UVM Introduction

SyoSil, 2025

Contents

UVM Introduction

UVM History – SystemVerilog Branch

- 2009: Accellera sets up a committee to develop a single standard verification methodology (UVM)
- Dec 2009: Committee adopts OVM v2.1 (over VMM) as the baseline for UVM
  - Mar 2010: Committee adopts OVM v2.1.1 as the baseline
- Apr 2010: UVM Early Adopter Kit (UVM EA 1.0) release
  - OVM 2.1.1 with "o" changed to "u" everywhere
- DVCon 2011: uvm-1.0.p1 release
  - Initial public release
- DAC 2011: uvm-1.1 release
  - Bug fixes
  - Update the User's Guide and Reference Manual
  - Determined there is problems with Phasing as adopted
- 2011 – 2015: various bug fixes and minor update releases (1.1a – 1.2)
- 2017: IEEE release 1800.2-2017
- 2020: IEEE release 1800.2-2020

1.0-beta3
• on Jul 8, 2020

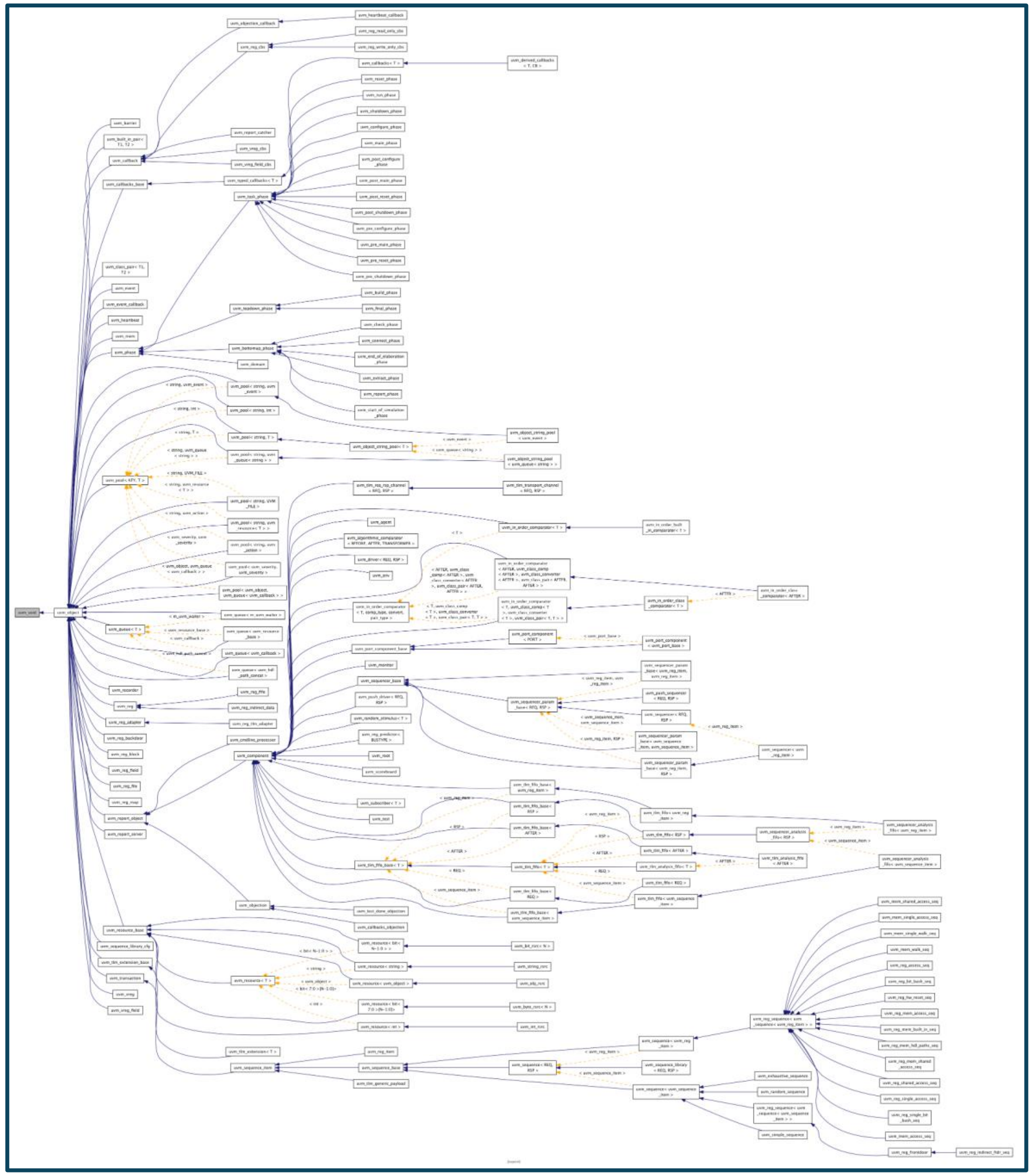
UVM History – PyUVM/SystemC Branch

- PyUVM Releases
  - 3.0.0 - Jun 29, 2024
  - 2.9.0 - Oct 15, 2022
  - ...
  - 2.0 - Sep 5, 2021
  - ...
  - 1.0 - Feb 21, 2021

- UVM-SC Releases
- 1.0-beta6 - Jul 1, 2024
- 1.0-beta5 - Mar 15, 2023
- 1.0-beta4 - Apr 12, 2022
- 1.0-beta3 - Jul 8, 2020
- 1.0-beta2 - Nov 15, 2018
- 1.0-beta1 - Dec 7, 2017
- 1.0-alpha1 - Dec 4, 2015

What is UVM?

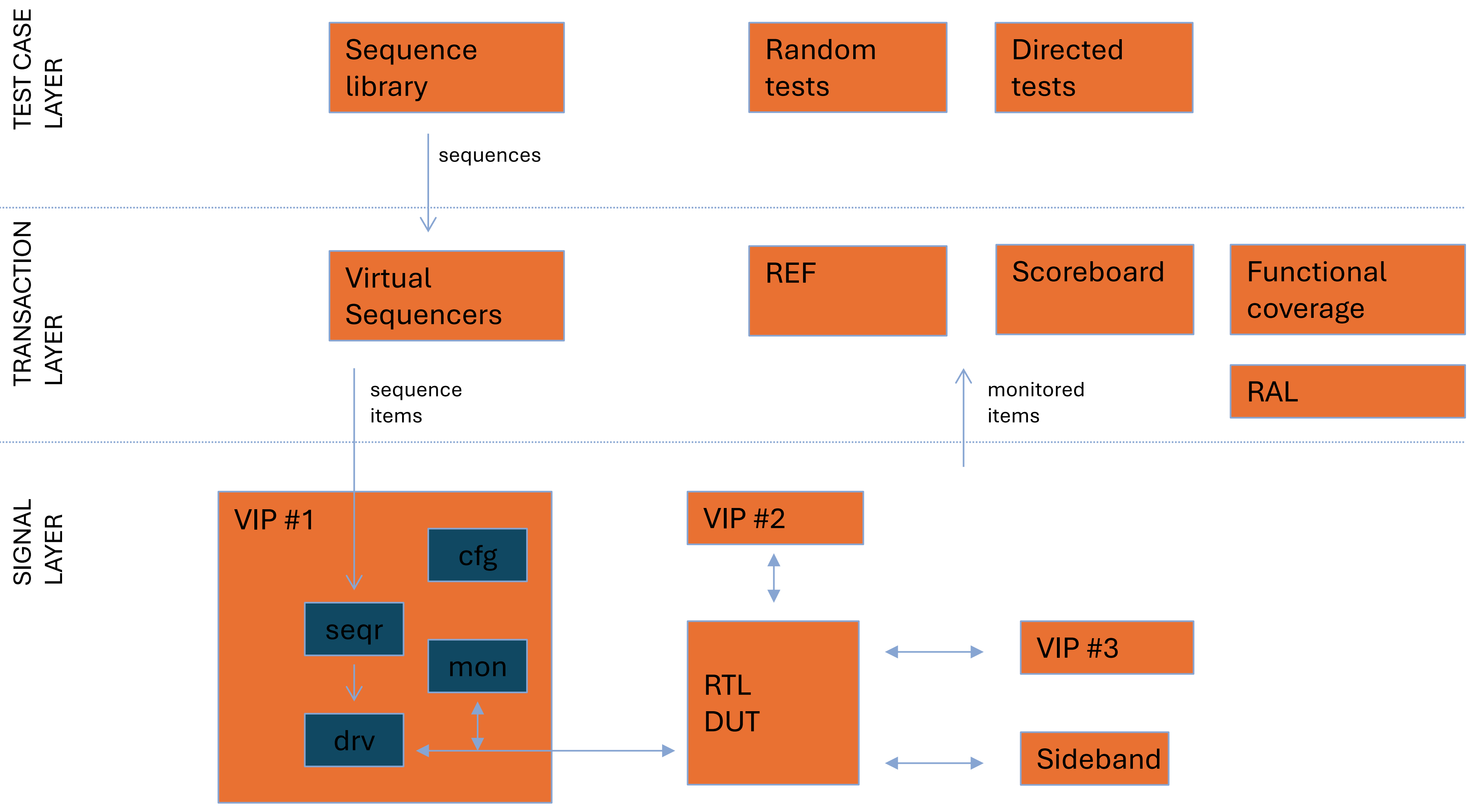
- A concrete and usable implementation of a CRV framework
- A base class library - **BCL** implemented in SystemVerilog
  - OOP with bells and whistles
  - OOP Design Patterns (Read the book ☺)
- Source code available – **Open Source**
- Provides base classes for the core objects in a **CRV** testbench
  - Structure
  - Simulation control
  - Ensure reusability



