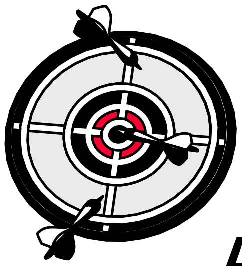
Agenda: Day 1

DAY 1

1	OOP Inheritance Review
2	UVM Structural Overview
3	UVM Transaction
4	UVM Sequence

Unit Objectives



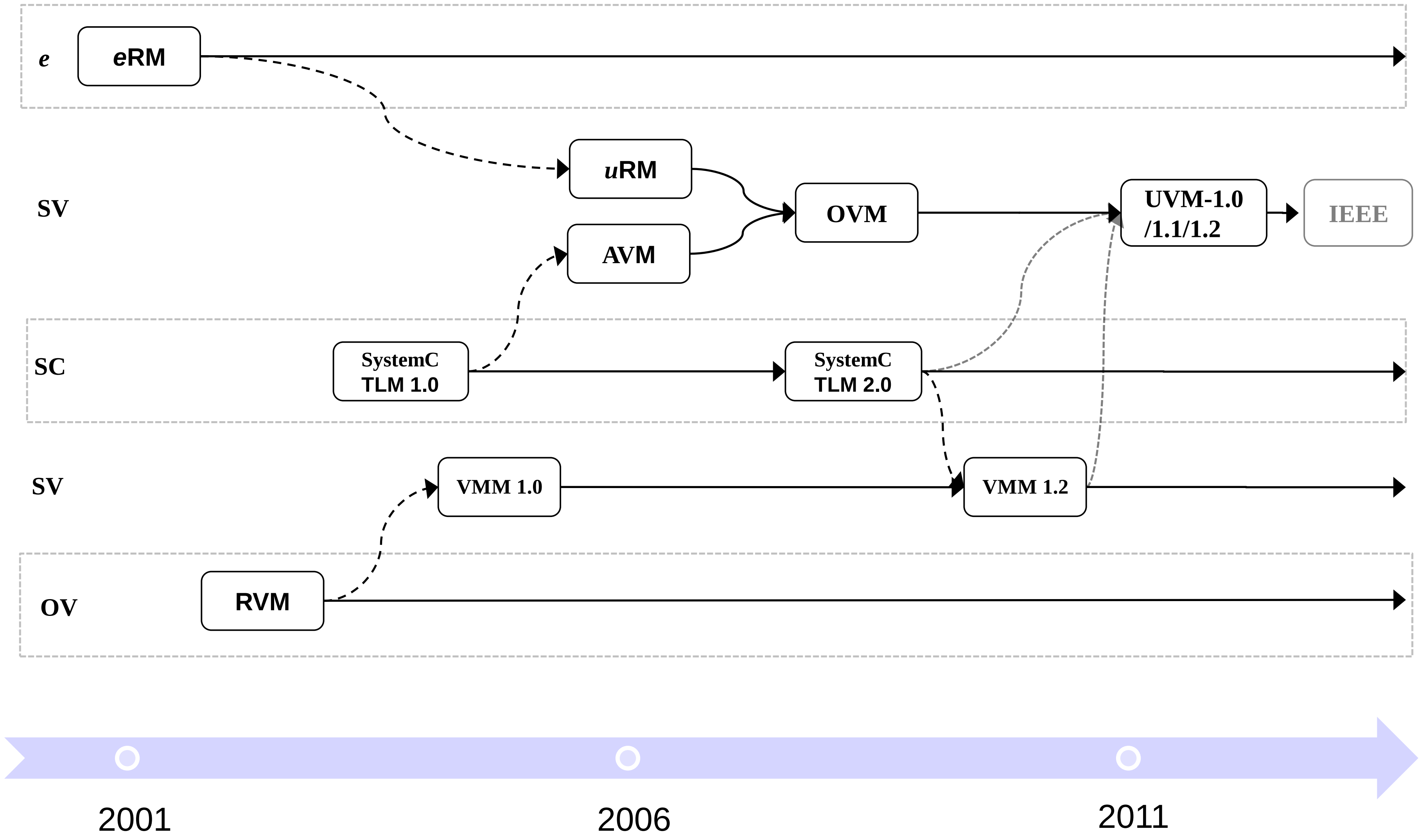
After completing this unit, you should be able to:

- Describe the process of reaching verification goals
- Describe the UVM testbench architecture
- Describe the different components of a UVM testbench
- Bring different components together to create a UVM environment

UVM - Universal Verification Methodology

- **An effort (by an Accelerate committee) to define a standard verification methodology & base class library**
  - Uses classes and concepts from VMM, OVM
  - UVM-1.2 has been submitted to IEEE for standardization
- **Related Websites:**
  - UVM Public Website – http://www.accellera.org/community/uvm
  - Mantis (Bug Tracking) – http://eda.org/svdb/view_all_bug_page.php
- **Synopsys verification video, blog & SNUG:**
  - http://blogs.synopsys.com/vip-central/
  - http://www.synopsys.com/Community/SNUG/Pages/default.aspx
  - http://www.synopsys.com/Support/Training/Pages/ces-training-videos-2016.aspx
  - https://www.youtube.com/user/synopsys

Origin of UVM



Verification Goal

- **Ensure full conformance with specification:**
  - Must avoid **false** passes

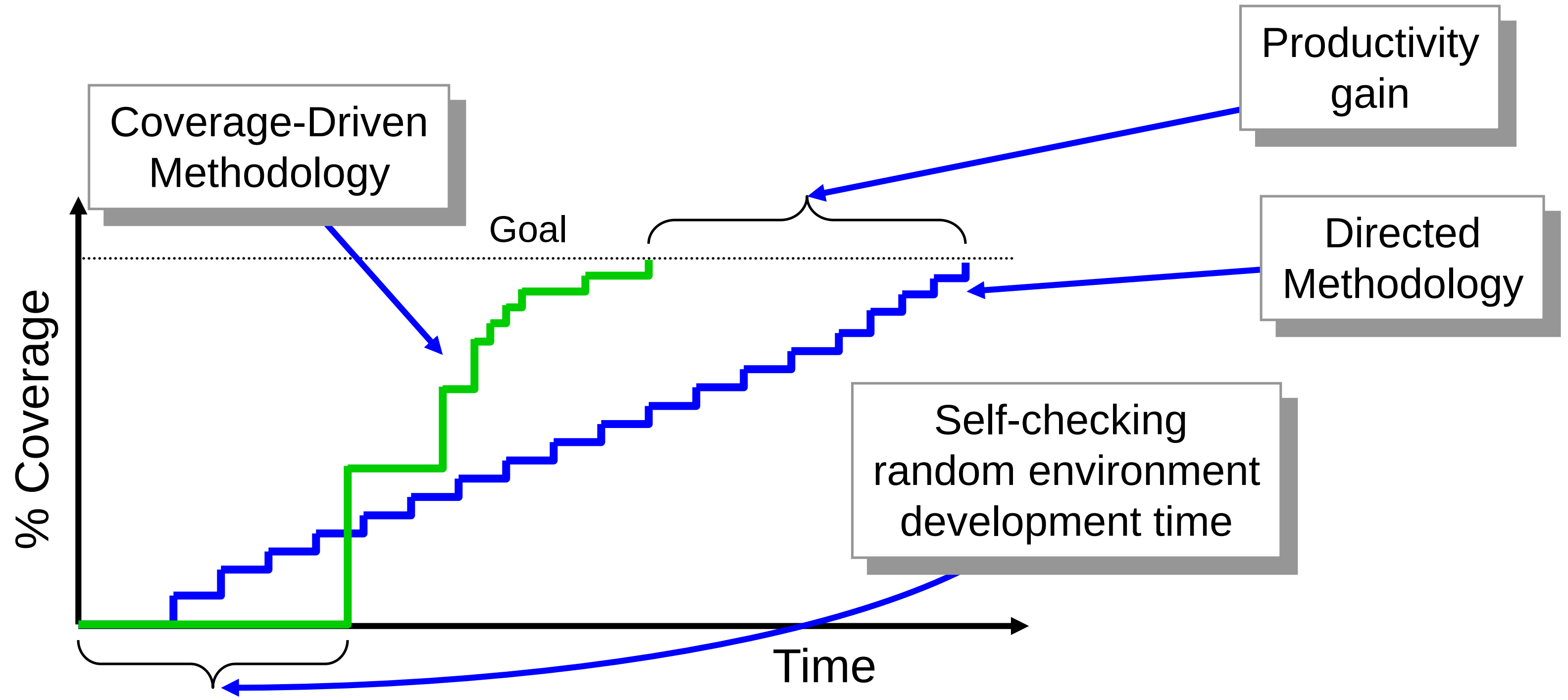
Testbench Simulation result	RTL code	
	Good	Bad(bug)
Pass	√ Tape out!	???
Fail	Debug testbench	Debug RTL code

False pass results in shipping a bad design

How do we achieve this goal?

