# UVM library basics

- Reference

Universal Verification Methodology (UVM) 1.2 Class Reference by Accellera Systems Initiative (Accellera)

# UVM library basics

- UVM library, Library Base Classes, the uvm_object class
- uvm_component class, UVM configuration mechanism, TLM in UVM
- UVM factory, UVM message facilities, callbacks

# UVM library

- The UVM Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments in SystemVerilog.
  - Provides detailed reference information for each user-visible class in the UVM library
- Divided as UVM classes and utilities into categories pertaining to their role or function

- Globals
- Base
- Reporting
- Factory
- Phasing
- Configuration and Resources
- Synchronization
- Containers
- Policies
- TLM
- Components
- Sequencers
- Sequences
- Macros
- Register Layer
- Command Line Processor

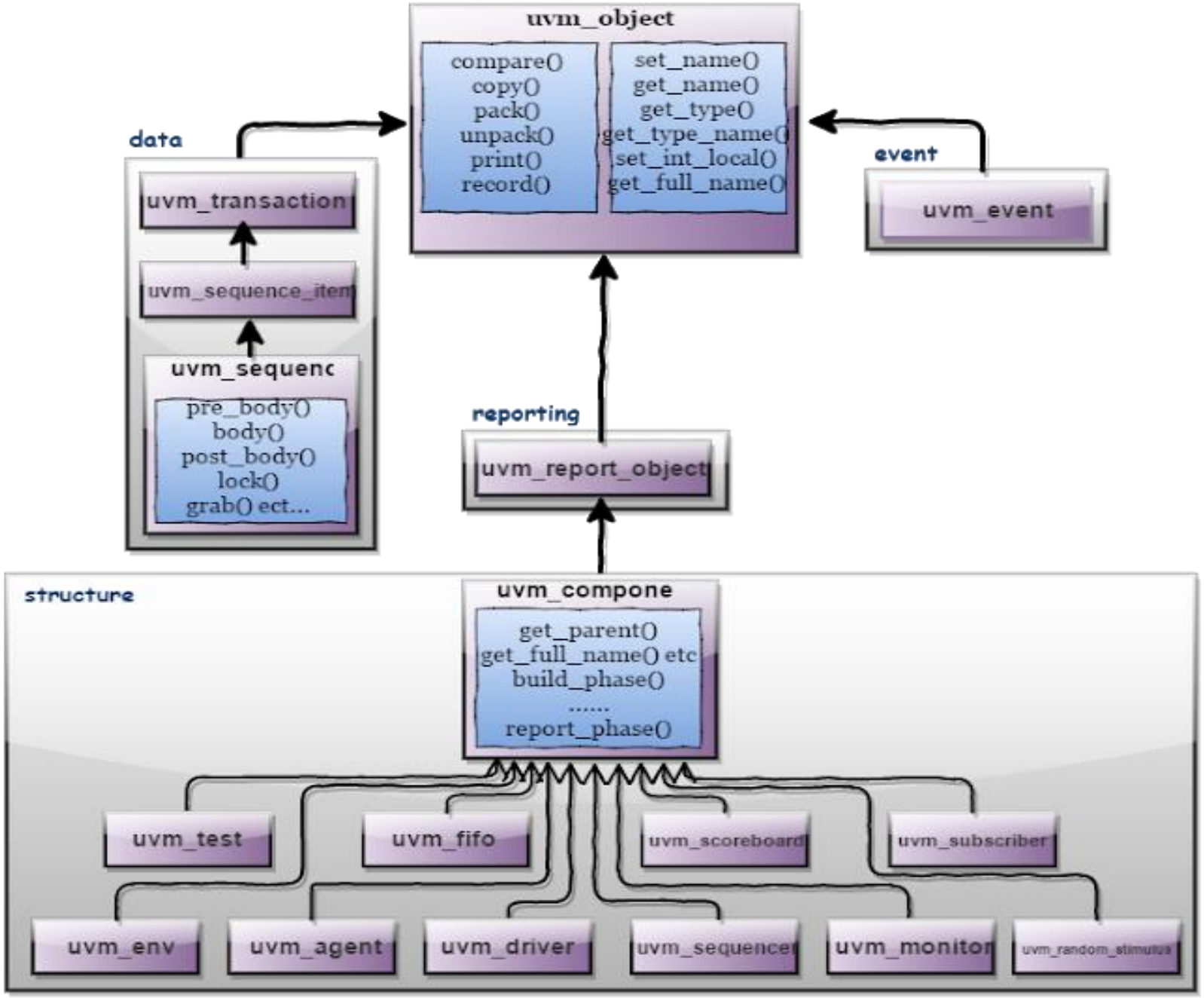
# UVM Class Libraries

- UVM has class libraries (a set of base classes with methods defined in it)
- Needed for development of well-constructed, reusable SV verification environment (done by extending the base classes).
- Three main types of UVM classes:
  - **uvm_object**
  - **uvm_transaction**
  - **uvm_component**