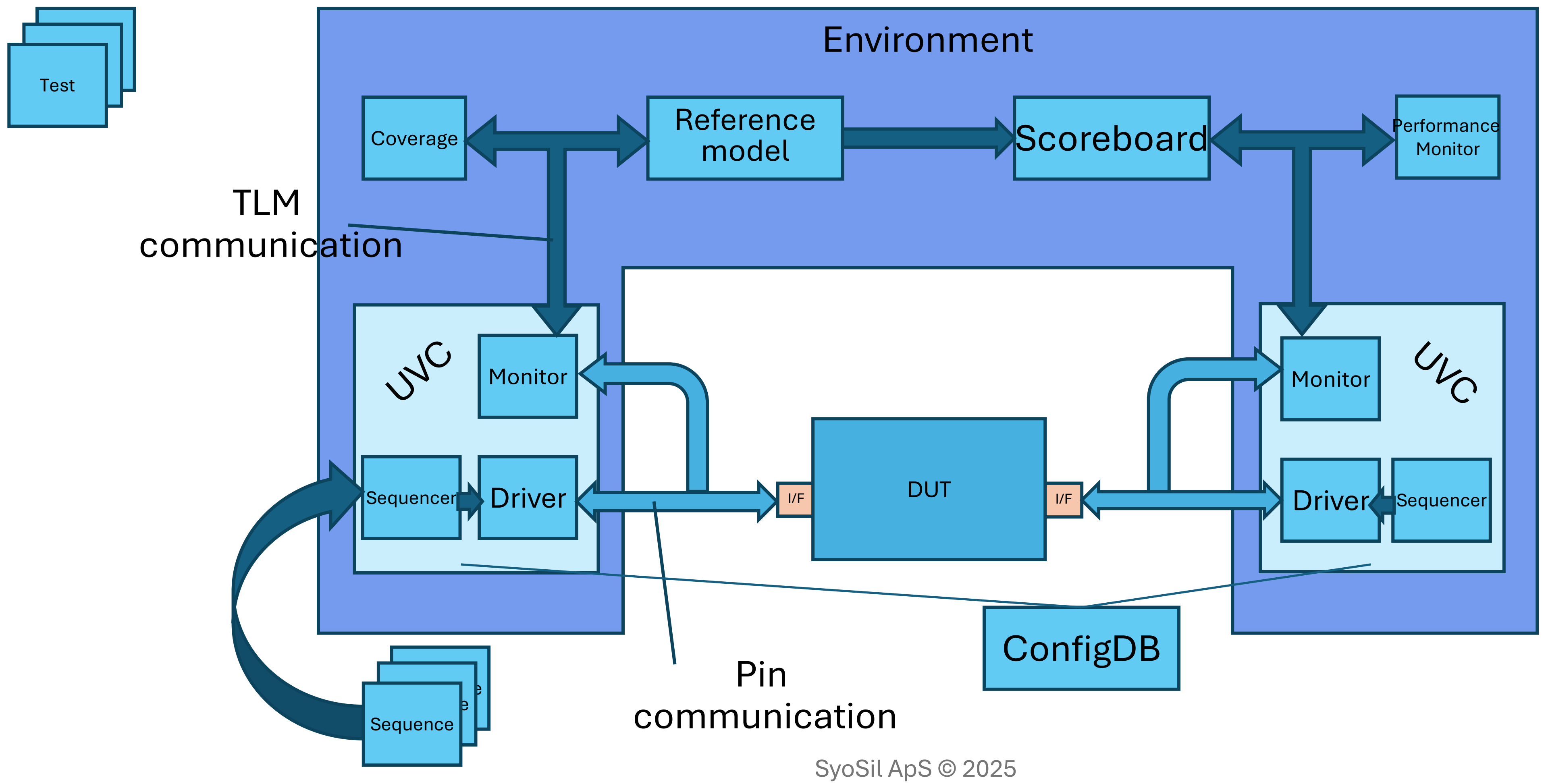
What is PyUVM?

- **PyUVM** is a clean implementation of IEEE 1800.2 in Python
  - PyUVM uses **CocoTB** to interact with simulators and schedule simulation events.
- It is a port (rewriting) and **NOT** an machine translation
- Link to GitHub: https://github.com/pyuvm/pyuvm

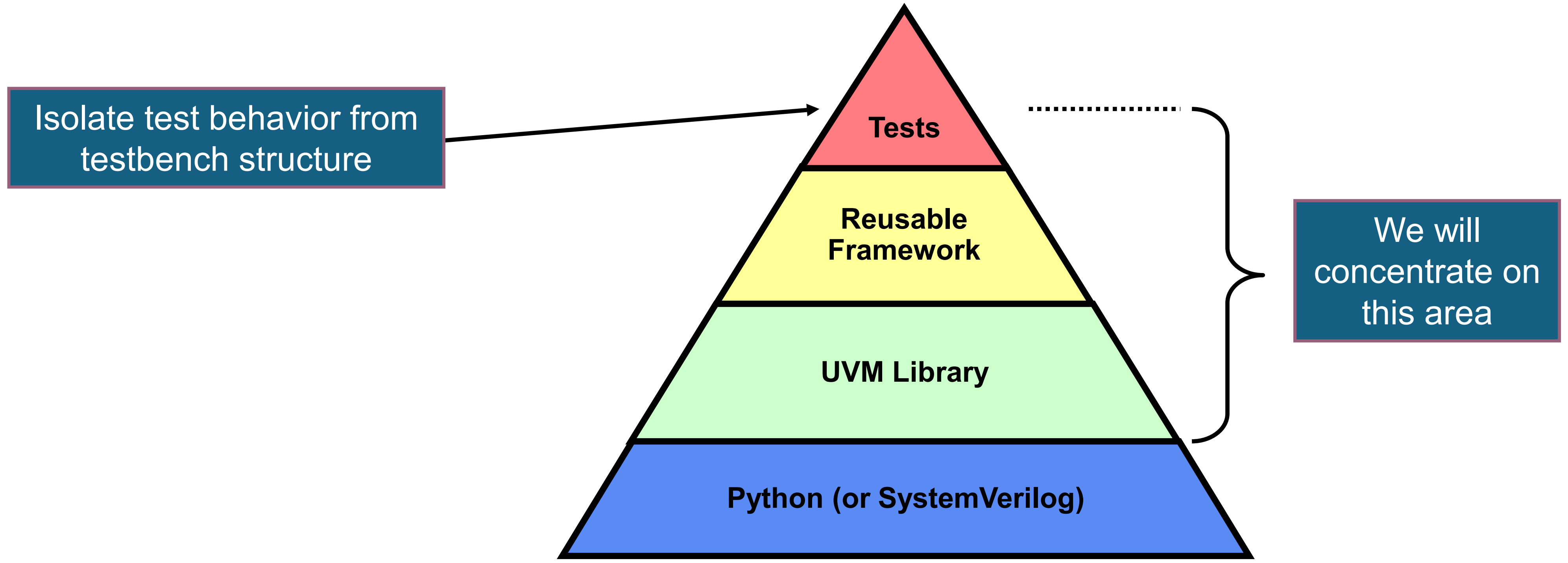
General CRV Test Bench Structure



UVM-SV/UVM-SC/**PyUVM** Structural Overview



Building on a Reusable Foundation


Isolate test behavior from testbench structure
Tests
Reusable Framework
UVM Library
Python (or SystemVerilog)
We will concentrate on this area

