The American University in Cairo

CENTER OF NANOELECTRONICS & DEVICES

CND212: Digital Testing and Verification

Configuration Database

Component Configuration

- A configurable testbench provides a degree of freedom to the verification engineer
- UVM has an internal database table in which we can store values under a given name
- Provided information can be retrieved later in various parts of the testbench
- uvm_config_db is a mechanism for defining component configurations
  - It is structured for hierarchical configurations.
  - Focused on component instances
  - Can be used globally

Component Configuration

- Mechanism for configuring object properties

```
uvm_config_db#(type)::set(context, inst_name, field, value)
```

```
uvm_config_db#(type)::get(context, inst_name, field, var)
```

Component Configuration: set

```
uvm_config_db#(type)::set(context, inst_name, field, value)
```

```
// Set from the environment
master_agent_config agt_cfg;
uvm_config_db#(master_agent_config)::set(this, "mst_agt","cfg", agt_cfg);

// Set from the program/module. Context of "null" or "uvm_root::get()"
master_interface mst_intf;
uvm_config_db#(virtual master_interface.TB)::set(null,"uvm_test_top.env.mst_agt",
                "vif", mst_intf.TB);

// Set global for test bench from the env
uvm_config_db#(int)::set(this, "*","enable_coverage", 0);
```

Full-Hierarchy-Path to UVM Object	Field	Value
uvm_test_top.env.mst_agt	cfg	agt_cfg
uvm_test_top.env.mst_agt	vif	mst_intf.TB
uvm_test_top.*	enable_coverage	0

Component Configuration: get

```
uvm_config_db#(type)::get(context,inst_name, field, local_name)
```

```
// Set from the environment
uvm_config_db#(int)::get(this, "","count", mon_count);

// Set from the program/module. Context of "null" or "uvm_root::get()"
master_interface vif;
uvm_config_db#(virtual master_interface.TB)::get(this,"","vif", vif);
```

Configuring a Component Interface

```
class router_env extends uvm_env;
  //utils macro and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(int)::set(this, "i_agt[10]", "port_id", 10);
  endfunction
endclass
```

```
class agent extends uvm_agent;
  //constructor not shown

    int port_id = -1; //default value
    `uvm_component_utils_begin(agent)
    `uvm_field_int(port_id, UVM_ALL_ON)
    `uvm_component_utils_end

    virtual function void build_phase(..);
        uvm_config_db#(int)::get(this, "", "port_id", port_id);
        uvm_config_db#(int)::set(this, "*", "port_id", port_id);
    endfunction
endclass
```

Configuring a Component Interface

- In driver/monitor
  - Call uvm_config_db#()::get() in build_phase
  - Check for correctness in end_of_elaboration_phase

```
class driver extends uvm_driver#(packet); // other code not shown
  virtual router_io vif;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(virtual router_io)::get(this,"","vif",vif);
  endfunction

  virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    if (vif == null) begin
      `uvm_fatal("CFGERR", "Driver DUT interface not set");
    end
  endfunction
endclass
```

Configuring a Component Interface

- In agent
  - Call uvm_config_db#()::get() in build_phase to retrieve interface
  - Call uvm_config_db#()::set() in build_phase to set interface for children of agent

```
class input_agent extends uvm_agent;   // other code not shown
  virtual router_io vif;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(virtual router_io)::get(this, "", "vif", vif);
    uvm_config_db#(virtual router_io)::set(this, "*", "vif", vif);
  endfunction
endclass
```

UVM Factory

Why do we need Factory?

- UVM factory gives a mechanism to improve the flexibility, scalability, and reusability of the testbench
- Factor allows the user to substitute an existing class object with any of its inherited child class objects.
- User can use the “factory overriding” feature given by UVM to swap instances of an old class with instances of a new class.
- Each user-defined class needs to be registered in the factory.

Factories in UVM

- Factory registration
  - `uvm_object_utils (Type)
  - `uvm_component_utils (Type)
- Construct object using static proxy class method
  - ClassName obj = ClassName::type_id::create(..);
- Class overrides
  - Set_type_override_by_type(...);
  - Set_inst_override_by_type(...);

How Overriding Happens

- A factory can be thought of as a look-up table
- The component wrapper class can be accessed using type_id
  - It is used in create method
  - Returns a resultant handle
  - using the polymorphism concept
- The factory override mechanism returns a derived type handle using a base type handle.
- When the create method is called for the base class type, uvm_factory will return a pointer to an object of a derived class type.