# UVM Class Libraries

- **uvm_object** - All components and transactions derive from uvm_object, which defines an interface of core class-based operations: create, copy, compare, print, sprint, record, etc.
- It also defines interfaces for instance identification (name, type name, unique id, etc.) and random seeding.
- **uvm_transaction** - The uvm_transaction is the root base class for UVM transactions, which, unlike uvm_components, are transient in nature.
- It extends uvm_object to include a timing and recording interface.
- Simple transactions can derive directly from uvm_transaction, while sequence-enabled transactions derive from uvm_sequence_item
- **uvm_component** - The uvm_component class is the root base class for all UVM components.
- Components are quasi-static objects that exist throughout simulation.
- This allows them to establish structural hierarchy much like modules and program blocks.

# UVM Class Libraries



# UVM Class Libraries

- Base class hierarchy: uvm_void → uvm_root → uvm_object → uvm_report_object → uvm_component → uvm_transaction
- uvm_void – serves as base class for all UVM classes
- uvm_root – top level class, implicit, created automatically during simulation, accessed via uvm_package using the varialble uvm_top
- uvm_object – provides operational methods such as create, copy, clone, compare, print, record, etc.
- uvm_report_object – reporting facility such as messages, warnings, errors
- uvm_component – phasing, reporting, recording, factory (to create new components)
- uvm_transaction – simple transaction - timing and recording interface
  - Sequence enabled to be derived from uvm_sequence_item

