Virtual sequences

This page answers: *the routing table, the packets and the backpressure are each driven by a different agent. How do you make them change together, from one place?*

In Lab 04 the tests did the work themselves, one step at a time. In each phase a test set the sinks, wrote the routing table, sent the packets, drained, read `DROP_CNT` and cleared it, and each step waited for the one before it to finish. In Lab 05, Part 4 you'll want to move routes to the other output while packets are still going in, and to change how both sinks behave at the same time. The APB agent, the source agent and both sinks' agents then all have work to do at once, and you decide what each of them does and when. You do this with a **virtual sequence**, which is a sequence that starts other sequences on several agents' sequencers and changes those agents' knobs, but makes no items of its own.

This page builds one for a small design of our own, a FIFO that holds four beats and has a valid/ready stream on each side. The files are in `guide-examples/virtual-sequences/`. A source agent drives the FIFO's input, and its sequence, `SrcSeq`, sends one beat per item. A sink driver drives `out_ready` on the output side, and how often it's ready depends on `ready_pct`, a knob in a configuration object. Neither of the two knows about the other, and the sink has no sequencer at all. Lab 05's sinks do have one, but a virtual sequence steers them the way you steer this sink, by changing their configuration rather than by sending them items, so this sink is a good stand-in for them.

A sequence given the environment

An ordinary sequence runs on its agent's sequencer, which passes its items to the driver. A virtual sequence has no driver to feed, so in pyuvm it runs on no sequencer at all, and it needs some other way to reach every agent that it steers. (In SystemVerilog UVM a virtual sequence usually runs on a virtual sequencer, which holds a handle to each agent's sequencer, and the sequence reaches the virtual sequencer through a handle called `p_sequencer`.) pyuvm has no `p_sequencer`, so the sequence is given the environment when it's made, and keeps it:

```
class BaseVseq(uvm_sequence):
    def __init__(self, name="vseq", env=None):
        super().__init__(name)
        self.env = env
        self.traffic_done = False

    def info(self, msg):
        self.env.logger.info(f"[VSEQ] {msg}")
    ...
    async def send(self, n, base):
        await SrcSeq(f"src_{base:02x}", n, base).start(self.env.src.sqr)

    def sink(self, pct):
        self.env.snk_cfg.ready_pct = pct
        self.info(f"sink ready {pct}% of cycles")
```

`send()` runs the source agent's own sequence on the source agent's sequencer, as a test would. `sink()` only writes the knob, without starting anything, and the sink's driver reads the new value at its next rising edge. The test starts the virtual sequence with `start()` and no sequencer:

```
v = cocotb.start_soon(self.VSEQ("v", self.env).start())
```

Lab 05's `RouterTest` starts its virtual sequence the same way, with `await RouterLab05Vseq("vseq", self.env).start()`. Lab 05's `RouterBaseVseq` gives you helpers of the same kind, among them `route()`, `ctrl()`,

`burst()`, `send_rand()`, `sink()`, `stall()` and `drain()`, and most of them are the base test's steps from Lab 04, rewritten as methods of a sequence.

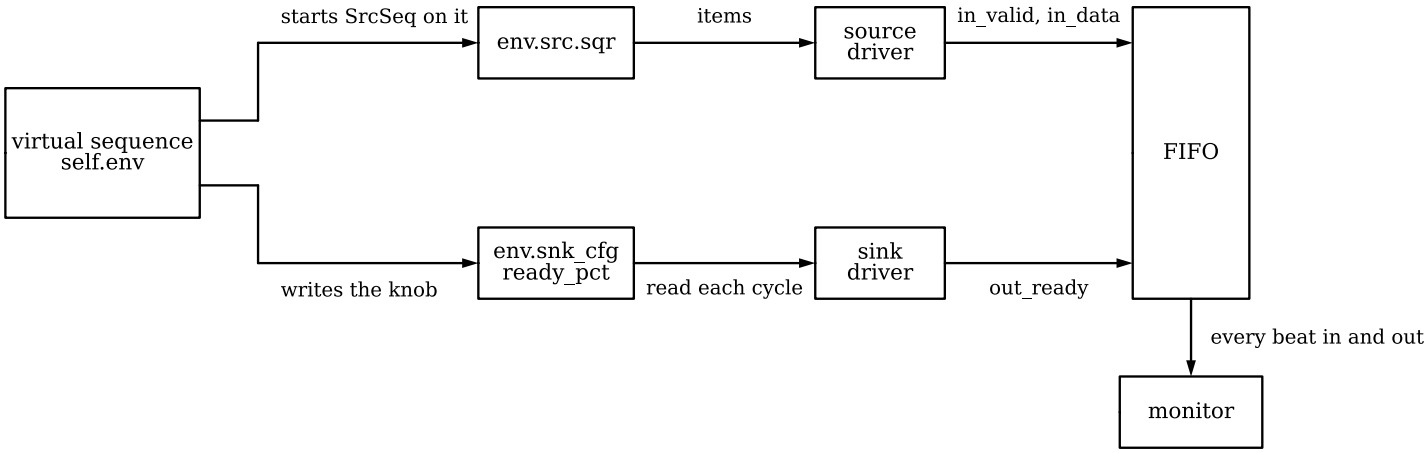


Figure 1: The example's testbench: one virtual sequence steering two drivers

Our example's testbench. Using the environment it was given, the virtual sequence reaches the source through the source's sequencer and the sink through the sink's configuration object. The monitor checks every beat, whatever caused it.