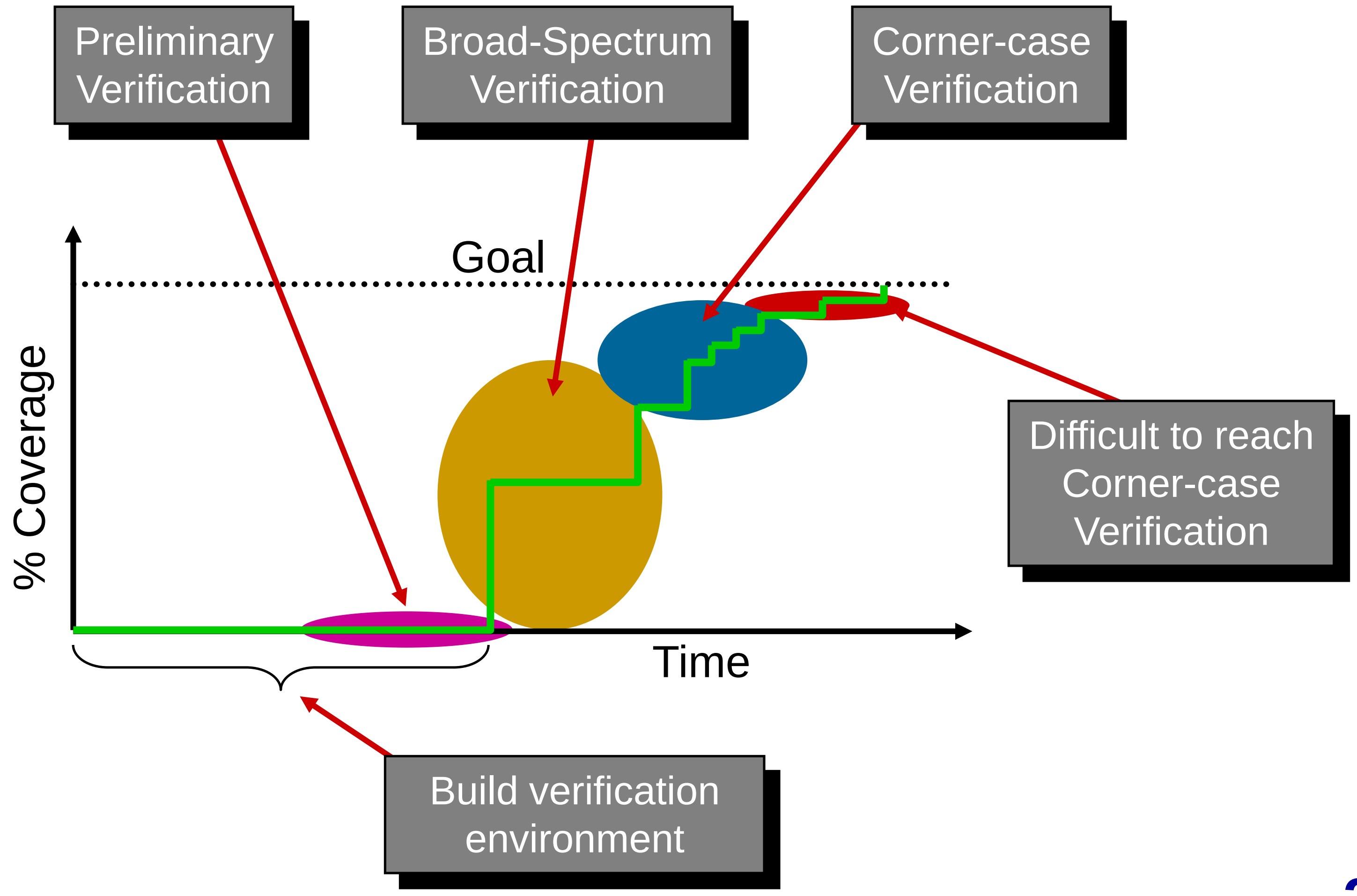
Coverage-Driven Verification

- Focus on uncovered areas
- Trade-off authoring time for run-time
- Progress measured using functional coverage metrics



Phases of Verification

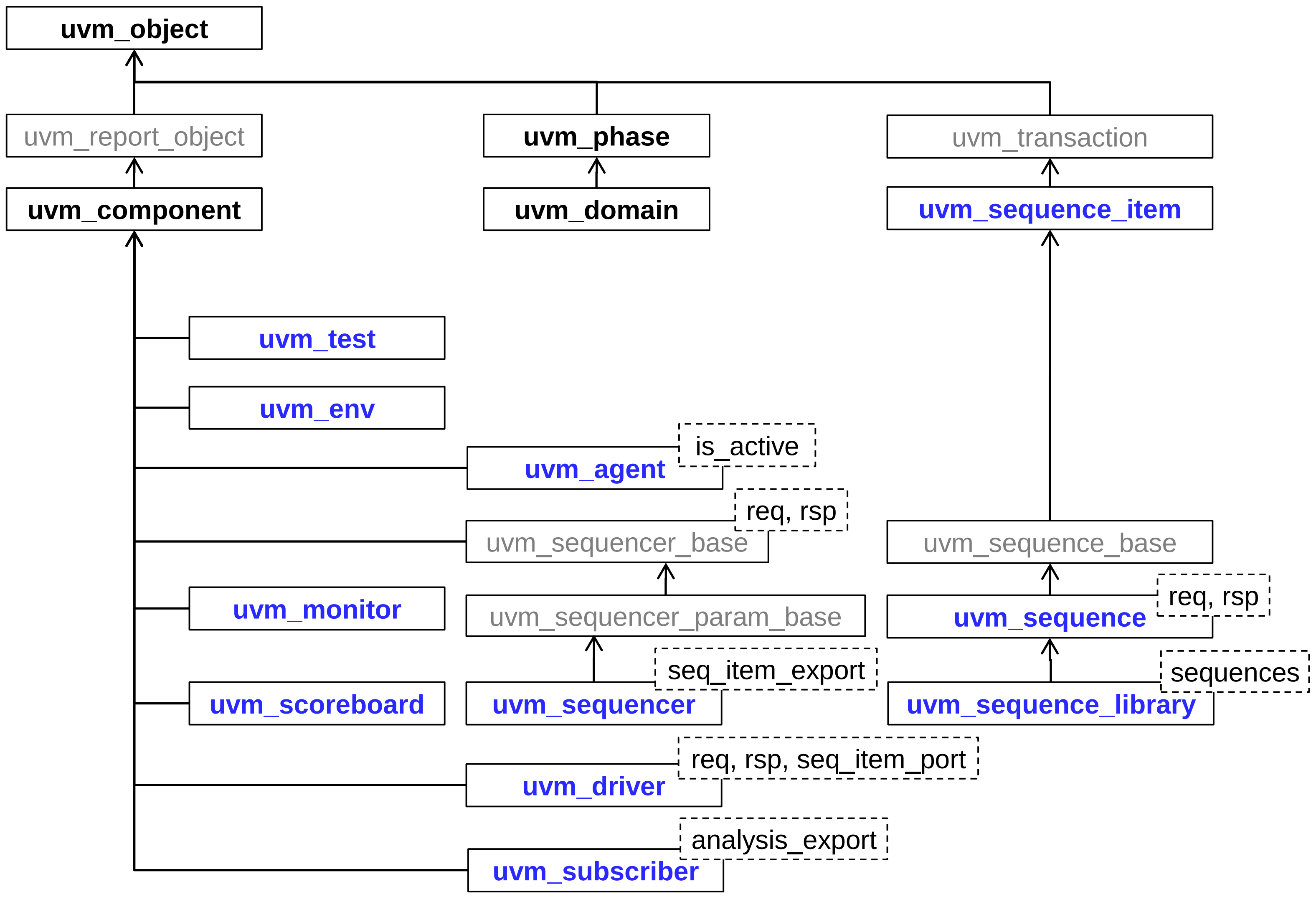
Start with fully random environment. Continue with more and more focused guided tests


Preliminary Verification
Broad-Spectrum Verification
Corner-case Verification
Goal
% Coverage
Time
Difficult to reach Corner-case Verification
Build verification environment

