The factory and config_db

This page answers: *how does a test change what the environment builds, and how it's set up, without editing the environment?*

UVM gives a test two ways to do it, the factory and `config_db`, and Lab 04 has you use both. In Part 2 you make a second copy of the output agent, and `config_db` tells each copy which output it's on. In Part 3 you write a stall test that changes a knob that the output drivers read, one output at a time. In Part 4 you write a test that sends packets to disabled routes by telling the factory to build a different packet class, and the sequence that makes the packets stays as it is. Part 5 then asks what the extra lines of the UVM testbench bought you, and these two mechanisms are most of the answer.

The example in `guide-examples/the-factory-and-config-db/` compiles the files of the monitors page example unchanged, so it has that example's environment. The environment has an agent that drives SimProc's ALU, and the agent's monitor feeds a scoreboard, a coverage collector and a history. Each of the example's four tests changes what that environment does without editing a line of it.

The factory

Every UVM class in the labs and in this example registers itself with the factory through a utils macro, `` `uvm_object_utils(alu_item) `` in an object and `` `uvm_component_utils(alu_env) `` in a component. The labs make almost everything by asking the factory for it, as the example's sequence does for each vector:

```
req = alu_item::type_id::create("req");
```

With no override set, that returns an `alu_item`, as `new()` would. But a test can tell the factory to return a subclass of `alu_item` to all the code that asks for an `alu_item`, and that code doesn't have to change.

`fac_equal_test` uses an override to give every vector equal operands. The override substitutes `alu_equal_item`, a subclass of `alu_item` that replaces one of `alu_item`'s two constraints:

```
class alu_equal_item extends alu_item;
  `uvm_object_utils(alu_equal_item)
  constraint c_same { same == 1; same -> B == A; }
  ...
endclass
```

A constraint with the same name as one in the parent class replaces it, so `alu_equal_item` has the parent's `c_op` and this `c_same`. (Verilator 5.050 does this as the standard says, and Lab 04's `router_noroute_pkt` replaces `c_dst` the same way.) The test sets the override in its `build_phase`, before `super.build_phase()` builds the environment:

```
function void build_phase(uvm_phase phase);
  alu_item::type_id::set_type_override(alu_equal_item::get_type());
  super.build_phase(phase);
endfunction
```

and later, in its `end_of_elaboration_phase`, it prints the factory's overrides with `uvm_factory::get().print(0)`:

```
#### Factory Configuration (*)
No instance overrides are registered with this factory
Type Overrides:
  Requested Type  Override Type
```

```
  --------------  --------------
  alu_item        alu_equal_item
```

The sequence still calls `alu_item::type_id::create("req")` and still randomizes what it gets, so the one override line in the test gives every one of its 200 vectors `A == B`. Run against Lab 01's three ALU bugs, the test catches none of them:

```
0 of 3 caught.
Still hiding from this test:
  A01_sub_operands_swapped
     hides behind: SUB rX, rX, the idiom for clearing a register
  A02_nand_is_nor
     hides behind: NAND rX, rX, the only way to write NOT
  A03_zero_flag_low_nibble
     hides behind: every check that compares results and not flags
```

That's DD104's lesson about equal operands again. With `A == B`, `A - B` and `B - A` are both 0, and `~(A & B)` and `~(A | B)` are both `~A`, so A01 and A02 give the right answers. The coverage collector sorts the vectors by operation and by whether the two operands are equal, and in this run it reported only 65% of its bins hit, because no vector had unequal operands.

An override changes every object the factory makes of that type, and that can include objects you meant to leave alone. Lab 04's `close_holes()` needs two packets exactly as `router_pkt` defines them, so it makes them with `new()`, which doesn't go through the factory. When we made them through the factory, `router_inject_test`'s override turned them into `router_noroute_pkt`s too, and `randomize()` failed on 4 of 5 seeds (Lab 04, Part 4, step 3).

