VERIFY THIS!

An introduction to UVM

DAY 1 DAY 2 DAY 3 DAY 4 DAY 5 DAY 6 DAY 7 DAY 8

github.com/leandrotozzi/verifythis · SystemVerilog · Verilator · UVM 2020.3.1

How to use this course

Especially if you are taking it on your own

- **What you need to know first:** Verilog or VHDL, and having simulated something. Object-oriented programming is **not** required — day 2 is exactly that. The only thing the course takes for granted is **SystemVerilog**: if you come from VHDL or Verilog-2001, half an hour with `docs/systemverilog-para-el-que-viene-de-vhdl.md` (in Spanish) before starting
- It is **seven days of class**, of 4 to 5 and a quarter hours — **≈ 31 h**, and each agenda carries its own. Plus an **optional day 8**, after the wrap-up. On your own, count double: half of it goes into running the examples, and that is the half that teaches
- `s` opens the **presenter notes** in another window · `n` shows them down here, without leaving the page. That is where what the instructor would say out loud lives: the classic trap, the reason behind the number, the mistake everybody makes the first time. **Do not skip them**
- Every day closes with a **review** of clickable options and with a **self-checking exercise**: `cd code/ejercicios/d1 && bash run.sh`. The names of the other eighteen are in `code/ejercicios/README.md`

How to use this course

Running it, moving around it, and where to look when something breaks

- **Run the examples**: `make u4/tests` runs one, `make u4` the whole unit. A verification course that is only read is worth nothing. A UVM example takes **a minute and a half the first time and fifteen seconds the second**, if you have `ccache` installed — and if you do not have Linux, the short road is GitHub Codespaces, which already ships with it
- `0` – `8` jump to the cover and to each day · `i` opens the index · `Esc` shows the whole deck · and the corner ribbon unfolds the cheat sheet of the day
- **If you are on your own, read the book:** `en/libro/day1.html` is this same course straight through, with the notes already inside the text. Keep the deck alongside for the reviews
- **When something does not work, go to the end:** the appendices *The debug toolbox* —what to look at for each symptom— and *The 21 silent traps* — everything that compiles, runs and lies—. Those are the two slides to print

Agenda

Day 1 · ≈ 4 h · units 1 and 2

Unit 1 · Why we verify

- Trends
- What UVM is
- The VTALU spec
- The verification plan

Unit 2 · The testbench without UVM

- The conventional testbench
- Functional coverage
- Interfaces and BFM · the `clocking block`

Two exercises, and the second one is solved with the wave viewer

- **And closing the day**, going back to the first slide

By the end of the day you can:

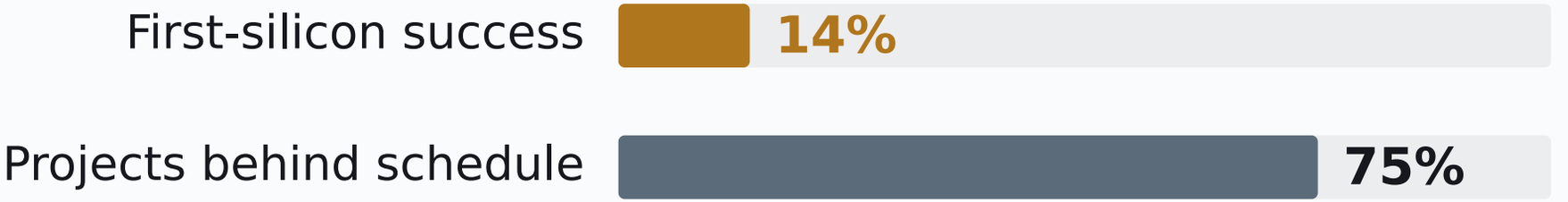
- **Write** a five-column verification plan for a DUT you have not seen before
- **Prove** that your scoreboard checks something: put a bug into the DUT on purpose and watch it fail
- **Read** the number a `covergroup` gives, say which row of the plan is missing, and what to look at first when it reads 0 %
- **Explain** which race a `clocking block` avoids — and why they are optional anyway

Six out of seven

And the bug that reached silicon passed a green regression

How IC/ASIC projects end up

IC/ASIC, 2024



14% is the lowest in 20 years of the survey. Behind schedule: 75%, against a 66% historical figure.

Data: Wilson Research Group / Siemens EDA, 2024 study. Own chart.

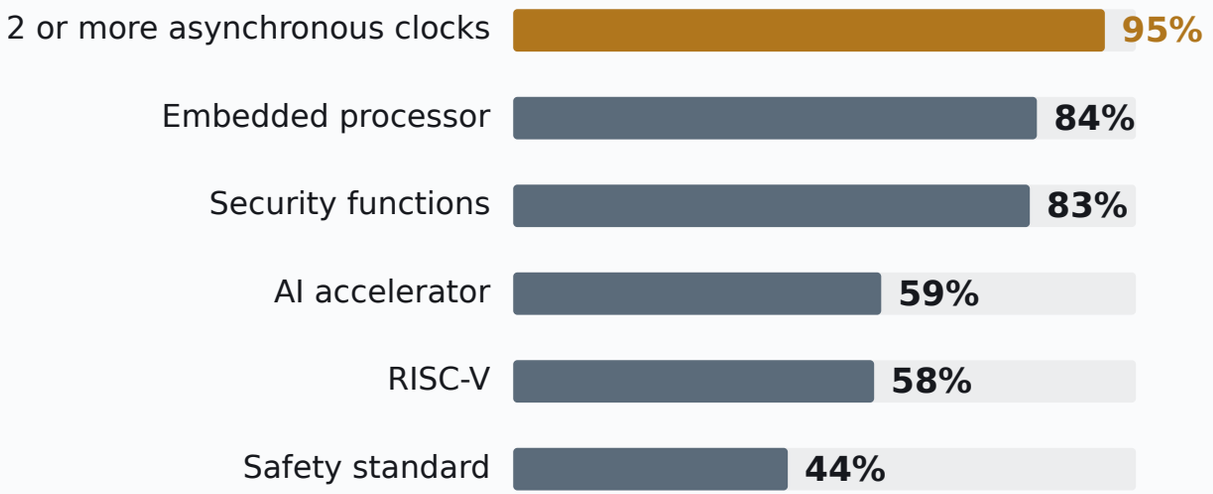
- **14 %** of projects get first silicon right. Six out of seven go back to the fab, and it is the worst figure in twenty years of the survey
- Every functional bug that reached silicon went through a testbench first. That testbench ran, finished, and **said everything was fine**
- Nobody tapes out with a `FAILED` in the log. You tape out with a `PASS`
- Today has a single question: **how do you know you verified?** And *"I ran a lot of tests"* is not an answer

Trends

Why they come back: what is inside a chip today

What today's IC/ASIC have inside

IC/ASIC, 2024



RISC-V nearly doubled since the previous study; AI accelerators went from ~30% to 59%.

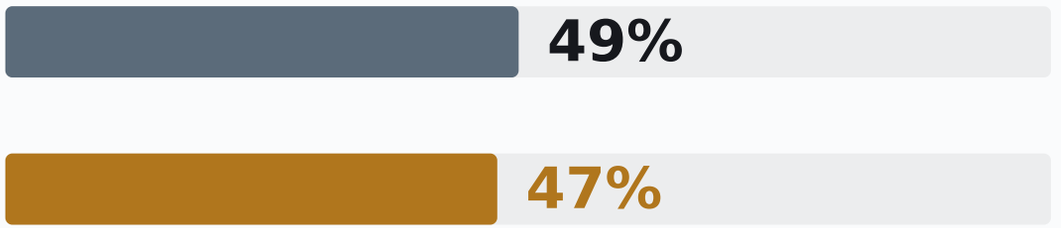
Data: Wilson Research Group / Siemens EDA, 2024 study. Own chart.

Trends

Verification is not a stage, it is half the work

Where the time goes

IC/ASIC, 2024



On processor projects the verification:design ratio reaches 5 to 1.

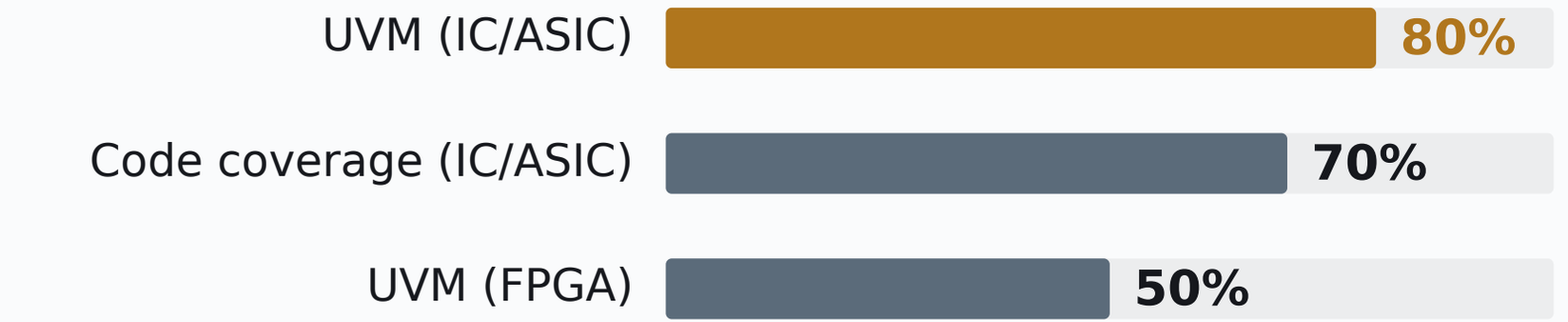
Data: Wilson Research Group / Siemens EDA, 2024 study. Own chart.

Trends

And it is verified, mostly, with UVM

What people verify with

IC/ASIC and FPGA, 2024



Formal is growing: 5.8% a year in property checking, 8.7% in automatic applications.

Data: Wilson Research Group / Siemens EDA, 2024 study. Own chart.

Is SystemVerilog the *COBOL* of electronics?

Language	Reserved Keywords
ANSI COBOL 85	357
SystemVerilog	248
VHDL 2008	115
Verilog 95	102
C#	102
C++ 20	92
Python 3	35

> *" No academic computer scientists participated in the design of COBOL; all of those on the committee came from commerce or government"* Sound familiar?*

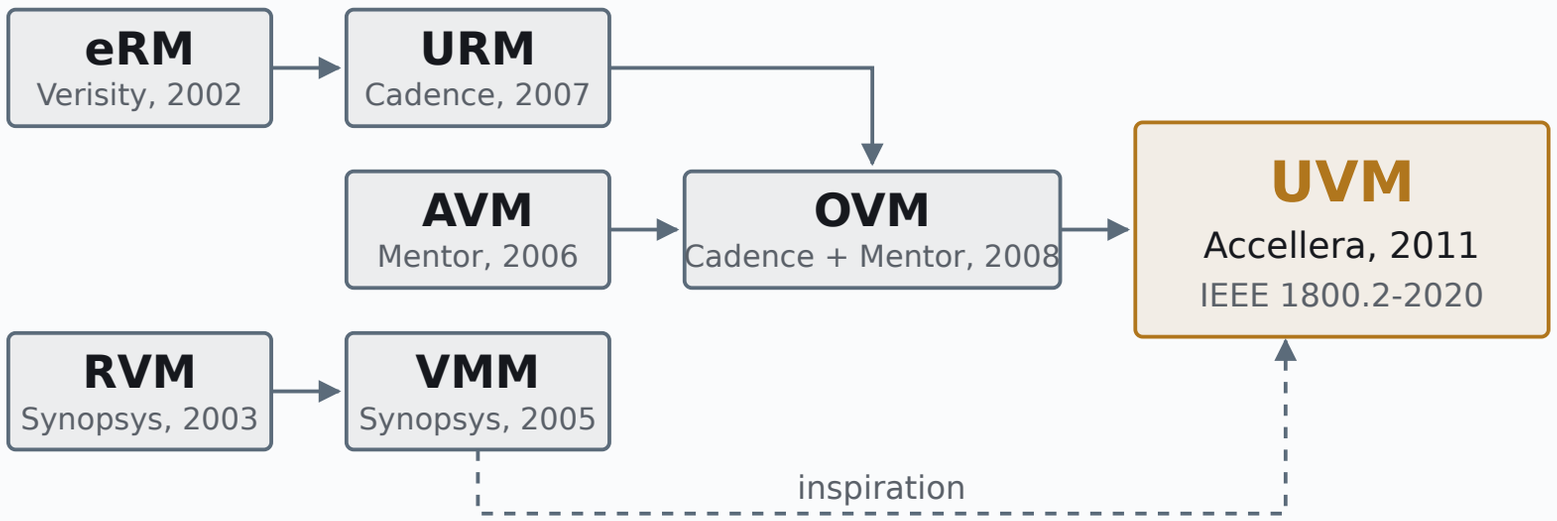
Introduction

What is UVM?

- **U**niversal **V**erification **M**ethodology: a **class library** written in SystemVerilog, plus a pile of **conventions** on how to use it
- It is not a language and it is not a simulator. Everything we are going to see in the course can be written by hand — and on days 1 and 2 we are going to write it by hand
- It is made by **Accellera**, the same consortium that standardized SystemVerilog, and it is **IEEE 1800.2**. The course uses the reference implementation, `uvm-core 2020.3.1`
- It comes from OVM (Cadence + Mentor), with ideas from VMM (Synopsys) and from eRM (Verisity): the methodologies of four vendors that competed, until the industry got tired of translating testbenches between tools
- And there is the goal, and it is a single one: **that the testbench of the person next to you looks like yours**

Introduction

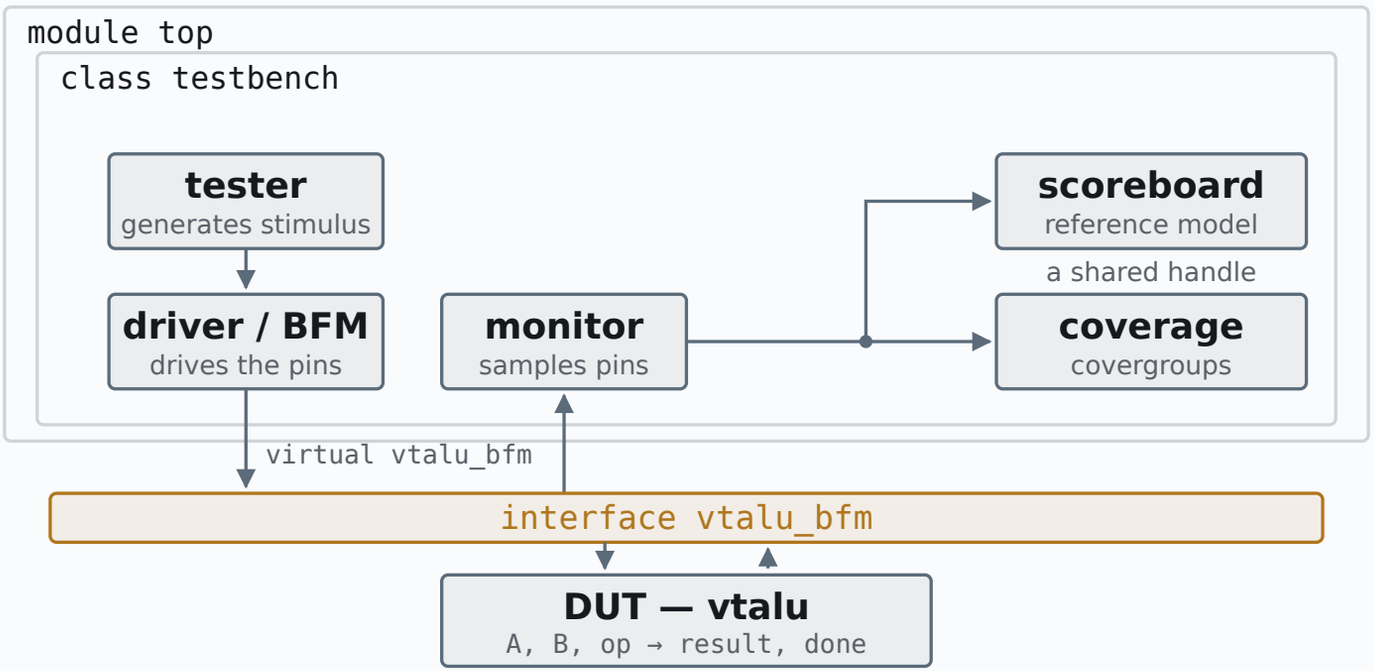
Where it comes from



OVM (2008) merged Mentor's AVM with Cadence's URM; Accellera took OVM 2.1.1 as the base for UV
VMM contributed ideas, not code. IEEE standardized UVM in 1800.2-2017 and then in 1800.2-2020.