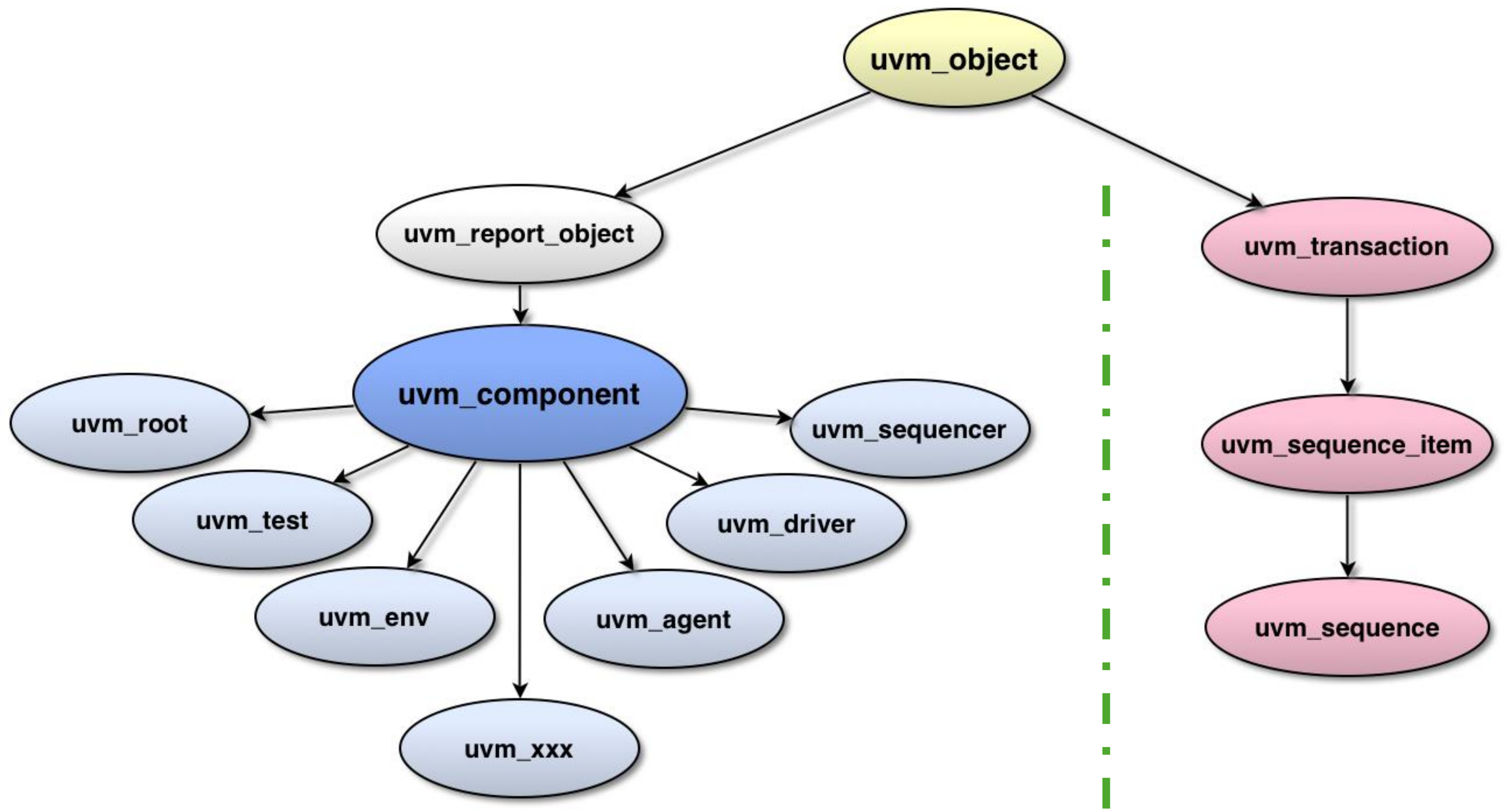
PyUVM Library Module

- PyUVM library is defined within a module called `pyuvm`
  - Class declarations
  - Static singletons (Singleton Pattern - en.wikipedia.org/wiki/Singleton_pattern)
  - Convenience methods
- Testbench components typically
  - Inherit from base classes in `pyuvm`
  - Instantiate classes from `pyuvm`
  - Access static singletons in `pyuvm`
  - Call convenience methods in `pyuvm`

■ Sometimes these singletons and convenience methods are referred to as "global objects" or "global methods"
  - Since `pyuvm` is imported everywhere it appears as if they are "global" while technically their scope is the `pyuvm`

UVM Fundamentals
Base Objects

Base Classes

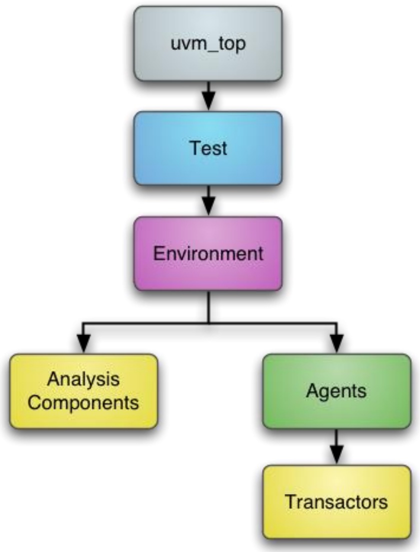


Structural component base classes

Transaction item and stimulus generation base classes

UVM Top - (`uvm_top`)

- **`uvm_top`** is a singleton of **`uvm_root`**
  - It is created and managed by UVM
  - The top-level component of all test components
    - All testbench components are created with a parent argument
    - If a component is created with the parent argument as "null," then it becomes a child of **`uvm_top`**
      - This is typically not a good thing and will cause problems if done accidently
      - Else becomes a child of the specified parent



- **`uvm_top`** has a number of methods used to control simulation
  - These methods and their usage will be explained as they are used or needed

UVM component - (`uvm_component`)

- UVM base library class
- Used for static testbench structure
- May only be created at the start of simulation
- Once created, present throughout simulation
- Are phased – We will get to that later
- `uvm_env`, `uvm_driver`, `uvm_agent`, `uvm_monitor`

UVM object – (`uvm_object`)

- UVM base library class
- Used for dynamic objects that are created and destroyed throughout the test, e.g.
  - Stimuli (randomized transactions)
  - Observed traffic (recorded transactions)
  - Scoreboarded & reference model transactions
- Provides print, copy, compare, pack, unpack and record methods
- **NOT** phased
- uvm_sequence_item, uvm_sequence

Tests (`uvm_test`)

- Root of the UVM test bench hierarchy
- Instantiates
  - Environment configuration
  - Environment
  - Connects top level module intefaces with environment
- Enables per test ENV configuration
- Typically
  - Starts clocks
  - Releases reset
  - Starts a top level sequence

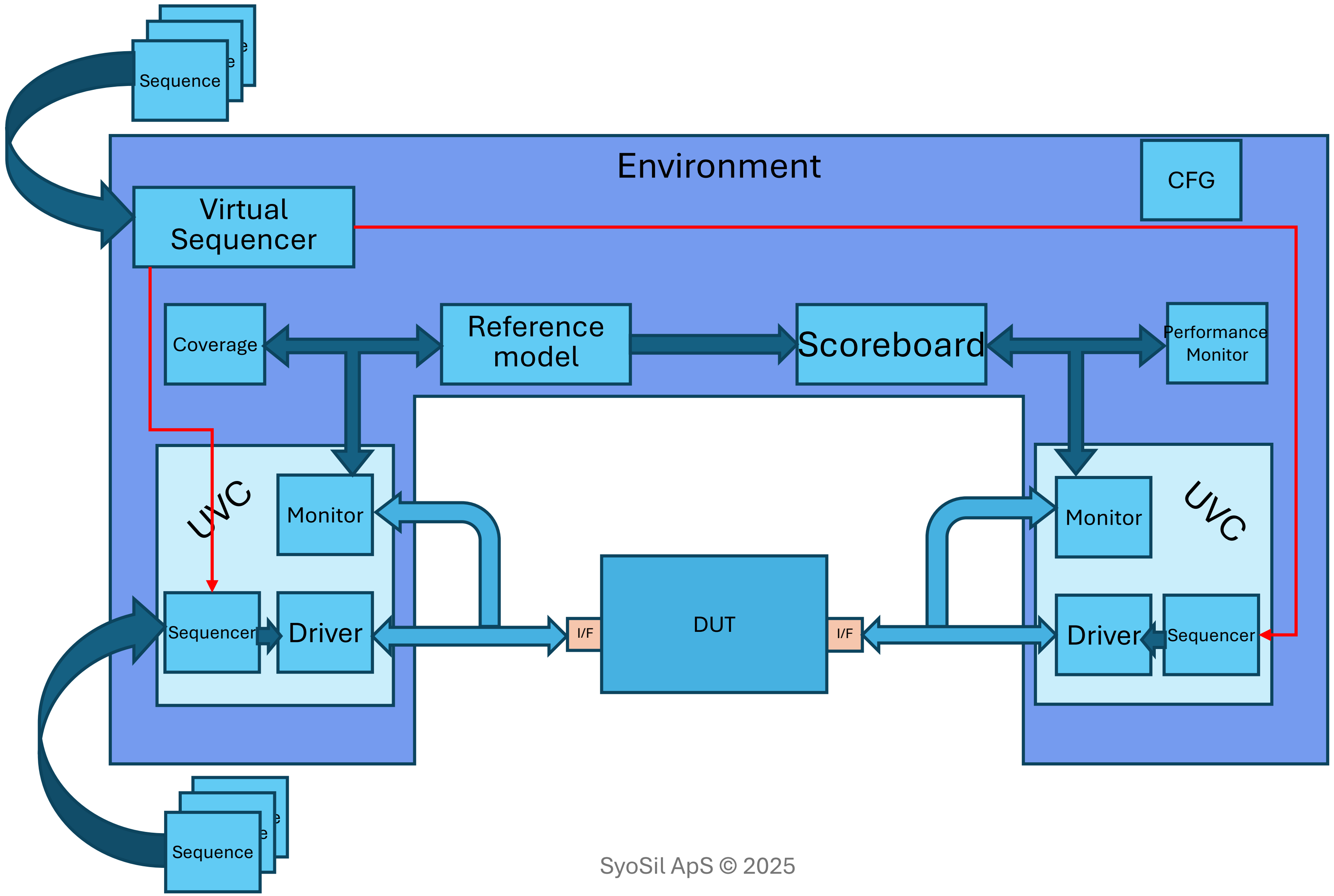
Environment (`uvm_env`)

