what it contains, how it should be used, and methodology recommendations to enable VIP reuse.
4. An Errata document. Describes API changes made in the current release of the BCL relative to the officially approved standard.

We decided as a committee to release the BCL, UG, and Errata document more often than the standards document. This would allow the UVM implementation and User's Guide to be more nimble, responsive, and fluid according to end user needs. This also required however that we detail any changes in the API Natural Docs relative to the approved standard so that EDA companies, 3rd party vendors, and end user developers understood the differences.

For this version of errata it is based on the Accellera approved UVM 1.0 version approved on February 18th, 2011. This API spec can be found here:

http://www.accellera.org/activities/vip/

And is called the "Class Reference Manual".

The formatting for this Errata document is as follows:

~~Text shown crossed and red~~ removes existing material. Text shown underlined and blue adds new material without disturbing the existing material.

This document is organized according to the main chapters found in the API UVM spec.

Base:

CHANGE SET #1: Add to uvm_transaction.

BCL LOCATION: distrib/src/base/uvm_transaction.svh
PDF LOCATION: page 22

The uvm_transaction class is the root base class for UVM transactions. Inheriting all the methods of uvm_object, uvm_transaction adds a timing and recording interface.

Use of the class *uvm_transaction* as a base for user-defined transactions is deprecated. Its subtype, uvm_sequence_item, shall be used as the base class for all user-defined transaction types.

CHANGE SET #2: Add default values to uvm_phase::new()

LOCATION: distrib/src/base/uvm_phases.svh
PDF LOCATION: page 55

new

```
function new(string         name       = "uvm_phase",
             uvm_phase_type phase_type = UVM_PHASE_SCHEDULE,
             uvm_phase      parent     = null                              )
```

CHANGE SET #3: Change uvm_phase::find()

LOCATION: distrib/src/base/uvm_phases.svh
PDF LOCATION: page 56

find

```
function uvm_phase find(uvm_phase phase ~~string name~~,
                    bit          stay_in_scope      = 1                    )
```

Locate the phase node with the specified *phase* IMP and return its handle. With *stay_in_scope* set, searches only within this phase's schedule or domain.

~~Locate a phase node with the specified *name* and return its handle. Look first within the current schedule, then current domain, then global~~

CHANGE SET #4: Replace uvm_phase::add_phase() and add_schedule() with add()

LOCATION: distrib/src/base/uvm_phases.svh
PDF LOCATION: page 57

add

```
function void add(uvm_phase phase,
                  uvm_phase with_phase   = null,
                  uvm_phase after_phase  = null,
                  uvm_phase before_phase = null                              )
```

Build up a schedule structure inserting phase by phase, specifying linkage
Phases can be added anywhere, in series or parallel with existing nodes

phase	handle of singleton derived imp containing actual functor. by default the new phase is appended to the schedule
with_phase	specify to add the new phase in parallel with this one
after_phase	specify to add the new phase as successor to this one
before_phase	specify to add the new phase as predecessor to this one

~~add_schedule~~

```
~~function void add_schedule(uvm_phase schedule,~~
~~                          uvm_phase with_phase = null,~~
~~                          uvm_phase after_phase = null,~~
~~                          uvm_phase before_phase = null)~~
```

~~Build up schedule structure by adding another schedule flattened within it.~~

~~Inserts a schedule structure hierarchically within the enclosing schedule's graph. It is essentially flattened graph-wise, but the hierarchy is preserved by the 'm_parent' handles which point to that schedule's begin node.~~

~~schedule - handle of new schedule to insert within this one~~
~~with_phase - specify to add the schedule in parallel with this phase node~~
~~after_phase - specify to add the schedule as successor to this phase node~~
~~before_phase - specify to add the schedule as predecessor to this phase node~~

~~add_phase~~

```
~~function void add_phase(uvm_phase phase,
                        uvm_phase with_phase = null,
                        uvm_phase after_phase = null,
                        uvm_phase before_phase = null)~~
```

~~Build up a schedule structure inserting phase by phase, specifying linkage~~

~~Phases can be added anywhere, in series or parallel with existing nodes~~

~~*phase*~~	~~handle of singleton derived imp containing actual functor. by default the new phase is appended to the schedule~~
~~*with_phase*~~	~~specify to add the new phase in parallel with this one~~
~~*after_phase*~~	~~specify to add the new phase as successor to this one~~
~~*before_phase*~~	~~specify to add the new phase as predecessor to this one~~