An override can also be set from the command line instead of in a test class. UVM reads `+uvm_set_type_override=alu_item,alu_equal_item` as the same override, so it works on `mon_rand_test`, which knows nothing about `alu_equal_item`. On the ALU with bug A01, the run prints:

```
UVM_INFO @ 0: reporter [UVM_CMDLINE_PROC] Applying type override from the command line:
+uvm_set_type_override=alu_item,alu_equal_item
```

and ends with `MISSED: 0 UVM_ERROR`. The same test without the plusarg catches A01 with 33 errors. A plusarg needs no rebuild, and Lab 04 uses one the same way on the router in Part 4, step 6.

The factory can also override a single instance, named by its path in the tree, instead of every object of a type. The course's labs use only type overrides. So does `mon_alias_test` on the monitors page, which replaced the monitor class in an environment that has only one monitor, so the type override reached only that monitor.

config_db

`config_db` is a table of values that one part of the testbench puts in and a component takes out. The code that sets a value names the components that may read it, by a path in the component tree. When the first argument is `this`, the path is relative to the component that makes the call:

```
uvm_config_db#(bit)::set(this, "env.sb", "check_flags", 1);     // in the test
void'(uvm_config_db#(bit)::get(this, "", "check_flags", check_flags));  // in the scoreboard
```

The test's full name is `uvm_test_top`, so its `set` is for `uvm_test_top.env.sb`. The scoreboard's `get` has an empty path, so it looks for a value set for the scoreboard's own full name, which is also `uvm_test_top.env.sb`. A path

can contain wildcards. The top module isn't a component, so it sets each virtual interface with a `null` context and a full path, such as `"uvm_test_top.env.agent.*"`, and the `*` lets every component inside the agent find it.

The example's scoreboard checks only `ALUout` unless `check_flags` is set, and then it checks N and Z as well. `fac_flags_test` sets it, and catches all three bugs:

```
      ok  the golden design                       0 UVM_ERROR, 0 UVM_FATAL       0.2s
--------------------------------------------------------------------------------
  caught  tier 1  A01_sub_operands_swapped        33 UVM_ERROR                   0.2s
  caught  tier 1  A02_nand_is_nor                 26 UVM_ERROR                   0.2s
  caught  tier 1  A03_zero_flag_low_nibble        8 UVM_ERROR                    0.2s
--------------------------------------------------------------------------------
 3 of 3 caught.
```

`fac_typo_test` makes the same `set` for `"env.scoreboard"`, and the environment has no component by that name. The run ends with 0 warnings and 0 errors. The scoreboard's `get` finds nothing, so `check_flags` keeps its default of 0, and the test misses A03, as `mon_rand_test` does. To see what went wrong, run with `+UVM_CONFIG_DB_TRACE`, which logs every `set` and `get`:

```
UVM_INFO  third_party/uvm/src/base/uvm_config_db_implementation.svh(485)  @  0:  reporter
[CFGDB/SET]  Configuration  scope='uvm_test_top.env.scoreboard'  name='check_flags'  (type
PARAMTYPEDTYPE 'T') set accessor=uvm_test_top = (PARAMTYPEDTYPE 'T') 1
UVM_INFO third_party/uvm/src/base/uvm_config_db_implementation.svh(485) @ 0: reporter [CFGDB/
GET] Configuration scope='uvm_test_top.env.sb' name='check_flags' (type PARAMTYPEDTYPE 'T')
read accessor=uvm_test_top.env.sb = null (failed lookup)
```

The `set` was for `uvm_test_top.env.scoreboard`, and the `get` looked in `uvm_test_top.env.sb`. A knob with a sensible default can be read the way `check_flags` is, with `void'(...)`, but a component that can't work without a value should stop the run when the `get` fails. Every driver and monitor in Lab 04 stops the run with `` `uvm_fatal("NOVIF", ...) `` if it gets no virtual interface. The output agent's driver and monitor also need an `out_cfg`, and they stop with `` `uvm_fatal("NOCFG", ...) `` if they don't get one.

