The factory and ConfigDB

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

In pyuvm a test uses the factory to change what the environment builds, and `ConfigDB` to change how it's set up, and Lab 04 has you use both. In Part 2 you make a second output agent from the same class, and `ConfigDB` tells each agent which output it's on. The stall test you write in Part 3 stalls one output at a time by changing a knob that the output drivers read. Part 4's test sends packets to disabled routes by telling the factory to make a subclass of the packet class instead, and the sequence that makes the packets doesn't change. Part 5 then asks what the extra lines of the pyuvm testbench bought you, and its answer is these three changes and how few lines each one took.

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

The factory

As soon as Python defines a class that derives from a pyuvm class, pyuvm registers it with the factory under the class's name. The labs make most of their components and many of their sequence items by asking the factory for them, as the example's sequence does for each vector:

```
req = AluItem.create("req")
```

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

`FacEqualTest` uses an override to give every vector equal operands. The override puts `AluEqualItem` in place of `AluItem`, and `AluEqualItem` is a subclass of `AluItem` that replaces `AluItem`'s constraint `c_same`:

```
@vsc.randobj
class AluEqualItem(AluItem):
    @vsc.constraint
    def c_same(self):
        self.same == 1
        with vsc.implies(self.same == 1):
            self.B == self.A
```

`AluItem`'s `c_same` makes `same` 1 in one vector in eight, and makes `B` equal to `A` when it is. `AluEqualItem`'s makes `same` 1 in every vector. A pyvsc constraint is a method, and a Python method with the same name as one in the parent class replaces it, so `AluEqualItem` has this `c_same` and not its parent's. Lab 04's `NoroutePkt` replaces `RouterPkt`'s `c_dst` the same way. The test sets the override in its `build_phase`, before `super().build_phase()` builds the environment:

```
def build_phase(self):
    uvm_factory().set_type_override_by_type(AluItem, AluEqualItem)
    super().build_phase()
```

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

```
--- overrides ---

Overrides:
AluItem                  : Type Override: AluEqualItem ||
```

The sequence still calls `AluItem.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`. The history's first three vectors were `OR A=29 B=29`, `NAND A=e8 B=e8` and `ADD A=5d B=5d`. 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
```

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. DD104 showed the same two bugs hiding behind equal operands. The scoreboard doesn't check the flags, so A03 stays hidden too. The coverage collector reported `[COV] alu: 13 of 20 bins hit`, because no vector had unequal operands, so the `differ` bin and the six cross bins that include it were never hit.

The constructor and the factory

An override changes only the objects that the factory makes. `FacCtorTest` sets the same override and then makes one item each way:

```
[CTOR] AluItem.create('x') made AluEqualItem
[CTOR] AluItem('x') made AluItem
```

Lab 04's `close_holes()` picks the DST of every packet it sends, so it makes its packets as plain `RouterPkt`s, with the constructor, in `new_packet()` and in `RouterPkt.make()`, where a test's override can't reach them. When we made its two random packets with `create()` instead, `RouterInjectTest`'s override turned them into `NoroutePkt`s too, and the test ended `BROKEN: SolveFailure` on 2 of seeds 1 to 20 (Lab 04, Part 4, step 3).

Components go through the factory too, when they're made with `create()`. On the monitors page, `MonAliasTest` swapped in a different monitor with an override of `AluMonitor`, and `MonConnectTest` swapped in a different environment with an override of `MonEnv`. Both overrides took effect because `MonAgent` makes its monitor with `AluMonitor.create()` and the base test makes the environment with `MonEnv.create()`. A component made with its constructor can't be swapped that way. In Lab 04, the input and APB agents make their sequencers like that, with `uvm_sequencer("sqr", self)`.

pyuvm clears every override at the start of each test, so one test's overrides never reach the next. An override can also apply to a single component, named by its path in the tree, with `uvm_factory().set_inst_override_by_type(original, override, path)`, but the course's labs use only type overrides.

An override from the command line

pyuvm 4.0.1 reads no plusargs of its own, so it can't take an override from the command line the way SystemVerilog UVM's `+uvm_set_type_override` does. Lab 04's base test reads a

plusarg that the course defines, `+OVERRIDE=<class>,<class>`, and before it builds anything it calls `uvm_factory().set_type_override_by_name()` with the two class names. The example's base test, `MonBaseTest`, does the same, so the override also works on `MonRandTest`, which knows nothing about `AluEqualItem`:

```
make run TEST=MonRandTest BUG=A01 PLUSARGS=+OVERRIDE=AluItem,AluEqualItem
```

On the ALU with bug A01 the run logged