Introduction

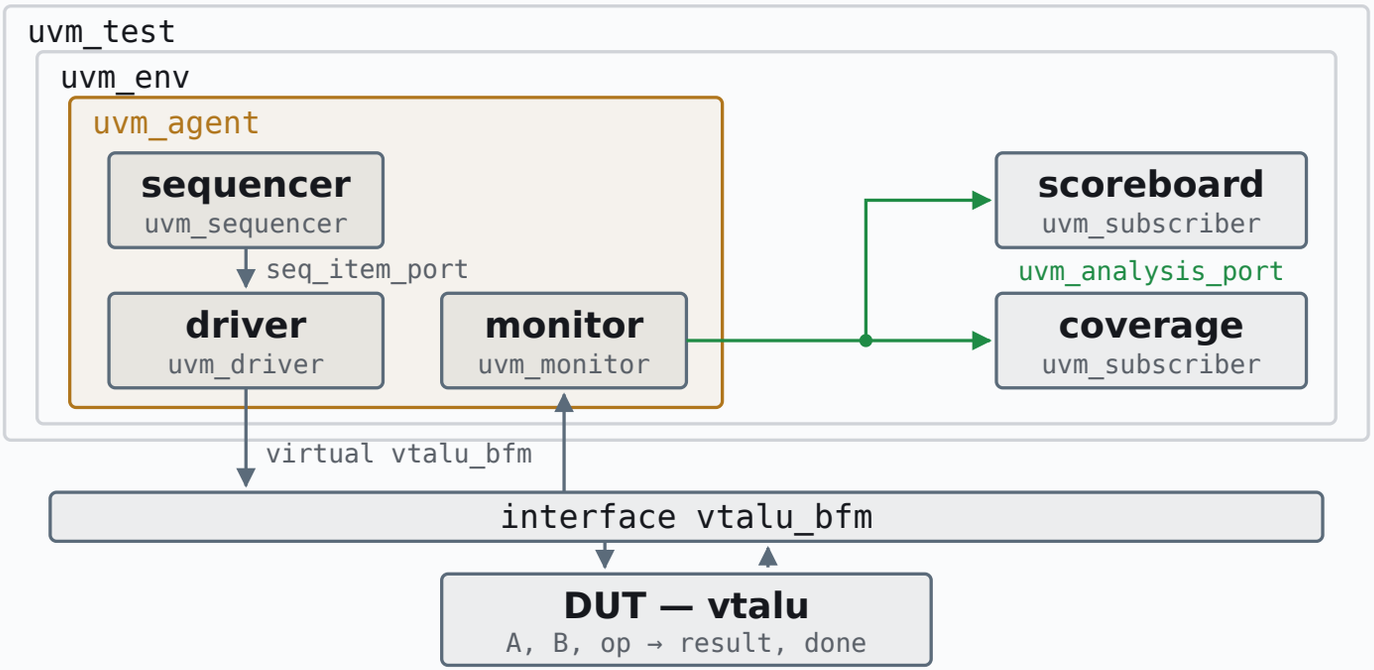
A SystemVerilog testbench, from the inside



The interface is the boundary: classes and transactions on one side, pins and RTL on the other.

Introduction

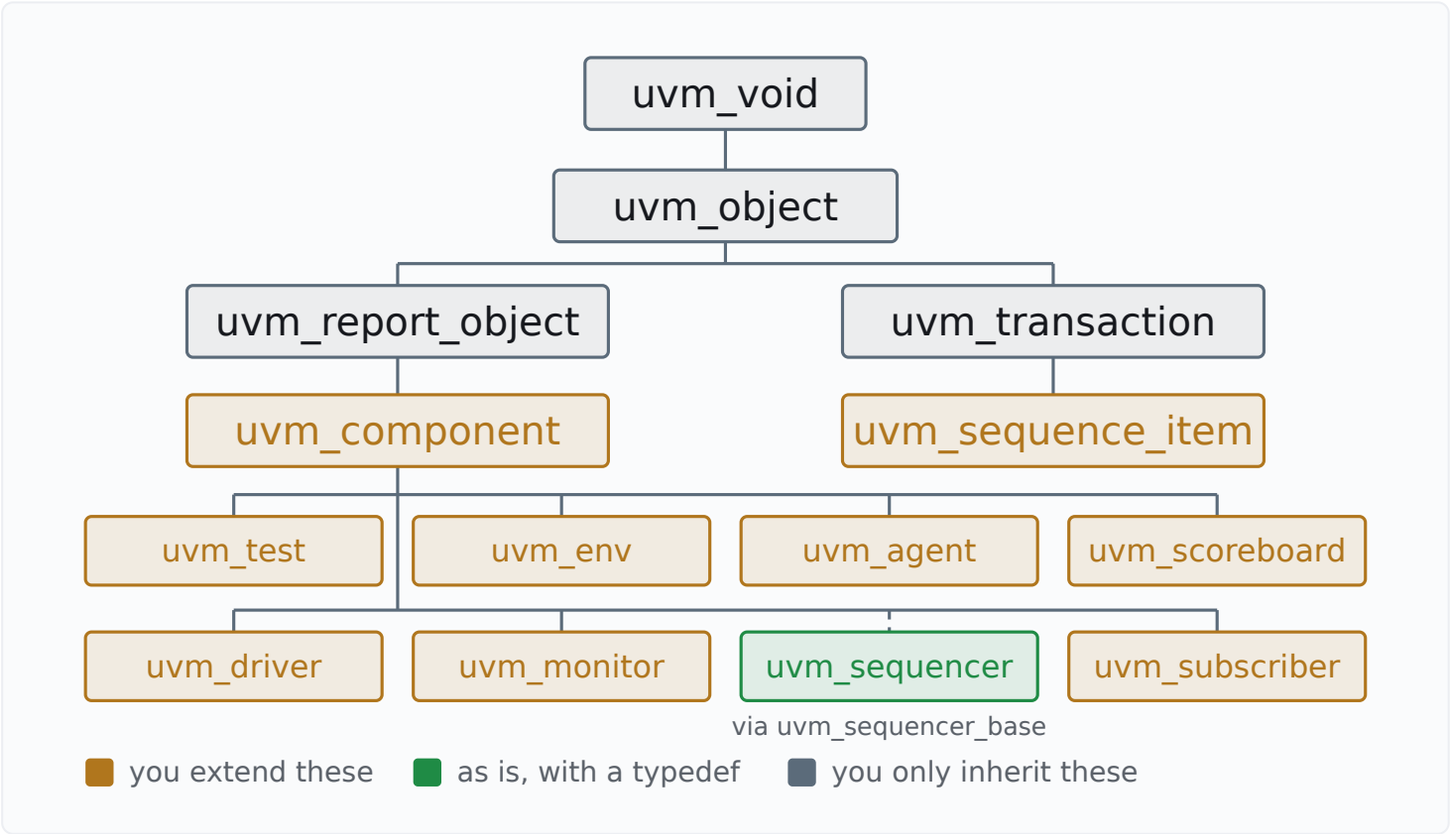
The same testbench, with UVM structure



Same pieces, another structure: test → env → agent, and the connections become analysis ports.

Introduction

The class tree

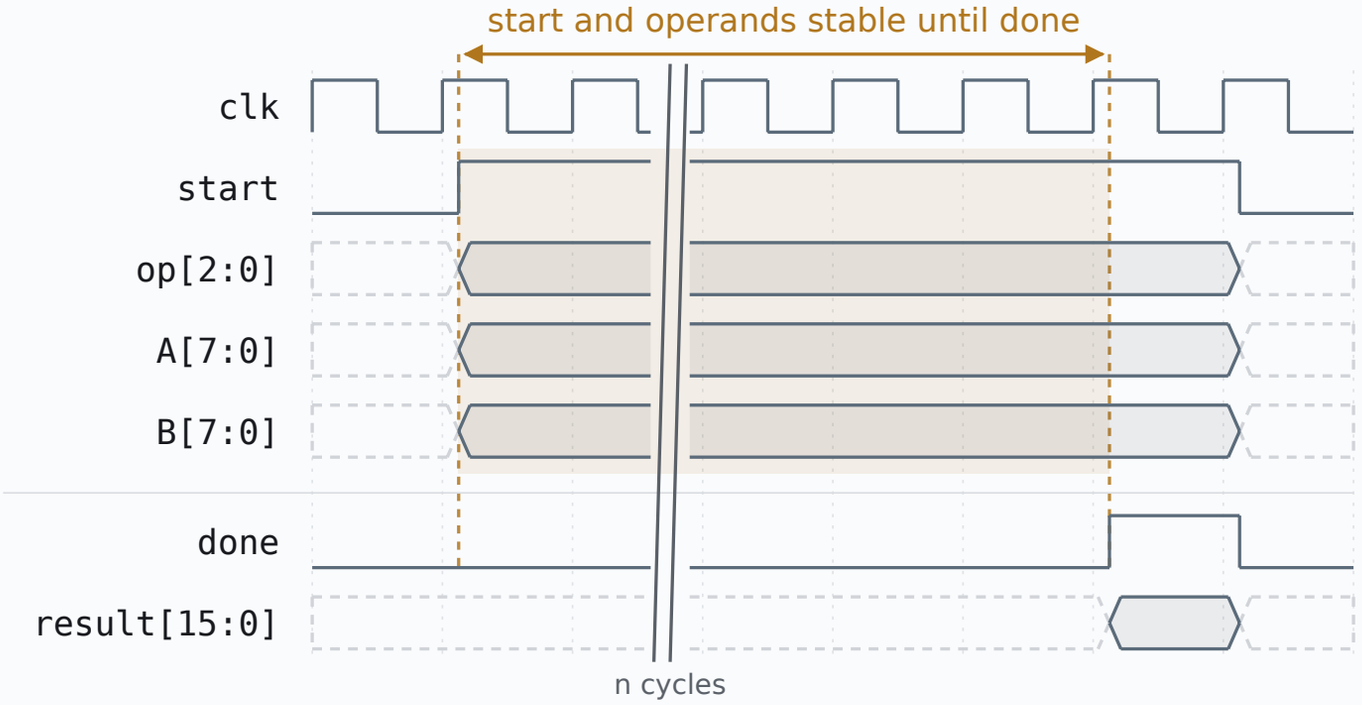


Introduction

Run more tests writing less code

- That is the promise, and it has one condition: **the env and the components barely change** from one test to another. What changes is the stimulus
- For that to work, the testbench has to leave **hooks**: virtual methods, the factory and callbacks. A test hooks in there and does not touch the structure
- A new test is then one of these three things, and none of them touches the env:
  - adding **constraints** to reach a corner case
  - doing a factory **override** to swap one class for another
  - injecting errors or delays with **callbacks**
- And on top of all that, the lever that pays off the most: **the same test with hundreds of different seeds**

The VTALU spec



The testbench waits `done` to sample `result`: the latency depends on the operation.

- *start* has to stay at 1 and the operands stable until the operation finishes
- *done* goes up when the operation finishes: on the multiplication it is a one-cycle pulse; on add/sub/and/xor it is a **level**, and it stays up while *start* stays up
- *reset_n* is asynchronous and active low; the diagram starts with the DUT already out of reset
- *ovf* comes out together with *done*, and it is the **borrow of the subtraction**: it is 1 when `A < B`. For the other operations it is always 0

The VTALU spec

Operation	Opcode	Cycles	`ovf`
no_op	3'b000	—	0
add_op	3'b001	1	0
sub_op	3'b010	1	`A < B`
and_op	3'b011	1	0
xor_op	3'b100	1	0
mul_op	3'b101	4	0
free	3'b110	1	0
rst_op	3'b111	—	—

- **Cycles** counts `clk` edges with `start` up, up to and including the one that raises `done`: one on the single-cycle ops, four on the multiplication. The plan and the day 7 assertions count the same way
- `rst_op` is not decoded by the RTL — the DUT sees it as *unused*. It is the testbench convention for pulsing `reset_n`, and that is why it is in the `operation_t` of the testbench
- **`3'b110` is free on purpose**: it is today's exercise

The VTALU spec

Single Cycle: Add - Sub - AND - XOR

```
// The two resets are deliberately different: the one on the result is
// SYNCHRONOUS and the one on done is ASYNCHRONOUS. Not an oversight: the spec.
always_ff @(posedge clk) begin
   if (!reset_n) begin
      result_1c <= 16'h0000;
      ovf_1c    <= 1'b0;
   end else if (start) begin
      ovf_1c <= (op == 3'b010) && (A < B);
      case (op)
         3'b001:  result_1c <= {8'h00, A} + {8'h00, B};
         3'b010:  result_1c <= {8'h00, A} - {8'h00, B};
         3'b011:  result_1c <= {8'h00, A} & {8'h00, B};
         3'b100:  result_1c <= {8'h00, A} ^ {8'h00, B};
         default: ;  // 3'b110 is free: it is the day 1 exercise
      endcase
   end
end

// done goes high on the edge if start is high and the op is not no_op
always_ff @(posedge clk or negedge reset_n) begin
   if (!reset_n) done_1c <= 1'b0;
   else done_1c <= start && (op != 3'b000);
end
```

- Two `always_ff` and nothing else: one registers `A op B`, the other raises `done`
- The resets **are not the same**: the one on the result is **synchronous** —only `clk` in the sensitivity list—, the one on `done` is **asynchronous**
- `done <= start && (op != no_op)`: that is why on these four operations `done` is a **level** and stays up while `start` is

The VTALU spec

Multi Cycle: Multiplication

```
// Four register stages. done is pipelined alongside the data, and each
// stage clears itself with (& ~done_mult): that is what makes done a
// one-cycle pulse instead of staying high for as long as start does. It is
// subtle -- it is copied straight from the VHDL.
always_ff @(posedge clk or negedge reset_n) begin
   if (!reset_n) begin
      done_mult <= 1'b0;
      done3 <= 1'b0;
      done2 <= 1'b0;
      done1 <= 1'b0;
      a_int <= 8'h00;
      b_int <= 8'h00;
      mult1 <= 16'h0000;
      mult2 <= 16'h0000;
      result_mult <= 16'h0000;
   end else begin
      a_int <= A;
      b_int <= B;
      mult1 <= a_int * b_int;
      mult2 <= mult1;
      result_mult <= mult2;
      done3 <= start & ~done_mult;
      done2 <= done3 & ~done_mult;
      done1 <= done2 & ~done_mult;
      done_mult <= done1 & ~done_mult;
   end
end
```