Run More Tests, Write Less Code

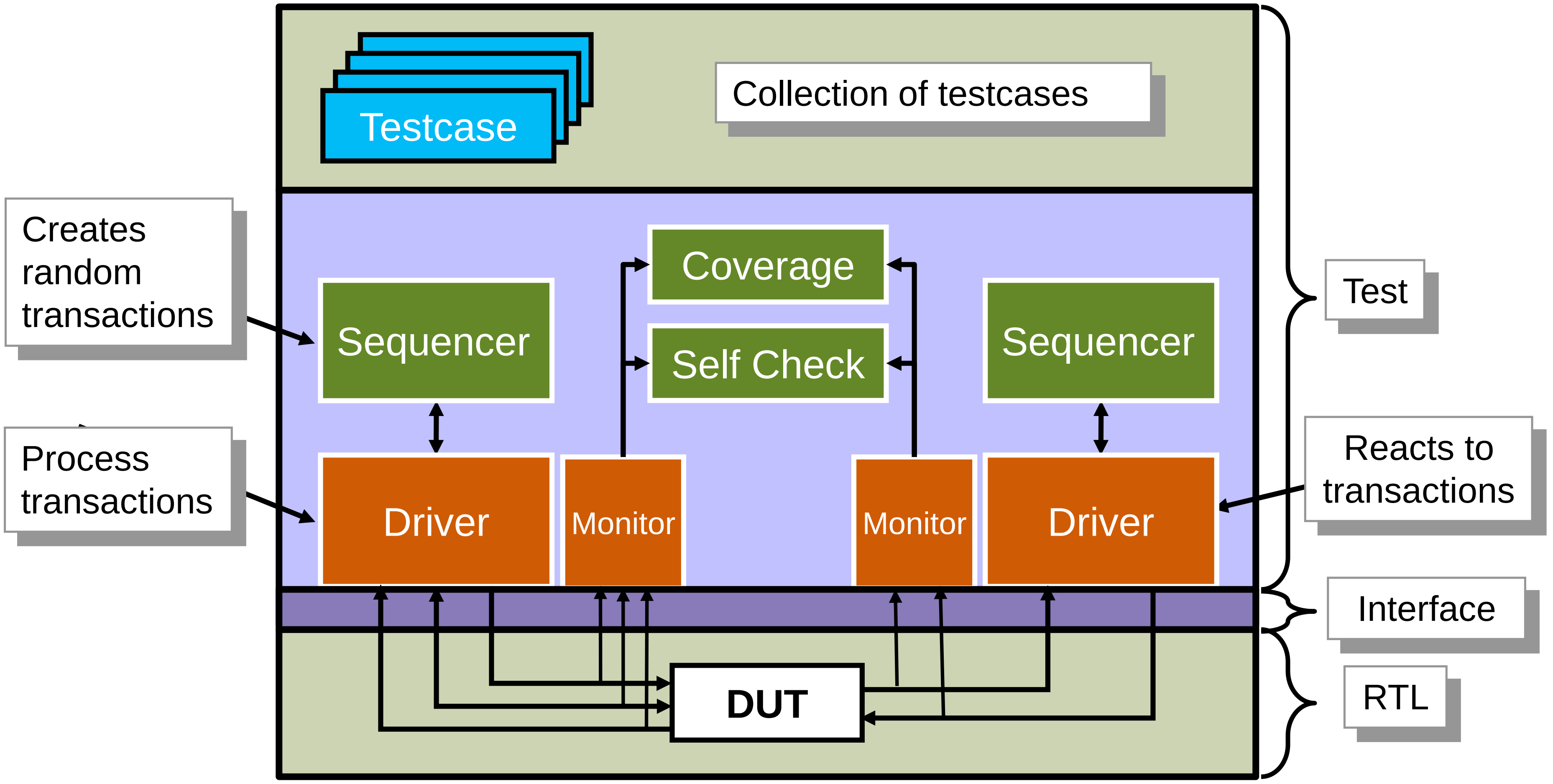
- **Environment and component classes rarely change**
  - Sends good transactions as fast as possible
  - Keeps existing tests from breaking
  - Leave "hooks" so test can inject new behavior
    - Virtual methods, factories, callbacks
- **Test extends testbench classes**
  - Add constraints to reach corner cases
  - Override existing classes for new functionality
  - Inject errors, delays with callbacks
- **Run each test with hundreds of seeds**

UVM Class Tree (Partial)

