The UVM skeleton

This page answers: *what are the pieces of a UVM testbench, and in what order does UVM run them?*

In Lab 01 you read a whole UVM testbench, the ALU's, from its tests down to its driver, monitor and scoreboard, and you added the flag check to its scoreboard. Labs 02 and 03 used only one piece of UVM, the test. It extended `dd105_base_test` so that `make` could pick it with `+UVM_TESTNAME` and the graders could read UVM's report summary at the end of the log. Inside its `run_phase` it fetched the virtual interfaces from `config_db`, made the environment with `new()` and ran it like any other class. In Lab 04 the router testbench becomes UVM classes too, and Part 5 of that lab counts what this costs in lines against your Lab 03 testbench. Much of what it counts is the structure that this page describes. The drivers, the monitors and the scoreboard become components in a tree, UVM builds the tree and connects it in a fixed order of phases, and objections decide when the run ends.

The example for this page, `guide-examples/the-uvm-skeleton/`, has no design at all. It has a test, an environment, two agents with one child each, and a component called `mon` that loops forever the way a monitor does. Each of them prints a line in every function phase, and the test, the two children and `mon` also print in the run phase, so you can watch the order in which UVM calls them.

Objects and components

Every UVM class in a testbench extends either `uvm_object` or `uvm_component`. (In the library, `uvm_component` is itself built on `uvm_object`, but a testbench uses the two in different ways.)

A **`uvm_object`** is data: a packet, an APB transfer, a set of knobs for an agent. You make one whenever you need it, pass it around and drop it, and many of them come and go in a run. Lab 04's `router_pkt`, `stream_beat`, `apb_item` and `out_cfg` are objects.

A **`uvm_component`** is one of the testbench's fixed parts, such as a driver, a monitor, a scoreboard or the test. It's made once, at the start, and lasts for the whole run. Every component has a name and a parent, given to its constructor as `new(name, parent)`, and the parents link the components into a tree. Lab 04's tree looks like this:

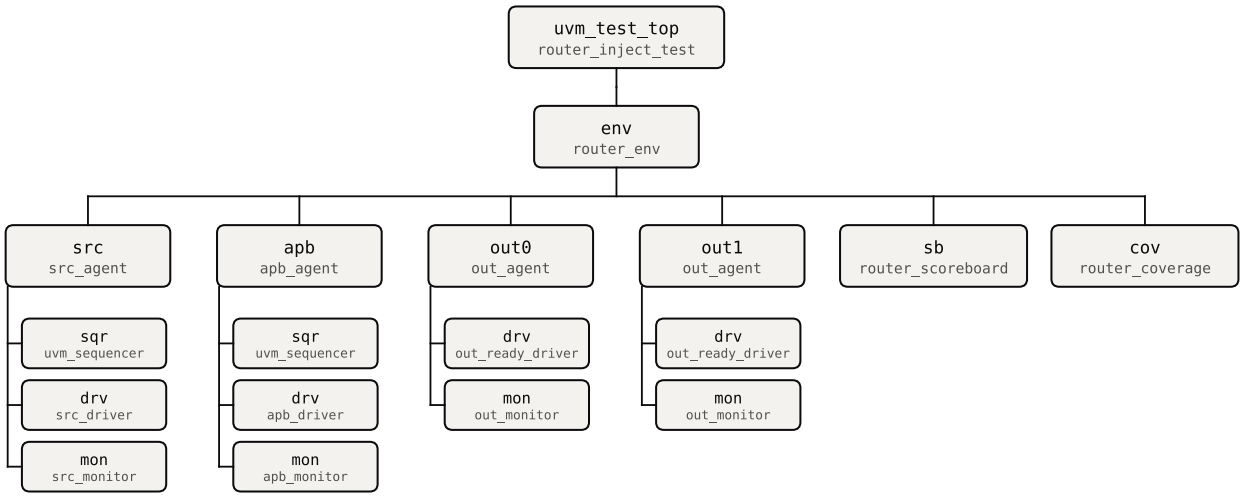


Figure 1: The components of Lab 04's testbench. Each box shows an instance name and, under it, the class.

The components that `router_base_test`, `router_env` and the agents build in Lab 04, once you've added `out1` in Part 2. When a component reports a message, UVM prints the component's name as a path down this tree, such as `uvm_test_top.env.out1.mon` for the monitor on output 1.