- It is a pipeline: the operands get registered, they get multiplied, and the product goes through two more registers before coming out on `result_mult`: four edges
- The `done` travels down **the same chain** — `done3`, `done2`, `done1` — so it arrives exactly with the data and not before
- The `& ~done_mult` of each stage is what turns the chain off on its own: that is why here `done` is a **one-cycle pulse**, even if `start` stays up

The VTALU spec

Top Level

```
assign es_mult = (op == 3'b101);

assign start_1c = es_mult ? 1'b0 : start;
assign start_mult = es_mult ? start : 1'b0;
assign result_mux = es_mult ? result_mult : result_1c;
assign done = es_mult ? done_mult : done_1c;
```

- The top computes nothing: it instantiates the two blocks and **decodes** the opcode
- `es_mult = (op == mul_op)`. The `start` is routed to only one of them, and `result_mux` and `done` come out of the same side. `ovf` follows the same pattern right below the excerpt — `assign ovf = es_mult ? 1'b0 : ovf_1c;` — and `result_mux` only comes out through `result` after the `+VTALU_BUG` XOR
- That is why the bus protocol is a single one even though inside there are two different latencies: what the testbench sees is `start` → `done`

The VTALU spec

How to run it

- The DUT lives in `code/vtalu_dut/`, in **SystemVerilog**
- The simulator of the course is **Verilator** (free, no licence):

```
make u4/tests          # one UVM example:  ~1 min 30 the first time
make u4/tests          # the second, with ccache:      ~15 seconds
make               # the 38 examples
```

- The ones that use UVM compile **the whole library** the first time. Install `ccache` before starting: the course detects it on its own and does not recompile twice
- Every example has its `run.sh`; the common flags are in `code/verilator/common.sh`

The VTALU spec

Looking at the real waves

- The diagram on the first slide is **drawn**. The real one you generate yourself, and it is the tool 47 % of the work gets done with: debugging

```
cd code/u2/convencional
VLT_TRACE=1 bash run.sh      # compiles with --trace and dumps vtalu.vcd
gtkwave vtalu.vcd            # or surfer, or whatever viewer you use
```

- `VLT_TRACE=1` turns on two things at once: Verilator's `--trace` and the top's `$dumpfile`/`$dumpvars`, which without the flag would not compile
- It costs nothing and it is free, same as the simulator. When the testbench hangs on `while (done == 0)`, this is what is going to tell you why

The VTALU spec

Summary of the unit

- The **protocol** is a single sentence, and it has to be respected: `start` at 1 and the operands stable **until** `done` **goes up**
- The fine print that is going to hang the first testbench: `done` is a one-cycle **pulse** on the multiplication and a **level** on the single-cycle ones
- Inside there are **two blocks** with different latencies —one and four cycles— and the top **decodes** the opcode. Outside, the bus is a single one
- `ovf` is one more output, and it belongs to `sub_op` and to nobody else
- `rst_op` **does not exist for the RTL**: it is a testbench convention for pulsing `reset_n`, and that is why it enters the coverage
- The DUT does not validate the opcode: the free one **raises** `done` **all the same**, with the `result` of the previous operation. Catching it is the **verification plan's** job, not the design's
- And the tool used 47 % of the time is already installed: `VLT_TRACE=1` and GTKWave

The verification plan

The most professional deliverable of the discipline, and the cheapest

- It is a **table**. It gets written **before** the testbench — if it gets written after, it describes what the testbench already does instead of what the spec asks for
- It comes out of the spec you have just read, not out of the testbench that does not exist yet
- And it is what gets delivered: the day 7 capstone is handed in with the plan filled, which is exactly what gets handed in on a project

The verification plan

The five columns

Column	The question it answers	What does not go in
Feature	which part of the spec does this row come from?	the name of a testbench file
Scenario	which concrete situation has to be provoked?	*"test the ALU"*
Stimulus	who provokes it: the random or a directed case?	*"by hand"*
Check	who says it went right?	*"you look at the wave"*
Measure	which bin gets filled when it happens?	*"you see it in the log"*

- One row per **scenario**, not per feature: a feature with three corner cases is three rows, and they get closed one at a time

The verification plan

Two rules that come out of the columns

- **If the measure column is empty, nobody is going to find out that the scenario never happened.** A case the random never touched and that has no bin is indistinguishable from one that happened a thousand times
- And one distinction that orders the check column: **the scoreboard checks *what* the DUT computes; the assertions check *how* it is talked to**

The conventional testbench

The VTALU plan, with three columns to fill

Feature	Scenario	Stimulus	Check	Measure
ALU	the six operations			
ALU	operands at `00` and at FF			
reset	operate after a reset			
mult	a mult after a single-cycle one			
ALU	the same operation twice in a row			
sub	subtract too much: `A < B`, and `ovf` goes up			

- Six rows that come out of the spec, and none of the right-hand columns yet. This section fills **two**: the stimulus provokes them and the scoreboard says whether the result was right
- The third one —**which bin gets filled when it happens**— is the next section
- The testbench has three parts because the plan has three columns to fill

The conventional testbench

The stimulus: a thousand operations, and the protocol by hand

```
repeat (1000) begin
   @(negedge clk);
   op_set = get_op();
   A = get_data();
   B = get_data();
   start = 1'b1;
   case (op_set) // handle the start signal
     no_op: begin
        @(posedge clk);
        start = 1'b0;
     end
     rst_op: begin
        reset_n = 1'b0;
        start = 1'b0;
        @(negedge clk);
        reset_n = 1'b1;
     end
     default: begin
        // Wait for an EDGE with done up, not for done itself: on the
        // one-cycle ops done is a level still up from the previous op,
        // and a plain wait(done) drops this one without a sound.
        enviadas++;
        do @(negedge clk); while (done == 0);
        start = 1'b0;
     end
   endcase // case (op_set)
end
$finish;
```

- A thousand rounds: pick the operation, pick the operands, raise `start`, wait for **an edge with `done` up**, lower `start`
- The `case` is there because of the fine print of the spec: `no_op` does not raise `done`, `rst_op` pulses `reset_n`, and the rest wait
- `get_op()` and `get_data()` bias the random towards the edges —00 and FF— to reach the cases the plan asks for. It is *constrained random* written by hand
- `enviadas` gets counted here and `chequeadas` in the scoreboard, and a `final` demands they match: it is the testbench checking itself
- Notice who knows about the protocol here: **the tester**. In interfaces and BFM that moves to the BFM and in the body of the `repeat` shrinks to one line

The conventional testbench

The self-checking: predict and compare

```
// Sent by the tester, compared by the scoreboard: the final block below
// demands that the two match.
int enviadas, chequeadas;

// Scoreboard loop: checks that everything came out right
always @(posedge done) begin : scoreboard
   shortint predicted_result;
   bit      predicted_ovf;
   #1;
   case (op_set)
     add_op: predicted_result = A + B;
     sub_op: predicted_result = A - B;
     and_op: predicted_result = A & B;
     xor_op: predicted_result = A ^ B;
     mul_op: predicted_result = A * B;
   endcase // case (op_set)

   // ovf belongs to sub and to nobody else: with 8-bit inputs and a 16-bit
   // output, neither the addition nor the multiplication can overflow.
   predicted_ovf = (op_set == sub_op) && (A < B);

   if ((op_set != no_op) && (op_set != rst_op)) begin
     chequeadas++;
     if (predicted_result != result || predicted_ovf != ovf)
       $error ("FAILED: A: %0h  B: %0h  op: %s result: %0h ovf: %0b",
               A, B, op_set.name(), result, ovf);
   end
end : scoreboard
```

- An `always @(posedge done)`: every time the DUT says it finished, the scoreboard predicts the result and compares it
- The `#1` is not decoration: without it the signals get read at the same instant `done` goes up and a delta race can be swallowed
- `no_op` and `rst_op` do not raise `done`, so this block should not run with them. The `if` is defensive —with no prediction for those two it would compare garbage— and inside goes `chequeadas++`, the other half of the counter
- The covergroup —the third leg— is the section that follows

The conventional testbench

How do you know the scoreboard checks anything?

```
FAILED: A: 15  B: ce  op: xor_op result: da ovf: 0
```

- A thousand operations and not one error. Now **the same testbench with the DUT broken on purpose**: `VTALU_BUG=1 bash run.sh` flips bit 0 of the result
- It fails on the first comparison. That —and not the earlier `PASS`— is what proves the scoreboard looks at the result
- A scoreboard that **never saw an error** is not proven: it may have compared against itself, or compared nothing at all
- `make mutante` does this with the three testbenches of days 1 and 2, and fails if any of them does **not** fail. It is the first answer to the question of the day

The conventional testbench

Unit summary

- The testbench has **three parts** because the plan has three columns to fill: stimulus, self-checking and measure. Today they are loose in one file
- The stimulus **biases the random towards the edges** — `00` and `FF` — because uniform randomness almost never visits them
- The `#1` before reading the signals is not decoration: without it you read in the very instant the DUT writes, and the result is undefined
- A `PASS` proves nothing until you have seen the `FAILED`: **`VTALU_BUG=1`** is the way to see it, and `make mutante` the way not to forget
- And what to look at for interfaces and BFM: **the tester knows the protocol**. It moves `start` and waits for `done` by hand

Functional coverage

When are you done verifying?

- *Code coverage* —lines, branches, toggle— the simulator gives you for free. It says which RTL got **executed**
- *Functional coverage* you write yourself. It says which **scenarios of the spec** went through the DUT
- They are not the same thing, and the first one misleads: a test that always sends `add_op` with `A=B=8'h01` executes almost every line of the adder and verified nothing
- Functional coverage is the *contract* between the spec and the testbench: if a case of the spec has no bin, nobody is going to find out it is missing

Functional coverage

From the verification plan to the covergroup

The six rows of the plan from the previous section, translated:

From the plan	How it gets measured
the six operations	`coverpoint op_set` with one bin per operation
operands at `00` and at FF	`cross` of A, B and the operation
operate after a reset	transition bin `rst_op => op`
a mult after a single-cycle one	transition bin
the same operation twice in a row	repetition bin `[* 2]`
subtract too much: `A < B`, and `ovf` goes up	`coverpoint borrow` with the `hubo_borrow` bin

- The six rows stop being good intentions: each one is a line of code, and the simulator tells you which ones have not happened yet

Functional coverage

Anatomy: it is three steps, not one

```
covergroup op_cov;                      // 1. WHAT gets measured
   coverpoint op_set;
endgroup

op_cov oc;                              // 2. it gets INSTANTIATED: it is a type, like a class
initial oc = new();

always @(posedge clk) oc.sample();      // 3. WHEN it gets counted
```

- A `covergroup` is a **type**: declaring it measures nothing, it has to be `new()`ed
- `sample()` is the one that counts. Without `sample()` the report reads **0 %** and the code compiles just the same
- *Where* to sample is a design decision. Here, the edge the **DUT reads on**, never the same one the tester writes on; when the TB has monitors it will be *"every time the monitor sees a command"*

Functional coverage

bins: the buckets that have to be filled

```
coverpoint op_set;                    // automatic bins: one per value of the enum

coverpoint op_set {
   bins single_cycle[] = {[add_op : xor_op], rst_op, no_op};  // [] -> one per value
   bins multi_cycle    = {mul_op};                            // no [] -> a single one
}
```

- Without `{ }`, SystemVerilog creates one bin per value. That works for an `enum`; for an `int` the tool chops the range into **64** buckets — `auto_bin_max` — that correspond to no row of the plan
- The brackets `[]` **hand out**: one bucket per value of the range. Without them, every value falls into the same bucket
- A coverpoint is covered when **all** of its bins have at least one hit. A bin that cannot be filled pins your number down forever

Functional coverage

The VTALU covergroup

```
   //op_cov: make sure every operation, and every interaction between them, is covered
   covergroup op_cov;
      coverpoint op_set {
         bins single_cycle[] = {[add_op : xor_op], rst_op,no_op};
         bins multi_cycle = {mul_op};

         // => are transitional coverage. From => To
         // Transition bins. Verilator 5.052 does NOT implement them yet: any form
         // with "=>" or "[* n]" aborts with an Internal Error. Minimal repro in
         // code/verilator/repro-cg-transition.sv. They stay because they are part
         // of the topic; when Verilator supports them, drop the ifndef, nothing else.
`ifndef VERILATOR
         bins opn_rst[] = ([add_op:mul_op] => rst_op);
         bins rst_opn[] = (rst_op => [add_op:mul_op]);

         bins sngl_mul[] = ([add_op:xor_op],no_op => mul_op);
         bins mul_sngl[] = (mul_op => [add_op:xor_op], no_op);

         bins twoops[] = ([add_op:mul_op] [* 2]);
         bins manymult = (mul_op [* 3:5]);
`endif
      }
   endgroup
```

- `single_cycle[]` generates six bins —one per operation—, `multi_cycle` a single one
- What is inside `` `ifndef VERILATOR `` are the transition bins, which come further on in this same section

Functional coverage

ignore_bins: saying out loud what does not get covered

```
// Check the corners where the inputs are all 0s or all 1s
all_ops : coverpoint op_set {
   ignore_bins null_ops = {rst_op, no_op};}

a_leg: coverpoint A {
   bins zeros = {8'h00};
   bins others= {[8'h01:8'hFE]};
   bins ones  = {8'hFF};
}

b_leg: coverpoint B {
   bins zeros = {8'h00};
   bins others= {[8'h01:8'hFE]};
   bins ones  = {8'hFF};
}
// VTALU rev2: the borrow of the subtraction, the row of the plan that the
// scoreboard can only close by looking at TWO outputs. Without this bin, a
// subtraction that never went negative looks exactly like one that did.
borrow: coverpoint ((op_set == sub_op) && (A < B)) {
   bins hubo_borrow = {1};
   ignore_bins resto = {0};
}
```

- `all_ops` **ignores** `rst_op` and `no_op`: it makes no sense to ask for "an addition with the operands at 0x00" when the operation is a reset
- `a_leg` and `b_leg` bring 256 values down to the three buckets that matter: all zeros, the middle, all ones
- An impossible bin you leave in place keeps your coverage pinned below 100 % and nobody knows why. `ignore_bins` turns it into a written decision

Functional coverage

cross: the combinations, and how not to drown

```
op_00_FF : cross a_leg, b_leg, all_ops;    // 3 x 3 x 5 = 45 bins, almost all meaningless
```

- A `cross` is the **cartesian product** of its coverpoints: it grows fast and most of the combinations are in nobody's spec
- `binsof(x) intersect {v}` reads as *"the bins of `x` that contain `v`"*
- They combine with `&&` and `||`, and what is left over gets thrown away with `ignore_bins`
- That way the 45 bins come down to the 11 the plan actually asked for

Functional coverage

The VTALU cross

```
bins mul_00 = binsof (all_ops) intersect {mul_op} &&
              (binsof (a_leg.zeros) || binsof (b_leg.zeros));

bins mul_FF = binsof (all_ops) intersect {mul_op} &&
              (binsof (a_leg.ones) || binsof (b_leg.ones));

bins mul_max = binsof (all_ops) intersect {mul_op} &&
               (binsof (a_leg.ones) && binsof (b_leg.ones));

ignore_bins others_only =
               binsof(a_leg.others) && binsof(b_leg.others);
```