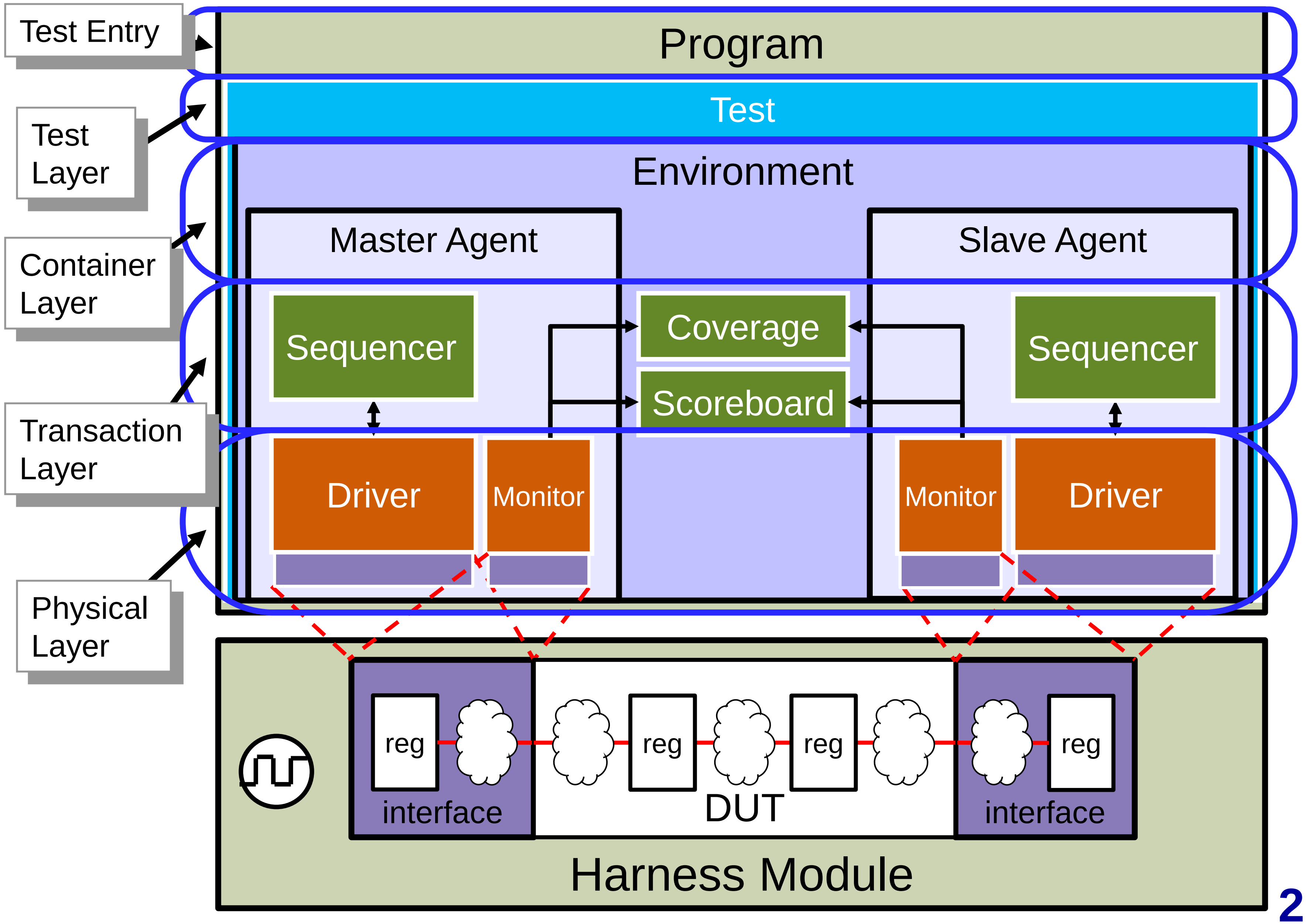


Typical Testbench Architecture

- **SystemVerilog testbench structure**

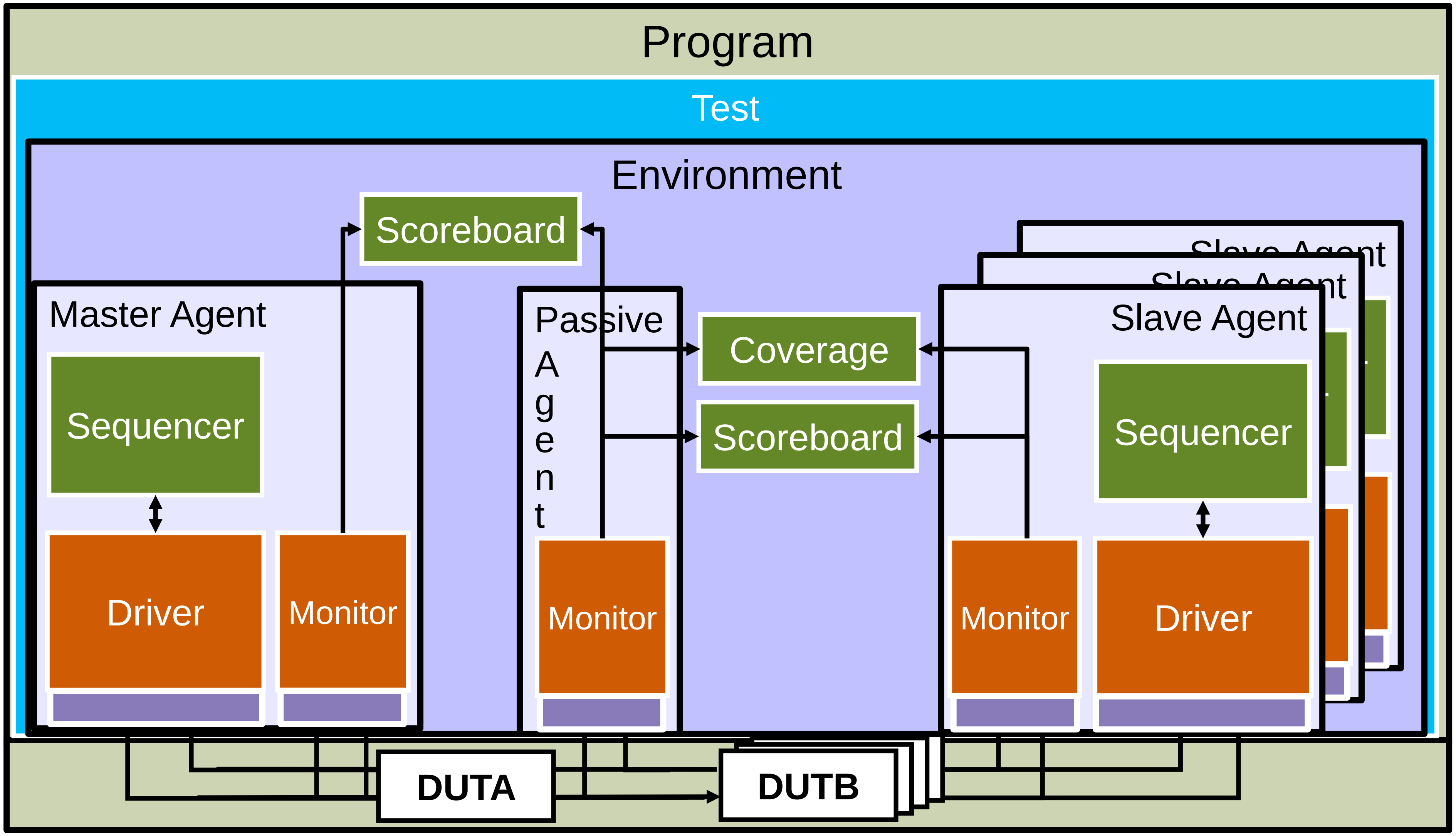


UVM Testbench Architecture



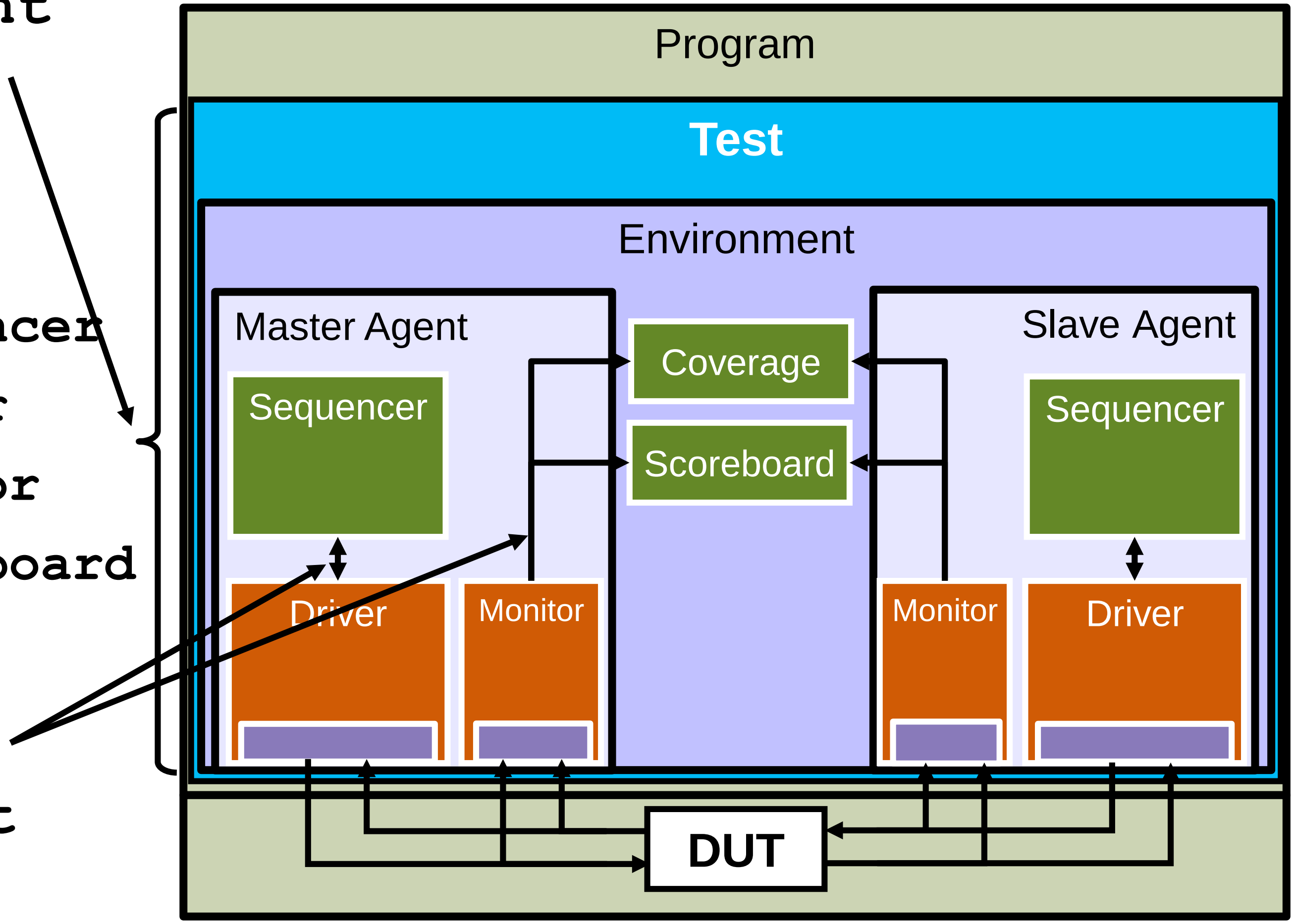
UVM Structure is Scalable

- Agents are the building blocks across test/projects



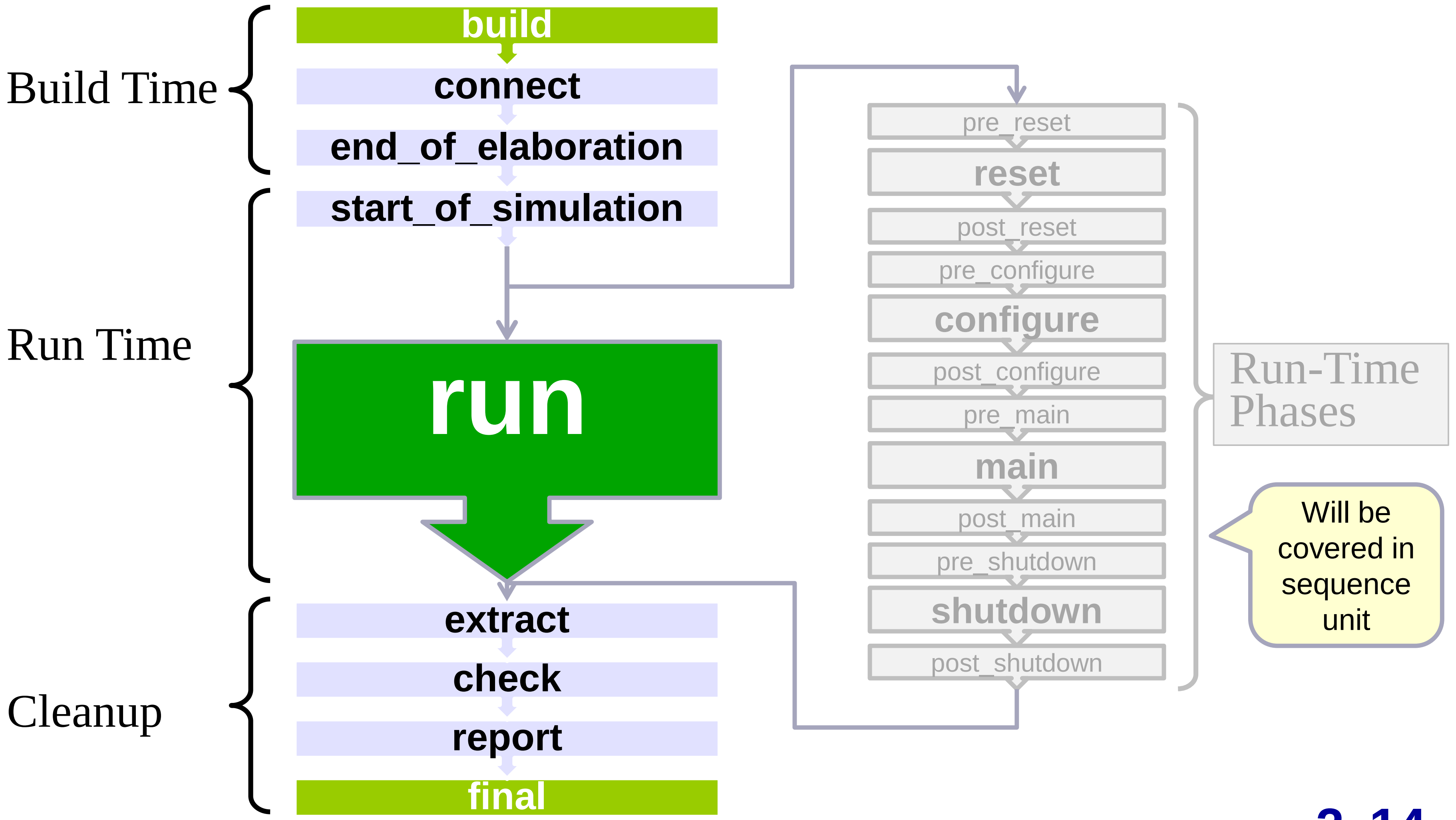
Structural Class Support in UVM

- Structural & Behavioral
  - `uvm_component`
    - `uvm_test`
    - `uvm_env`
    - `uvm_agent`
    - `uvm_sequencer`
    - `uvm_driver`
    - `uvm_monitor`
    - `uvm_scoreboard`
- Communication
  - `uvm_*_port`
  - `uvm_*_socket`
- Transaction
  - `uvm_sequence_item`
  - `uvm_sequence`



Structural Functional Support - Phasing

- The run phase executes concurrently with the scheduled run-time phase tasks



UVM Hello World Example

- UVM tests are derived from `uvm_test` class
- Execution of test is done via global task `run_test()`

```
program automatic test;
  import uvm_pkg::*;
  class hello_world extends uvm_test;
    `uvm_component_utils(hello_world)
    function new(string name, uvm_component parent);
      super.new(name, parent);
    endfunction
    virtual task run_phase(uvm_phase phase);
      phase.raise_objection(this);
      `uvm_info("TEST", "Hello World!", UVM_MEDIUM);
      phase.drop_objection(this);
    endtask
  endclass
  initial
    run_test();
endprogram
```

Test base class

Create and register test name

DUT functional verification code resides in one of the task phases

Message

Use phase objection mechanism to stay in phase and execute content of method

Execute test

Compile and Simulate

- Compile with **`-ntb_opts uvm-1.2`** switch
- Specify test to run with **`+UVM_TESTNAME`** switch

test.sv

```
program automatic test;  import uvm_pkg::*;
  class hello_world extends uvm_test;
    `uvm_component_utils(hello_world)
    function new(string name, uvm_component parent);
      super.new(name, parent);
    endfunction
    virtual task run_phase(uvm_phase phase);
      phase.raise_objection(this);
      `uvm_info("TEST", "Hello World!", UVM_MEDIUM);
      phase.drop_objection(this);
    endtask
  endclass

  initial run_test();