Types of Factory Component Overrides

- Type override
  - In a type override, a substitute component class type is created instead of an original component class in the testbench hierarchy. This applies to all instances of that component type.

```
function void set_type_override_by_type (uvm_object_wrapper original_type,
                                         uvm_object_wrapper override_type,
                                         bit replace = 1)
```

def = 1

```
function void set_type_override_by_name(string original_type_name,
    string override_type_name,
    bit replace = 1)
```

Types of Factory Component Overrides

- Instance override
  - Unlike type override which overrides all instances of the type, instance override overrides only specified positions in the UVM component hierarchy.

```
function void set_inst_override_by_type (uvm_object_wrapper original_type,
                                         uvm_object_wrapper override_type,
                                         string full_inst_path)
```

```
function void set_inst_override_by_name (string original_type_name,
                                         string override_type_name,
                                         string full_inst_path)
```

Type Override: Example

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    set_type_override_by_type(component_A::get_type(),
component_B::get_type());
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction
```

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    factory.set_type_override_by_name("component_A",
"component_B");
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
endfunction
```

Type Override: Example

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);


component_A::type_id::set_type_override(component_B::get_type());
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction
```

Instance Override: Example

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    set_inst_override_by_type(component_A::get_type(),
component_B::get_type(), "*");
    comp_A = component_A::type_id::create("comp_A", this);

    factory.print();
  endfunction
```

```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    factory.set_inst_override_by_name("component_A", "component_B",
"*");
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction
```

Instance Override: Example

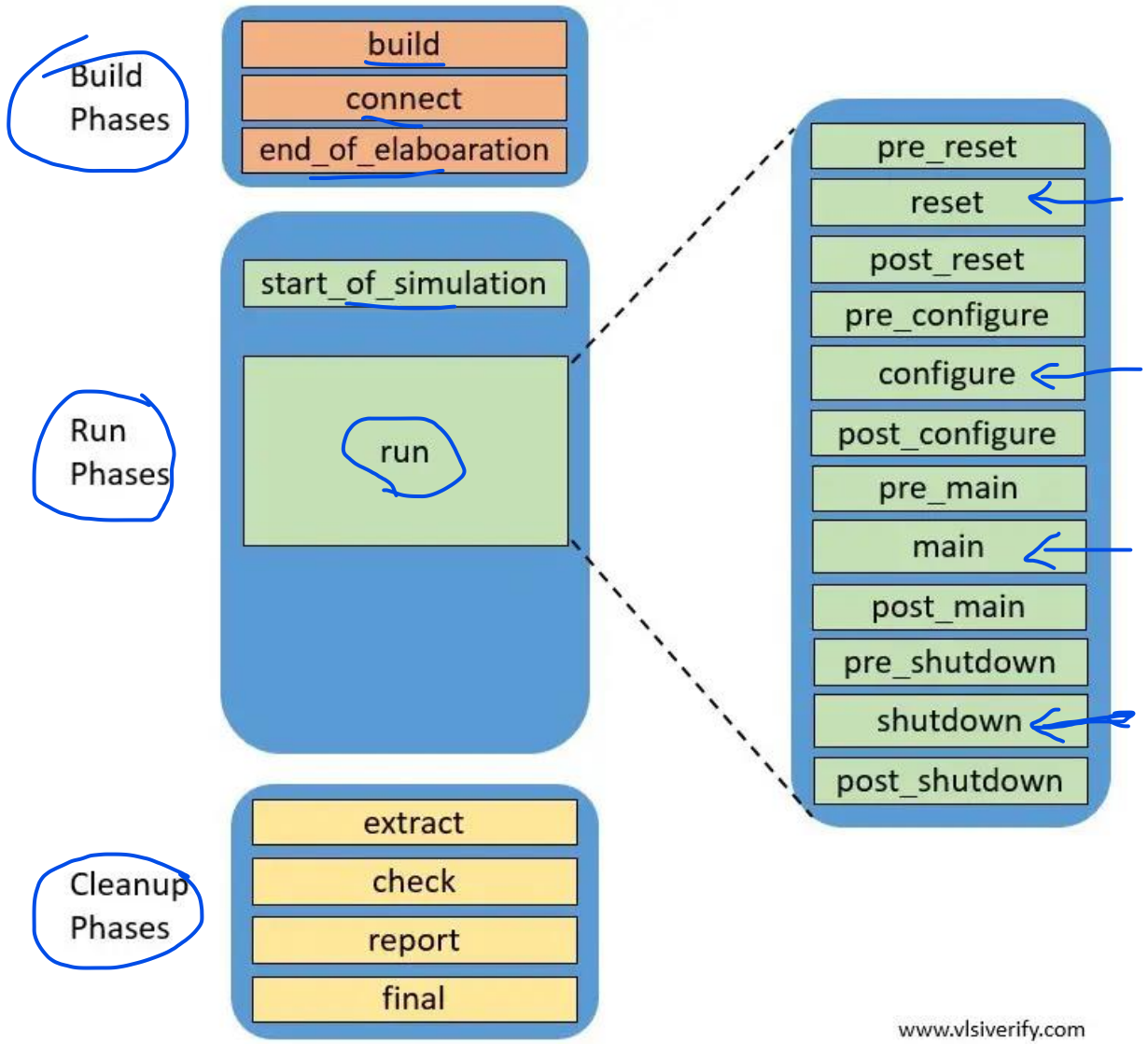
```
function void build_phase(uvm_phase phase);
    uvm_factory factory = uvm_factory::get();
    super.build_phase(phase);

    component_A::type_id::set_inst_override(component_B::get_type(), "*");
    comp_A = component_A::type_id::create("comp_A", this);
    factory.print();
  endfunction
```