- `mul_00` reads straight through: *a multiplication where A **or** B is 0x00*. The other eight bins of the cross are these two lines with another `op`
- `mul_max` is the only one with `&&`: it asks for **both** legs at 0xFF: the **maximum product**, `FF` × `FF` = `FE01`. It does not overflow — 8 bits by 8 fit in 16
- Verilator 5.052 **ignores** `binsof` / `intersect` (`%Warning-COVERIGN`) and measures the whole cross: that is why the number is not a commercial tool's

Functional coverage

Transition bins: when the order matters

```
bins opn_rst[] = ([add_op:mul_op] => rst_op);   // A and then B
bins sngl_mul[] = ([add_op:xor_op], no_op => mul_op);
bins twoops[]  = ([add_op:mul_op] [* 2]);       // twice in a row
bins manymult  = (mul_op [* 3:5]);              // between 3 and 5 times in a row
```

- Three rows of the plan are not values, they are **sequences**: "after a reset", "a mult after a single-cycle one"
- Verilator 5.052 still **does not compile them** (Internal Error). In the code they are inside `` `ifndef VERILATOR ``: they are part of the topic and read just the same
- Minimal repro in `code/verilator/repro-cg-transition.sv`. The day Verilator supports them, the `ifndef` gets deleted and nothing else

Functional coverage

The knobs: when a bin counts as covered

```
coverpoint op_set {
   option.at_least = 10;                  // a bin only counts at 10 hits
   bins single[] = {[add_op:xor_op]};
}
coverpoint A { option.auto_bin_max = 8; } // 8 automatic buckets, not 64
```

- `option.at_least` is the one that matters most: by default it is **1**, so a bin that happened **once** already shows up as covered. The corner case that came up once in a thousand runs is not verified, and the 100 % says it is
- `option.auto_bin_max` bounds the automatic bins. Without `{ }` and without this knob, an 8-bit coverpoint does not give 256 buckets: it gives **64**, handed out by the tool and not by the spec
- `option.weight` changes how much a coverpoint weighs in the total, and `type_option.merge_instances` adds every instance into a single number instead of reporting them separately
- Verilator 5.052 honours `auto_bin_max`, and `at_least` **only written in the coverpoint**: in the covergroup it ignores it without warning. It is in `docs/en/verilator.md`

Functional coverage

Reading the number

```
$ make u2/convencional
Coverage Summary:
  covergroup : 86.8% (66/76)
```

- 66 of 76 bins filled, with **1000 randomizations** and **615 operations on the bus** —`no_op` and `rst_op` get drawn and never sent, and the testbench prints that right before the Verilator report—. The other rows come out `0/0`: `--coverage-user` leaves code coverage out
- The 10 that are missing are **a single value**: `all_ops.auto_5` and its nine crosses. It is `3'b110`, which the enum **does not have** — Verilator hands out the automatic bins by the base type, `bit [2:0]`, and not by enum member
- Which means no row of the plan was left uncovered, and on Questa this gives 100 %. The number is not the goal: the question that helps is **which** bin is missing, and only the per-bin report says that, `obj_dir/top/coverage.dat`
- Run → look at what is missing → write the directed test → run again. That is called *coverage closure*, and it is what a verification engineer spends the day on

Functional coverage

The verification plan, whole and in one table

#	Feature	Scenario	Stimulus	Check	Measure
1	ALU	the six operations	random	scoreboard	`coverpoint op_set`
2	ALU	operands at `00` and at FF	`dist` biased to the edges	scoreboard	cross `op_00_FF`
3	mult	maximum product: FF × FF	directed case	scoreboard, 16 bits	bin `mul_max`
4	reset	operate after a reset	`rst_op` interleaved	scoreboard	bin `rst_op => op`
5	mult	a mult after a single-cycle one	random	scoreboard	transition bin
6	ALU	the same operation twice in a row	random	scoreboard	bin `[* 2]`

#	Feature	Scenario	Stimulus	Check	Measure
7	sub	subtract too much: `A < B`, and `ovf` goes up	random	scoreboard, **two outputs**	bin `hubo_borrow`
8	sub	`A == B`: the result is 0 and `ovf` does **not** go up	random	scoreboard, two outputs	bin `sub_00`/`sub_FF`
9	ovf	`ovf` does not go up for any other operation	random	assertion	`c_ovf`
10	protocol	the operands are not touched with `start` up	random	assertion	`cover property`
11	protocol	done arrives, and before 5 cycles	random	assertion	`cover property`
12	protocol	no_op is the only one that does not answer	random	assertion	`cover property`

- The three right-hand columns are **the three parts of the testbench** of this section. What ties them together is the plan, and until now you had not seen it whole
- The two subtraction rows are the ones that force the scoreboard to look at **two outputs**: if it only compares `result`, it passes green with `ovf` stuck at zero
- The **last four** are checked by no scoreboard: they are `ovf` and **protocol** rules, and get checked where they happen —the assertions
- The twelve rows are numbered the same as in **`docs/en/verification-plan.md`**, which also says which file each one lives in and carries the capstone template

Functional coverage

Summary of the unit

- **Code coverage ≠ functional coverage.** The first one measures which RTL got executed; the second, which scenarios of the spec happened
- A `covergroup` is a **type**: it has to be instantiated with `new()` and **somebody has to call** `sample()`. Without that the report reads 0 % without a warning
- The **bins** are the buckets. `[]` hands out one per value; without brackets, the whole range goes into a single one
- `ignore_bins` **is executable documentation**: it says out loud what does not get covered and why. An impossible bin you leave in place pins your number down
- A `cross` is the cartesian product and grows extremely fast: without bounding it, the coverage never closes
- `option.at_least` **is 1 by default**: a bin that happened once already shows up as covered, and a rare case with one hit is not verified
- The number is not the goal. The goal is the **plan**; the coverage only says how much of the plan was met

Interfaces and BFM

First: the signals stop being loose

```
interface vtalu_bfm;
   // brings in operation_t. e.g. (no_op  = 3'b000)
   import vtalu_pkg::*;

   byte unsigned        A;
   byte unsigned        B;
   bit                  clk;
   bit                  reset_n;
   wire          [ 2:0] op;
   bit                  start;
   wire                 done;
   wire          [15:0] result;
   wire                 ovf;
   operation_t          op_set;

   assign op = op_set;

   // The BFM is a first step towards modularity
   // It takes care of driving every low-level signal
   // and keeps the protocol in one single place

   // DUT clock
   initial begin
      clk = 0;
      forever begin
         #10;
         clk = ~clk;
      end
   end
```

- An `interface` is a **bundle** of signals with a name of its own. Nine wires that used to be declared in the `top` now live together
- The clock gets generated inside: the interface is not a wire, it is a model of the bus
- Connecting the DUT becomes `.A(bfm.A)`, `.clk(bfm.clk)`... and a new signal gets declared **once**: the modules that use it see it appear without touching their ports
- What is not needed here and you will see in somebody else's RTL: the `modport`, which is the same interface with the directions set for one side of the wire

Interfaces and BFM

Then: the protocol hides inside a task

```
// The protocol handling can now be changed in one single place.
// Fix it here once, and the fix propagates through the rest of the code
task send_op(input byte iA, input byte iB, input operation_t iop,
             output shortint alu_result);

   op_set = iop;

   if (iop == rst_op) begin
      @(posedge clk);
      reset_n = 1'b0;
      start = 1'b0;
      @(posedge clk);
      #1;
      reset_n = 1'b1;
   end else begin
      @(negedge clk);
      A = iA;
      B = iB;
      start = 1'b1;
      if (iop == no_op) begin
         @(posedge clk);
         #1;
         start = 1'b0;
      end else begin
         do @(negedge clk); while (done == 0);
         start = 1'b0;
      end
   end

endtask : send_op
```

- `send_op(A, B, op, result)` translates *"do an addition"* into the signal wiggling the DUT expects. That is a **Bus Functional Model**
- Inside it is all the fine print of the spec: the `reset_n` pulse, the `no_op` that does not wait for `done`, and the `do ... while (done == 0)` of the rest
- From here on **nobody moves `bfm.start` by hand**. Whoever wants an operation asks for it

Interfaces and BFM

The top, with the BFM in the middle

- The testbench is still split into the same three pieces —tester, scoreboard and coverage—, and not one of them changed its name
- What changed is **how they connect to the DUT**: there are no loose signals travelling any more, there is a single interface the three of them share

```
module top;
   vtalu_bfm bfm ();
   // Tester: generates the stimulus
   tester tester_i (bfm);
   // Coverage: provides functional coverage
   coverage coverage_i (bfm);
   // ScoreBoard: checks the results
   scoreboard scoreboard_i (bfm);

   vtalu DUT (
       .A(bfm.A),
       .B(bfm.B),
       .op(bfm.op),
       .clk(bfm.clk),
       .reset_n(bfm.reset_n),
       .start(bfm.start),
       .done(bfm.done),
       .ovf(bfm.ovf),
       .result(bfm.result)
   );
endmodule : top
```

Interfaces and BFM

Scoreboard: the same check, now talking to the BFM

```
module scoreboard (
    vtalu_bfm bfm
);
   import vtalu_pkg::*;

   always @(posedge bfm.done) begin
      shortint predicted_result;
      bit      predicted_ovf;
      #1;
      case (bfm.op_set)
         add_op: predicted_result = bfm.A + bfm.B;
         sub_op: predicted_result = bfm.A - bfm.B;
         and_op: predicted_result = bfm.A & bfm.B;
         xor_op: predicted_result = bfm.A ^ bfm.B;
         mul_op: predicted_result = bfm.A * bfm.B;
      endcase  // case (op_set)

      // ovf belongs to sub and to nobody else: with 8-bit inputs and a 16-bit
      // output, neither the addition nor the multiplication can overflow.
      predicted_ovf = (bfm.op_set == sub_op) && (bfm.A < bfm.B);

      if ((bfm.op_set != no_op) && (bfm.op_set != rst_op))
         if (predicted_result != bfm.result || predicted_ovf != bfm.ovf)
            $error(
                "FAILED: A: %0h  B: %0h  op: %s result: %0h ovf: %0b",
                bfm.A,
                bfm.B,
                bfm.op_set.name(),
                bfm.result,
                bfm.ovf
            );

   end
endmodule : scoreboard
```

- It is the conventional testbench's `scoreboard`: it predicts and compares with the same logic
- The only thing that changed is where it gets the signals from: they used to be variables of the same module, now they are `bfm.A`, `bfm.B`, `bfm.op_set`
- And that is why it is a **separate module** now: it can live in its own file because it shares nothing with the tester except the interface

Interfaces and BFM

Tester: it asks for operations, it does not move wires

```
initial begin
   byte unsigned iA;
   byte unsigned iB;
   operation_t   op_set;
   shortint      result;

   bfm.reset_alu();
   repeat (1000) begin : random_loop
      op_set = get_op();
      iA = get_data();
      iB = get_data();
      bfm.send_op(iA, iB, op_set, result);
   end : random_loop
   $finish;
end  // initial begin
```

- The tester no longer touches `start` or waits for `done`: it calls `bfm.send_op(...)` and forgets about the protocol
- The handshake ended up **written once**, inside the BFM. If tomorrow the DUT asks for two cycles of `start`, one file gets touched
- That is the definition of *Bus Functional Model*: the task translates *"do an addition"* into the signal wiggling the DUT expects

Interfaces and BFM

The protocol rule, written in the simulator's language

```
// IMMEDIATE: a statement. It checks the present, where the thread goes by.
assert (op_set != no_op || done == 0)
   else $error("no_op should not raise done");

// CONCURRENT: a declaration with a clock. It plugs itself in once and
// checks EVERY edge of the simulation, on its own, forever.
a_operandos_estables :
assert property (@(negedge clk) start |=> $stable(A) && $stable(B));
```

- The rule —*while* `start` *is up, the operands are not touched*— is already written **twice**: in prose in the spec, and wrapped inside `send_op()` here
- An **assertion** is that same sentence a third time, but **executable**: it does not merely comply with the rule, it **checks** it
- And it checks it where it happens —in the interface, on the exact edge— and not at the end, comparing results

Interfaces and BFM

Where each thing gets checked

- The conventional testbench's **scoreboard** checks the **result**: `A op B` against what the DUT returned. It is the half this course develops to the end
- An **assertion** checks the **protocol**: who moves, when, and for how many cycles. It is the other half
- A DUT that returns `result` correctly but drops `done` one cycle early **passes every test** of the next six days. Nobody is watching it
- It is seen whole in **the assertions section**, with the same VTALU and the same `run.sh`

Interfaces and BFM

The hole the BFM leaves: the timing

```
// 1. Sample AT the edge. The DUT always_ff updates d_out with a
//    nonblocking assignment, which lands after this initial already ran:
//    you read the OLD value.
@(posedge clk);
en_el_flanco = d_out;

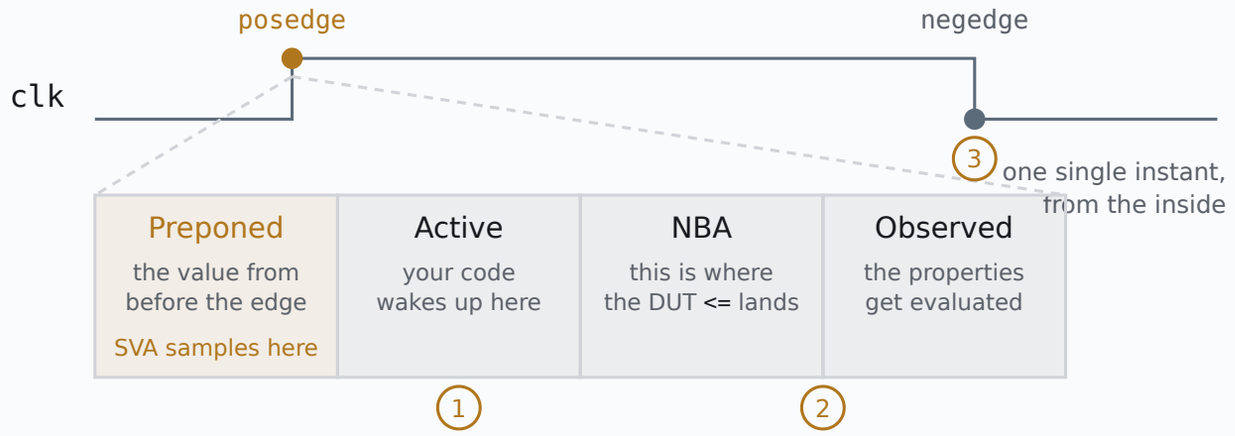
// 2. The same edge, one delta later. The nonblocking assignment landed
//    already and you read the NEW value. That "#1" is visible nowhere and
//    changes the result.
@(posedge clk);
#1;
un_delta_despues = d_out;

// 3. The opposite edge: the trick the whole course BFM uses. It works,
//    but it asks whoever reads it to know why.
@(negedge clk);
en_el_flanco_opuesto = d_out;
```

- The BFM wrapped **the protocol**. What it did **not** wrap is *when* it drives and *when* it samples: that is still decided task by task
- Three ways of reading the same data: `0`, `11` and `11`. Two values, but **three mechanisms**, and the middle one depends on a `#1` that is visible nowhere
- The `@(negedge clk)` of our BFM is the third one. It works — and it asks whoever reads it to know why

Interfaces and BFM

Why the edge alone is not enough: the regions



Three ways to read the same data, with the DUT computing `d_in + 1 = 11`:

1	`@(posedge clk);`	read in Active, before the NBA	`0`
2	`@(posedge clk); #1;`	the NBA already landed	`11`
3	`@(negedge clk);`	the opposite edge, half a cycle later	`11`

The `#1` in the middle works, and it works by accident: it never says which race it avoids.