CHANGE SET #5: Add 'hier' arg with default value=0 to uvm_phase::get_schedule()

LOCATION: distrib/src/base/uvm_phases.svh
PDF LOCATION: page 58

get_schedule

```
function uvm_phase get_schedule(bit hier = 0)
```

Returns the topmost parent schedule node, if any, for hierarchical graph traversal

---

CHANGE SET #6: Add 'hier' arg with default value=0 to uvm_phase::get_schedule_name() plus additional changes

LOCATION: distrib/src/base/uvm_phases.svh
PDF LOCATION: page 58

get_schedule_name

```
function string get_schedule_name(bit hier = 0 )
```

Returns the schedule name associated with this phase node

~~Accessor to return the schedule name associated with this schedule~~

CHANGE SET #7: Add the following methods in uvm_phases:
- ***find_by_name()***
- ***get_full_name()***
- ***get_domain()***
- ***get_imp()***
- ***get_domain_name()***

LOCATION: distrib/src/base/uvm_phases.svh
PDF LOCATION: N/A

find_by_name

```
function uvm_phase find_by_name(string name,
                                bit    stay_in_scope = 1          )
```

Locate a phase node with the specified *name* and return its handle. With *stay_in_scope* set, searches only within this phase's schedule or domain.

get_full_name

```
virtual function string get_full_name()
```

Returns the full path from the enclosing domain down to this node. The singleton IMP phases have no hierarchy.

get_domain

```
function uvm_domain get_domain()
```

Returns the enclosing domain

get_imp

```
function uvm_phase get_imp()
```

Returns the phase implementation for this this node. Returns null if this phase type is not a UVM_PHASE_LEAF_NODE.

get_domain_name

```
function string get_domain_name()
```

Returns the domain name associated with this phase node

CHANGE SET #8: Add to sync and unsync relationship to uvm_phase before the description of the sync function. Change sync and unsync API's:

LOCATION: distrib/src/base/uvm_phases.svh
PDF LOCATION: page 59

sync and unsync

Add soft sync relationships between nodes

Summary of usage

```
target::sync(.source(domain)

               [,.phase(phase)[,.with_phase(phase)]]);

target::unsync(.source(domain)

                 [,.phase(phase)[,.with_phase(phase)]]);
```

Components in different schedule domains can be phased independently or in sync with each other. An API is provided to specify synchronization rules between any two domains. Synchronization can be done at any of three levels:

- the domain's whole phase schedule can be synchronized
- a phase can be specified, to sync that phase with a matching counterpart
- or a more detailed arbitrary synchronization between any two phases

Each kind of synchronization causes the same underlying data structures to be managed. Like other APIs, we use the parameter dot-notation to set optional parameters.

When a domain is synced with another domain, all of the matching phases in the two domains get a 'with' relationship between them. Likewise, if a domain is unsynched, all of the matching phases that have a 'with' relationship have the dependency removed. It is possible to sync two domains and then just remove a single phase from the dependency relationship by unsyncing just the one phase.

sync

```
function void sync(    uvm_domain     target,
                       uvm_phase      phase          = null,
                       uvm_phase      with_phase     = null          )
```

Synchonize two domains, fully or partially

target	handle of target domain to synchronize this one to
phase	optional single phase in this domain to synchronize, otherwise sync all
with_phase	optional different target-domain phase to synchronize with, otherwise use *phase* in the target domain

unsync

```
function void unsync(uvm_domain target,
                     uvm_phase   phase      = null,
                     uvm_phase   with_phase = null                  )
```

Remove synchronization between two domains, fully or partially

target	handle of target domain to remove synchronization from
phase	optional single phase in this domain to un-synchronize, otherwise unsync all
with_phase	optional different target-domain phase to un-synchronize with, otherwise use *phase* in the target domain

---

CHANGE SET #9: Add to uvm_domain.

BCL LOCATION: distrib/src/base/uvm_phases.svh
PDF LOCATION: page 61

uvm_domain

Phasing schedule node representing an independent branch of the schedule. Handle used to assign domains to components or hierarchies in the testbench

Summary

uvm_domain

Phasing schedule node representing an independent branch of the schedule.