- UVM Component
- Instantiates UVCs, topology similar to DUT
- Container for cross-UVC testbench functionality
  - Virtual sequencer
  - Reference model
  - Coverage
- Hierarchical -> may contain other environments
- Primary unit of re-use from block to system testbenches

Environment Sequencer (`uvm_sequencer`)

- UVM Component
- Called ‘virtual sequencer’ in UVM terminology
- Coordinates sequences that access multiple UVCs
  - In order to promote (vertical) reuse no sequences should refer directly to UVC sequencers. Instead use the virtual sequencer.
- Environment sequences are DUT specific but can often be re-used at system level

UVM Environment with Virtual Sequencer



UVM Fundamentals
Phasing

Fundamentals: UVM Phasing

- All testbenches have a natural chain of events
- Formalized in UVM as ‘phases’
- All phases have a `<phasename>_phase` method
- You can make your own phases and phase schedules – not for the faint of heart

Phase Management

- Fortunately, we don't have to define and create our own phases!
  - UVM pre-defines and declares a set of phases

- The general form of the phase method is

  `<phasename>_phase(self)`

    - Phase methods are inherited from `uvm_component`
      - Do nothing by default
      - Components "participate" in a phase by overriding the inherited phase method
- The component phases are synchronized by UVM
  - All components finish one phase before the next phase is started
- During a phase, UVM will progress through all components in an orderly fashion
  - Each component will get a chance to execute its behavior for the phase
  - Depending on the phase, the order can be top-down, bottom-up, or concurrent (i.e., no order)

Implicit Phasing Methods

- Override these inherited methods in your components

`def build_phase()`
- For constructing child components and other structural things

`def connect_phase()`
- For connecting up the ports of components, etc.

`def end_of_elaboration_phase()`
- Set up after components are built and constructed

`def start_of_simulation_phase()`
- Set up before simulation starts

`def run_phase()`
- Run behavior of the component – generate stimulus and check results

`def extract_phase()`
- Retrieve and process information from scoreboards, monitors, etc.

`def check_phase()`
- Check that DUT behavior is correct and identify testbench execution errors

`def report_phase()`
- Display or log simulation results

`def final_phase()`
- Complete any outstanding testbench actions

Fundamentals: UVM Phasing

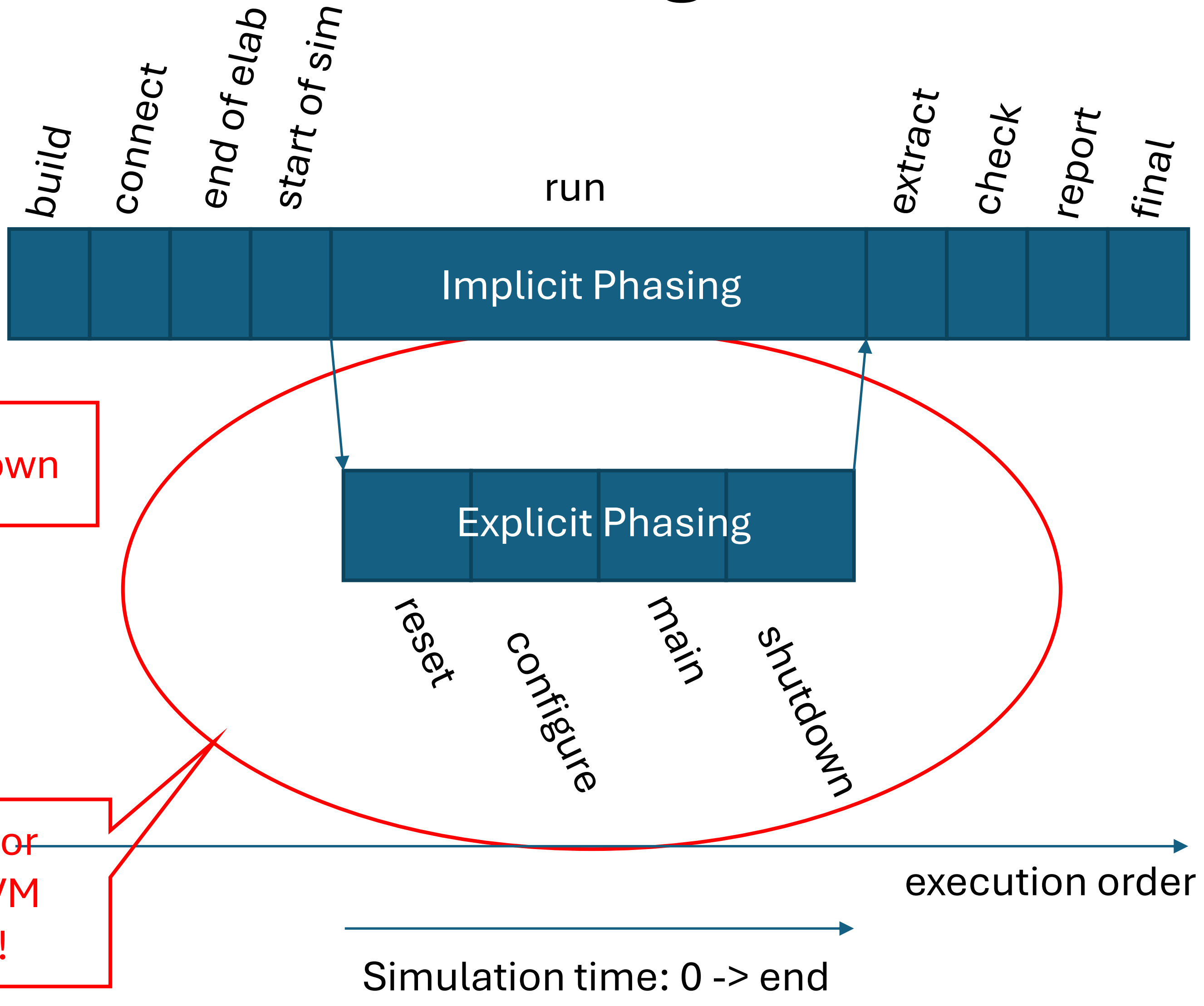
```
class uvm_component(uvm_report_object):
     ...
    def __init__(self, name, parent):
     ...
    def build_phase(self):
     ...
    def connect_phase(self):
     ...
    def end_of_elaboration_phase(self):
     ...
    def start_of_simulation_phase(self):
     ...
    async def run_phase(self):
    ...
    def extract_phase(self):
     ...
    def check_phase(self):
     ...
    def report_phase(self):
     ...
    def final_phase(self):
     ...
```

Top Down

Bottom Up

Parallel execution
CocoTB coroutine

Not for PyUVM Yet!



UVM Fundamentals
ConfigDB

Configuration database a.k.a ConfigDB

- The **ConfigDB** class is a singleton database that stores objects using keys.
  - It is much like a Python dictionary except that the keys must be strings and we can control the visibility to the data using the UVM hierarchy.
  - Key/value store → A Python dictionary ☺

- pyuvm version of ConfigDB() is much simpler than the SystemVerilog
  - Don’t have to deal with parameterized types etc..

- Used for ‘resources’
  - Recommendation is to use it for config information and interface handles, but potentially any object instance
  - When used more widely then it will potentially open up for “Spaghetti code”

ConfigDB Object

- API methods
  - `set()`
    - Place or modify a resource in the resource database
  - `get()`
    - Fetch the value of an existing resource in the resource database
  - There are other API methods, but they are rarely if ever used
    - `wait_modified()` will block until a value is set or updated