endprogram
```

Test name

Compile with vcs: (using UVM in VCS installation)

```
vcs -sverilog -ntb_opts uvm-1.2 test.sv
```

Simulate with:

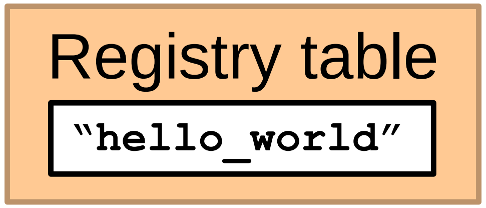
```
simv +UVM_TESTNAME=hello_world
```

```
UVM_INFO @ 0: reporter [RNTST] Running test hello_world ...
UVM_INFO ./test.sv(10) @ 0: uvm_test_top [TEST] Hello World!
```

Inner Workings of UVM Simulation

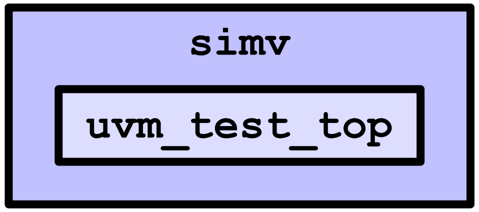
- **Macro registers the class in factory (`uvm_factory::get()`)**

```
class hello_world extends uvm_test;
  `uvm_component_utils(hello_world)
```



- **UVM package contains a singleton `uvm_root` object (`uvm_root::get()`)**

```
import uvm_pkg::*;
```



- **`run_test()` causes this singleton `uvm_root` object to retrieve the test name from `+UVM_TESTNAME` and create a test component called `uvm_test_top`**

```
initial
  run_test();
```

```
simv +UVM_TESTNAME=hello_world
```

```
build_phase
connect_phase
end_of_elaboration_phase
start_of_simulation_phase
run_phase
extract_phase
check_phase
report_phase
final_phase
```

- **Then, the `uvm_root` object executes the phase methods of components**

User Report Messages

- **Print messages with UVM macros**

```
`uvm_fatal("CFGERR", "Fatal message");
`uvm_error("RNDERR", "Error message");
`uvm_warning("WARN", "Warning message");
`uvm_info("REGRESS", "Regression message", UVM_LOW);
`uvm_info("NORMAL", "Normal message", UVM_MEDIUM);
`uvm_info("TRACE", "Tracing execution", UVM_HIGH);
`uvm_info("FULL", "Debugging operation", UVM_FULL);
`uvm_info("DEBUG", "Verbose message", UVM_DEBUG);
```

Failure messages are set to **UVM_NONE**

More verbose

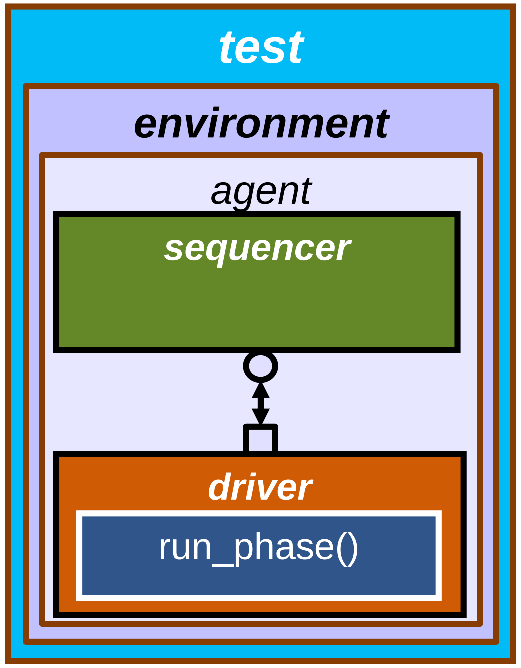
Info messages need to specify verbosity

- **Verbosity filter defaults to `UVM_MEDIUM`**
  - User can modify run-time filter via **`+UVM_VERBOSITY`** switch

```
simv +UVM_VERBOSITY=UVM_DEBUG +UVM_TESTNAME=hello_world
```

UVM Simple Structure Example

- **Structural classes**
  - Test class
    - **`uvm_test`**
  - Environment class
    - **`uvm_env`**
  - Agent class
    - **`uvm_agent`**
  - Sequence execution class
    - **`uvm_sequencer`**
  - Driver class
    - **`uvm_driver`**