- An edge is not an indivisible instant: inside it there are **regions**, in a fixed order
- The `<=` of the `always_ff` **lands in NBA**: whoever reads in *Active* reads the old value
- The `#1` crosses the NBA, the opposite edge crosses it with room to spare, and SVA needs neither of the two: it samples in *preponed*, before everything

Interfaces and BFM

The clocking block: the edge and the delta, declared once

```
interface reg_bfm (input bit clk);
   byte unsigned d_in, d_out;

   // The only two timing lines in the testbench:
   //   input  #1step  samples the STABLE value right before the edge, which is
   //                  what the hardware sees. Never what the nonblocking
   //                  assignment just wrote on that same edge.
   //   output #0      drives ON the edge, in the nonblocking region, so the DUT
   //                  does not see it until the next edge.
   clocking cb @(posedge clk);
      default input #1step output #0;
      output d_in;
      input  d_out;
   endclocking

   modport tb (clocking cb);
endinterface : reg_bfm
```

- `input #1step` — sample the value **stable just before** the edge, which is the one the hardware sees. Never the one the nonblocking assignment just wrote
- `output #0` — drive **on** the edge, with the nonblocking assignments, so the DUT does not see it until the next edge
- Driving becomes `cb.d_in <= value`; sampling, reading `cb.d_out`; waiting, `@(cb)`. **No `#1` in any task**
- And the value read is the same at any point of the cycle: the race got closed by declaration, not by habit

Interfaces and BFM

Are they mandatory? No — and it is worth knowing why

The question	The answer
Can a correct TB be written **without** clocking blocks?	**Yes.** With NBA in the driver and scheduler discipline, there is no race
What should **a team** use?	**Use them** on synchronous interfaces — it is what the lowRISC/OpenTitan style guide says

- They are not UVM's: they are **SystemVerilog's**. UVM neither asks for them nor knows them
- **Dave Rich** —the reference at Verification Academy— has held since 2014 that they are *optional*: what avoids the race is understanding the scheduler, not the clocking block
- The two positions answer different questions: one is **language correctness**, the other is **team policy**. Both are right

Interfaces and BFM

The price: two names for the same wire

```
// The two reads, at the SAME instant, on the SAME wire.
por_cb = bfm.cb.d_out;
crudo  = bfm.d_out;

$display("at the same instant, on the same wire:");
$display("  bfm.cb.d_out (what the clocking block sampled) : %0d", por_cb);
$display("  bfm.d_out    (the live wire)                   : %0d", crudo);
$display("");
$display("They differ by one cycle, and no warning says so. A monitor that");
$display("reads through the clocking block and a scoreboard that reads the");
$display("raw wire do not fail every time: they fail when the data changes.");
```

- With the signal inside a clocking block there are **two** ways of reading it: `bfm.d_out` is the live wire, `bfm.cb.d_out` is what got sampled `#1step` before the edge
- On a signal that changes every cycle they **differ by one cycle**: the example prints `3` and `4`, at the same instant and on the same wire
- Nobody warns you. It compiles, it runs, and the scoreboard fails **only when the data changes**
- The rule: if a signal went into the clocking block, **the whole protocol reads it through there** — and you wait on `@(bfm.cb)`, not on `@(posedge bfm.clk)`

Interfaces and BFM

Summary of the unit

- An `interface` is a **bundle of named signals**: adding a wire stops being touching six port lists
- A **BFM** is the interface **plus the protocol**: `send_op(A, B, op, result)` translates *"do an addition"* into the signal wiggling
- All the fine print of the spec —the reset pulse, waiting for `done`, the `no_op` that does not answer— ends up written **once**
- From here on **nobody moves** `bfm.start` **by hand**. Whoever wants an operation asks for it
- The tester, the scoreboard and the coverage become **separate modules**, each one in its file, connected by the BFM
- And this does not get abandoned: the agent's driver keeps calling **this same task**
- The rule the BFM **complies with** can also be **checked**, with an assertion that lives in this same interface. That is the assertions section
- And the `clocking block` closes the last hole: the edge and the delta stop being decided task by task and get declared **once**

Review · Day 1

1 of 7 · The question of the day

A thousand random operations, the scoreboard reported no error and the log ends in `PASS`. What are you missing to say the DUT is verified?

a. Nothing: a thousand operations without an error is a verified DUT

b. Running more seeds until code coverage reaches 100 %

c. Knowing what the log would have said with a bug inside: run it with the mutated DUT ✓

d. Replacing the scoreboard with assertions, which check the protocol on the exact edge and not at the end

Watch it fail — the `PASS` of a scoreboard that never saw an error says nothing: it may not have compared. `VTALU_BUG=1` flips one bit of the result and the testbench has to fail; `make mutante` demands it for the three testbenches of days 1 and 2.

Review · Day 1

2 of 7 · The ALU spec

While the VTALU is running an operation, what do `start` and the operands have to do?

a — `start` drops right away: it is a start pulse, and the DUT already latched everything

b — It makes no difference: the DUT registers them on the first edge

c — The operands have to change on every cycle

d — `start` at 1 and the operands still until `done` goes up ✓

Stable until `done` — it is the DUT protocol, and it is exactly the reason the BFM exists: to wrap that rule in a single place so that no test forgets it. The first distractor describes a real protocol —pulse start, latched operands— that this DUT does not have.

Review · Day 1

3 of 7 · Functional coverage

The test runs 1000 random operations and the *code* coverage reads 100 %. What does that tell you about the verification?

a That the DUT is verified and the verification plan can be closed

b That the testbench has no bugs

c That few scenarios are missing: 100 % already walked the whole design

d Very little: it measures the RTL that ran, not the spec ✓

Very little — code coverage measures the DUT; functional coverage measures the spec. A feature the designer never wrote gives 100 % of lines and 0 % of what matters, and the report is not going to tell you.

Review · Day 1

4 of 7 · covergroup

You declare a `covergroup`, you `new()` it, you run a thousand operations and the report reads 0 %. What is the first thing to look at?

- **a** That nobody is calling `sample()` ✓
- **b** That the bins are badly defined and match no value at all
- **c** That the DUT is not answering
- **d** That `ignore_bins` are missing

The `sample()` — a covergroup does not sample itself: somebody has to call it, on the edge or when a transaction arrives. Without that call the code compiles, runs, and the report reads 0 without a single warning.

Review · Day 1

5 of 7 · Interfaces and BFM

What does the testbench gain when the protocol moves into a BFM?

a It stops talking in signals and starts talking in operations ✓

b It simulates faster: one BFM task costs the simulator less than moving wires

c The testbench becomes synthesizable

d You save having to declare a `clk`

It stops talking in signals — the BFM translates *one operation* into *a handshake of signals*. The tester never touches a wire again; the scoreboard and the coverage keep reading them, through the interface, until a monitor sends them transactions. It is the first step towards UVM.

Review · Day 1

6 of 7 · The verification plan

In the verification plan, what goes in the *Measure* column?

a How long the scenario takes to run, so the regression can be estimated

b The name of the testbench file that covers that row

c Which bin gets filled when the scenario happens ✓

d How many times the test has to be run to call it covered

Which bin gets filled — the five columns are *Feature*, *Scenario*, *Stimulus*, *Check* and *Measure*, and this is the one that costs the most: it forces you to decide **beforehand** what is going to be counted. If it is left empty, nobody is going to find out that the scenario never happened — a case the random never touched and that has no bin is indistinguishable from one that happened a thousand times.

Review · Day 1

7 of 7 · clocking block

Are clocking blocks needed to write a UVM testbench without races?

a. Yes: without a clocking block, driver and DUT always compete on the same edge

b. Yes, and they are also part of the UVM library

c. No: NBA in the driver plus scheduler discipline is enough ✓

d. No: the `uvm_driver` replaces them, since it already samples in the right region

They are not needed, and they are worth using anyway — they belong to **SystemVerilog**, not to UVM, and no `uvm_driver` samples for you. What avoids the race is understanding the scheduler: a driver that drives with `<=` against a DUT that registers with `<=` is already deterministic. The clocking block does not replace that understanding, it **wraps** it — and that is where it pays off: reusable agents, VIP, gate-level and protocols with setup/hold in the spec.

Exercise · Day 1 · 1 of 2

A new operation, end to end

```
$ cd code/ejercicios/d1 && bash run.sh
```

- The VTALU has a **free opcode**, `3'b110`. Teach it to shift, and teach the TB that it has to verify it too
- It gets touched in **three** places: the RTL, the stimulus with its check, and **the measure** — the bin that does not exist today
- The `run.sh` fails until it prints `EXERCISE OK` **and the report closes with 77 covered bins**; the solution is right next to it

Exercise · Day 1 · 2 of 2

The waves: when the log is not enough

```
$ cd code/ejercicios/d1b && bash run.sh
```

- The run aborts with **a single line**: `FAILED: A: e5 B: 0 op: mul_op result: fe01 ovf: 0`. `e5 * 00` is `0`, not `fe01`. The scoreboard is right, and so is the DUT: what got it wrong is the BFM's **handshake**
- That is where what the log gives you ends. The `run.sh` leaves **`ondas.vcd`** next to it: `gtkwave ondas.vcd`
- **Stage 1:** two times that only exist in the viewer, in `respuesta.txt`. **Stage 2:** the line of the BFM that explains them
- It is the only exercise of the course that is **not solved by reading the log**

Six out of seven, again

Closing the day

- The regression that sent the bug to the fab said `PASS`. Today you have three ways of knowing whether that `PASS` was worth anything, and all three are columns of one table
- **What had to be tested?** A plan: one row per scenario, written before the testbench, from the spec
- **Who said it was right?** A scoreboard that predicts and compares — and that you watched fail with a bug inside
- **How much of the plan passed?** A `covergroup` with one bin per row, and the per-bin report, not the percentage
- The tester already asks for operations instead of moving wires: that was the BFM. What you still cannot do is **change the stimulus without editing the file**. Tomorrow is that

Yesterday we left...

Where the testbench stands

- A testbench that **works**: it sends stimulus, checks the result and measures functional coverage. The verification plan says which row each thing closes
- The BFM took the signals inside an `interface`, and the testbench went on to call `send_op()` instead of moving wires
- And one problem was left open: **changing the stimulus means editing the file**. A tester that only multiplies is copying the one that is there and deleting five lines

Agenda

Day 2 · ≈ 4 h · unit 3 · the OOP UVM takes for granted

- Classes and extensions
- Polymorphism
- Static variables and methods
- Parameterized classes
- The factory pattern
- A testbench without a single module

By the end of the day you can:

- **Extend** a class and change its behaviour **without copying it**, and say what `super` does
- **Change** the class a testbench uses **without touching** the code that uses it, with a factory
- **Write** the day 1 testbench without a single module

Classes and extensions

Why OOP in a testbench?

- The conventional testbench is a **module**: it gets elaborated once, it exists from `time 0` until the `$finish`, and it is always the same one
- A stimulus looks nothing like that: it shows up, it travels through the testbench, it gets compared and it gets thrown away. That is not a module, it is an **object**
- Modules = *structure*, they get elaborated. Classes = *data and behaviour*, they get created and destroyed while the simulation runs
- And it is not optional: **UVM is a class library**. `uvm_test`, `uvm_component`, `uvm_object`. Without this, the rest of the course cannot be read

Classes and extensions

What a `struct` cannot do

```
typedef struct {
   int length;
   int width;
} rectangle_struct;

typedef struct {int side;} square_struct;

module top_struct;
   rectangle_struct rectangle_s;
   square_struct    square_s;
   initial begin
      rectangle_s.length = 50;
      rectangle_s.width = 20;
      $display("rectangle area: %0d", rectangle_s.length * rectangle_s.width);
      square_s.side = 50;
      $display("square area: %0d", square_s.side ** 2);
   end
endmodule
```

- The `struct` holds data and **nothing else**: the area calculation has to be written outside, and again in every place where it is needed
- `square_struct` has no relationship at all with `rectangle_struct`, even though a square is a particular case of a rectangle
- It does not randomize itself, it does not compare itself, it does not print itself

Classes and extensions

The first class: data, methods and constructor

```
class rectangle;
   int length;
   int width;

   function new(int l, int w);
      length = l;
      width = w;
   endfunction

   function int area;
      return length * width;
   endfunction
endclass : rectangle

module top_rectangle;

   rectangle rectangle_h;

   initial begin

      rectangle_h = new(.l(50), .w(20));

      $display("rectangle area: %0d", rectangle_h.area());

   end
endmodule : top_rectangle
```

- The class puts the **data** (`length`, `width`) together with **what gets done with it** (`area()`). The area gets computed in a single place
- `new()` is the *constructor*: it runs once, when the object gets created
- `.l(50)` is argument passing **by name**. UVM uses it all the time

Classes and extensions

A handle and an object are not the same thing

```
rectangle rectangle_h;              // 1. a HANDLE. No object yet: it is null
rectangle_h = new(.l(50), .w(20));  // 2. new() creates the OBJECT and returns its handle
```

- Declaring a handle **creates nothing**. Until the `new()` it is `null`, and using it there blows up at simulation time, not at compile time
- The handle looks like a pointer, but it takes no arithmetic: there is no `h + 1`
- The memory of a `struct` the simulator reserves the moment it sees it; that of an object, only at the `new()`
- And there is no `free()`: when no handle is left pointing at it, the object gets collected on its own

Classes and extensions

Extending: `extends` and `super`

```
class rectangle;
   // Data members
   int length;
   int width;

   function new(int l, int w);
      length = l;
      width = w;
   endfunction

   function int area();
      return length * width;
   endfunction
endclass

// Extending classes!
class square extends rectangle;
   // Override Constructor
   function new(int side);
      // super: the call to the parent method
      super.new(.l(side), .w(side));
   endfunction

endclass
```

- `square extends rectangle` inherits `length`, `width` and `area()` **without copying a single line**. A square is a rectangle with a constraint, and the code says it that way
- The child's constructor calls `super.new()`: the top part of the object also has to be built, and it goes **first**
- Extending is not copying: if tomorrow `area()` changes, it changes for both