Two components can set the same value for the same component. `fac_prec_test` sets `check_flags` to 1 for the scoreboard, and it also overrides the environment with a subclass, `fac_env`, that sets `check_flags` to 0 for the same scoreboard. The environment is built after the test, so its `set` comes second:

```
[TEST] uvm_test_top set check_flags = 1 for "env.sb"
[ENV] uvm_test_top.env set check_flags = 0 for "sb"
```

The scoreboard still checked the flags, and reported `200 vectors checked with N and Z: 0 wrong`. During the build phase, a `set` made higher in the component tree wins over one made lower down, even when the lower one comes later, so the test's 1 beat the environment's 0. That rule lets an environment give its components defaults that any test can override.

Objects by handle

`config_db` can hold an object as well as a number. For an object it stores the handle, so the component that gets the value and the one that set it share one object. Lab 04 uses that for the output agents. `router_base_test` makes an `out_cfg` for each output, sets output 0's for `"env.out0*"` and output 1's for `"env.out1*"`, and keeps the two handles. Both output agents are the same class. Each one gets its virtual interface and its `out_cfg` from `config_db`, because its name matches the path that they were set for. Nothing else differs between the two.

Because the test kept the handles, your Part 3 stall test can change `ocfg[p].ready_pct` in the middle of a run. Each `out_ready_driver` reads `ready_pct` again every cycle, so it follows the change at the next clock edge.

Write the type of a virtual interface out in full, as in `virtual stream_src_if`, and don't reach it through a `typedef`. On Verilator 5.050, a class in a package that reached an interface through a `typedef` of the virtual interface read 0 from every clocking-block input, with no warning, so the labs' packages don't use one. `labs/ref/README.md` has the probe that showed it.

Check yourself

1. `fac_equal_test` and `mon_rand_test` both extend `mon_base_test`, and `fac_equal_test` adds a type override. It catches none of the three bugs, and `mon_rand_test` catches two of them. Did the override change the sequence?

   Answer: A1, in Answers at the end.

2. A test sets a knob for its scoreboard, and the run is no different from a run without the `set`. Nothing printed an error. What do you look at?

   Answer: A2, in Answers at the end.

3. Lab 04's `close_holes()` makes its two random packets with `new()`. What happened in `router_inject_test` when they were made through the factory instead?

   Answer: A3, in Answers at the end.

Further reading

- ChipVerify's UVM Factory Override covers type and instance overrides, and the difference in argument order between the factory's methods and the component's. Its last example calls methods on a `factory` variable that it never declares, as if UVM had a global by that name. The course's UVM library (1800.2-2020.3.2) has none, so use `uvm_factory::get()` there.
- UVM config database, for the API, the wildcards and the precedence rules.
- uvm_config_db Examples works through four cases with `+UVM_CONFIG_DB_TRACE` logs, and has a table of nine `set()` calls and the components that each one reaches. Its third case depends on a `set()` that the page never shows.
- Configure Components, on configuration objects like Lab 04's `out_cfg`.
- Passing the Virtual Interface, on how the top module's `set` reaches the driver and the monitor.

Next: Lab 04: The same testbench in UVM.

Answers

A1. Check yourself

No. The sequence's code is the same, and it still asks the factory for an `alu_item` and randomizes it. The factory hands back an `alu_equal_item` instead, and that class's `c_same` decides the operands. Every item the sequence made had `A == B`, which hides A01 and A02, and the scoreboard didn't check the flags, so it missed A03 as it does in `mon_rand_test`.

A2. Check yourself

Run it with `+UVM_CONFIG_DB_TRACE` and compare the scope in the `CFGDB/SET` line with the scope in the scoreboard's `CFGDB/GET` line. If the `get` says `failed lookup`, the `set` didn't name the component that reads the value, as in `fac_typo_test`. Without the trace, the run's log says nothing about it.

A3. Check yourself

The test's override reached them too, so both were `router_noroute_pkt`s. `close_holes()` switches `c_dst` off for the packet to DST 1 through a `router_pkt` handle, and that left the subclass's own `c_dst` on. Whenever the packet's

`pre_randomize()` decided against a disabled route, that `c_dst` required an enabled one. DST 1 was disabled, so `randomize()` failed, which happened on 4 of 5 seeds (Lab 04, Part 4, step 3).