Because it runs on no sequencer, a virtual sequence can't send an item itself. `VseqItemTest` runs a virtual sequence whose `body()` calls `start_item()`. pyuvm raised a `UVMSequenceError` at that first call, and the graders called the run BROKEN:

```
BROKEN: UVMSequenceError   0.27 s   log: build/runs/VseqItemTest-s1/run.log
  VseqItemTest: UVMSequenceError: Tried start_item in a virtual sequence v
```

A pyuvm sequence has no logger of its own, because it isn't a component, so `info()` reports through the environment's logger. Every message from the virtual sequence therefore carries the environment's path, `[uvm_test_top.env]`, and you tell its lines from the environment's own by the `[VSEQ]` tag at the start of each message. In Lab 06 you'll pass the environment's logger to each register sequence you make, and the register sequences report through it.

Doing two things at once

The test starts our virtual sequence, `DemoVseq`. Its first step stalls the sink, and then sends six beats in one branch while a second branch waits 15 cycles and makes the sink ready in every cycle again. `par()` starts each coroutine it's given as a cocotb task, and then waits for each task in turn, so it returns when all of them are done:

```
@staticmethod
async def par(*coros):
    tasks = [cocotb.start_soon(c) for c in coros]
    for t in tasks:
        await t
```

and the first step is

```
self.sink(0)

async def beats():
    await self.send(6, 0x10)
    self.info("step 1: all 6 beats have gone in")

async def release():
    await self.cycles(15)
    self.info(f"step 1: the FIFO holds {int(d.level.value)} beats")
    self.sink(100)

await self.par(beats(), release())
```

It printed:

```
    10.00ns INFO     ..tual-sequences/test_vseq.py(172) [uvm_test_top.env]: [VSEQ] sink ready
0% of cycles
    160.00ns INFO      ..tual-sequences/test_vseq.py(172) [uvm_test_top.env]: [VSEQ] step 1:
the FIFO holds 4 beats
   160.00ns INFO     ..tual-sequences/test_vseq.py(172) [uvm_test_top.env]: [VSEQ] sink ready
100% of cycles
    200.00ns INFO      ..tual-sequences/test_vseq.py(172) [uvm_test_top.env]: [VSEQ] step 1:
all 6 beats have gone in
```

By 160 ns the FIFO held four beats and was full, and `send()` still had two of its six beats to send. It could only return once the sink had taken some beats, and the sink would only take beats once something set `ready_pct` again. If you wrote the two one after the other, `send()` and then `sink(100)`, they would deadlock, because `send()` waits for the sink and the line that releases the sink waits for `send()` to return. With `+SERIAL` the example runs them in that order, and the test's own limit of 2,000 cycles ended the run:

```
 20010.00ns ERROR     ..tual-sequences/test_vseq.py(281) [uvm_test_top]: [TEST] the virtual
sequence is still running after 2000 cycles; the FIFO holds 4 beats
 20010.00ns INFO     ..tual-sequences/test_vseq.py(147) [uvm_test_top.env.mon]: [MON] 4 beats
in, 0 out, 0 out of order
```

With `par()` the two branches run at the same time, and the step ended at 200 ns, when the last beat had gone in.

`par()` awaits each task, and awaiting a cocotb task re-raises any exception that the task raised, so a branch that fails also fails the sequence and the test. One of `labs/ref`'s probes of cocotb 2.0.1 waits on a task's `complete` trigger instead, and never calls the task's `result()`. That test passes, and the task's exception is lost. So if you wait on `complete`, as our test does to give up after 2,000 cycles, call the task's `result()` afterwards, which raises the exception if there was one.

A loop that ends with the traffic

The second step sends 30 beats while another branch keeps changing how often the sink is ready. That branch can't know how long the traffic will take, so it loops until the traffic branch sets a flag, `traffic_done`:

```
self.traffic_done = False
```

```
async def traffic():
    await self.send(30, 0x40)
    self.traffic_done = True
    self.info("step 2: all 30 beats have gone in")

async def backpressure():
    while not self.traffic_done:
        self.sink(random.choice([0, 30, 100]))
        await self.cycles(random.randint(4, 12))

await self.par(traffic(), backpressure())
```

The loop set the sink eleven times between 200 ns and 1,070 ns, and the last beat went in at 1,130 ns. The loop saw the flag the next time it checked, at 1,150 ns, and then `par()` returned. Step 4 of the sequence you write in Lab 05, Part 4 has the same shape, with `backpressure()` as the looping branch and a third branch that moves routes. If the traffic branch didn't set `traffic_done`, the loop would go on changing the sink for ever, and `par()` would never return.