`set()`

```
def set(cntxt, inst_name, field_name, value)
```

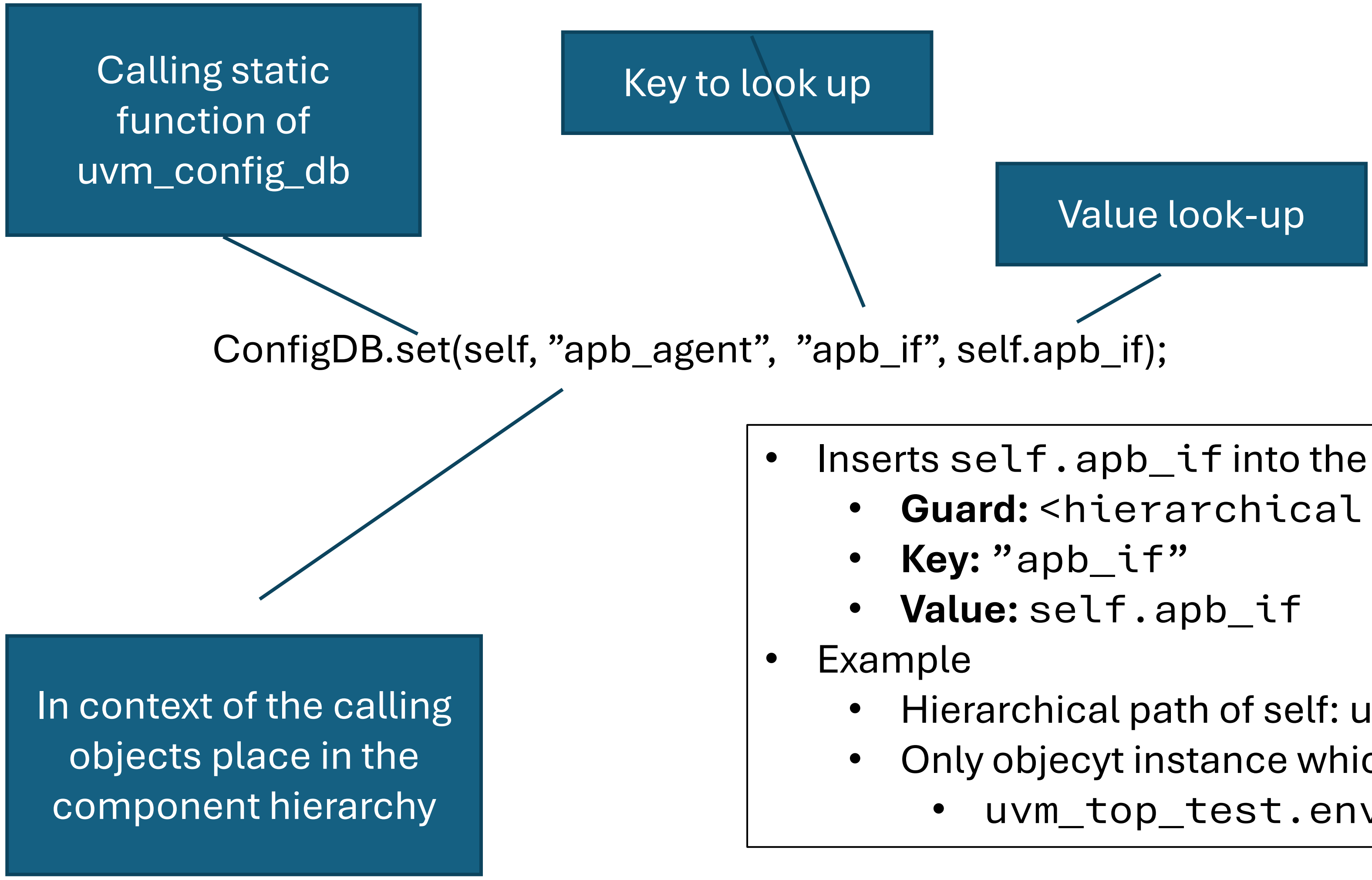
- Create a new or update an existing resource at `field_name` with `value`
  cntxt, inst_name, field_name, value, T=None
  - The full name of the `cntx` component is concatenated with the `inst_name` to form the scope of resource
    - If `cntx` is `null` then it is replaced with `uvm_top`
- If a resource with the same type and the same context already exists then it's value is updated
  - Else a new resource is created
- Since `set()` is a static method it may be called from anywhere

`get()`

```
def get(cntxt, inst_name, field_name, value)
```

- Get the value for an existing resource at `field_name` with `value`
  - The full name of the cntx component is concatenated with the `inst_name` to form the scope of resource
- Since `get()` is a static method it may be called from anywhere

ConfigDB Example


Calling static function of uvm_config_db
Key to look up
Value look-up
In context of the calling objects place in the component hierarchy

```
ConfigDB.set(self, ”apb_agent”,  ”apb_if”, self.apb_if);
```

- Inserts `self.apb_if` into the ConfigDB *dictionary*
  - **Guard:** `<hierarchical path of self>` + `”apb_agent”`
  - **Key:** `”apb_if”`
  - **Value:** `self.apb_if`
- Example
  - Hierarchical path of self: `uvm_top_test.env`
  - Only objecyt instance which can access
    - `uvm_top_test.env.apb_agent`

UVM Fundamentals
Factory

Introducing the UVM Factory

- A well-established object-oriented best practice, a creation pattern is called the **Factory Pattern** (en.wikipedia.org/wiki/Abstract_factory_pattern)
- The **`factory`** is singleton of the **`uvm_factory`** class
  - Used to create objects dynamically
  - Its main benefit is that the type of object that is constructed can be specified at ***runtime*** for maximum flexibility
    - a.k.a., ***"Polymorphic Construction"***
- The **`factory`** itself is not a part of the testbench component hierarchy
  - Its scope is the **`pyuvm`**
- The **`factory`** can be used to create anything derived from **`uvm_object`**
  - Testbench components or transaction objects
- The **`factory`** provides an ***override*** mechanism
  - Helps when making modifications to an existing testbench
  - More on overrides in a later section

UVM Factory

- Standard OOP pattern
- Allows run-time control of created object types through ‘factory overrides’
- Factory overrides can be specified by type or by instance
- Allows for very flexible stimulus patterns
- Can have performance impact!

Registering Types with the Factory

- In SystemVerilog UVM then for the factory to be able to create a component
  - Must first ***register*** that type with the factory
  - Limitation
    - Limits class constructor arguments to just name and parent
- Registration is done by using macros
    - **`` `uvm_component_utils(``*`typename`*`)`**
    - **`` `uvm_object_utils(``*`typename`*`)`**
- In **`pyuvm`**
  - Don’t have a uvm_component_utils() macro that registers the class with the factory. In pyuvm, all classes that extend uvm_void are all registered with the factory.

Example using the factory

- Use normal constructor:

```
class TinyTest(uvm_test):
    def build_phase(self):
        self.tc = TinyComponent("tc", self)
```

- Use factory by using the create() method:

```
class TinyFactoryTest(uvm_test):
    def build_phase(self):
        self.tc = TinyComponent.create("tc", self)
```

Factory Overwrite

- Type Override
  - `set_type_override_by_type(<original class>, <overriding class>)`
  - `set_type_override_by_name(<"original class name">, <“overriding class name”>)`

- Inst Override
  - `set_inst_override_by_type(<original class>, <overriding class>, <"UVM hierarchy path">)`
  - `set_inst_override_by_name(<"original class name">, <“overriding class name”>, <"UVM hierarchy path)`

Factory Overwrite Example

- Class MediumComponent which should replace TinyComponent:

```
class MediumComponent(TinyComponent):
    async def run_phase(self):
        self.raise_objection()
        self.logger.info("I'm medium size.")
        self.drop_objection()
```

- Overrides TinyComponent with MediumComponent:

```
class MediumFactoryTest(TinyFactoryTest):
    def build_phase(self):
        uvm_factory().set_type_override_by_type(TinyComponent, MediumComponent)
        super().build_phase()
```

UVM Fundamentals
Macros

UVM Field Macros in SystemVerilog

- Registers the component or object with the factory
- Provides standardized implementations of print, copy, compare and pack/unpack operation
- Macros are provided for most types *except* struct and more-than-1D arrays
- Can be custom implemented by overriding `do_yyy` functions

UVM Field Macros in SystemVerilog

```
class cl_sdt_seq_item#(pk_sdt::t_params PARAMS) extends uvm_sequence_item;

  rand tp_access                          access; // SDT access type
  rand logic[PARAMS.SDT_AWIDTH-1:0] addr;   // SDT address
  rand logic[PARAMS.SDT_DWIDTH-1:0] data;   // SDT data
  rand int                                dummy;

  `uvm_object_param_utils_begin(cl_sdt_seq_item#(PARAMS))

    `uvm_field_enum (tp_access, access, UVM_ALL_ON)
    `uvm_field_int  (addr,              UVM_ALL_ON)
    `uvm_field_int  (data,              UVM_ALL_ON)
    `uvm_field_int  (dummy,             UVM_ALL_ON | UVM_NOCOMPARE)

  `uvm_object_utils_end
  ...
endclass: cl_sdt_seq_item
```