Classes and extensions

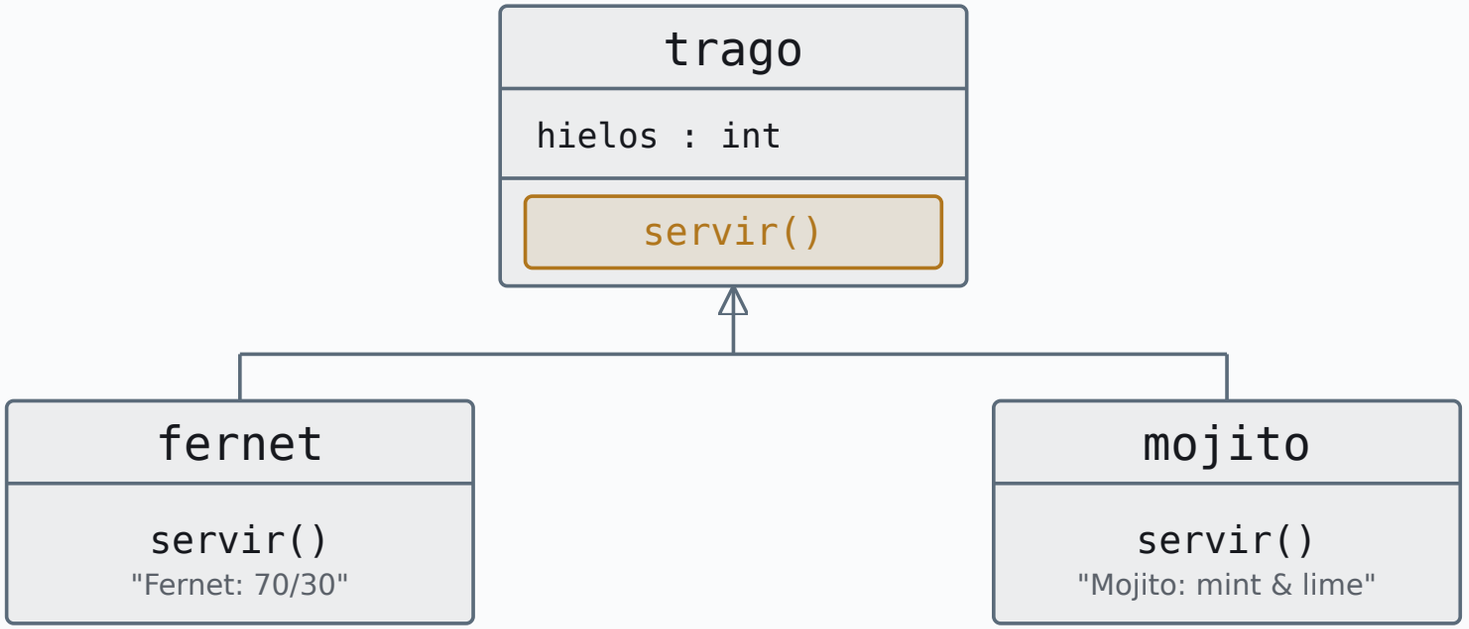
Where all of this comes back

What you have just seen	What it is called in UVM
class with data and methods	`uvm_object` — the *transactions* of day 5
class that also lives in the TB tree	`uvm_component` — driver, monitor, scoreboard
`new()`	the constructor, with `name` (and `parent` if it is a component)
`extends`	every test, every tester, every agent of the course
`super.new()`	the mandatory first line of every UVM constructor

- One piece is missing: storing a `square` object in a `rectangle` variable. That is the unit that follows

Polymorphism

A `trago` variable, a `fernet` object: which `servir()` runs?




```
trago_h = fernet_h; trago_h.servir();
```

Without virtual the variable type wins (trago, $fatal). With virtual the object wins (fernet).

- A `fernet` **is** a `trago`, so storing it in a `trago` variable compiles and is legal
- The question is the other one: when you call the method through that variable, does SystemVerilog look at the **type of the variable** or at the **type of the object**?
- The default answer is the one you do not want, and the word that changes it is a single one

Polymorphism

The three classes: a `trago` that does not know how to serve itself

```
class trago;
   int hielos = -1;

   function new(int a);
      hielos = a;
   endfunction : new

   function void servir();
      $fatal(1, "A generic trago cannot be served: order a real one.");
   endfunction : servir

endclass : trago

class fernet extends trago;

   function new(int hielos);
      super.new(hielos);
   endfunction : new

   function void servir();
      $display("Fernet: 70/30, and the coke last");
   endfunction : servir

endclass : fernet
```

- `trago` defines `servir()` with a `$fatal`: the base class does not know what it gets served with, and it says so loudly
- `fernet` and `mojito` extend `trago` and **redefine** `servir()`, each one with its own
- `mojito` —which does not fit on the screen— is the same as `fernet`, with another `$display`

Polymorphism

Without `virtual` the variable rules, and that breaks

```
fernet_h = new(15);
fernet_h.servir();
$display("The fernet has %0d ice cubes", fernet_h.hielos);

mojito_h = new(1);
mojito_h.servir();
$display("The mojito has %0d ice cubes", mojito_h.hielos);

// The variable is trago, the object is fernet.
trago_h = fernet_h;
trago_h.servir();  // ** Fatal: A generic trago cannot be served
$display("The trago has %0d ice cubes", trago_h.hielos);

trago_h = mojito_h;
trago_h.servir();
$display("The trago has %0d ice cubes", trago_h.hielos);
```

- The first two calls work: the variable `fernet_h` is of type `fernet` and so is the object
- The third one **explodes**. `trago_h` points at a `fernet`, but since the method is not virtual, SystemVerilog resolves at **compile time** by the type of the variable, and calls `trago`'s `servir()`
- It is called *static binding*: the decision of which code to run is already taken before the object exists

Polymorphism

Virtual methods: one word, and now the object rules

```
virtual function void servir();
   $fatal(1, "A generic trago cannot be served: order a real one.");
endfunction : servir
```

- It is the **only** difference between `01-sin-virtual` and `02-virtual`: the keyword `virtual` on the method of the base class
- With that the call gets resolved at **simulation time**, looking at the object inside the variable — *dynamic binding*
- `trago_h.servir()` now serves a fernet or a mojito, without whoever wrote that line knowing which one they are going to get
- That is the whole idea: **whoever uses the object stops being the one who picks the type**

Polymorphism

Abstract classes: moving the error to compile time

- The `$fatal` of `trago` finds out **late**: it compiles, it runs, and it explodes in the middle of the simulation. In an overnight regression that is a lost run
- SystemVerilog allows declaring an **abstract class** — `virtual class` —: it only serves as a base, **it cannot be instantiated**
- Inside it can declare **pure virtual methods**: methods with no body
- Extending the class **forces** you to redefine them. If not, it is a **compile error**
- The error did not go away: it moved to a moment where it comes out cheap

Polymorphism

`pure virtual`: the same example, with no `$fatal` possible

```
// Abstract class: it only works as a base, it cannot be instantiated.
virtual class trago;
   int hielos = -1;

   function new(int a);
      hielos = a;
   endfunction : new

   // pure virtual: no body. Extending it forces you to redefine it.
   pure virtual function void servir();

endclass : trago
```

- `virtual class trago` no longer has a `servir()` that explodes: **it has no body**, and that is why there is nothing that can run wrong
- The commented-out line of the `top` — `trago_h = new(3)` — does not compile: an abstract class does not get instantiated
- If you delete `mojito`'s override, the compiler says *"Class 'mojito' extends 'trago' but is missing implementation for 'servir'"* and that is the end of it

Polymorphism

Summary of the unit

- **Without `virtual` the variable rules**, and it is decided at compile time. **With `virtual` the object rules**, and it gets resolved during simulation. One word, and it changes everything
- The underlying idea, the one that comes back in every section that follows: **whoever uses the object stops being the one who picks the type**
- In SystemVerilog `virtual` is **overloaded**: virtual method, virtual class and virtual interface are three different things — and in UVM you are going to write all three

Static variables

The global variable you can actually defend

- In a testbench there is always something there is **only one of**: the error counter, the list of in-flight transactions, the handle to the configuration
- The easy answer is a global variable, and it is the one nobody can debug afterwards: you do not know who wrote it, or from where, or when
- `static` is the same idea, but with a last name attached. The variable lives **in the class**, not in the object, and to touch it you have to name the class: `bandeja_de_fernet::vasos`
- That `::` is the whole argument. The day the value looks odd, a grep tells you exactly who put it there
- And it is not a loose OOP topic: the **factory** and the `uvm_config_db` are the two most important static variables of a UVM testbench

Static variables

- `bandeja_de_fernet` has no methods and no constructor: it is a class that exists **only** to be the place where the queue lives
- Nobody instantiates it. It gets named with the `::` operator, which tells the compiler *"look this variable up in the namespace of that class"*
- And there is the advantage over the global: `bandeja_de_fernet::vasos` says where it is declared. A loose `vasos` does not

```
class bandeja_de_fernet;
   // Static queue
   static fernet vasos[$];

endclass : bandeja_de_fernet

module top;
   initial begin
      fernet fernet_h;
      fernet_h = new(2, "the one at table 7");
      // :: operator
      //----------------
      //   The variable in the bandeja_de_fernet class is reached by
      //   naming the class and putting the :: operator before the name
      //   That tells the compiler we want a static variable inside
      //   the namespace of the class
      bandeja_de_fernet::vasos.push_back(fernet_h);
      fernet_h = new(3, "the one at table 4");
      bandeja_de_fernet::vasos.push_back(fernet_h);
      fernet_h = new(15, "the one at the bar");

      // class::static_variable. You know right away where it is declared
      bandeja_de_fernet::vasos.push_back(fernet_h);
```

```
      $display("Fernets on the tray:");
      foreach (bandeja_de_fernet::vasos[i]) $display(bandeja_de_fernet::vasos[i].get_name());
   end
endmodule : top
```

Static variables

- There is **a single** copy in memory, no matter how many objects get instantiated
- And it exists even if none gets instantiated: the `static` does not wait for the `new()`

```
class pepe;
   static    int    cant;
   int    val = -1;

   function new(int a);
      // Static Variable
      cant++;
      val = a;
   endfunction : new

endclass : pepe

//---------------------------------------
module top_pepes;
   initial begin
      pepe pepe1_h, pepe2_h;

      pepe1_h = new(10);
      pepe2_h = new(254654);

      $display("PEPE1 VAL = %0d", pepe1_h.val);
      $display("PEPE2 VAL = %0d", pepe2_h.val);

      // There is only one copy of a static variable in memory,
      // no matter how many instances of the class exist
      $display("NUMBER OF PEPES = %0d", pepe::cant);

   end
endmodule : top_pepes
```

Static variables

Static methods

- In the two previous examples the queue was out in the open, so the day it gets swapped for another structure you have to go and fix the whole testbench
- The version that survives a code review is two words: the variable `protected`, and **static access methods** that are the only door
- It is the same thing UVM does with the factory: nobody touches the type dictionary, you ask it with `type_id::create()`

```
class bandeja_de_fernet;

   protected static fernet vasos[$];

   static function void bandeja_fernet(fernet l);
      vasos.push_back(l);
   endfunction : bandeja_fernet

   static function void lista_fernets();
      $display("Fernets on the tray:");
      foreach (vasos[i]) $display(vasos[i].get_name());
   endfunction : lista_fernets

endclass : bandeja_de_fernet

module top;

   initial begin
      fernet fernet_h;
```

```
      fernet_h = new(2, "the one at table 7");
      bandeja_de_fernet::bandeja_fernet(fernet_h);
      fernet_h = new(3, "the one at table 4");
      bandeja_de_fernet::bandeja_fernet(fernet_h);
      fernet_h = new(15, "the one at the bar");
      bandeja_de_fernet::bandeja_fernet(fernet_h);
      bandeja_de_fernet::lista_fernets();
   end

endmodule : top
```

Static variables

Summary of the unit

- **A single copy** no matter how many objects you instantiate, **it exists even if you instantiate none**, and to touch it you have to name the class with `::`
- That `::` is the whole argument: the day the value looks odd, a `grep` tells you who put it there. But `static` solves *where it lives*, not *who touches it* — that one is `protected` plus static access methods
- Where it comes back: `uvm_config_db#(T)::set(...)` and `type_id::create()`, the two most important static variables of a UVM testbench

Parameterized classes

It is not a class: it is a mould

- The `bandeja_de_fernet` of the previous section works for fernets. For mojitos you have to write another one just like it, with one word changed
- You already solved that in RTL and you do not call it OOP: a FIFO does not get copied for every data width, it gets parameterized. `#(.awidth(8), .dwidth(16))`
- A parameterized class is the same thing, with **types** instead of numbers: `bandeja #(fernet)` and `bandeja #(mojito)`
- And there is a consequence that surprises: every different `#(...)` is a **different class**, generated at compile time. They share nothing, not even the `static`
- Why it matters: every `#(...)` you see in the rest of the course is this. `uvm_analysis_port #(T)`, `uvm_subscriber #(T)`, `uvm_sequence #(T)`

Parameterized classes

- Before the new syntax, the parameter you already know: a RAM that does not get copied for every width, it gets instantiated with `#(.awidth(8), .dwidth(16))`
- Every instantiation with different parameters produces **different hardware**, and nobody argues about it
- *Parameterized class definitions* are that same thing on the software side, with one difference: the parameter can be a **type**

```
module RAM #(
    awidth,
    dwidth
) (
    input wire [awidth-1:0] address,
    inout wire [dwidth-1:0] data,
    input we
);

   initial $display("awidth: %0d  dwidth %0d", awidth, dwidth);
   // code to implement RAM

   //...

endmodule  // RAM

module top;
   wire [ 7:0] address;
   wire [15:0] data;
   wire        write_enable;
```

```
    RAM #(
        .awidth(8),
        .dwidth(16)
    ) my_ram (
        address,
        data,
        write_enable
    );
endmodule  // top
```

Parameterized classes

- Version with static methods: the tray does not get instantiated, you ask it with `::`
- The queue is `static` and even so there are **two**: `bandeja#(fernet)` and `bandeja#(mojito)` are two different classes, each one with its own

```
// Parameterized class
// the T (type) parameter is what says which
// kind of queue this is.
class bandeja #(
    type T
);

   protected static T vasos[$];

   static function void bandeja_trago(T l);
      vasos.push_back(l);
   endfunction : bandeja_trago

   static function void lista_tragos();
      $display("Drinks on the tray:");
      foreach (vasos[i]) $display(vasos[i].get_name());
   endfunction : lista_tragos

endclass : bandeja

module top;

   initial begin
      fernet fernet_h;
      mojito mojito_h;
      fernet_h = new(15, "the one at the bar");
      bandeja#(fernet)::bandeja_trago(fernet_h);
      fernet_h = new(15, "the one at table 4");
```

```
      bandeja#(fernet)::bandeja_trago(fernet_h);

      mojito_h = new(1, "the one at table 7");
      bandeja#(mojito)::bandeja_trago(mojito_h);
      mojito_h = new(1, "the one at the counter");
      bandeja#(mojito)::bandeja_trago(mojito_h);

      $display("-- Fernets --");
      bandeja#(fernet)::lista_tragos();
      $display("-- Mojitos --");
      bandeja#(mojito)::lista_tragos();
   end

endmodule : top
```

Parameterized classes

Variables with parameters

- Here the tray gets instantiated like any object and you have to have the handle to use it — it stops being visible from the whole testbench
- It does exactly the same thing as the previous version. What changes is the scope, and that is the decision: `static` **when there really is only one, instantiated when there can be more than one**

```
module top;

   // Since bandeja is not static, it has to be instantiated
   // the parameter is a variable type
   // Using a type that does not have the get_name()
   // method is a syntax error
   fernet fernet_h;
   mojito mojito_h;

   bandeja #(fernet) bandeja_de_fernet;
   bandeja #(mojito) bandeja_de_mojito;

   initial begin
      bandeja_de_fernet = new();
      fernet_h = new(15, "the one at the bar");
      bandeja_de_fernet.bandeja_trago(fernet_h);
      fernet_h = new(15, "the one at table 4");
      bandeja_de_fernet.bandeja_trago(fernet_h);

      bandeja_de_mojito = new();
      mojito_h = new(1, "the one at table 7");
      bandeja_de_mojito.bandeja_trago(mojito_h);
```

```
      mojito_h = new(1, "the one on the sidewalk");
      bandeja_de_mojito.bandeja_trago(mojito_h);

      $display("-- Fernets --");
      bandeja_de_fernet.lista_tragos();
      $display("-- Mojitos --");
      bandeja_de_mojito.lista_tragos();
   end

endmodule : top
```

Parameterized classes

Summary of the unit

- A parameterized class **is not a class**: it is the recipe. And every different `#(...)` is a **different class**, which shares nothing with the others — **not even the `static`**
- The limit: it gets resolved **at compile time**, so the type has to be known there. Picking at runtime is the section that follows
- Where it comes back: `uvm_analysis_port #(T)`, `uvm_subscriber #(T)`, `uvm_sequence #(T)` — every `#(...)` in the rest of the course is this

The factory pattern

Who decides the type?