UVM Component Phasing

UVM Component Phasing

- The phases are an important concept in uvm that applies to all testbench components.
- Each testbench component is derived from uvm_component and has predefined phases.
- Each component cannot move to the next phase unless the current phase execution is completed for all components. This provides proper synchronization between all components.
- UVM phases are executed in a certain order and all are virtual methods.

UVM Component Phasing



UVM Component Phasing: Build Phase

- Phases in this category are executed at the start of the UVM testbench simulation, where the testbench components are
  - constructed,
  - configured and
  - connected.
- All the build phase methods and functions are executed in zero simulation time.

UVM Component Phasing: Run-time phase

- Once testbench components are created and connected in the testbench, it goes to the run phase where actual simulation time is consumed.
- The run_phase for all components are executed in parallel.
- The run_phase and pre_reset phase both start at the same time.

UVM Component Phasing: Clean-up Phase

- The clean-up phase is used to collect information from
  - functional coverage monitors and
  - scoreboards
- It checks whether the coverage goal has been reached or the test case has passed.
- The cleanup phases will start once the run phases are completed.
- It is implemented as functions and works from the bottom to the top of the component hierarchy.
- Analysis components may use the extract, check, and report phase.

UVM Reporting

UVM Reporting

- UVM Reporting or Messaging has a rich set of message-display commands & methods to alter the numbers & types of messages that are displayed without re-compilation of the design.

- UVM Reporting has the concepts of
  - Severity,
  - Verbosity and
  - Simulation handling behavior.

- Each reporting method can be independently specified and controlled.

UVM Reporting

- Severity
  - Severity indicates importance
  - Examples are Fatal, Error, Warning & Info
- Verbosity
  - Verbosity indicates filter level
  - Examples are None, Low, Medium, High, Full & Debug
- Simulation Handling Behavior
  - Simulation handling behavior controls simulator behavior
  - Simulation Handling Behavior is the Action taken by the Simulator which is dependent from the Severity being produced by the Verification Environment.
  - Examples are Exit, Count, Display, Log, Call Hook & No Action

UVM Reporting

- UVM Reporting provides Macros to embed report messages.
- The following Macros can be used:
  - `uvm_info(string ID, string MSG, verbosity);
  - `uvm_error(string ID, string MSG);
  - `uvm_warning(string ID, string MSG);
  - `uvm_fatal(string ID, string MSG);

Severity	Simulator Action
uvm_fatal	uvm_display & uvm_exit
uvm_error	uvm_display & uvm_count
uvm_warning	uvm_display
uvm_info	uvm_display

Simulator Action	Description
uvm_exit	Exit from simulation immediately
uvm_count	Increment global error count
uvm_display	Display message on console
uvm_log	Captures messages in a named file
uvm_call_back	Calls callback method
uvm_no_action	Do nothing

UVM Objections

UVM Objections

- The uvm_objection class provides a mechanism to coordinate status information between two or more components, objects.
- The uvm_objection class is extended from uvm_report_object.
- The objection deals with the concept of raise and drop objection which means the internal counter is increment and decrement respectively.
- Each participating component and object may raise or drop objections asynchronously.
- The objection has to be raised before starting any process and drop it once it is completed.

Managing Objections

- The UVM phasing mechanism uses objections to coordinate with each other and the phase should be ended when all objections are dropped. They can be used in all UVM phases.
- It allows proceeding for the “End of test”.
- The simulation time-consuming activity happens in the run phase.
- If all objections dropped for run phases, it means simulation activity is completed.
- The test can be ended after executing clean up phases.

Managing Objections

```
task reset_phase( uvm_phase phase);
  phase.raise_objection(this);
  ...
  phase.drop_objection(this);
endtask

task run_phase(uvm_phase phase);
  phase.raise_objection(this, "Raised Objection");
  ...
  phase.drop_objection(this, "Dropped Objection");
endtask
```

Lab Task