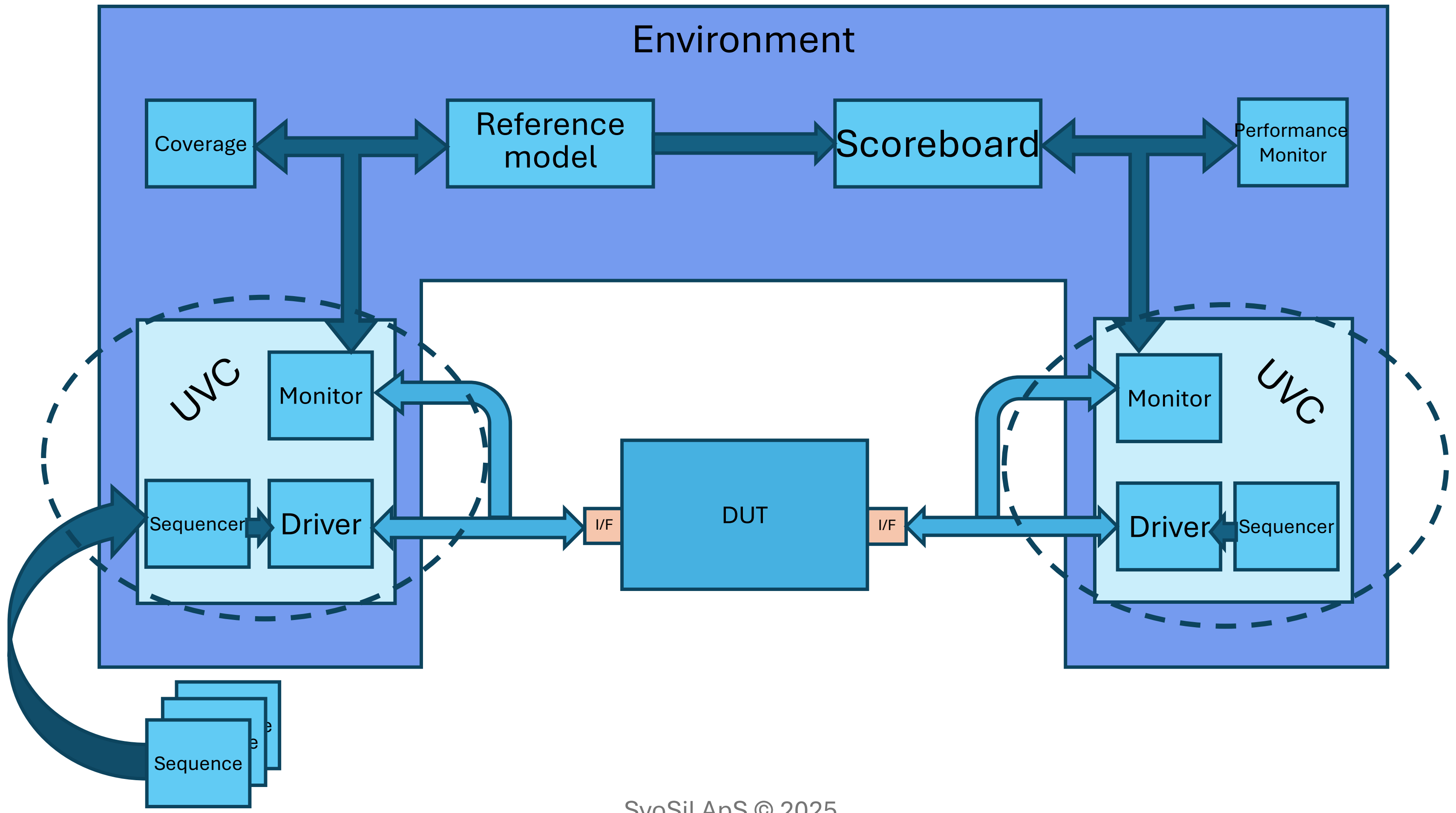
Disable compare for `dummy` member variable

UVM Field Macros in PyUVM

- Components or objects already registered with the factory
- No need for field macros
  - `__eq__`
    - Replaces do_compare
  - `__str__`
    - Is used to generate a string which for instance can be printed
  - `do_copy(self, rhs)`
    - Creates a new copy of the object

UVM Fundamentals
UVC

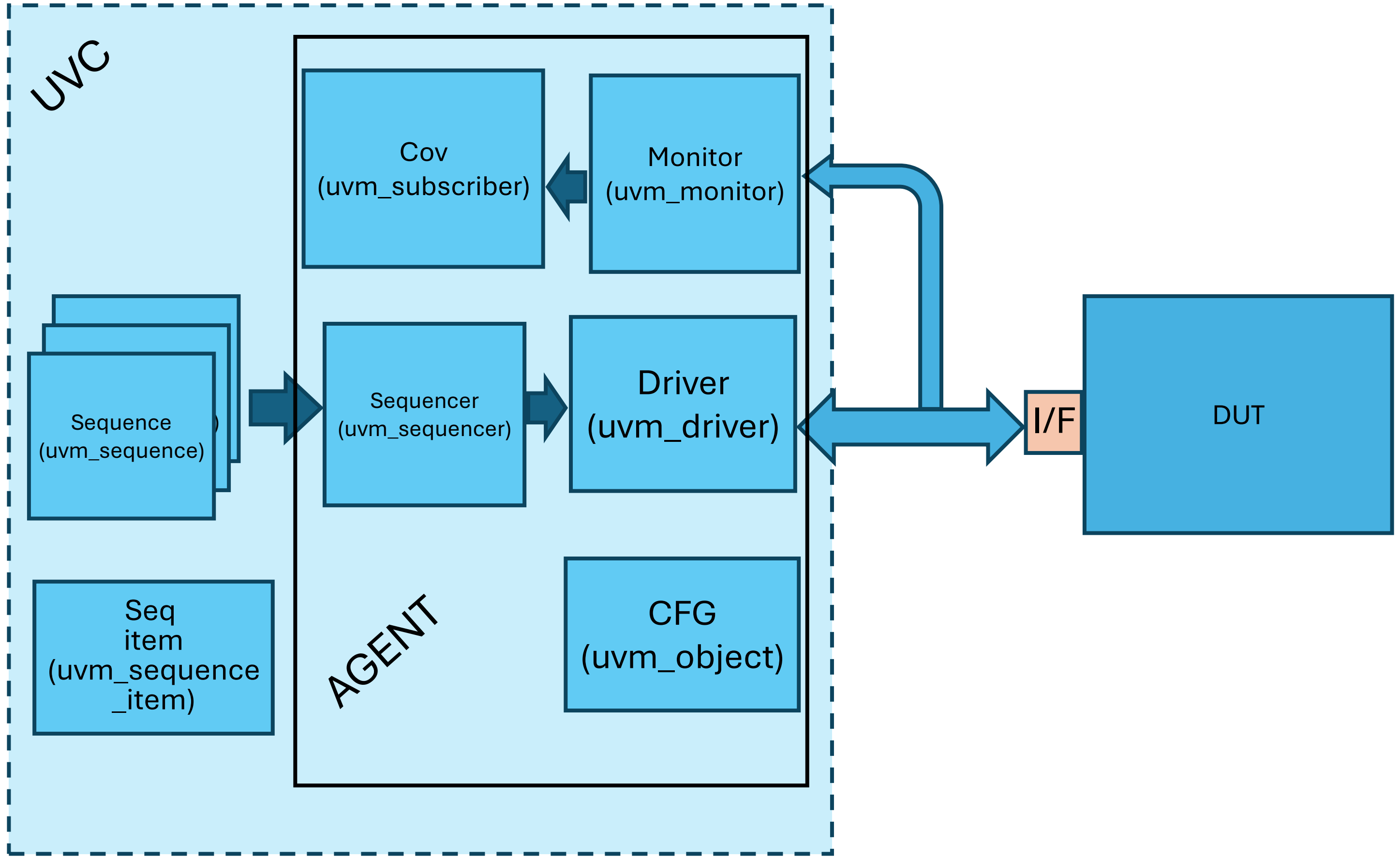
UVM Verification Components (UVCs)



UVC

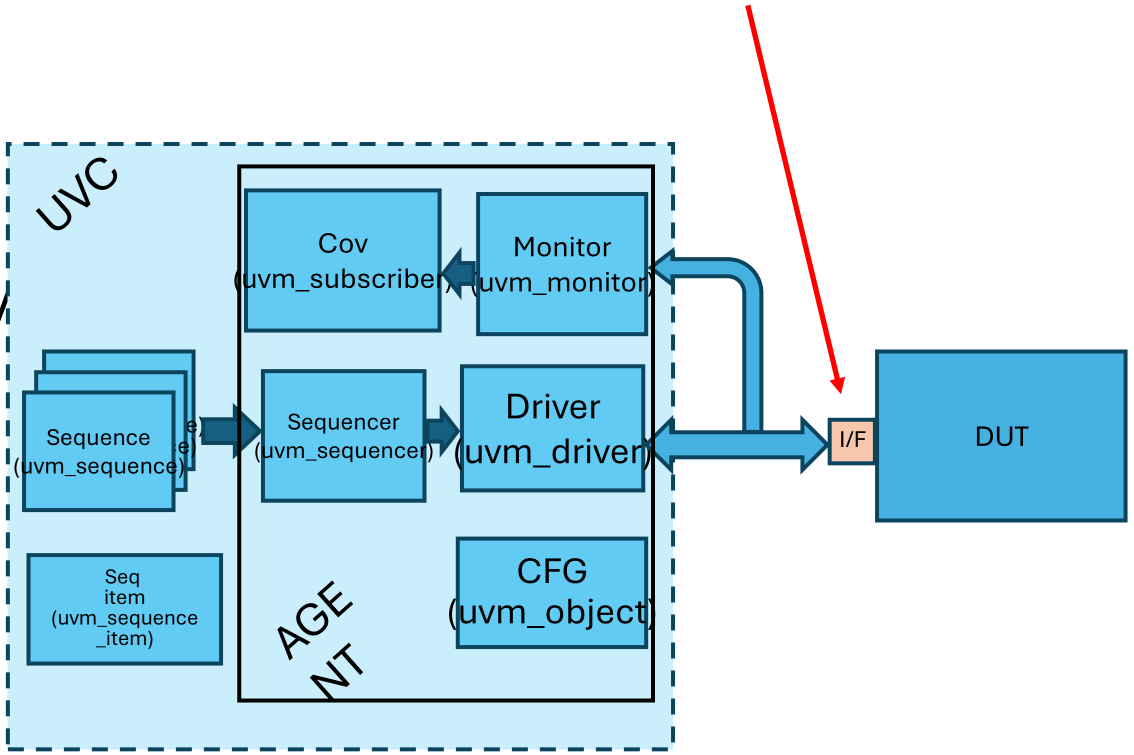
- At least following elements:
  - Interface
  - Sequence item
  - Monitor
  - Driver
  - Sequencer