Class Hierarchy

uvm_void
uvm_object
uvm_phase
uvm_domain

Class Declaration

```
class uvm_domain extends uvm_phase
```

Methods

get_domains	Provies a list of all domains in the provided *domains* argument.
get_uvm_schedule	
get_common_domain	Get the "common" domain, which consists of the common phases that all components execute in sync with each other. ~~Get the common domain objection which consists of the common phases that all components executed together (build, connect, ..., report, final).~~
add_uvm_phases	Appends to the given *schedule* the built-in UVM phases.
get_uvm_domain	Get a handle to the singleton *uvm* domain
new	Create a new instance of a phase domain.

Methods

get_domains

```
static function void get_domains(output uvm_domain domains[string])
```

Provies a list of all domains in the provided *domains* argument.

get_uvm_schedule

```
static function uvm_phase get_uvm_schedule()
```

get_common_domain

```
static function uvm_domain get_common_domain()
```

Get the "common" domain, which consists of the common phases that all components execute in sync with each other. Phases in the "common" domain are build, connect, end_of_elaboration, start_of_simulation, run, extract, check, report, and final. ~~Get the common domain objection which consists of the common phases that all components executed together (build, connect, ..., report, final).~~

add_uvm_phases

```
static function void add_uvm_phases(uvm_phase schedule)
```

Appends to the given *schedule* the built-in UVM phases.

get_uvm_domain

```
static function uvm_domain get_uvm_domain()
```

Get a handle to the singleton *uvm* domain

new

```
function new(string name)
```

Create a new instance of a phase domain.

TLM:

CHANGE SET #11: Change uvm_pair as follows:

BCL LOCATION: distrib/src/comps/uvm_pair.svh
PDF LOCATION: page 347

uvm_pair classes

This section defines container classes for handling value pairs.

Contents

uvm_class_pair ~~uvm_pair~~ #(T1,T2)

Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.

Summary

uvm_class_pair ~~uvm_pair~~ #(T1,T2)

Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.

Class Hierarchy

uvm_void
uvm_object
uvm_class_pair#(T1,T2)

Class Declaration

```
class uvm_class_pair #(
__type T1 = int,
__       T2 = T1
) extends uvm_object
```

Variables	
T1 first	The handle to the first object in the pair
T2 second	The handle to the second object ~~second variable~~ in the pair
Methods	
new	Creates an instance that holds a handle to two objects ~~of uvm_pair that holds two built-in type values~~.

Variables

T1 first

```
T1 first
```

The handle to the first object in the pair

T2 second

```
T2 second
```

The handle to the second object ~~second variable~~ in the pair

Methods

new

```
function new (string name = "",
              T1     f    = null,
              T2     s    = null                                    )
```

Creates an instance that holds a handle to two objects ~~of uvm_pair that holds two built-in type values~~. The optional name argument gives a name to the new pair object.

uvm_built_in_pair #(T1,T2)

Container holding two variables of built-in types (int, string, etc.). The types are specified by the type parameters, T1 and T2.

Summary

uvm_built_in_pair #(T1,T2)

Container holding two variables of built-in types (int, string, etc.)

Class Hierarchy

uvm_void
uvm_object
~~uvm_transaction~~
uvm_built_in_pair#(T1,T2)

Class Declaration

```
class uvm_built_in_pair #(
  type T1 = int,
       T2 = T1
) extends uvm_object ~~uvm_transaction~~
```

Variables	
T1 first	The first value in the pair
T2 second	The second value in the pair
Methods	
new	Creates an instance that holds two built-in type values ~~of uvm_pair that holds a handle to two elements, as provided by the first two arguments~~.

VARIABLES

T1 first

```
T1 first
```

The first value in the pair

T2 second

```
T2 second
```

The second value in the pair

Methods

new

```
function new (string name = ""                                    )
```

Creates an instance that holds two built-in type values ~~of uvm_pair that holds two built-in type values~~. The optional name argument gives a name to the new pair object.

CHANGE SET #12: Change uvm_tlm_generic_payload as follows.