```
    0.00ns INFO    ..-and-scoreboards/mon_env.py(171) [uvm_test_top]: [OVERRIDE] the factory
makes AluEqualItem for every AluItem
```

and ended `MISSED: 0 errors`. The same test without the plusarg ends `caught: 30 errors`. A plusarg goes to the simulator when it starts, so changing one rebuilds nothing. Lab 04 has you use `+OVERRIDE` the same way on the router, in Part 4, step 6.

ConfigDB

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

```
ConfigDB().set(self, "env.sb", "check_flags", True)                 # in the test
self.check_flags = ConfigDB().get(self, "", "check_flags", False)   # 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`. Its last argument, `False`, is the value it returns if nobody set one.

The example's scoreboard checks only `ALUout` unless `check_flags` is `True`, and then it checks `N` and `Z` as well. `FacFlagsTest` sets it to `True` for `"env.sb"`, and it catches all three bugs:

```
      ok  the golden design                       0 errors                      2.5s
------------------------------------------------------------------------------
  caught  tier 1  A01_sub_operands_swapped        32 errors                     2.5s
  caught  tier 1  A02_nand_is_nor                 18 errors                     2.5s
  caught  tier 1  A03_zero_flag_low_nibble        19 errors                     2.5s
------------------------------------------------------------------------------
 3 of 3 caught.
```

A path in a `set()` can hold wildcards. They're globs, as in a shell, not regular expressions. A `*` matches any run of characters, dots included, so `"env.*"` reaches every component below `env`. A `get()` names one component, and pyuvm raises `UVMError` for a wildcard in its path. Lab 01's pyuvm test puts its `AluIf` in `ConfigDB` with `None` as the context and the path `"*"`, so that every component can find it. The course's `RouterBaseTest`, which Lab 04's base test derives from, does the same with the router's handle, under the name `DUT`.

When nothing reads the value

`FacTypoTest` makes the same `set()` for `"env.scoreboard"`, and the environment has no component by that name. The scoreboard's `get()` found nothing, so `check_flags` kept its default of `False` and the test missed A03 (`2 of 3 caught.`). The run printed no warning, and it passed on the golden design. To see what went wrong, turn on `ConfigDB`'s trace with `ConfigDB().is_tracing = True`, which logs every `set()` and every `get()` that finds a value. The test does that before its `set()`:

```
     0.00ns INFO     uvm.logger_holder126772875082400   CFGDB/SET Context: uvm_test_top  --
uvm_test_top.env.scoreboard check_flags=True
[TEST] uvm_test_top set check_flags = True for "env.scoreboard"
       0.00ns  INFO       uvm.logger_holder126772875082400    CFGDB/SET  Context:     --   *
vif=<sq_agent.AluIf object at 0x734c9c2fba40>
          0.00ns  INFO          uvm.logger_holder126772875082400      CFGDB/GET  Context:
uvm_test_top.env.agt.drv    --    uvm_test_top.env.agt.drv  vif=<sq_agent.AluIf  object  at
0x734c9c2fba40>
          0.00ns  INFO          uvm.logger_holder126772875082400      CFGDB/GET  Context:
uvm_test_top.env.agt.mon    --    uvm_test_top.env.agt.mon  vif=<sq_agent.AluIf  object  at
0x734c9c2fba40>
        0.00ns  INFO        uvm.logger_holder126772875082400     CFGDB/GET  Context:     --
vif=<sq_agent.AluIf object at 0x734c9c2fba40>
```

The `set()` was for `uvm_test_top.env.scoreboard`, and there's no `GET` line for `check_flags` at all, because the trace leaves out a `get()` that finds nothing. Compare the path in each `SET` line with the full names of the components that should read it. (The last `GET` is the sequence's. A sequence has no place in the component tree, so its `get()` passes `None` as the context.)

A knob with a sensible default can be read the way `check_flags` is, with a default in the `get()`. A component that can't work without a value should get it with no default, and then a `get()` that finds nothing raises `UVMConfigItemNotFound` and ends the run. `FacMissingTest` swaps in a scoreboard that gets `check_flags` with no default, and nothing sets it:

```
 BROKEN: UVMConfigItemNotFound   1.86 s   log: build/runs/FacMissingTest-s1/run.log
     FacMissingTest:  UVMConfigItemNotFound:  "Component  uvm_test_top.env.sb  has  no  key:
check_flags
   ConfigDB().get() asked for a name nobody set. That is configuration,
   not a check.
```

Every driver and monitor in Lab 04 gets the router's handle with no default, and the output agents' drivers and monitors get their `OutCfg` the same way, so if either one is missing, the run stops in the build phase.