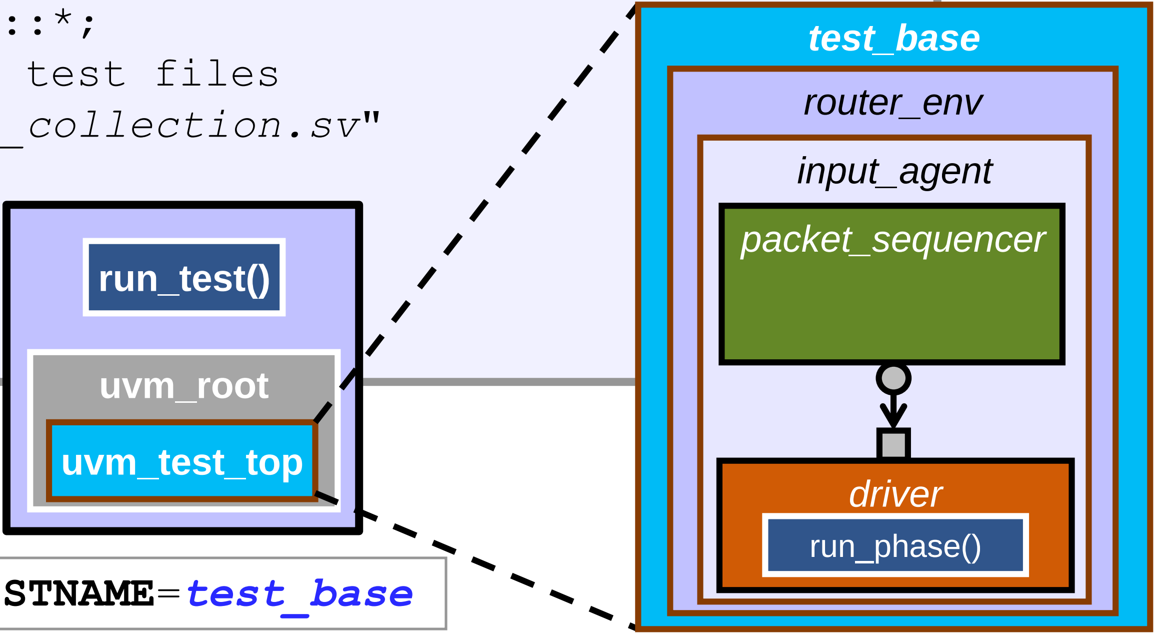


Starting UVM Execution

- **Can be in SystemVerilog `program` or `module`**
  - Includes test class files
  - Start UVM execution in **`initial`** block
    - **`uvm_root`** singleton object will construct and execute the test specified via the **`+UVM_TESTNAME`** switch

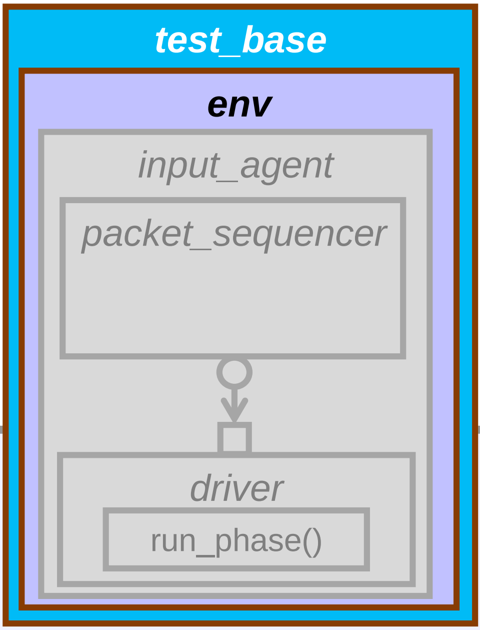
```
program automatic test;
  import uvm_pkg::*;
  // include the test files
  `include "test_collection.sv"
  initial begin
    run_test();
  end
endprogram
```

```
simv +UVM_TESTNAME=test_base
```



Structural Classes - Test

- **UVM test class is the top component of test structure**
  - Extends from the **`uvm_test`** base class
  - Creates environment object



```
class test_base extends uvm_test;
  router_env env;
  `uvm_component_utils(test_base)
  // constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = router_env::type_id::create("env", this);
  endfunction
endclass
```

Construct environment object with UVM factory create mechanism

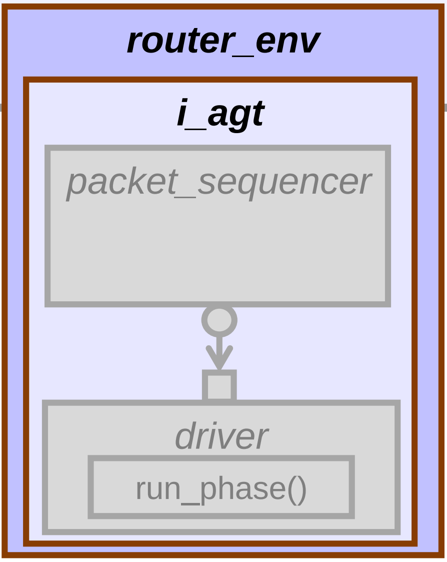
Structural Classes - Environment

- **Encapsulates DUT specific Verification Components**
  - Encapsulate agents, scoreboard and coverage
    - Scoreboard and coverage will be addressed in later units

```
class router_env extends uvm_env;
  input_agent i_agt;
  // utils macro and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    i_agt = input_agent::type_id::create("i_agt", this);
  endfunction
endclass
```

Extends from **uvm_env**

In **build phase**, create agent object



Structural Classes - Agent

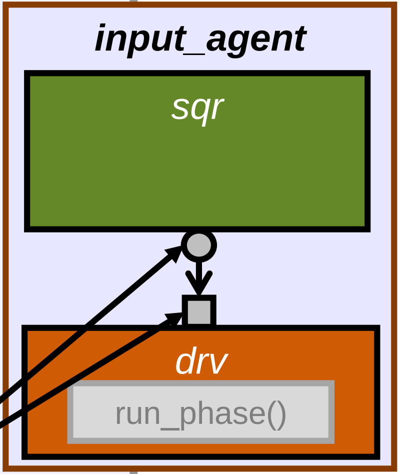
- **Encapsulate sequencer, driver and monitor in agent**
  - In the example code, monitor is left off for simplicity (more on monitor and agent in later unit)

```
class input_agent extends uvm_agent;
  typedef uvm_sequencer #(packet) packet_sequencer;
  packet_sequencer sqr; driver drv;
  // utils macro and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    sqr = packet_sequencer::type_id::create("sqr", this);
    drv = driver::type_id::create("drv", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    drv.seq_item_port.connect(sqr.seq_item_export);
  endfunction
endclass
```

Extend from **uvm_agent**

In **build phase**, construct components

In **connect phase**, connect **built-in TLM ports**

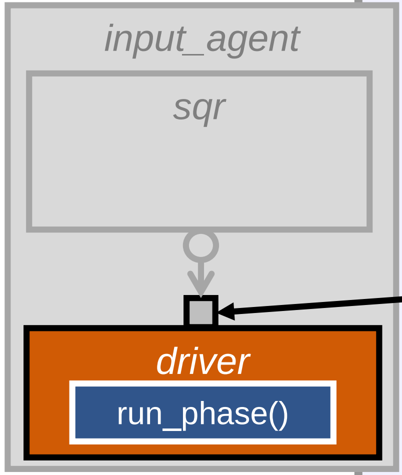


Structural Classes - Driver

- **Driver class extends from `uvm_driver` class**
  - `uvm_driver` class has a built-in TLM port

Must be typed to the sequence item class

```
class driver extends uvm_driver #(packet);
  `uvm_component_utils(driver)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      `uvm_info("DRV_RUN", req.sprint(), UVM_MEDIUM);
      seq_item_port.item_done();
    end
  endtask
endclass
```



Driver implements **run phase** as an infinite loop to process all sequence items received from sequencer. **`item_done()`** in TLM port must be called before retrieving next item.

Structural Classes - Debug

- **Topology of the test can be printed with**

  `uvm_root::get().print_topology()`

```
UVM_INFO @ 0.0ns: reporter [UVMTOP] UVM testbench topology:
-----------------------------------------------------------------
Name                         Type                    Size  Value
-----------------------------------------------------------------
uvm_test_top                 test_base               -     @510
  env                        router_env              -     @517
    i_agt                    input_agent             -     @529
      drv                    driver                  -     @666
        rsp_port             uvm_analysis_port       -     @681
          recording_detail   uvm_verbosity           32    UVM_FULL
        sqr_pull_port        uvm_seq_item_pull_port  -     @673
          recording_detail   uvm_verbosity           32    UVM_FULL
        port_id              integral                32    -1
        recording_detail     uvm_verbosity           32    UVM_FULL
      sqr                    uvm_sequencer           -     @557
```

```
function void test_base::start_of_simulation_phase(uvm_phase phase);
  uvm_root::get().print_topology();  // defaults to table printer
endfunction
```

Print Format Can Be Specified

- **Topology can also be printed in tree format**

```
UVM_INFO @ 0.0ns: reporter [UVMTOP] UVM testbench topology:
uvm_test_top: (test_base@510) {
  env: (router_env@517) {
    i_agt: (input_agent@529) {
      drv: (driver@666) {
        rsp_port: (uvm_analysis_port@681) {
          recording_detail: UVM_FULL
        }
        sqr_pull_port: (uvm_seq_item_pull_port@673) {
          recording_detail: UVM_FULL
        }
        port_id: -1
        recording_detail: UVM_FULL
      }
      sqr: (uvm_sequencer@557) {
        rsp_export: (uvm_analysis_export@564) {
          recording_detail: UVM_FULL
```

```
function void test_base::start_of_simulation_phase(uvm_phase phase);
  uvm_root::get().print_topology(uvm_default_tree_printer);
endfunction
```

