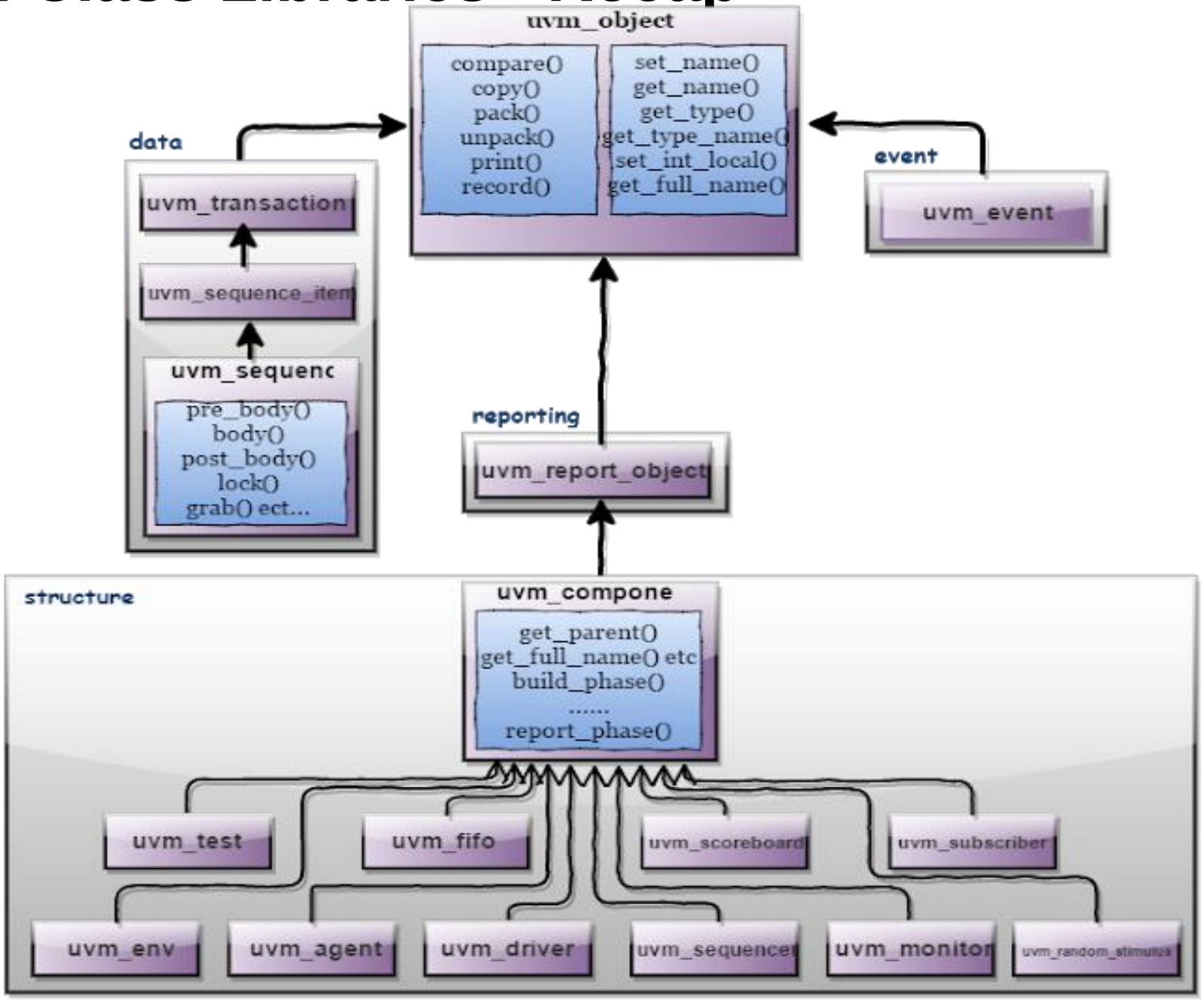
UVM component

TLM in UVM

Resource Database

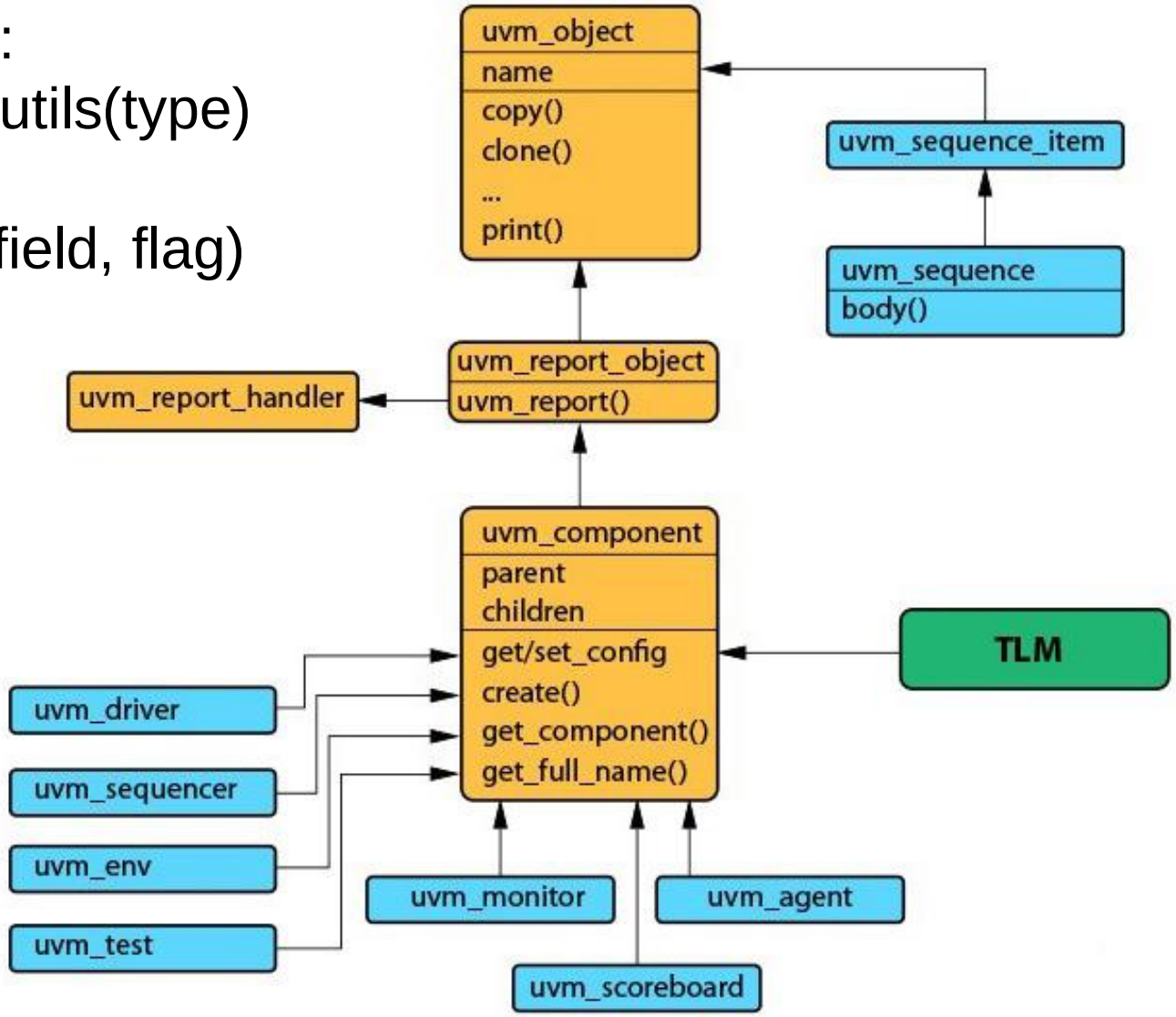
UVM Factory

UVM Class Libraries - Recap

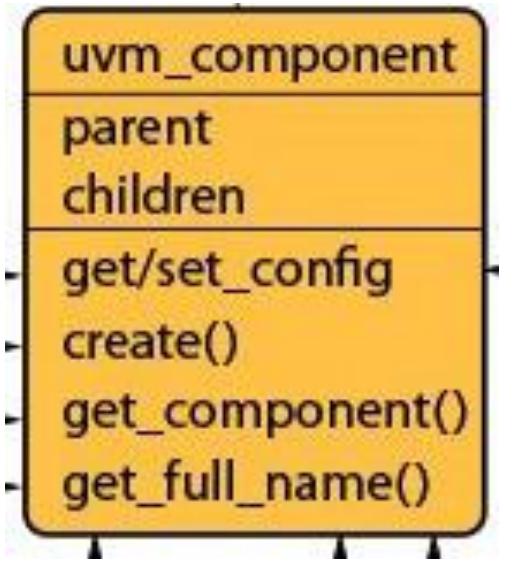


uvm_object - Recap

Utility macros:
`uvm_object_utils(type)
Field macros:
`uvm_field_*(field, flag)



UVM Component



- Components are quasi-static objects that exist throughout simulation
- Uniquely addressable with hierarchical path (env.agent.driver)
- Defines configuration, reporting, transaction recording, & factory interfaces

Universal Verification Component (UVC)

Components

Driver