None of these sequences checks anything. In this example the monitor does the checking, and in the labs the scoreboard and the assertions do. The monitor watches both sides of the FIFO in every cycle and checks every beat that comes out, whatever is driving the FIFO. At the end of the run, after all three steps, it printed `[MON] 44 beats in, 44 out, 0 out of order`.

Two sequences on one sequencer

The third step starts two source sequences at once on the same sequencer. Each sends four beats. The first numbers its beats from `80`, and the second from `a0`:

```
await self.par(self.send(4, 0x80), self.send(4, 0xA0))
```

and the beats came out of the FIFO as

```
step 3: the sink took 80 a0 81 a1 82 a2 83 a3
```

pyuvm's sequencer keeps one queue. Each `start_item()` puts its item at the back of it, and the sequencer hands items to the driver from the front, in the order they were asked for. Each sequence asks for its next item only after the driver has finished its last one, so the two sequences alternated item by item. Sometimes that's what you want, for example to mix two traffic streams together. It also means that if you start a second sequence on a sequencer while another one is running there, the two share the driver, and pyuvm 4.0.1's sequencer has no way for one sequence to keep the driver to itself. Each of Lab 05's sinks runs a sequence that never ends, and Part 4 asks you why `sink()` and `stall()` change the sink's `OutCfg` rather than starting a sequence on the sink's sequencer.

Configuration changes and the scoreboard

With a virtual sequence you can change the configuration while traffic is moving, and then the scoreboard has to apply each change from the right cycle. In Lab 04 the tests wrote the routing table only between phases, so the scoreboard could apply a write as soon as it saw it. In Lab 05 a route can change between one header and the next, and the specification says that a header accepted in the cycle in which the write completes still uses the old route. So in Lab 05 the APB monitor stamps each transfer with the cycle it completed in, and the scoreboard applies a route write only at the first header of a later cycle. Part 4 asks you why it goes by the cycle rather than by the order in which the two monitors report.

Adding a virtual sequence to a test can also change the random values that the test draws after it. Every random value in the labs comes from Python's `random`, one after another, so a sequence that draws values early in a test moves every value drawn after it. Making a pyuvm sequence and starting it draws nothing, though, so only the sequence's own draws move the values after them. Regressions and seeds has the measurements. In Lab 06, Part 5 you'll see the packets of Labs 04 and 05 change because of eight values drawn early in `RouterTest`.

Check yourself

1. In step 1, why can't `self.sink(100)` simply follow `await self.send(6, 0x10)` in the same branch?

   Answer: A1, in Answers at the end.

2. What does `par()` wait for in step 2, and what would happen if the traffic branch forgot to set `traffic_done`?

   Answer: A2, in Answers at the end.

3. You want the four beats from `80` to go in before the four from `a0`, with both sends still started by the virtual sequence. What do you change?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's pages describe SystemVerilog UVM, where a virtual sequence usually runs on a virtual sequencer and reaches the agents' sequencers through `p_sequencer`. pyuvm has neither, and the labs use the pattern on this page, a sequence given the environment.

- UVM Virtual Sequence: both of SystemVerilog UVM's styles, `p_sequencer` and `start(null)`; the second is the one closest to pyuvm's. Correction: two of its three example virtual sequences create their child sequences in `pre_body()` and start them without a parent, against the page's own advice.
- UVM Virtual Sequencer: the sequencer that holds handles, which pyuvm doesn't need. Its log was made from a different version of the code than the page shows, so the names in it don't match.
- UVM Sequence Arbitration: how a SystemVerilog UVM sequencer chooses between sequences that run on it at the same time. pyuvm 4.0.1′s sequencer has one order, first come first served.
- How to execute sequences via start(). Its example test starts a sequence without raising an objection, which works only because that sequence takes no simulation time.

Next: Lab 05: Protocol rules and backpressure.

Answers

A1. Check yourself

`send()` returns only when all six beats have gone into the FIFO, and the FIFO holds four. The fifth beat waits for `in_ready`, which comes back only when the sink takes a beat, and the sink takes beats only after `sink(100)`. In the same branch as `send()`, `sink(100)` would never be reached. The `+SERIAL` run puts it there, and after 2,000 cycles its monitor had counted 4 beats in and 0 out.

A2. Check yourself

It waits for both tasks to finish. Without the flag the backpressure loop never ends, so the virtual sequence never returns, the test never drops its objection, and the run goes on until something stops it. In our example that's the test's limit of 2,000 cycles; in the labs it's the test's watchdog (`+MAX_CYCLES`) or `make`'s run timeout.

A3. Check yourself

Await them one after the other instead of in `par()`: `await self.send(4, 0x80)` and then `await self.send(4, 0xA0)`. The second sequence then starts only when the first has returned.