- Every `new()` you write is a **hardcoded** decision: that line is always going to build that class, and to change it you have to edit it and recompile
- In a testbench that does not scale. You want the same `env` sending additions today and multiplications tomorrow, picked from the command line
- The parameterized classes of the previous unit are not enough either: the `#(...)` gets resolved at compile time, and here the decision has to happen **at runtime**
- The factory is the answer: instead of building it yourself, you **ask** for the object from somebody who knows which type to hand over
- It is the most visible pattern of UVM, and with it you understand `type_id::create()` and `set_type_override()`, which are going to be in every file of the rest of the course

The factory pattern

- A design pattern, not a language feature: a class whose only job is to **build objects for you**
- You ask it for an object by describing which one you want —a string, an enum— and it hands it back already built, of whichever subtype corresponds
- Before the solution, the problem. Look at what happens when the `new()` is written in the place where the object gets used:

```
// ILLUSTRATION -- not compiled. In the previous example, new() gets called every
// time an object is needed: building a new set of drinks means editing the source.
bandeja_de_fernet = new();
fernet_h = new(15, "the one at the bar");
bandeja_de_fernet.bandeja_trago(fernet_h);
fernet_h = new(15, "the one at table 4");
bandeja_de_fernet.bandeja_trago(fernet_h);

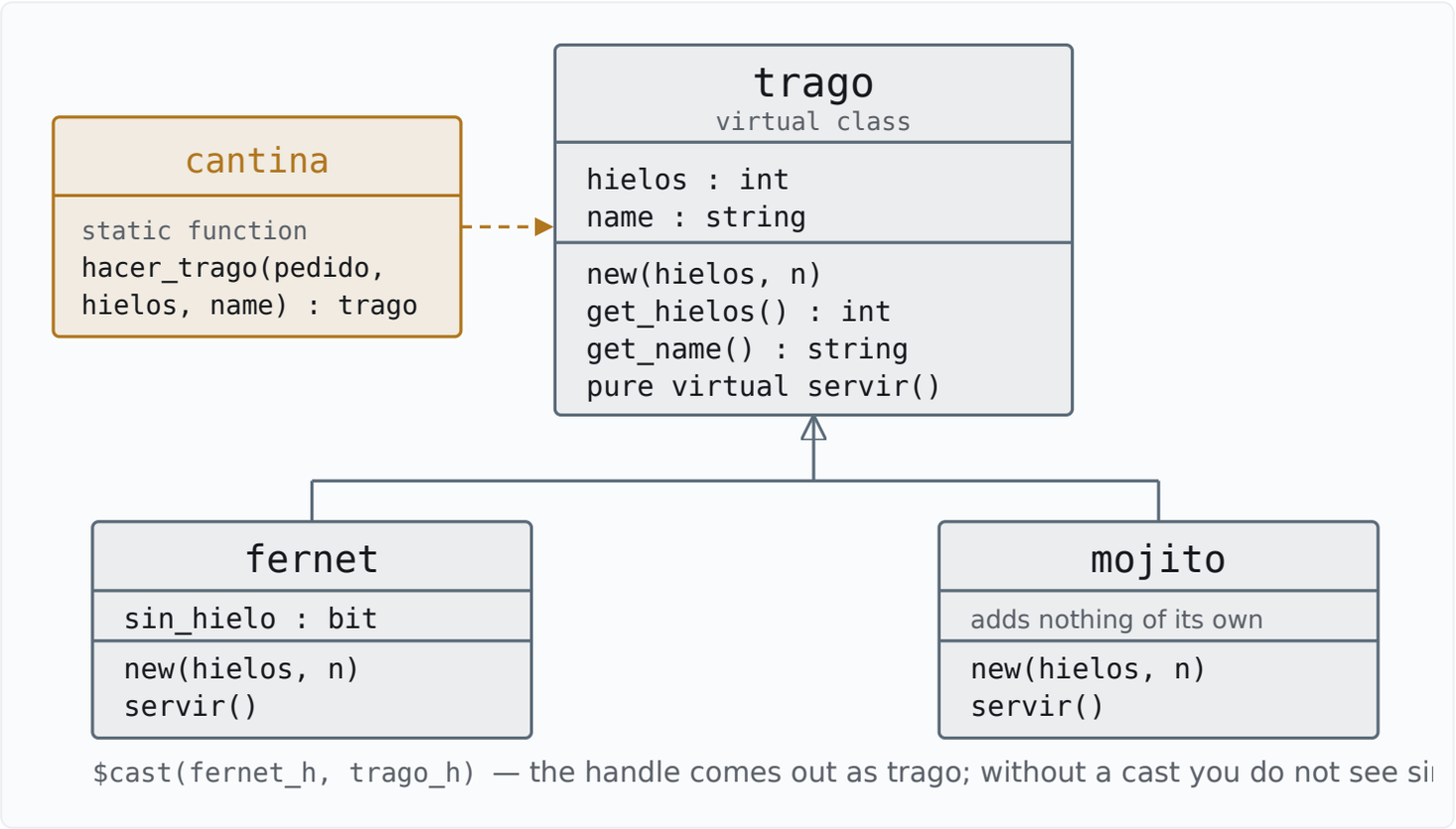
bandeja_de_mojito = new();
mojito_h = new(1, "the one at table 7");
bandeja_de_mojito.bandeja_trago(mojito_h);

mojito_h = new(1, "the one on the sidewalk");
bandeja_de_mojito.bandeja_trago(mojito_h);

// Consider the case where a set of random drinks has to be read from a file
// dynamically.
//    --> Doing that with NON hardcoded code is impossible
//        You would need a script that generates the source, recompile it...
```

The idea is to create data of random types without changing the code.

The factory pattern



`$cast(fernet_h, trago_h)` — the handle comes out as trago; without a cast you do not see si

- **The factory** is the method at the top: it takes an argument and returns an object of the type that argument asks for
- **Polymorphism** is what makes it possible: `fernet` and `mojito` extend `trago`, so both fit in a `trago` variable
- Without the second there is nowhere to put what the first returns. They are the two halves of the same tool

The factory pattern

- `virtual` is enough as long as you use methods the base class already declares. But `fernet` adds a field of its own — `sin_hielo` — that `trago` knows nothing about
- To reach it you have to bring the handle down to its real type, and that is `$cast`: it converts the variable of the second argument to the class of the first
- It checks **at runtime** and returns 0 if it does not work, so it never goes alone: always inside an `if` with its `$fatal`

```
//////////////////////////////////////////////////////////////////////
// Part of the factory Class!
//////////////////////////////////////////////////////////////////////
class cantina;

   // Static method that builds the drinks
   static function trago hacer_trago(string pedido, int hielos, string name);
      mojito nuevo_mojito;
      fernet nuevo_fernet;
      case (pedido)
         "fernet": begin
            nuevo_fernet = new(hielos, name);
            return nuevo_fernet;
         end

         "mojito": begin
            nuevo_mojito = new(hielos, name);
            return nuevo_mojito;
         end
```

```
            default: begin
               $fatal(1, {"No such drink: ", pedido});
               return null;
            end

         endcase  // case (pedido)

      endfunction : hacer_trago

   endclass : cantina
```

- And `$cast` in use, in the `top`: the factory always returns a `trago`, and to reach `sin_hielo` you have to bring it down to `fernet`

```
      // Using the factory!
      trago_h = cantina::hacer_trago("fernet", 15, "the one at the bar");
      trago_h.servir();

      // Reaching a member of fernet (derived from trago) through a trago variable
      // requires casting it to fernet.
      if (!$cast(fernet_h, trago_h)) $fatal(1, "Could not cast trago_h to fernet_h");

      if (fernet_h.sin_hielo) $display("And on top of that it comes warm!");
      bandeja#(fernet)::bandeja_trago(fernet_h);

      if (!$cast(fernet_h, cantina::hacer_trago("fernet", 2, "the one at table 4")))
         $fatal(1, "Could not cast the cantina drink to fernet_h");
```

The factory pattern

Python Style

```
class Trago:
    """Factory class that builds a Trago."""

    @staticmethod
    def factory(kind):
        """Factory Method."""
        if kind == "CubaLibre":
            return CubaLibre()
        if kind == "Fernet":
            return Fernet()
        if kind == "Mojito":
            return Mojito()
        if kind == "Whiscola":
            return Whiscola()
        raise ValueError("No such trago: " + kind)
```

- The same factory, in a language that has nothing to do with hardware: a static method with an `if` per type and a `raise` for what does not exist
- And the same annoyance: adding a drink forces you to **edit the factory**
- The whole file is in `code/u3/factory/factory.py` and it runs with `python3`: it generates twenty drinks at random and serves them all the same way

The factory pattern

Summary of the unit

- Every `new()` is a **hardcoded decision**. The factory moves it to runtime: instead of building, you **ask** for the object from somebody who knows which type to hand over
- **It does not work without polymorphism** —somebody who manufactures and a base variable to store it in are both needed— and the **`$cast` returns 0** instead of aborting, so it never goes alone: always inside an `if` with its `$fatal`
- Where it comes back: `` `uvm_component_utils `` **registers the class on its own** —the `case` you have just written by hand— and `+UVM_TESTNAME` picks it from the command line

A testbench without a single module

The same testbench, without a single module

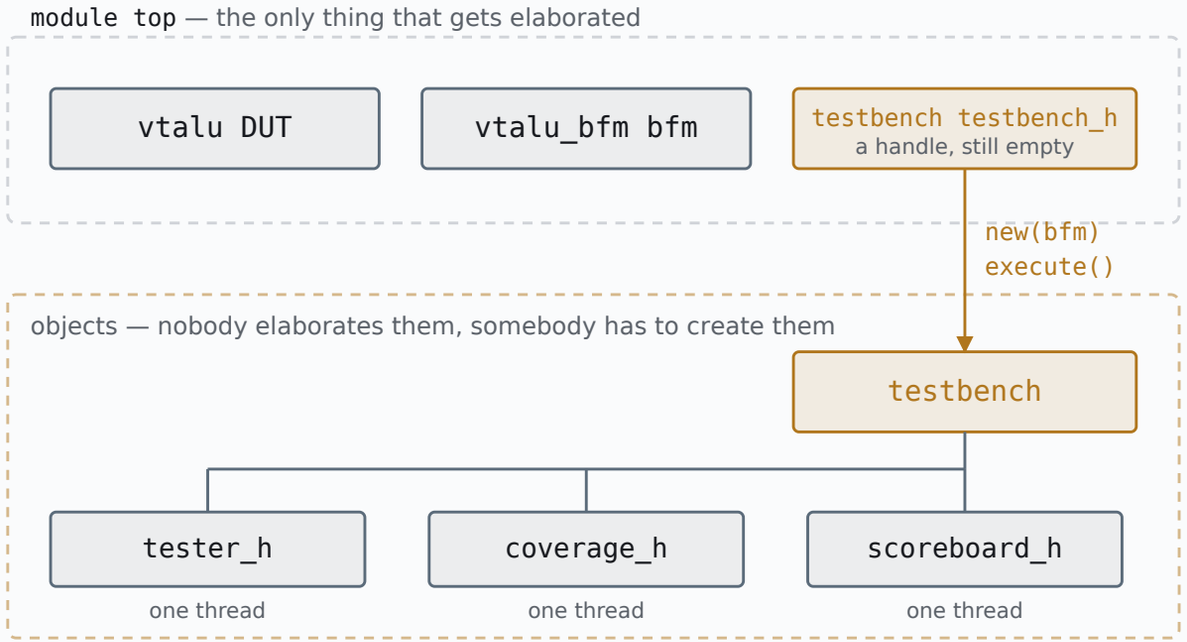
- This section **adds no functionality**: it verifies the same as the interfaces and BFM testbench
- What changes is what it is made of. The three modules —tester, scoreboard, coverage— become three **classes**, and the `initial` becomes a method
- The piece that binds them appears: a `testbench` class that instantiates the others, hands them the BFM and launches their threads. Nobody instantiates it for you
- It is the **last version without UVM**, and that is why it is worth looking at closely: everything this class does by hand, from the tests on the library is going to do
- The question to walk out of the section with: *which part replaces which?*

A testbench without a single module

- A testbench of modules works until it has to be reused: to change the stimulus you have to edit the module, and to have two versions you have to copy it
- With classes, changing the stimulus is **extending a class**, and the two versions live side by side. That is all OOP buys you here

The VTALU TB, piece by piece

- top: Instantiates the testbench class
- testbench: Top-Level Class
- tester: Generates the stimulus
- scoreboard: Checks that the VTALU is working
- coverage: Captures the functional coverage information



The three of them get the same BFM through the constructor, as a `virtual vtalu_bfm`.
From the tests on, UVM builds this tree: `build_phase`, `config_db` and one `run_phase` per piece.