- Usually also:
  - Configuration object
  - Sequence library
  - Coverage collection



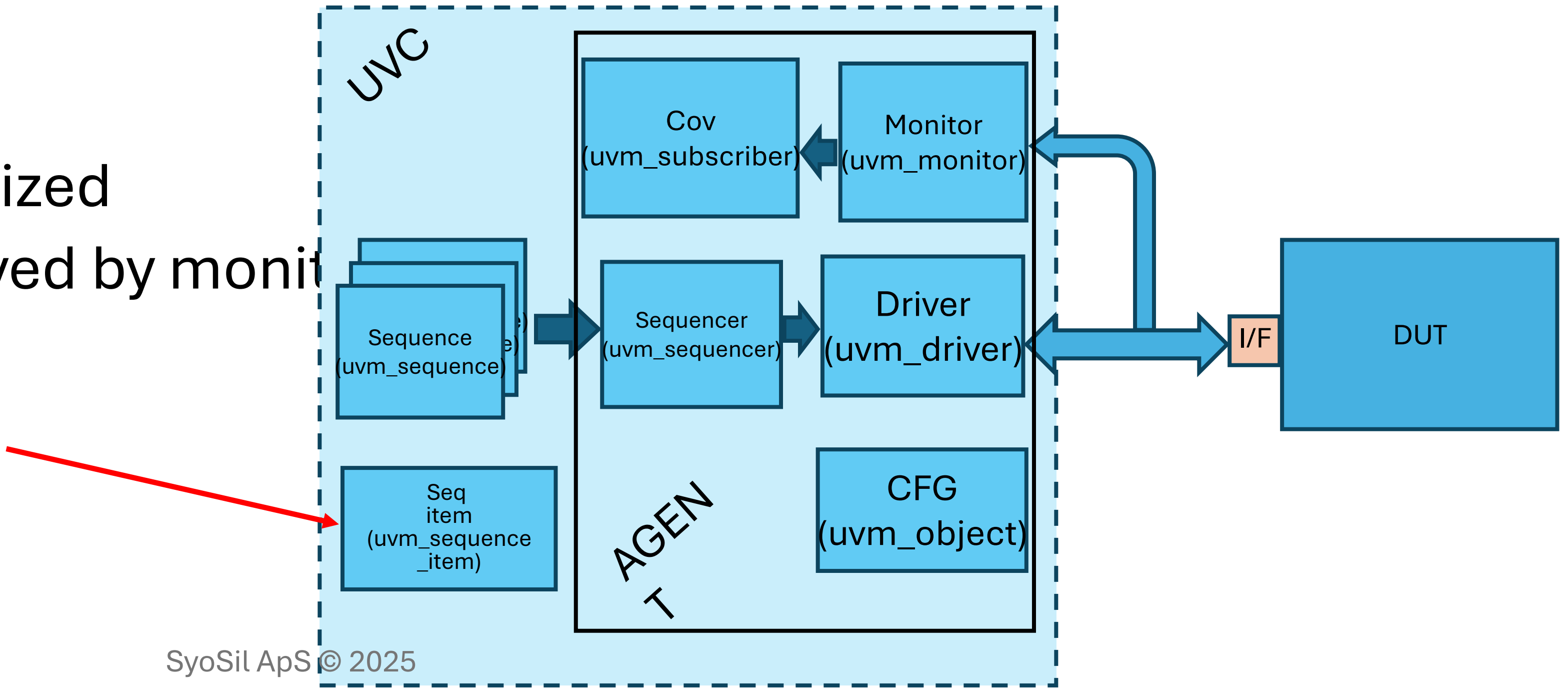
Interface (non-UVM)

- Necessary for UVM to hook up to RTL DUT
  - Using virtual interface reference
- Contains
  - Wires for DUT pin hookup
  - Assertions (SVAs) & coverage metrics
  - Potentially modports, clocking blocks
  - Potentially helper code for the UVM driv



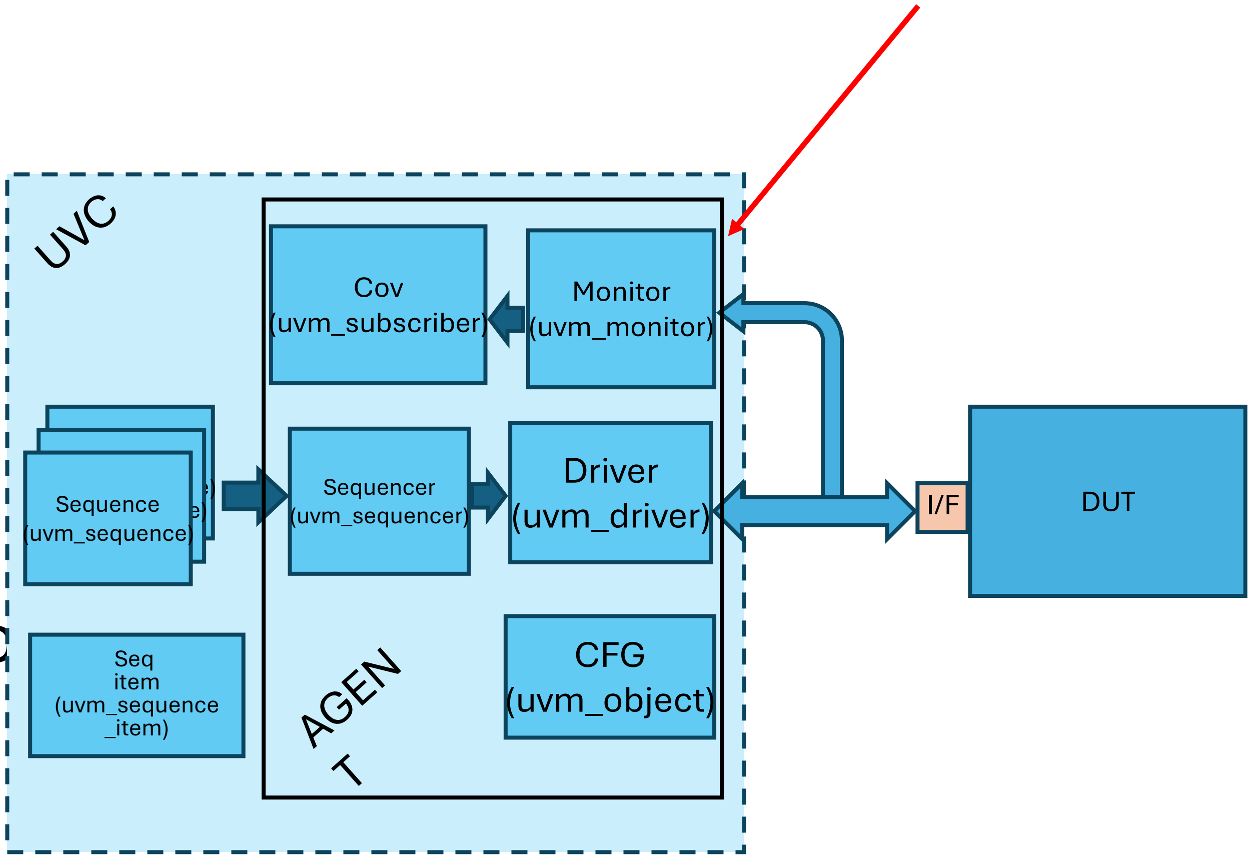
Sequence item (uvm_sequence_item)

- UVM Object
- Capturing a **transaction** on the abstract level
  - Rand data members
  - Constraints
- Used for (at least):
  - Stimuli: Can be randomized
  - Recorded traffic, observed by monit
  - Coverage analytics
  - REF/SCB