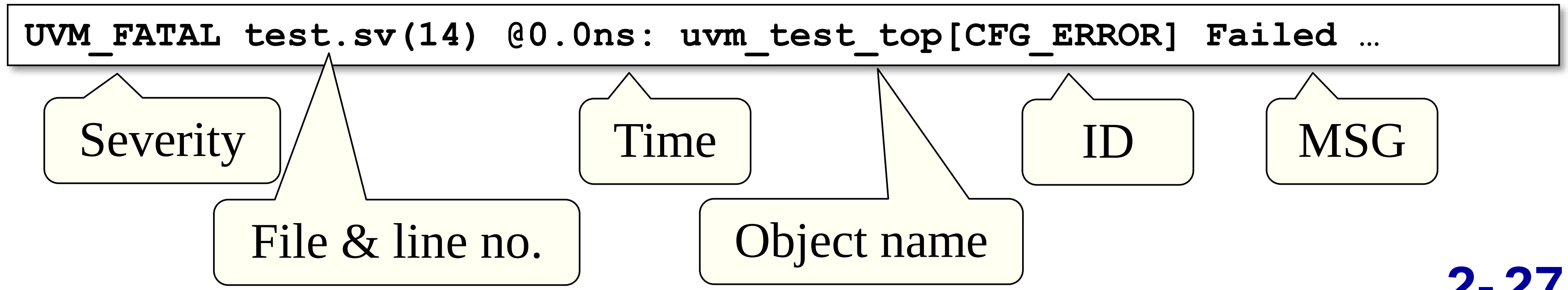
General Debugging with Report Messages

- **Create messages with macros:**

```
`uvm_fatal(string ID, string MSG)
`uvm_error(string ID, string MSG)
`uvm_warning(string ID, string MSG)
`uvm_info(string ID, string MSG, verbosity)
```

Example:

```
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  if (!cfg.randomize()) begin
    `uvm_fatal("CFG_ERROR", "Failed Configuration randomization");
  end
endfunction
```



```
UVM_FATAL test.sv(14) @0.0ns: uvm_test_top[CFG_ERROR] Failed …
```

Default Simulation Handling

Severity	Default Action
UVM_FATAL	UVM_DISPLAY | UVM_EXIT
UVM_ERROR	UVM_DISPLAY | UVM_COUNT
UVM_WARNING	UVM_DISPLAY
UVM_INFO	UVM_DISPLAY

Action	Description
UVM_EXIT	Exit from simulation immediately
UVM_COUNT	Increment global error count. Set count for exiting simulation with `+UVM_MAX_QUIT_COUNT=` run-time switch
UVM_DISPLAY	Display message on console
UVM_LOG	Captures message in a named file
UVM_CALL_HOOK	Calls callback method
UVM_NO_ACTION	Do nothing

User Filterable Code Block

- **Control filtering of block of code**
  - Based on the uvm_report mechanism

Verbosity

Severity

ID

```
if (uvm_report_enabled(UVM_HIGH, UVM_INFO, "CODE_BLOCK")) begin
  $display("Code Block to be filtered");
end
```

- **Does not get tracked by system**

ID will not be reported

```
** Report counts by id
[Comparator Match]    154
[Comparator Mismatch]      6
[DRV_RUN]    160
```

Command Line Control of Report Messages

- **Control verbosity of components at specific phases or times**
  - **id** argument can be _ALL_ for all IDs or a specific id
    - Wildcard for id argument not supported

+uvm_set_verbosity=<*comp*>,<*id*>,<verbosity>,<phase>
+uvm_set_verbosity=<*comp*>,<*id*>,<verbosity>,time,<time>

```
+uvm_set_verbosity=uvm_test_top.env.agt.*,_ALL_,UVM_FULL,build
+uvm_set_verbosity=uvm_test_top.env.agt.*,_ALL_,UVM_FULL,time,800
```

- **Control report message action (like set_report_*_action)**

+uvm_set_action=<*comp*>,<*id*>,<severity>,<action>

```
+uvm_set_action=uvm_test_top.env.*,_ALL_,UVM_ERROR,UVM_NO_ACTION
```

- **Control severity (like set_report_*_severity_override)**

+uvm_set_severity=<*comp*>,<*id*>,<current severity>,<new severity>

```
+uvm_set_severity=uvm_test_top.*,BAD_CRC,UVM_ERROR,UVM_WARNING
```

Test For Understanding

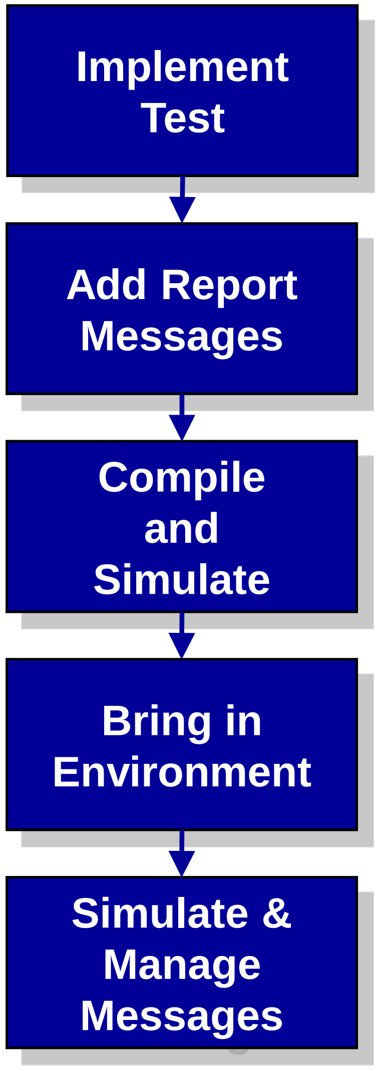
- **How long does the following run for in simulation?**

```
program automatic test;
  import uvm_pkg::*;

  class hello_world extends uvm_test;

    `uvm_component_utils(hello_world)

    function new(string name, uvm_component parent);
      super.new(name, parent);
    endfunction

    virtual task run_phase(uvm_phase phase);
      #50ns;
      `uvm_info("TEST", "Hello World!", UVM_MEDIUM);
    endtask

  endclass

  initial
    run_test();

endprogram
```

Lab 1 Introduction

45 min

Implement test, environment and report messages



Key Structural Concept: <u>Parent-Child</u>

- **Parent-child relationships**
  - Set up at component creation
  - Establishes component phasing execution order
  - Establishes component hierarchical path for component configuration and factory override
  - <u>Composition hierarchy – Not OOP hierarchy</u>
- **Phase execution order**
  - **Each component follows the <u>same</u> sequence of phase execution**

```
build
connect
end_of_elaboration
start_of_simulation
run*
extract
check
report
final
```

- **Search path allow tests to:**
  - **Configure specified components**
  - **Override specified components in environment**

Key Component Concepts: Logical Hierarchy

```
class test_base extends uvm_test;
  environment env;
  `uvm_component_utils(test_base)
  function new(string name, uvm_component parent);
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = environment::type_id::create("env", this);
  endfunction
```

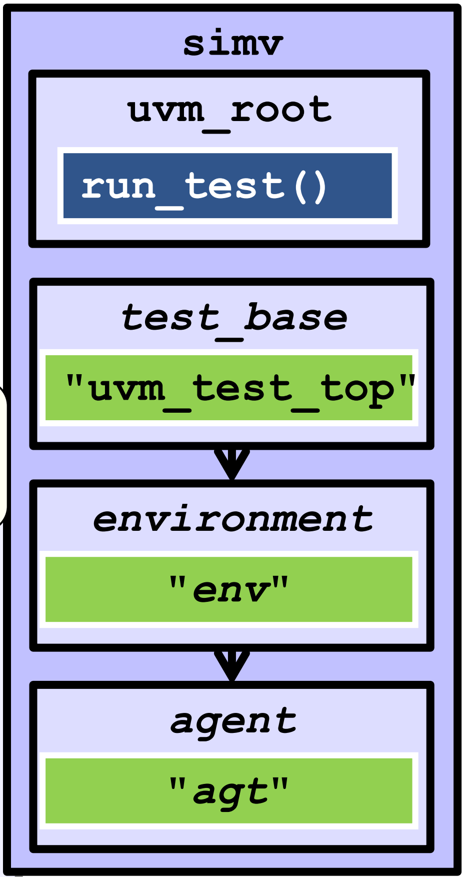
```
class environment extends uvm_env;
  agent agt;
  `uvm_component_utils(environment)
  function new(string name, uvm_component parent);
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agt = agent::type_id::create("agt", this);
  endfunction
```

```
class agent extends uvm_agent;
  `uvm_component_utils(agent)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
  virtual task class_task(…);
endclass
```