- Gets transactions from sequencer:
  get_next_item(item)
- Drives transactions on the DUT interface:
  send_to_dut(item)
- Indicates to the sequencer that it is done:
  item_done()
- Connects to the DUT via a SV virtual interface

Monitor

- Collects information from the DUT
- Contains events, status, checkers & coverage
- Monitor is independent of driver

Sequencer

- Controls generation of stimulus
- Upon request from driver, generates sequences of transactions

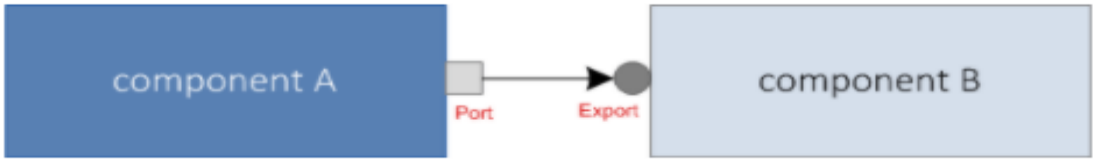
Agent

- Contains instances of sequencer, driver & monitor
- Configurable: is_active flag indicates whether agent is active or passive
- Is_active == 1 – all agent components constructed
- Is_active == 0 – only monitor

Transaction Level Modeling (TLM)

- A modeling style to build highly abstract models of components and systems.
- Data is represented as transactions that flow in and out of different components via TLM interfaces.

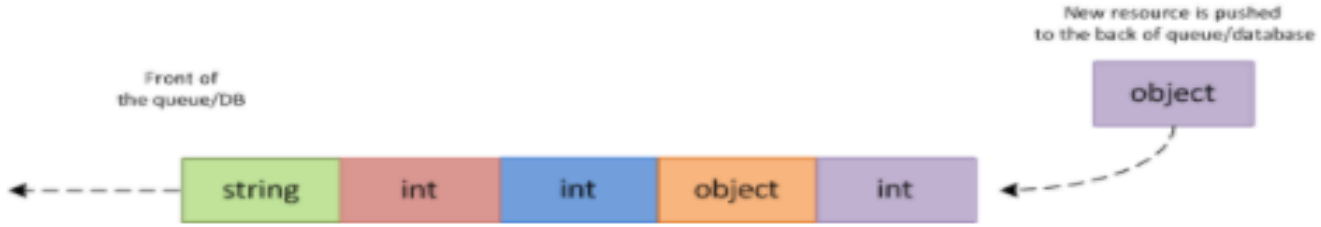
```
class simple_packet extends uvm_object;
    `uvm_object_utils (simple_packet)

    rand bit [7:0] addr;
    rand bit [7:0] data;
         bit       rwb;

    constraint c_addr { addr > 8'h2a; };
    constraint c_data { data inside {[8'h14:8'he9]};

endclass
```