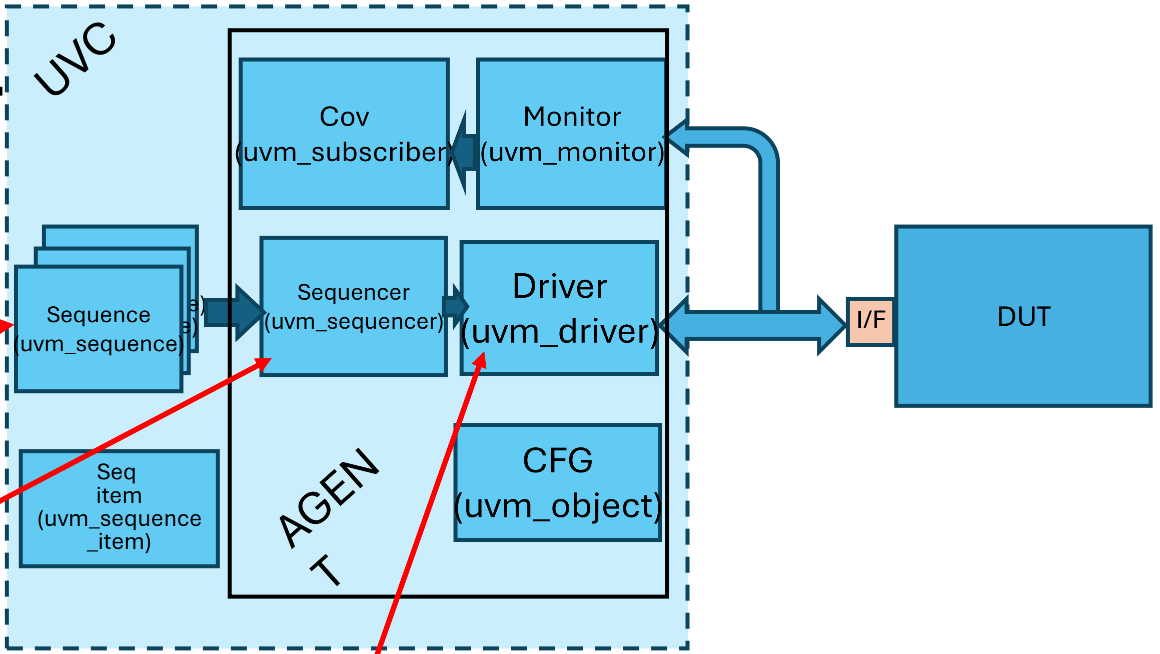


Monitor (uvm_monitor)

- UVM Component
- Monitors bus at pin level
- Publishes transactions for analysis
  - Coverage
  - Reference model & scoreboarding
- Responsible for validating bus proto
  - Contains protocol checks
  - May also contain some pin coverage



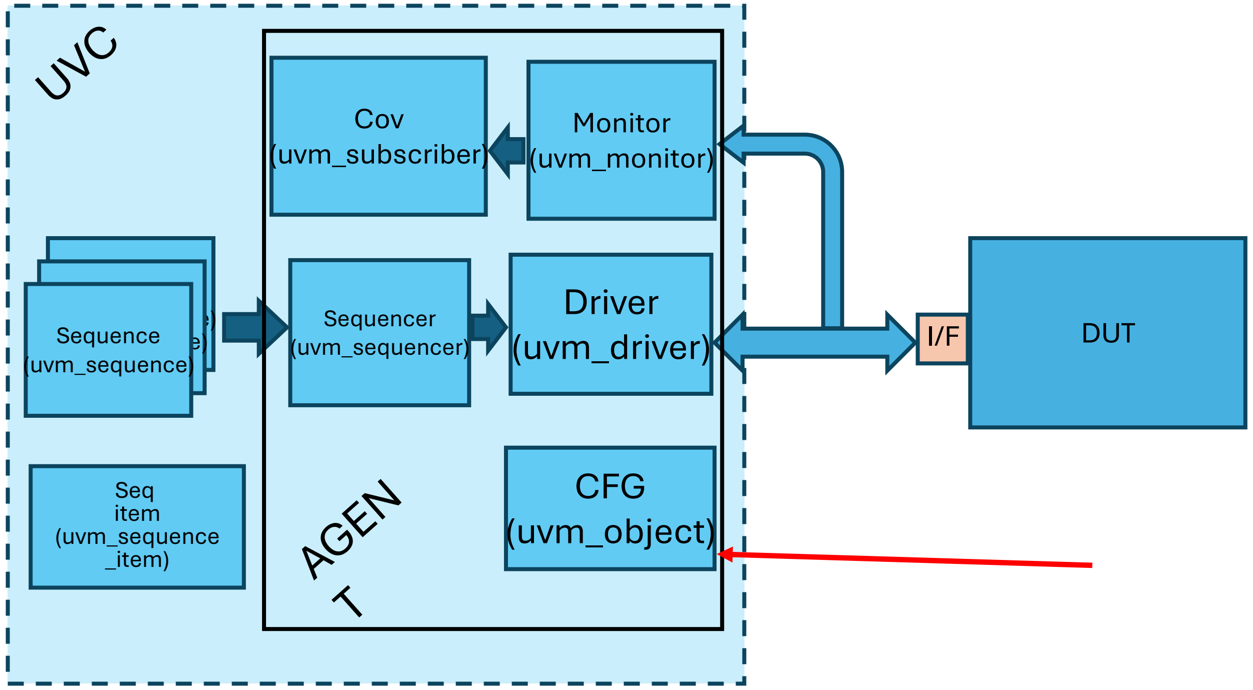
Sequences, Sequencer, Dr



- Sequences (uvm_sequence)
  - UVM Object
  - Sequences provide transactions
- Sequencer (uvm_sequencer)
  - UVM Component
  - Arbitrates between sequences requesting access to driver
  - Sequencer routes response to appropriate sequence
- Driver (uvm_driver)
  - UVM Component
  - Converts transactions to pin wiggles
  - Driver may provide transaction response
  - Communicates with sequencer through function calls (backed by TLM channels)

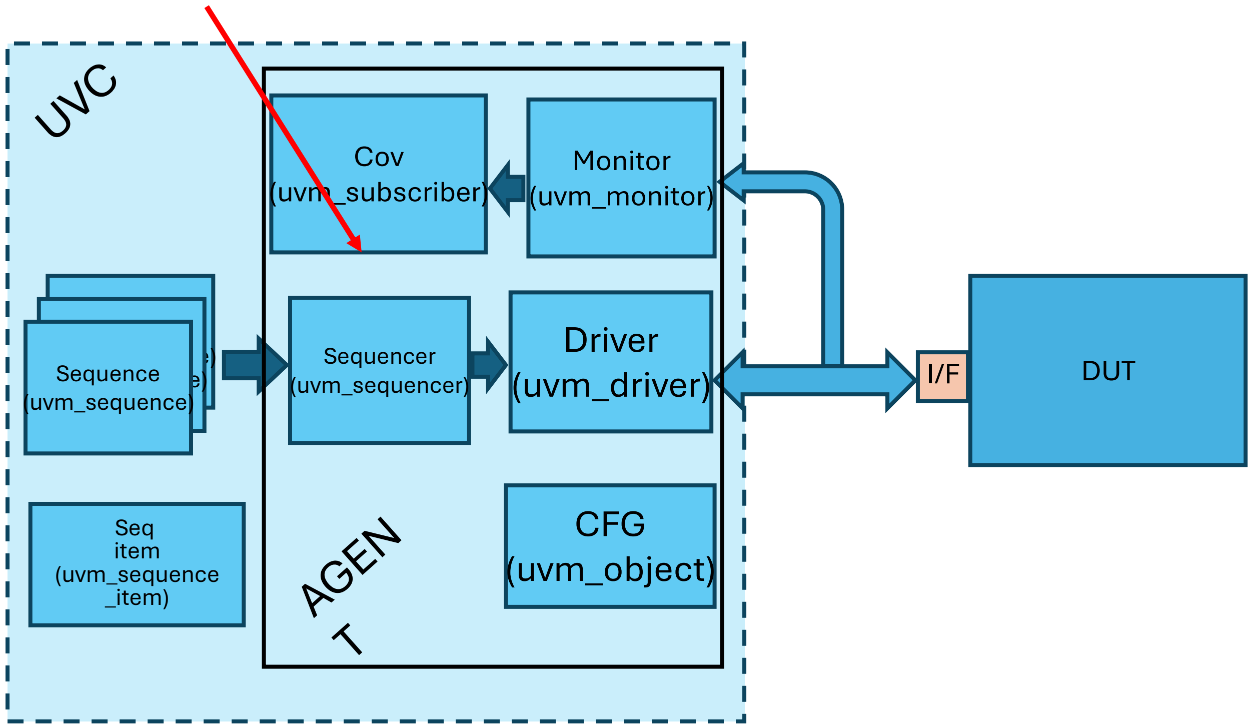
Configuration object (uvm_object)

- UVM Object
- Contains
  - Virtual interface reference
  - UVC configuration options, examples:
    - Protocol version
    - Enable coverage
    - DUT bypass
    - Coverage configuration
    - Allowed transaction IDs
    - Max outstanding transactions
- Collects relevant configuration options into single object → fewer configDB lookups
- One configuration object instance per UVC instance



Coverage collector (uvm_subscriber)

- UVM Component
- Subscribing to monitor observated sequence items
- Protocol centric coverage model
  - SV functional coverage
  - Examples:
    - Access types
    - Address map
    - Transaction Ids
- Extended from uvm_subscriber
  - Specializations should override the write() function.



UVM Fundamentals
Register Model

Register Model

- Perhaps Later! ☺

UVM Fundamentals
TLM Communication

Fundamentals: TLM communication

- Also Later! ☺
- Transaction Level Modelling
- Standardized way to communicate between components
- Communication happens in channels between initiators and targets
- Point-to-point or publisher-subscriber model

Questions