BCL LOCATION: distrib/src/tlm2/tlm2_generic_payload.svh
PDF LOCATION: page 243, 244

The elements in the byte enable array shall be interpreted as follows. A value of 8'h00 ~~0~~ shall indicate that that corresponding byte is disabled, and a value of 8'hFF ~~1~~ shall indicate that the corresponding byte is enabled.

(…)

If the byte enable pointer is not empty ~~is non-null~~, the target shall either implement the semantics of the byte enable as defined below or shall generate a standard error response. The recommended response status is UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE.

CHANGE SET #13: Change uvm_component as follows.

BCL LOCATION: distrib/src/base/uvm_component.svh
PDF LOCATION: starting on page 289

Phasing Interface	These methods implement an interface which allows all components to step through a standard schedule of phases, or a customized schedule, and also an API to allow independent phase domains which can jump like state machines to reflect behavior e.g.
build_phase	The Pre-Defined Phases::uvm_build_phase phase implementation

	method.
connect_phase	The Pre-Defined Phases::uvm_connect_phase phase implementation method.
end_of_elaboration_phase	The Pre-Defined Phases::uvm_end_of_elaboration_phase phase implementation method.
start_of_simulation_phase	The Pre-Defined Phases::uvm_start_of_simulation_phase phase implementation method.
run_phase	The Pre-Defined Phases::uvm_run_phase phase implementation method.
pre_reset_phase	The Pre-Defined Phases::uvm_pre_reset_phase phase implementation method.
reset_phase	The Pre-Defined Phases::uvm_reset_phase phase implementation method.
post_reset_phase	The Pre-Defined Phases::uvm_post_reset_phase phase implementation method.
pre_configure_phase	The Pre-Defined Phases::uvm_pre_configure_phase phase implementation method.
configure_phase	The Pre-Defined Phases::uvm_configure_phase phase implementation method.
post_configure_phase	The Pre-Defined Phases::uvm_post_configure_phase phase implementation method.
pre_main_phase	The Pre-Defined Phases::uvm_pre_main_phase phase implementation method.
main_phase	The Pre-Defined Phases::uvm_main_phase phase implementation method.
post_main_phase	The Pre-Defined Phases::uvm_post_main_phase phase implementation method.
pre_shutdown_phase	The Pre-Defined Phases::uvm_pre_shutdown_phase phase implementation method.
shutdown_phase	The Pre-Defined Phases::uvm_shutdown_phase phase implementation method.
post_shutdown_phase	The Pre-Defined Phases::uvm_post_shutdown_phase phase implementation method.
extract_phase	The Pre-Defined Phases::uvm_extract_phase phase implementation method.
check_phase	The Pre-Defined Phases::uvm_check_phase phase implementation method.
report_phase	The Pre-Defined Phases::uvm_report_phase phase implementation method.
final_phase	The Pre-Defined Phases::uvm_final_phase phase implementation method.
phase_started	Invoked at the start of each phase.
phase_ended	Invoked at the end of each phase.

(...)

build_phase

```
virtual function void build_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_build_phase phase implementation method.

(...)

connect_phase

```
virtual function void connect_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_connect_phase phase implementation method.

(...)

end_of_elaboration_phase

```
virtual function void end_of_elaboration_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_end_of_elaboration_phase phase implementation method.

(...)

start_of_simulation_phase

```
virtual function void start_of_simulation_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_start_of_simulation_phase phase implementation method.

(...)

run_phase

```
virtual task run_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_run_phase phase implementation method.

(...)

pre_reset_phase

```
virtual task pre_reset_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_pre_reset_phase phase implementation method.

(...)

reset_phase

```
virtual task reset_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_reset_phase phase implementation method.

(...)

post_reset_phase

```
virtual task post_reset_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_post_reset_phase phase implementation method.

(...)

pre_configure_phase

```
virtual task pre_configure_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_pre_configure_phase phase implementation method.

(...)

configure_phase

```
virtual task configure_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_configure_phase phase implementation method.

(...)

post_configure_phase

```
virtual task post_configure_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_post_configure_phase phase implementation method.

(...)

pre_main_phase

```
virtual task pre_main_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_pre_main_phase phase implementation method.