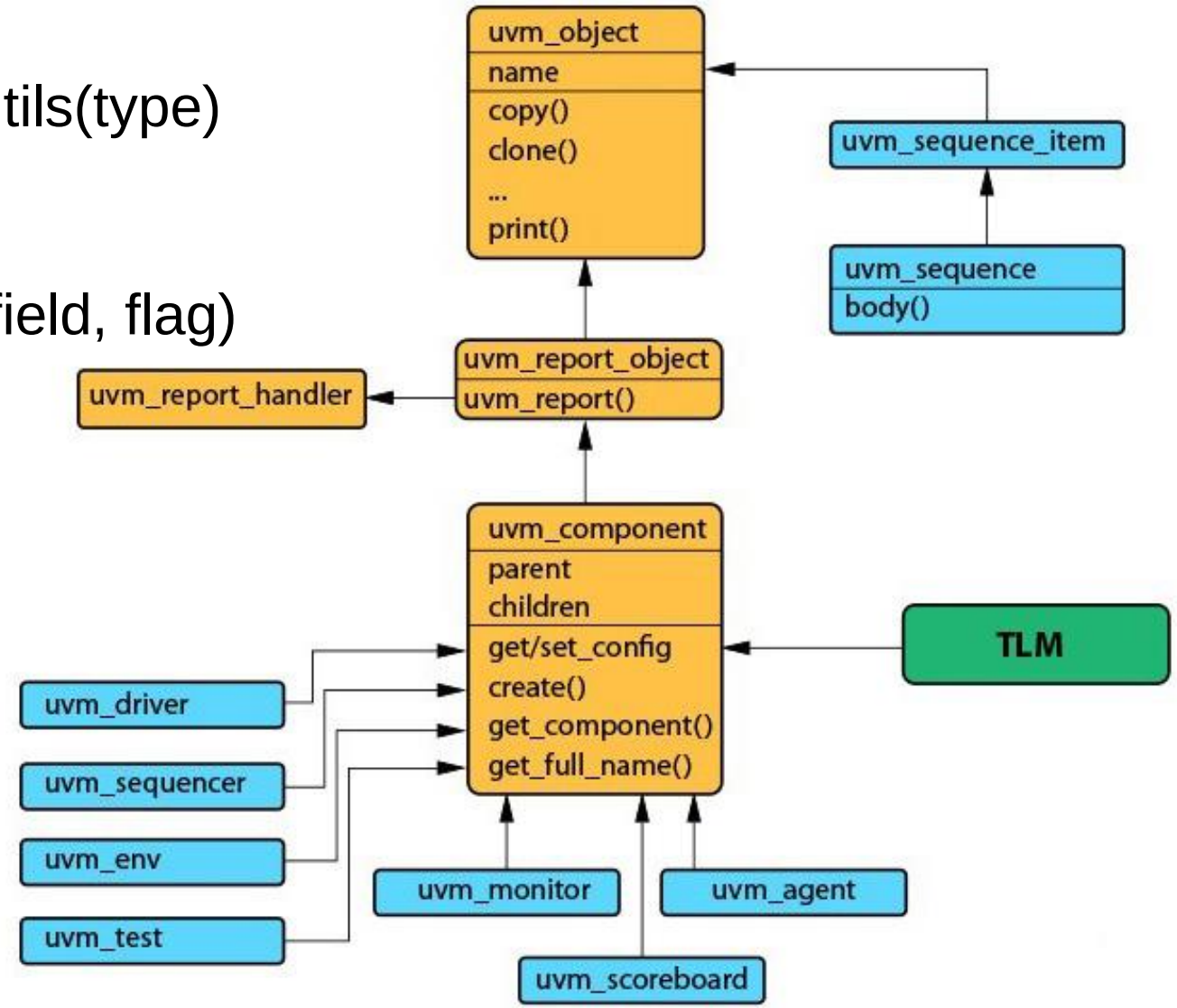
# uvm_object

- Utility macros:

  `uvm_object_utils(type)

- Field macros:

  `uvm_field_*(field, flag)



# uvm_object

- This is the main class which has operational methods (create, copy, clone, compare, print, record, etc..), instance identification fields (name, type name, unique id, etc.) and random seeding are defined in it.
  - uvm_transaction and uvm_component are derived classes
- Classes derived from uvm_object must implement the pure virtual methods such as create and get_type_name.
- Example: uvm_object is used to write a uvm_sequence item. The virtual methods in uvm_object can make sequence_item purposeful.
- uvm_object has utility macros & field macros.
  - Provide implementations of the basic methods such as create.
  - When you are trying to write objects without any field macros, its written as `uvm_object_utils(type).
- UVM field macros are useful typically for the implementation of methods such as copy, print, pack, unpack, etc.
  - They are invoked inside the `uvm_object_utils_begin and `uvm_object_utils_end.
  - The macro basically has two arguments as part of it: field and flag.

# UVM utility and field macros

- Utility macros
  - The utils macro is used primarily to register an object with the factory
  - Required to be inside every user defined class derived from uvm_object
- Object Utility
  - All classes derived directly from **uvm_object** or **uvm_transaction** require them to be registered using **`uvm_object_utils** macro.

```
import uvm_pkg::*;

class my_object extends uvm_object;
  `uvm_object_utils(my_object)

  function new(string name = "my_object");
    super.new(name);
  endfunction
endclass
```

# UVM component utility

- All classes derived directly or indirectly from **uvm_component** require them to be registered with the factory using **`uvm_component_utils** macro.
  - It is mandatory for the new function to be explicitly defined for every class defined directly or indirectly from uvm_component.
  - The new function takes the name of the class instance and a handle to the parent class where the object is instantiated.

```
import uvm_pkg::*;

class my_component extends uvm_component;
  `uvm_component_utils(my_component)

  //function new(string name = "my_object", uvm_component parent = null);
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass
```

# Creation of class object

- Class objects are created by calling the *type_id::create()* method
- This makes any child class object to be created & returned using factory mechanism

# Creation of class object

```
import uvm_pkg::*;

class my_object extends uvm_object;
  `uvm_object_utils(my_object)

  function new(string name = "my_object");
    super.new(name);
  endfunction
endclass
```

```
import uvm_pkg::*;

class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  my_test t;
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
    t = my_test::type_id::create("t", this);
  endfunction
endclass
```