The test is at the top of everything the testbench builds, and UVM always names it `uvm_test_top`. Above it sits UVM's own root, `uvm_root`, which you'll see in a printout of the tree later on this page. The test makes the

environment, `env`, and the environment makes the agents, the scoreboard and the coverage collector. An agent holds the components for one interface: a driver that drives its pins, a monitor that watches them, and, if the agent sends items, a sequencer that hands the driver its items.

The library has a class for each of these jobs: `uvm_test`, `uvm_env`, `uvm_agent`, `uvm_driver`, `uvm_monitor`, `uvm_sequencer`, `uvm_scoreboard` and `uvm_subscriber`. Most of them add little or nothing to `uvm_component`. `uvm_driver` adds the port that it gets items through, and `uvm_subscriber` adds the port that it receives transactions on; the next two pages describe both ports. `uvm_monitor` and `uvm_scoreboard` add nothing, and each is a plain component whose name says what it's for. In Lab 04′s output agent, `out_ready_driver` takes no items, so it extends `uvm_component` rather than `uvm_driver`.

Every UVM class also has a registration line, such as `` `uvm_object_utils(router_pkt) `` or `` `uvm_component_utils(router_env) ``, which tells UVM's factory that the class exists, and the components are made with `router_env::type_id::create("env", this)` instead of `new()`. The factory page explains what the factory does with them. For now, the registration is the reason that `+UVM_TESTNAME` can name a test that no other code in the testbench refers to.

run_test() and +UVM_TESTNAME

Every lab's top module makes the clock, the design and the interfaces, puts each interface into `config_db` for the components that will need it (in Labs 02 and 03, the test), and ends by calling `run_test()`. `run_test()` reads `+UVM_TESTNAME`, asks the factory for a component of that class, names it `uvm_test_top`, and starts the phases. That's how one build serves every test in a lab.

If the name doesn't match a registered class, no test is built. `make run UVM_TESTNAME=sk_phase_tset`, with two letters of the name swapped, prints these two lines among its other output:

```
UVM_WARNING @ 0: reporter [BDTYP] Cannot create a component of type 'sk_phase_tset' because
it is not registered with the factory.
UVM_FATAL @ 0: reporter [INVTST] Requested test from command line +UVM_TESTNAME=sk_phase_tset
not found.
```

and the simulator exits with status 0, which the last section of this page comes back to.

Phases

Instead of running each component as soon as it's made, UVM calls a fixed list of methods, the **phases**, on every component in the tree, and finishes each phase everywhere before it starts the next. A component overrides the phases it needs. The ones you'll write are:

Phase	Kind	Order in the tree	What goes in it
`build_phase`	function	top down	make the children; read configuration
`connect_phase`	function	bottom up	connect ports between children
`end_of_elaboration_phase`	function	bottom up	print or check the finished tree
`start_of_simulation_phase`	function	bottom up	last settings before time moves

Phase	Kind	Order in the tree	What goes in it
`run_phase`	task	all at once	everything that takes simulated time
`extract_phase`, `check_phase`, `report_phase`	function	bottom up	collect results, look for leftovers, print summaries
`final_phase`	function	top down	the very end

The function phases take no simulated time. The run phase is a task, and UVM starts every component's `run_phase` at the same moment, each in its own thread. Here is the start of `sk_phase_test`'s output:

```
0    ns  build_phase              uvm_test_top
0    ns  build_phase              uvm_test_top.env
0    ns  build_phase              uvm_test_top.env.a0
0    ns  build_phase              uvm_test_top.env.a0.child
0    ns  build_phase              uvm_test_top.env.a1
0    ns  build_phase              uvm_test_top.env.a1.child
0    ns  build_phase              uvm_test_top.env.mon
0    ns  connect_phase            uvm_test_top.env.a0.child
0    ns  connect_phase            uvm_test_top.env.a0
0    ns  connect_phase            uvm_test_top.env.a1.child
0    ns  connect_phase            uvm_test_top.env.a1
0    ns  connect_phase            uvm_test_top.env.mon
0    ns  connect_phase            uvm_test_top.env
0    ns  connect_phase            uvm_test_top
```

The build phase has to go from the top down, because a component's children don't exist until its own `build_phase` makes them, so UVM can't call `env`'s `build_phase` until the test's `build_phase` has created `env`. Within each parent, the course's UVM library visited the children in name order (`a0`, `a1`, `mon`), although `env` made them in the order `mon`, `a1`, `a0`.

The connect phase goes from the bottom up, so by the time `env` connects its agents' monitors to the scoreboard, every agent has already connected its own pieces. A component connects ports in `connect_phase` and never in `build_phase`, because the component on the other end may not exist yet during `build_phase`. UVM finishes `build_phase` in the whole tree before any `connect_phase` starts, so every component exists by the time it connects.