(...)

main_phase

```
virtual task main_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_main_phase phase implementation method.

(...)

post_main_phase

```
virtual task post_main_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_post_main_phase phase implementation method.

(...)

pre_shutdown_phase

```
virtual task pre_shutdown_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_pre_shutdown_phase phase implementation method.

(...)

shutdown_phase

```
virtual task shutdown_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_shutdown_phase phase implementation method.

(...)

post_shutdown_phase

```
virtual task post_shutdown_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_post_shutdown_phase phase implementation method.

(...)

extract_phase

```
virtual function void extract_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_extract_phase phase implementation method.

(...)

check_phase

```
virtual function void check_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_check_phase phase implementation method.

(...)

report_phase

```
virtual function void report_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_report_phase phase implementation method.

(...)

final_phase

```
virtual function void final_phase(uvm_phase phase)
```

The Pre-Defined Phases::uvm_final_phase phase implementation method.

Components:

CHANGE SET #14: Change uvm_component::set_domain()

BCL LOCATION: distrib/src/base/uvm_component.svh
PDF LOCATION: page 301

set_domain

```
function void set_domain(uvm_domain domain,
                         int        hier     = 1            )
```

Apply a phase domain to this component and, if *hier* is set, recursively to all its children ~~(by default, also to it's children)~~.

Calls the virtual define_domain method, which derived components can override to augment or replace the domain definition of ita base class.

~~Get a copy of the schedule graph for this component base class as defined by virtual define_phase_schedule(), and add an instance of that to our domain branch in the master phasing schedule graph, if it does not already exist.~~

CHANGE SET #15: Delete uvm_component::get_schedule()

BCL LOCATION: distrib/src/base/uvm_component.svh
PDF LOCATION: page 301

~~get_schedule~~

```
~~function uvm_domain get_schedule()~~
```

~~Return handle to the phase schedule graph that applies to this component~~

CHANGE SET #16: Replace uvm_component::define_phase_schedule() with define_domain()

BCL LOCATION: distrib/src/base/uvm_component.svh
PDF LOCATION: page 301

define_domain

```
virtual protected function void define_domain(uvm_domain domain)
```

Builds custom phase schedules into the provided *domain* handle.

This method is called by set_domain, which integrators use to specify this component belongs in a domain apart from the default 'uvm' domain.

Custom component base classes requiring a custom phasing schedule can augment or replace the domain definition they inherit by overriding <defined_domain>. To augment, overrides would call super.define_domain(). To replace, overrides would not call super.define_domain().

The default implementation adds a copy of the *uvm* phasing schedule to the given *domain*, if one doesn't already exist, and only if the domain is currently empty.

Calling set_domain with the default *uvm* domain (see <uvm_domain::get_uvm_domain>) on a component with no *define_domain* override effectively reverts the that component to using the default *uvm* domain. This may be useful

If a branch of the testbench hierarchy defines a custom domain, but some child sub-branch should remain in the default *uvm* domain, call set_domain with a new domain instance handle with *hier* set. Then, in the sub-branch, call set_domain with the default *uvm* domain handle, obtained via uvm_domain::get_uvm_domain().

Alternatively, the integrator may define the graph in a new domain externally, then call set_domain to apply it to a component.

~~define_phase_schedule~~

```
~~virtual protected function uvm_phase                    string name~~
~~define_phase_schedule(uvm_domain domain,~~
```

~~Builds and returns the required phase schedule subgraph for this component base~~
~~Here we define the structure and organization of a schedule for this component base~~

~~type (uvm_component). We give that schedule a name (default 'uvm') and return a handle to it to the caller (either the set_domain() method, or a subclass's define_phase_schedule() having called super.define_phase_schedule(), ready to be added into the main schedule graph.~~
~~Custom component base classes requiring a custom phasing schedule to augment or replace the default UVM schedule can override this method. They can inherit the parent schedule and build on it by calling super.define_phase_schedule(MYNAME) or they can create a new schedule from scratch by not calling the super method.~~

CHANGE SET #17: Change uvm_component::stop() to stop_phase() as follows:
BCL LOCATION: distrib/src/base/uvm_component.svh
PDF LOCATION: page 303

stop_phase

virtual task stop_phase(uvm_phase phase ~~string ph_name~~)

The stop_phase task is called when this component's enable_stop_interrupt bit is set and <global_stop_request> is called during a task-based phase, e.g., run.