Resource Database

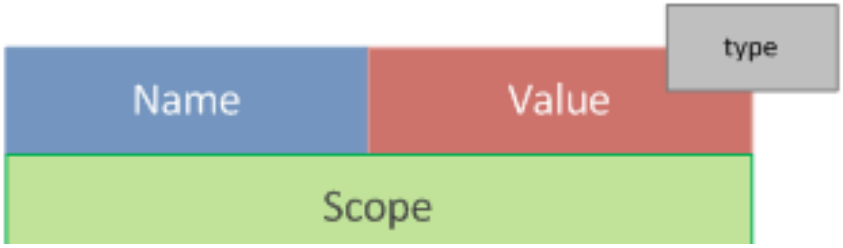
- A resource is a parameterized container that holds arbitrary data.
- Resources can be used to configure components, supply data to sequences, or enable sharing of information across disparate parts of the testbench.
- One can put any data type into the resource database, and have another component retrieve it later at some point in simulation, which is a very convenient feature to have.
- The global resource DB has both a name table and a type table into which each resource is entered. So, the same resource can be retrieved later by name or type.
- Multiple resources with the same name/type are stored in a queue and hence those which were pushed in earlier have more precedence over those placed later.

Resource Database



- Example:
- From the queue shown above, if a request to retrieve an item of type string is made, the queue is traversed from front to back, and the first occurrence of an object of string type will be returned.
- Now, consider the case where the items 2(red) and 3(blue) in the queue have the same scope, and a get_by_type() method is called for that particular scope.
- Then, item 2 will be returned since that sits earlier in the queue.
- Resources are added into the pool by calling set, and they are retrieved from the pool by get_by_name() or get_by_type()

Resource Database



Name	The "name" by which this resource is stored in the database. The same "name" has to be supplied to retrieve it later.
Value	The value that should be stored in the database for the given "name".
Scope	A regular expression that specifies the scope over which this resource is visible to other components in the testbench.
Type	The data type of the object that this resource contains. It can be a string, int, virtual interface, class object, or any other valid SystemVerilog data-type.

Configuration Database (UVM Config db)

- Provides access to centralized db, where type specific information can be stored and retrieved
- Can contain scalar objects, class handles, queues, lists, or even virtual interfaces
- All functions are static and must be called using :: scope operator

  uvm_config_db::set – store information in db

Configuration Database (UVM Config db)

uvm_config_db

All of the functions in uvm_config_db#(T) are static, so they must be called using the :: operator.

Class Hierarchy

uvm_resource_db#(T)

uvm_config_db

Class Declaration

```
class uvm_config_db#(
   type T = int
) extends uvm_resource_db#(T)
```

Methods

get	Get the value for *field_name* in *inst_name*, using component *cntxt* as the starting search point.
set	Create a new or update an existing configuration setting for *field_name* in *inst_name* from *cntxt*.
exists	Check if a value for *field_name* is available in *inst_name*, using component *cntxt* as the starting search point.
wait_modified	Wait for a configuration setting to be set for *field_name* in *cntxt* and *inst_name*.

Configuration Database (UVM Config db)

```
static function void set (  uvm_component cntxt,
                            string        inst_name,
                            string        field_name,
                            T             value);
```

- The static function of the class uvm_config_db is used to set a variable in the configuration database.
- cntxt is the hierarchical starting point of where the db entry is accessible.
- inst_name is a hierarchical path that limits the accessibility of the database entry.
- field_name is the label used as a lookup for the database entry
- T is the type of element being configured and can be scalar objects, class handles, queues, lists or even virtual interfaces
- Value is the value to be stored in the database.