In `end_of_elaboration_phase` the test asks UVM to print the tree it has built:

```
Name            Type           Size  Value
-----------------------------------------
<unnamed>       uvm_root       -     @64
  uvm_test_top  sk_phase_test  -     @150
    env         sk_env         -     @220
      a0        sk_agent       -     @244
        child   sk_leaf        -     @252
      a1        sk_agent       -     @236
        child   sk_leaf        -     @260
      mon       sk_watcher     -     @228
```

Objections

A driver loops forever asking for items, and a monitor loops forever watching pins, so the run phase can't wait for every `run_phase` task to return. Instead, UVM keeps the run phase going while any component holds an **objection**. A component that is in the middle of something calls `phase.raise_objection(this)`, and calls `phase.drop_objection(this)` when it's done, and the run phase ends once every objection has been dropped. A component can also set a **drain time**, which keeps the phase open for that long after the last objection drops, so that anything still in flight can arrive.

In `sk_phase_test` the test holds an objection for 100 ns and sets a drain time of 20 ns, `env.a1.child` holds one for 150 ns, and `env.mon` loops forever, printing every 40 ns:

```
0    ns  run_phase starts           uvm_test_top.env.a0.child
0    ns  run_phase starts           uvm_test_top.env.a1.child
0    ns  run_phase: still looping   uvm_test_top.env.mon
0    ns  run_phase starts           uvm_test_top
40   ns  run_phase: still looping   uvm_test_top.env.mon
80   ns  run_phase: still looping   uvm_test_top.env.mon
100  ns  drops after 100 ns         uvm_test_top
120  ns  run_phase: still looping   uvm_test_top.env.mon
150  ns  drops after 150 ns         uvm_test_top.env.a1.child
160  ns  run_phase: still looping   uvm_test_top.env.mon
170  ns  extract_phase              uvm_test_top.env.a0.child
```

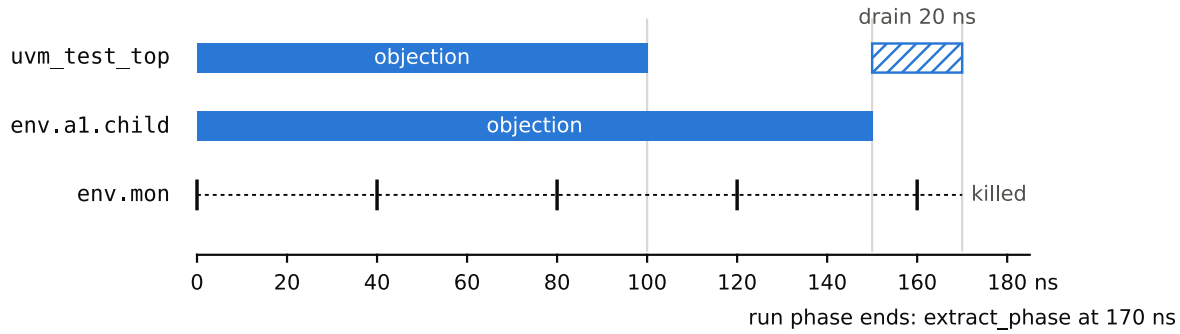


Figure 2: Objections in sk_phase_test: the test's from 0 to 100 ns, a1.child's from 0 to 150 ns, then 20 ns of drain time; the run phase ends at 170 ns.

The run phase of `sk_phase_test`. When the test dropped its objection at 100 ns, `a1.child` still held one. The phase ended 20 ns after that one dropped, at 170 ns, and UVM killed `mon`'s loop, which had been running the whole time without holding the phase open.

`sk_noobj_test` builds the same environment and raises no objection at all. Every `run_phase` still starts at 0 ns, and the phase ends in the same time step:

```
0    ns  run_phase starts           uvm_test_top
0    ns  extract_phase              uvm_test_top.env.a0.child
```

A new UVM test can pass this way without testing anything. If it starts its sequence in `run_phase` and forgets to raise an objection, the run ends before the first item reaches the driver. Lab 04's `router_base_test` raises its objection at the start of `run_phase`, runs the test's `body()` and drops the objection when `body()` returns. The

tests built on it override `body()` and leave `run_phase` alone, so none of them has to raise an objection itself. `router_base_test` also runs a watchdog beside `body()`, because a test that holds its objection forever never ends on its own either. If the test is still running after `+MAX_CYCLES` clock cycles, the watchdog reports a `UVM_ERROR`, and `run_phase` stops `body()` and drops the objection, so the run ends.