Establish parent-child relationship at creation

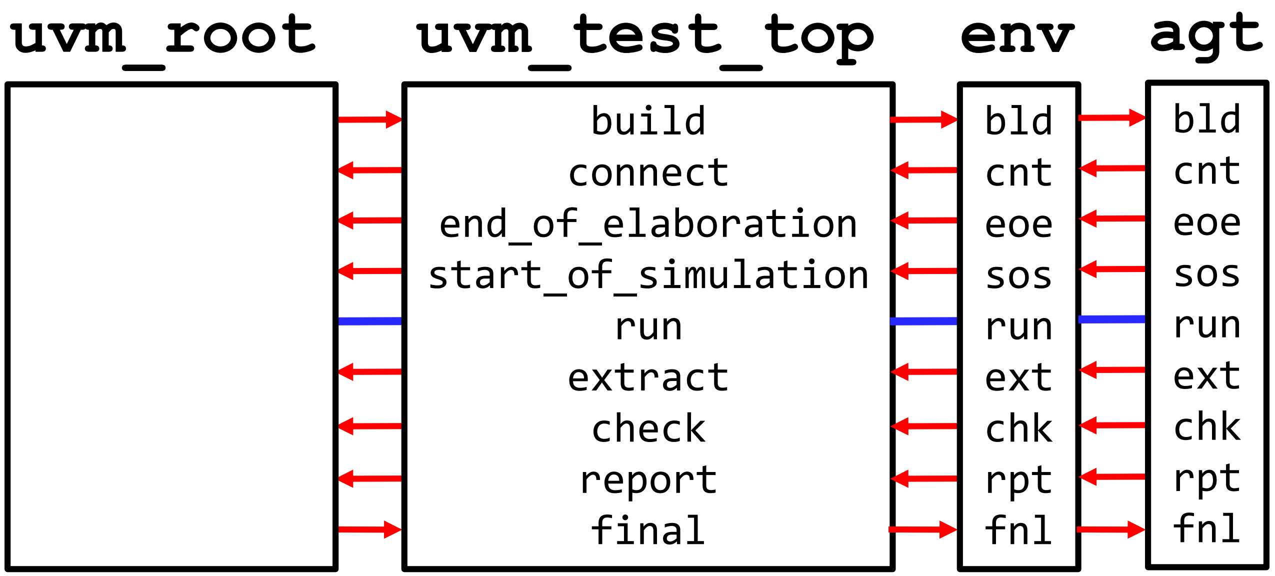
Set parent

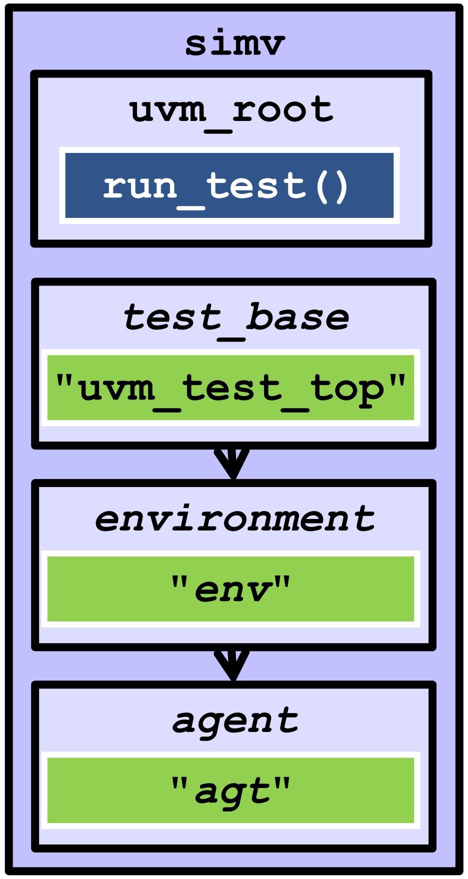
Parent handle



Key Component Concepts: Phase

- Parent-child relationship dictates phase execution order
  - Functions are executed bottom-up
    - Except for build and final phases which are executed top-down
  - Tasks are forked into concurrent executing threads





Key Component Concepts: Override

Simulate with:
```
simv +UVM_TESTNAME=test_new
```

```
class test_new extends test_base;  ...
  `uvm_component_utils(test_new)
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    set_inst_override_by_type("env.agt", agent::get_type(),
                                          new_agt::get_type());
  endfunction
endclass
```

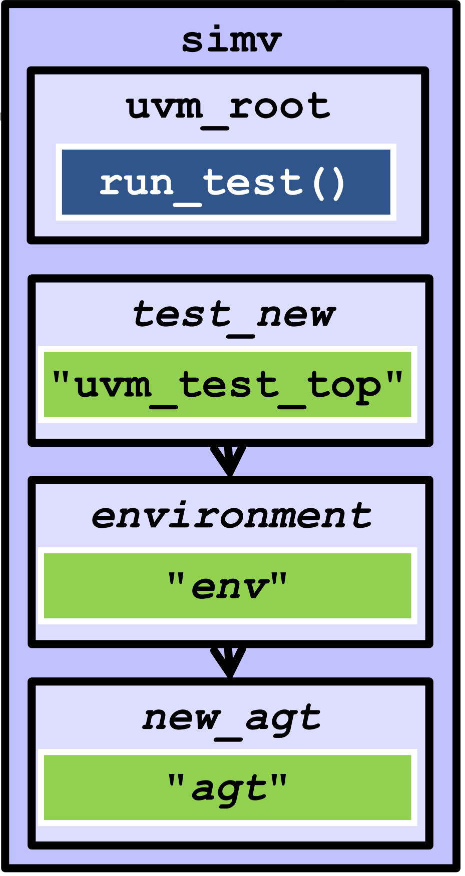
Use component hierarchical path to do component overrides

```
class environment extends uvm_env; ...
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agt = agent::type_id::create("agt", this);
  endfunction
endclass
```

create() used to build component

```
class new_agt extends agent;
  `uvm_component_utils(new_agt)
  function new(string name, uvm_component parent);
  virtual task class_task(…);
    // modified component functionality
  endtask
endclass
```

Modify operation



Unit Objectives Review

Having completed this unit, you should be able to:

- Describe the process of reaching verification goals
- Describe the UVM testbench architecture
- Describe the different components of a UVM testbench
- Bring different components together to create a UVM environment

Major Changes in UVM-1.2

Migration from UVM1.1 to UVM1.2

- **Non-Backward compatible changes outlined in migration document**
- **Migration script:**
  - `./bin/uvm11-to-uvm12.pl`
- **Release note:**
  - `release-notes.txt` lists mantis items and backward compatibility

VCS Support for UVM

Compiling UVM with VCS

- **Single compile flow**

```
% vcs -sverilog file.sv … -ntb_opts uvm-1.2 …
```

- **UUM compile flow**

Compile UVM library first with no source files

```
% vlogan -sverilog -work work1 -ntb_opts uvm-1.2
% vlogan -sverilog -work work1 -ntb_opts uvm-1.2 file1.v
% vlogan -sverilog -work work2 -ntb_opts uvm-1.2 file2.v
% vhdlan -work work3 file3.vhd
% vcs top … -ntb_opts uvm-1.2
```

When using the VPI-based backdoor access mechanism included in the UVM library, the "+acc" and "+vpi" command-line options must also be used.

UVM Reporter Control

Embed UVM Reporter Control in Test

```
class report_control extends test_base;
  UVM_FILE log_file = $fopen("log_file", "w") // log file
  // uvm_component_utils and constructor not shown
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_root::get().set_* (see method below)
  endfunction
  function void final_phase(uvm_phase phase);
    super.final_phase(phase);
    $fclose(log_file);
  endfunction
endclass
```

```
function void set_report_verbosity_level(int verbosity_level);
function void set_report_id_verbosity(string id, int verbosity);
function void set_report_severity_id_verbosity(uvm_severity severity, string id, int verbosity);
function void set_report_severity_action(uvm_severity severity, uvm_action action);
function void set_report_id_action(string id, uvm_action action);
function void set_report_severity_id_action(uvm_severity severity,string id, uvm_action action);
function void set_report_default_file(UVM_FILE file);
function void set_report_id_file(string id, UVM_FILE file);
function void set_report_severity_file(uvm_severity severity, UVM_FILE file);
function void set_report_severity_id_file(uvm_severity severity, string id, UVM_FILE file);
function void set_report_severity_override(uvm_severity cur_severity,uvm_severity new_severity);
function void set_report_severity_id_override(uvm_severity cur_severity, string id,
                                              uvm_severity new_severity);
```