Configuration Database (UVM Config db)

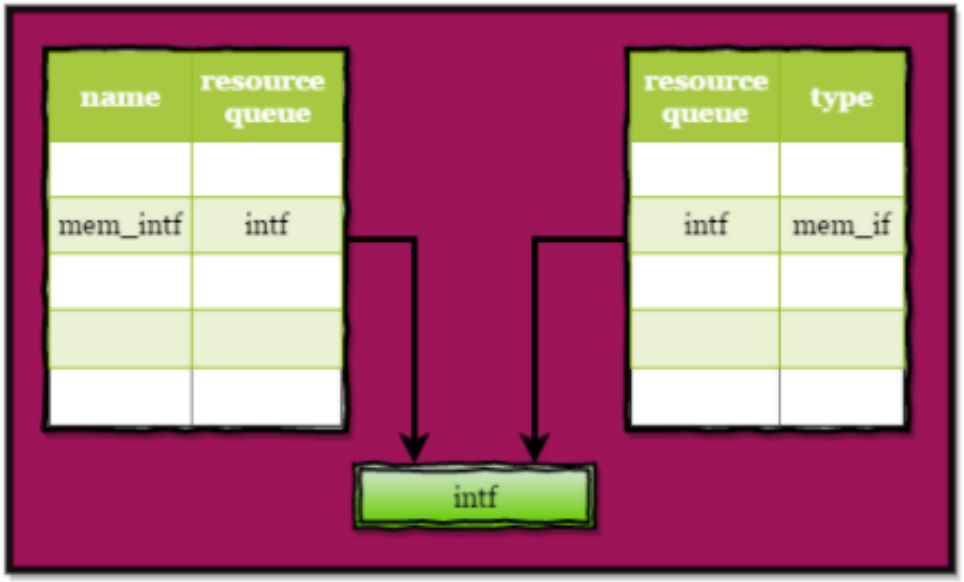
```
virtual function void build_phase (uvm_phase phase);
    ...
    uvm_config_db #(int) :: set (null, "uvm_test_top.m_env.m_apb_agent', "cov_enable", 1);
    ...
endfunction
```

- set() function will set a variable of name cov_enable at the path uvm_test_top.m_env.m_apb_agent with value 1.
- Use of set() creates a new or updates an existing configuration setting for field_name in inst_name from cntxt.
- This setting is made at cntxt with the full scope of the set being (cntxt, ".", inst_name).
- If cntxt is null, then the complete scope of getting the information will be provided by inst_name.

Configuration Database (UVM Config db)

- Example to show the setting of the interface handle intf, type mem_if, label mem_intf with global scope.

```
mem_if intf(clk,reset); //interface instance
uvm_config_db#(virtual mem_if)::set(null,"*","mem_intf",intf); //set method
```



The diagram shows how a resource whose name is mem_intf and type is mem_if is stored in the pool.

UVM Factory

- A mechanism allows the user to substitute an existing class object by any of its inherited class objects
  - To improve flexibility and scalability of the testbench
  - Ex: extend a driver to have few more test scenarios
  - Only classes extended from uvm_object and uvm_component are supported
- Basic steps
  - Registration
  - Construction
  - Overriding

UVM Factory

- Registration - Example

```
class packet extends uvm_object;
`uvm_object_utils(packet)
endclass


class driver extends uvm_component;
`uvm_component_utils(driver)
endclass
```

```
class monitor #(type T=int, int mode=0) extends uvm_component;
`uvm_component_param_utils(driver#(T,mode))
endclass
```

```
class packet #(type T=int, int mode=0) extends uvm_object;
`uvm_object_param_utils(packet #(T,mode))
endclass
```

UVM Factory

- Construction
- static method create() should be used. while constructing the uvm based components or uvm based objects , do not use new() constructor
- Syntax

```
static function T create(string name,
                         uvm_component parent,
                         string context = " ");
```

- Ex:

```
class_type object_name;

object_name = clss_type::type_id::creat("object_name",this);
```

UVM Factory

- Overriding
- Based on the needs, user could override the registered classes or objects