After the run phase

Checks that need the whole run go in the three function phases that follow the run phase, `extract_phase`, `check_phase` and `report_phase`. Lab 04's scoreboard uses `check_phase` to report, for each output, how many beats it expected and never saw, and the first of them. It uses `report_phase` to print the `[SB]` lines you'll read in every run. After `final_phase`, UVM prints its report summary, which counts the run's messages by severity and by ID.

Verilator ends a UVM run with exit status 0 even when the summary counts errors, so a script that trusts the exit status would call a failing test a pass. `dd105_base_test`, which every test in the course extends, fixes that in its `final_phase`, and in `pre_abort` when a `UVM_FATAL` ends the run early. If UVM counted no error or fatal, it prints `DD105: TEST PASSED`. Otherwise it prints the report summary itself, then `DD105: TEST FAILED`, and calls `$fatal`. `sk_error_test` reports one `UVM_ERROR` at 50 ns:

```
UVM_ERROR guide-examples/the-uvm-skeleton/sk_pkg.sv(186) @ 50000: uvm_test_top [CHECK] a check
failed at 50 ns
```

and, after the report summary, prints

```
DD105: TEST FAILED (1 UVM_ERROR, 0 UVM_FATAL)
[150000] %Fatal: dd105_pkg.sv:89: Assertion failed in dd105_pkg.dd105_base_test.m_end_of_test:
DD105: UVM counted errors, so the exit status is nonzero
```

and the simulator exits with status 1. The `UVM_ERROR` line has the same shape as the messages that the labs' testbenches report: the severity, the file and line that reported it, `@` and the time, the full name of the component, the message's ID in brackets, and the text. Some of the UVM library's own messages leave out the file and line, as the `[BDTYP]` and `[INVTST]` lines earlier on this page do. The time is in picoseconds, the simulator's precision, so `@ 50000` is 50 ns.

The base test can't help with the misspelt test name. No test was built, so the base test's `final_phase` and `pre_abort` never ran, and that run exited with status 0. That's why the course's `make run` reads the log as well as the exit status. It reported that run as `BROKEN: UVM_FATAL [INVTST]`.

Check yourself

1. Why can't a scoreboard connect a monitor's analysis port to itself in its `build_phase`?

   Answer: A1, in Answers at the end.

2. A test's `run_phase` starts a sequence of 200 packets on the input agent and doesn't raise an objection. What happens, and what would the scoreboard's `check_phase` see?

   Answer: A2, in Answers at the end.

3. A run's log ends with a `UVM_FATAL`, and the simulator exited with status 0. Was the test built?

   Answer: A3, in Answers at the end.

Further reading

- ChipVerify's Build Your First UVM Testbench walkthrough builds one testbench for a small register file over 17 pages, one component per page, and assembles it on the last, Full UVM Testbench. As you read it, note that

its testbench top module (UVM Testbench Top) never drives reset, which only the last page fixes. Its coverage collector (UVM Subscriber) has coverpoints that read a class member, so Verilator 5.050 drops the whole covergroup with a `COVERIGN` warning (see Functional coverage you can read).
- UVM Phases and UVM Objection cover every phase, including the twelve run-time sub-phases that this course doesn't use, and objections raised from sequences.
- UVM Component and UVM Object.

Next: Sequences, sequencers and drivers.

Answers

A1. Check yourself

The monitor may not exist yet. It's made in its agent's `build_phase`, and the agent's `build_phase` can run after the scoreboard's. In Lab 04, `env`'s children are built in name order, as `a0`, `a1` and `mon` were above, so `sb` is built before `src`, and `src.mon` doesn't exist yet when `sb`'s `build_phase` runs. By `connect_phase` every component has been built.

A2. Check yourself

The run phase ends at time 0, as `sk_noobj_test`'s did, before any packet reaches the pins. The scoreboard expected nothing and saw nothing, so `check_phase` finds no leftover beats. A scoreboard that also counts what it checked can catch this, as the one in Monitors, analysis ports and scoreboards does by reporting an error when it checked nothing.

A3. Check yourself

No. Every test in the course extends `dd105_base_test`, which turns a `UVM_FATAL` into a nonzero exit status in `pre_abort`, so a fatal with status 0 means that no test was built, as happens with a misspelt `UVM_TESTNAME`. `make tests` lists the names the lab knows.