Before a phase is abruptly ended, e.g., when a test deems the simulation complete, some components may need extra time to shut down cleanly. Such components may implement stop_phase to finish the currently executing transaction, flush the queue, or perform other cleanup. Upon return from stop_phase, a component signals it is ready to be stopped.

The *stop_phase* method will not be called if enable_stop_interrupt is 0.

The default implementation is empty, i.e., it will return immediately.

This method should never be called directly.

CHANGE SET #18: Add new method uvm_component::phase_ready_to_end() after phase_started() and before phase_ended().

BCL LOCATION: distrib/src/base/uvm_component.svh
PDF LOCATION: page 300

phase_ready_to_end

virtual function void phase_ready_to_end (uvm_phase phase)

Invoked when all objections to ending the given *phase* have been dropped, thus indicating that *phase* is ready to end. All this component's processes forked for the given phase will be killed upon return from this method. Components needing to consume delta cycles or advance time to perform a clean exit from the phase may raise the phase's objection.

```
phase.raise_objection(this,"Reason");
```

This effectively resets the wait-for-all-objections-dropped loop for *phase*. It is the responsibility of this component to drop the objection once it is ready for this phase to end (and processes killed).

Macros:

CHANGE SET #19: Remove macros related to new uvm_sequence_library class, which are not yet part of the approved standard.
BCL LOCATION: distrib/macros/uvm_sequence_defines.svh
PDF LOCATION: page 378

~~SEQUENCE LIBRARY~~

~~`uvm_add_to_sequence_library~~	~~Adds the given sequence *TYPE* to the given sequence library *LIBTYPE*~~
~~`uvm_sequence_library_utils~~	~~Declares the infrastructure needed to define extensions to the <uvm_sequence_library> class.~~

Globals:

CHANGE SET #20: Replace Enumerates for uvm_phase_type in GLOBALS:
BCL LOCATION: distrib/src/base/uvm_object_globals.svh
PDF LOCATION: page 603

uvm_phase_type

This is an attribute of a uvm_phase object which defines the phase type.

UVM_PHASE_IMP	The phase object is used to traverse the component hierarchy and call the component phase method as well as the *phase_started* and *phase_ended* callbacks. These nodes are created by the phase macros, `uvm_builtin_task_phase, `uvm_builtin_topdown_phase, and `uvm_builtin_bottomup_phase. These nodes represent the phase type, i.e. uvm_run_phase, uvm_main_phase.

UVM_PHASE_NODE	The object represents a simple node instance in the graph. These nodes will contain a reference to their corresponding IMP object.
UVM_PHASE_SCHEDULE	The object represents a portion of the phasing graph, typically consisting of several NODE types, in series, parallel, or both.
UVM_PHASE_TERMINAL	This internal object serves as the termination NODE for a SCHEDULE phase object.
UVM_PHASE_DOMAIN	This object represents an entire graph segment that executes in parallel with the 'run' phase. Domains may define any network of NODEs and SCHEDULEs. The built-in domain, *uvm*, consists of a single schedule of all the run-time phases, starting with *pre_reset* and ending with *post_shutdown*.

~~Every phase we define has a type. It is used only for information, as the type behavior is captured in three derived classes uvm_task/topdown/bottomup_phase.~~

~~UVM_PHASE_TASK - The phase is a task-based phase, a fork is done for each participating component and so the traversal order is arbitrary~~

~~UVM_PHASE_TOPDOWN - The phase is a function phase, components are traversed from top-down, allowing them to add to the component tree as they go.~~

~~UVM_PHASE_BOTTOMUP - The phase is a function phase, components are traversed from the bottom up, allowing roll-up / consolidation functionality.~~

~~UVM_PHASE_SCHEDULE_NODE - The phase is not an imp, but a dummy phase graph node representing the beginning of a VIP schedule of phases.~~

~~UVM_PHASE_ENDSCHEDULE_NODE - The phase is not an imp, but a dummy phase graph node representing the end of a VIP schedule of phases~~

~~UVM_PHASE_DOMAIN_NODE - The phase is not an imp, but a dummy phase graph node representing an entire domain branch with schedules beneath~~