When two components set the same value

`FacPrecTest` sets `check_flags` to `True` for `"env.sb"`. It also overrides the environment with a subclass, `FalseEnv`, which sets `check_flags` to `False` for its own `"sb"` before it builds the scoreboard. The environment is built after the test, so its `set()` comes second:

```
[TEST] uvm_test_top set check_flags = True for "env.sb"
[ENV] uvm_test_top.env set check_flags = False 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 `True` beat the environment's `False`. pyuvm stores each `set()` made during `build_phase` with a precedence of 1000 minus the depth of the component that made it, and a `get()` takes the value with the highest precedence. With that rule, an environment can give its components defaults, and a test can override them.

`FacGlobTest` is the same test, except that the test's `set()` is for `"env.*"`:

```
[TEST] uvm_test_top set check_flags = True for "env.*"
[ENV] uvm_test_top.env set check_flags = False for "sb"
```

This time the scoreboard reported `200 vectors checked: 0 wrong`, without the flags, and the test missed A03. Of the paths that match, pyuvm's `get()` first finds the most specific one that has a value named `check_flags`, here `uvm_test_top.env.sb` rather than `uvm_test_top.env.*`, and compares precedences only among the `set()`s for that one path. So a test that wants to override an environment's default has to name the component at least as exactly as the environment does. After the build phase, every `set()` gets the same precedence, 1000, so a later one for the same path replaces an earlier one.

Objects by handle

`ConfigDB` can hold an object as well as a flag like `check_flags`. It stores the object itself, so the component that gets the value and the one that set it share one object. Lab 04 uses that for the output agents. `RouterLabTest` makes an `OutCfg` for each output, sets output 0's for `"env.out0*"` and output 1's for `"env.out1*"`, and keeps the two handles:

```
self.ocfg = [OutCfg(0), OutCfg(1)]
for p in (0, 1):
    ConfigDB().set(self, f"env.out{p}*", "cfg", self.ocfg[p])
```

Both output agents are the same class, `OutAgent`. The driver and the monitor in `out1`, for example, have full names that start with `uvm_test_top.env.out1`, which matches `"env.out1*"`, so they get output 1's `OutCfg`, and its `port` tells them which pins to use. Nothing else differs between the two agents. Because the test kept the handles, your Part 3 stall test can change `self.ocfg[p].ready_pct` in the middle of a run. Each `OutReadyDriver` reads `ready_pct` again after every rising edge, so it follows the change at the next one.

Check yourself

1. `FacEqualTest` and `MonRandTest` both derive from `MonBaseTest`, and `FacEqualTest` adds a type override. It catches none of the three bugs, and `MonRandTest` 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 the constructor. What happened in `RouterInjectTest` when they were made with `create()` instead?

   Answer: A3, in Answers at the end.

Further reading

In pyuvm 4.0.1's source, the factory is in `s08_factory_classes.py`, and `ConfigDB` is at the end of `s13_uvm_component.py`, where a comment says why it matches globs and not regular expressions. ChipVerify's pages are about SystemVerilog UVM:

- UVM Factory Override covers type and instance overrides, and the difference in argument order between the factory's methods and the component's. pyuvm has only the factory's, called on `uvm_factory()`. The page's last example calls methods on a variable named `factory` that it never declares there.

- UVM config database, for the API, the wildcards and the precedence rules. Its `get()` returns 0 when it finds nothing; pyuvm's raises `UVMConfigItemNotFound` unless you give it a default. Its precedence rules leave out the first step of pyuvm's `get()`, which picks the most specific matching path, as `FacGlobTest` shows.
- Configure Components, on configuration objects like Lab 04's `OutCfg`.

Next: Lab 04: The same testbench in pyuvm.

Answers

A1. Check yourself

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

A2. Check yourself

Turn on the trace with `ConfigDB().is_tracing = True` and compare the path in the `CFGDB/SET` line with the scoreboard's full name. If there's no `CFGDB/GET` line for the knob, the scoreboard's `get()` found nothing, so the `set()` didn't name the component that reads the value, as in `FacTypoTest`. If the paths match, look for another `set()` for a more specific path, as in `FacGlobTest`.

A3. Check yourself

The test's override reached them too, so the packet for DST 0 was a `NoroutePkt`. Whenever its `pre_randomize()` set `to_disabled`, its `c_dst` asked for a disabled route, while the inline constraint asked for DST 0, the one enabled route. The solver found no solution, and the run ended `BROKEN: SolveFailure`, on seeds 11 and 15 of seeds 1 to 20 (Lab 04's solutions README).