# Field Macros

- `uvm_field_* used between *_begin and *_end utility macros are field macros
- They operate on class properties
- Provide automatic implementations of core methods like copy, compare and print.

```
class my_object extends uvm_object;
  rand bit [4:0] address;
  rand bit [2:0] data;
  string m_name = "MSIS";

  `uvm_object_utils_begin(my_object)

  `uvm_field_int(address, UVM_DEFAULT)
  `uvm_field_int(data, UVM_DEFAULT)
  `uvm_field_string(m_name, UVM_DEFAULT)

  `uvm_object_utils_end

  function new(string name = "my_object");
    super.new(name);
  endfunction
endclass
```

# Field Macros

- `uvm_field corresponding to the data type of each variable should be used :
  - int, bit, byte should use `uvm_field_int
  - String should use `uvm_field_string

# Field Macros

- Field macros accept at least two arguments: ARG, FLAG

| Argument | Description |
|---|---|
| *ARG* | Name of the variable, whose *type* should be appropriate for the macro that is used |
| *FLAG* | When set to something other than *UVM_DEFAULT* or *UVM_ALL_ON*, it specifies which data method implementations will not be included. For example, if FLAG is set to *NO_COPY*, everything else will be implemented for the variable except copy. |

# UVM Class Libraries

| FLAG | Description |
|---|---|
| UVM_ALL_ON | Set all operations on (default) |
| UVM_DEFAULT | Use the default flag settings |
| UVM_NOCOPY | Do not copy this field |
| UVM_NOCOMPARE | Do not compare this field |
| UVM_NOPRINT | Do not print this field |
| UVM_NOPACK | Do not pack or unpack this field |
| UVM_PHYSICAL | Treat as a physical field. Use physical setting in policy class for this field |
| UVM_ABSTRACT | Treat as an abstract field. Use the abstract setting in the policy class for this field |
| UVM_READONLY | Do not allow the setting of this field from the set_*_local methods |

# UVM Class Libraries

| A radix for printing and recording can be specified by OR'ing one of the following constants in the FLAG argument | |
|---|---|
| UVM_BIN | Print/record the field in binary (base-2) |
| UVM_DEC | Print/record the field in decimal (base-10) |
| UVM_UNSIGNED | Print/record the field in unsigned decimal (base-10) |
| UVM_OCT | Print/record the field in octal (base-8). |
| UVM_HEX | Print/record the field in hexadecimal (base-16) |
| UVM_STRING | Print/record the field in string format |
| UVM_TIME | Print/record the field in time format |

# UVM Object Print

```
import uvm_pkg::*;
typedef enum {red, blue} colors;

class my_object extends uvm_object;
  rand colors mycolor;
  rand bit [2:0] data;
  string my_class;
  constraint cons {data[2] ==1;}

  function new(string name = "my_object");
    super.new(name);
    my_class = name;
  endfunction

  `uvm_object_utils_begin(my_object)

  `uvm_field_enum(colors, mycolor, UVM_DEFAULT)
  `uvm_field_int(data, UVM_DEFAULT)
  `uvm_field_string(my_class, UVM_DEFAULT)

  `uvm_object_utils_end

endclass
```

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

  function void build_phase(uvm_phase phase);
    my_object t = my_object::type_id::create("t",this);
    t.randomize();
    t.print;
  endfunction:build_phase
endclass
```

```
module tb;
  initial begin
    run_test("my_test");
  end
endmodule
```

# UVM Reporting Functions

- Most of the verification components are inherited from uvm_report_object (they already have functions and methods to display messages)
- 4 basic reporting functions : info, error, warning, fatal, and these are used with 6 levels of verbosity.

```
typedef enum {
   UVM_NONE   = 0,
   UVM_LOW    = 100,
   UVM_MEDIUM = 200,
   UVM_HIGH   = 300,
   UVM_FULL   = 400,
   UVM_DEBUG  = 500
} uvm_verbosity;
```

- The verbosity level is required only for uvm_report_info.
- Use of uvm_report_fatal will exit the simulation

# UVM Reporting Functions

```
uvm_report_warning (get_type_name (), $sformatf ("Warning level message"));
uvm_report_error (get_type_name (), $sformatf ("Error level message"));
uvm_report_fatal (get_type_name (), $sformatf ("Fatal level message"));
```

- Verbosity Levels:
  - Controls whether a uvm_report_* statement gets displayed or not
  - If verbosity settings has been configured to UVM_HIGH, every uvm_report_* or `uvm_* message with verbosity level less than UVM_HIGH will be printed.
  - If configured to UVM_LOW, then only UVM_LOW and UVM_NONE lines will be dumped out.
  - Default configuration is UVM_MEDIUM