A testbench without a single module

The top: the only thing that is still a module

- The `top` is still a module, and it is going to stay that way until the end of the course: classes do not get elaborated, somebody has to exist before time 0
- It imports the package with the classes, instantiates the DUT and the BFM, and declares the handle to the `testbench`
- Inside the `initial`, two lines: `new(bfm)` and `execute()`

```
module top;

    // Class definitions and shared resources are kept in a package
    import vtalu_pkg::*;
    // The macros are defined in common too. UVM does this as well
    `include "vtalu_macros.svh"

    // The DUT and the BFM get instantiated
    vtalu DUT (
        .A(bfm.A),
        .B(bfm.B),
        .op(bfm.op_set),
        .clk(bfm.clk),
        .reset_n(bfm.reset_n),
        .start(bfm.start),
        .done(bfm.done),
        .ovf(bfm.ovf),
        .result(bfm.result)
    );

    vtalu_bfm bfm ();
```

```
    // A variable to hold our TB
    testbench testbench_h;

    initial begin
        // The testbench object gets built and launched
        testbench_h = new(bfm);
        testbench_h.execute();
    end

endmodule : top
```

A testbench without a single module

The `testbench` class: the one that assembles and starts

- There is **one** object above everything else that instantiates the others, hands them what they need and launches their threads. Every OOP testbench has one
- Here we write it by hand: three `new()` and a `fork ... join_none`
- From the `env` on that class is going to be called `uvm_env`, the `new()` are going to be `type_id::create()` in the `build_phase`, and the `fork` UVM is going to do on its own

```
class testbench;

   // virtual: the object-world equivalent of a module's port list. It tells
   // the compiler that this variable is going to receive a handle to an
   // interface at some point in the future. See the slide in the TB-in-objects section.
   virtual vtalu_bfm bfm;

   tester    tester_h;
   coverage  coverage_h;
   scoreboard scoreboard_h;

   function new(virtual vtalu_bfm b);
      bfm = b;
   endfunction : new

   task execute();
      tester_h = new(bfm);
      coverage_h = new(bfm);
      scoreboard_h = new(bfm);
```

```
      // One thread per object. join_none: execute() returns and the three go on.
      fork
         tester_h.execute();
         coverage_h.execute();
         scoreboard_h.execute();
      join_none
   endtask : execute
endclass : testbench
```

A testbench without a single module

The `tester` class: the stimulus, with no wires

- It sends a thousand random operations, with the same bias to the edges as the conventional testbench: the goal is still to fill the bins of the covergroup
- Against the modular version two things change and nothing else:
  - the BFM arrives through a **variable** —a *virtual interface*— instead of through a port list
  - the `initial` is now a method, `execute()`, that somebody has to call

```
// The initial block is replaced by an execute() task
task execute();
   byte unsigned     iA;
   byte unsigned     iB;
   shortint unsigned result;
   operation_t       op_set;
   bfm.reset_alu();
   op_set = rst_op;
   iA = get_data();
   iB = get_data();
   bfm.send_op(iA, iB, op_set, result);
   op_set = mul_op;
   bfm.send_op(iA, iB, op_set, result);
   bfm.send_op(iA, iB, op_set, result);
   op_set = rst_op;
   bfm.send_op(iA, iB, op_set, result);
   repeat (1000) begin : random_loop
      op_set = get_op();
      iA = get_data();
```

```
        iB = get_data();
        bfm.send_op(iA, iB, op_set, result);
        $display("%2h %6s %2h = %4h", iA, op_set.name(), iB, result);
      end : random_loop
      $finish;
    endtask : execute
```

A testbench without a single module

The scoreboard: the `always` became a task

- A sensitivity list does not exist in a class. What replaces it is a `forever` with the wait inside — same semantics, written as sequential code

```
// The always @(posedge done) of the modular version, as a forever loop: an
// object has no sensitivity list, it has a task that blocks.
task execute();
   shortint predicted_result;
   bit      predicted_ovf;
   forever begin : self_checker
      @(posedge bfm.done) #1;
      case (bfm.op_set)
         add_op: predicted_result = bfm.A + bfm.B;
         sub_op: predicted_result = bfm.A - bfm.B;
         and_op: predicted_result = bfm.A & bfm.B;
         xor_op: predicted_result = bfm.A ^ bfm.B;
         mul_op: predicted_result = bfm.A * bfm.B;
      endcase  // case (op_set)

      // ovf belongs to sub and to nobody else: with 8-bit inputs and a 16-bit
      // output, neither the addition nor the multiplication can overflow.
      predicted_ovf = (bfm.op_set == sub_op) && (bfm.A < bfm.B);

      if ((bfm.op_set != no_op) && (bfm.op_set != rst_op))
         if (predicted_result != bfm.result || predicted_ovf != bfm.ovf)
            $error(
                "FAILED: A: %0h  B: %0h  op: %s result: %0h ovf: %0b",
                bfm.A,
                bfm.B,
                bfm.op_set.name(),
                bfm.result,
                bfm.ovf
            );
```

```
    end : self_checker
endtask : execute
```

A testbench without a single module

The coverage class: the covergroup inside an object

```
// In a class, we don't need to declare variables to hold the covergroups, but we do need
// to call a constructor to create them.
 function new (virtual vtalu_bfm b);
   op_cov = new();
   zeros_or_ones_on_ops = new();
   bfm = b;
 endfunction : new

 task execute();
    forever begin  : sampling_block
       @(negedge bfm.clk);
       A = bfm.A;
       B = bfm.B;
       op_set = bfm.op_set;
       op_cov.sample();
       zeros_or_ones_on_ops.sample();
    end : sampling_block
 endtask : execute
```

- The covergroup is **the same** as the conventional testbench's, line by line. What changed is where it lives
- In a module, declaring the covergroup was enough. In a class it has to be built: the class's `new()` does `op_cov = new()`
- The `execute()` is the `always @(negedge clk)` from before, written as a `forever`: it copies the BFM signals and calls `sample()`

A testbench without a single module

The map of day 3: what replaces what

Here, by hand	From the tests on
`testbench.execute()` with three `new()`	the `build_phase` of the `env`
passing the BFM through the constructor	`uvm_config_db`
`fork ... join_none`	UVM runs one `run_phase` per component
`$finish` inside the tester	objections
picking the tester by editing the code	`set_type_override` of the factory
`$display` that gets put in and deleted	`` `uvm_info `` and the verbosity ceiling

- This is the **complete** testbench, without UVM and with nothing missing. It works
- Everything that comes on day 3 replaces one row of this table, **and nothing else**
- We come back to this slide at the close of the day, to cross them out

A testbench without a single module

Summary of the unit

- The day 1 testbench **without a single module**: the same three pieces, plus the class that binds them — `testbench` —, which is the direct ancestor of the `uvm_env`
- The `top` is still a **module** and it will be until the end, and the BFM arrives through a **virtual interface**: it is the only way a class has of touching signals
- It is the **last version without UVM**. From here on, every piece we write by hand has a class of the library that replaces it — `execute()` is going to be the `run_phase`

Review · Day 2

1 of 8 · Handle and object

`rectangle rectangle_h;` — after that line, how many objects are there?

- **a** None: `rectangle_h` is `null` until somebody calls `new()` ✓
- **b** One, with `length` and `width` at 0
- **c** One half built: the fields exist but the constructor has not run
- **d** It depends on whether the class has a constructor

None — declaring a handle reserves nothing. There is the difference with a `struct`, which the simulator reserves the moment it sees it. And using the handle before the `new()` does not fail at compile time: it blows up in the middle of the simulation.

Review · Day 2

2 of 8 · Polymorphism

A variable of type `trago` holds a `fernet` object. If `servir()` is not `virtual`, what gets executed?

a `fernet`'s `servir()`

b A compile error

c `trago`'s `servir()` ✓

d Both, the base class one first

`trago`'s — without `virtual`, SystemVerilog looks at the **type of the variable**, not at the object's. It is literally what `code/u3/polimorfismo/01-sin-virtual` prints: *"A generic trago cannot be served"*.

Review · Day 2

3 of 8 · Abstract classes

What do I gain by using an abstract class with `pure virtual` methods instead of a base class whose method does `$fatal`?

a Nothing, it is a matter of style

b That the error moves from simulation to the compiler ✓

c That the base class can be instantiated

d That the simulator can pick the most specific override at run time

The error moves to the compiler — with `$fatal` you find out halfway through the simulation that an override was missing. With `pure virtual` it does not compile. Catching it earlier is always cheaper.

Review · Day 2

4 of 8 · Static variables

I instantiate 10 objects of a class that has a `static` variable. How many copies are there in memory?

a 10, one per object

b 0 until somebody writes it: the memory is reserved on first access

c 1, shared by every instance ✓

d It depends on the simulator

A single one — and it exists even if you instantiate no object at all. That is what makes it useful for global TB data, and what makes it dangerous if you leave it public.

Review · Day 2

5 of 8 · Static methods

Why is it worth declaring the static variable `protected` and exposing it with static methods?

- **a** To be able to change the data structure without touching whoever uses it ✓
- **b** Because a `static` without `protected` is reserved once per instance
- **c** So that it takes up less memory
- **d** Because UVM demands it

To be able to change it later — if the queue is out in the open, the day you swap it for another structure you have to go and fix every place that touched it. Encapsulating is being able to change your mind.

Review · Day 2

6 of 8 · Parameterized classes

`bandeja#(fernet)` and `bandeja#(mojito)` have a `static` queue inside. Do they share the queue?

a. Yes, `static` is a single one for everybody

b. Yes: the parameter changes the type of the methods, not the static storage

c. It depends on whether they get instantiated or not

d. No: each specialization is a different class, with its own queue ✓

They do not share it — SystemVerilog generates **one class per combination of parameters**. `static` is unique inside each one of those classes, not across all of them. UVM leans on this all the time.

Review · Day 2

7 of 8 · The factory pattern

What problem does the factory pattern solve?

a Creating objects faster

b Deciding at run time which subtype to build ✓

c Avoiding having to declare classes

d Centralizing the `new()`s in one class, to be able to count and free them

Deciding the subtype at runtime — without hardcoding the `new`: you ask the factory for an object and it decides which one. It is the piece that is later going to let you change the stimulus of a whole test without touching the code of the `env`.

Review · Day 2

8 of 8 · $cast

When does a `$cast(destination, source)` succeed?

a Always: it converts any class into any other

b Only between classes with no inheritance

c When the two classes have the same fields, even if they are not related

d Only if the object in `source` is of the class of `destination` or of a derived one ✓

Only if the object allows it — `$cast` checks **at runtime** and returns 0 if it does not work. That is why UVM's factory is more comfortable than the `cantina` of the section: `type_id::create()` returns the right type and saves you the cast.

Exercise · Day 2

A tester that only multiplies

```
$ cd code/ejercicios/d2 && bash run.sh
```

- Extend `tester`, redefine `get_op()`, and have the testbench instantiate it
- It is going to carry on sending random operations. **That is where the exercise starts**
- Hint: polymorphism

Yesterday we left...

Where the testbench stands

- The day 1 testbench **without a single module**: three classes, one that binds them, and a `top` that only instantiates the DUT and the BFM
- The BFM gets into the classes through a **virtual interface**, which is the only way a class has of touching signals
- And the factory is already written **by hand**: a `case` that has to be edited every time a new type shows up. Today the one you do not edit shows up

Agenda

Day 3 · ≈ 4 h 30 · unit 4 · UVM walks in

- Tests
- Components and phases
- The env: structure and stimulus
- Reporting: verbosity and *actions*

By the end of the day you can:

- **Explain** why a UVM simulation ends at `t=0` if nobody raises an objection
- **Say** why `build_phase` runs top-down and `connect_phase` the other way round
- **Make** a scoreboard that fails say **something more** than "it failed"

Tests

Compile once, pick the test from the command line

- The object-based testbench has the stimulus **inside**: changing test means changing code and compiling again
- A thousand tests at five minutes of compilation each are **5000 minutes**: three and a half days of machine time to run nothing new
- UVM turns the order around: the testbench gets compiled **once**, and the test that runs is picked when the simulation starts

```
$ ./obj_dir/top/sim +UVM_TESTNAME=random_test
$ ./obj_dir/top/sim +UVM_TESTNAME=add_test     # the same binary
```

- That is what this unit buys, and what gets paid with the ceremony that comes: registration in the factory, a constructor with a fixed signature and phases

Tests

The top: instantiate, publish the interface, start

```
initial begin
   // null and "*": from the root and visible to the whole tree. See the
   // uvm_config_db slide.
   uvm_config_db#(virtual vtalu_bfm)::set(null, "*", "bfm", bfm);

   // UVM reads +UVM_TESTNAME, and builds THAT test with the factory.
   run_test();
```

- The `top` is still a **module**: it instantiates the BFM and the DUT the same as before. The only new thing is these two lines
- `run_test()` with no argument means *"whichever `+UVM_TESTNAME` says"*. With an argument — `run_test("random_test")` — that one becomes the **default**, and `+UVM_TESTNAME` still overrides it from the command line
- Before `run_test()` there is not a single UVM object alive: that is why the `set` goes here

Tests

uvm_config_db: the testbench's mailbox

- A global and **typed** database: the `#(...)` is part of the key
- Four arguments: *scope* (`cntxt`, `inst_name`), *name* and *data*
- The **scope** is a hierarchical path, and that is where the whole point is

```
// ILLUSTRATION -- not compiled. The two halves of the config_db, side by side:
// the set() lives in top.sv and the get() in tb_classes/driver.svh.
// -- IN THE TOP -- top is a module, not a uvm_component --------------------
//   cntxt = null : the scope starts at the UVM root
//   inst  = "*"  : visible to the whole component tree
uvm_config_db #(virtual vtalu_bfm)::set(null, "*", "bfm", bfm);

// -- IN A COMPONENT -- always inside build_phase ----------------------------
//   cntxt = this : the scope is the hierarchical path of THIS component
//   inst  = ""   : with no suffix, the path is exactly the component's
if (!uvm_config_db #(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
  `uvm_fatal("DRIVER", "Failed to get BFM")

// The 3rd argument ("bfm") is the NAME of the datum inside the database.
// If the string in the set and the one in the get are not identical, this still
// compiles and blows up at run time with a null pointer.
```

Tests

A test in three parts: 1 · register it

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

   virtual vtalu_bfm bfm;
```

- `random_test` extends `uvm_test`, which extends `uvm_component`: it is a node of the tree, not a loose object
- The `` `uvm_component_utils `` macro **enrols it in the factory**. Without that line, `+UVM_TESTNAME=random_test` finds nothing and UVM aborts
- The handle to the virtual interface gets **declared** here and **filled in** in the `build_phase`: they are two different moments

Tests

2 · the constructor and the `build_phase`

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

// The get goes in build_phase and not in the constructor: when the
// constructor runs, the tree does not exist yet.
function void build_phase(uvm_phase phase);
   if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
      `uvm_fatal("RANDOM TEST", "Failed to get BFM")
endfunction : build_phase
```

- The constructor of a `uvm_component` has a **fixed signature**: `name` and `parent`, in that order, and `super.new(name, parent)` as the first line
- The `get` of the config_db goes in `build_phase`, **not** in the constructor: when the constructor runs the tree is still being assembled
- `if (!get(...)) uvm_fatal`: the `get` returns a bit. Ignoring it leaves the handle at `null` and the failure shows up a hundred lines later, in another file

Tests

3 · the `run_phase`: this is where the simulation happens

```
task run_phase(uvm_phase phase);
   random_tester random_tester_h;
   coverage      coverage_h;
   scoreboard    scoreboard_h;

   phase.raise_objection(this);

   random_tester_h = new(bfm);
   coverage_h = new(bfm);
   scoreboard_h = new(bfm);

   fork
      coverage_h.execute();
      scoreboard_h.execute();
   join_none

   random_tester_h.execute();

   phase.drop_objection(this);
endtask : run_phase
```

- It is the same body as the `execute()` of the object-based testbench: tester, coverage and scoreboard, with the two observers in `fork ... join_none`
- What changes is **who calls it**: before it was called by the `top` module, now it is called by UVM when the run phase's turn comes
- The two new lines are the `raise` and the `drop` of the objection, which are the ones that decide when everything ends

Tests

Objections: who decides when it ends

```
task run_phase(uvm_phase phase);
   phase.raise_objection(this);    // "I still have work"
   ...                             // the stimulus
   phase.drop_objection(this);     // "done, as far as I am concerned it can end"
endtask
```

- Every `run_phase` runs **in parallel**: none of them can decide on its own when to finish. UVM counts, and the phase ends when the **last** objection drops
- Without `raise_objection` the simulation ends at **time 0**, and the test passes with 0 errors without having sent a single stimulus
- Without `drop_objection` it never ends: there is no error, the simulation just keeps running
- Both symptoms are **silent**, and that is why two plusargs exist: `+UVM_OBJECTION_TRACE` says who raised it and who dropped it, and `+UVM_TIMEOUT` puts a ceiling on the run

Tests

The end that is not the end: `drain_time`

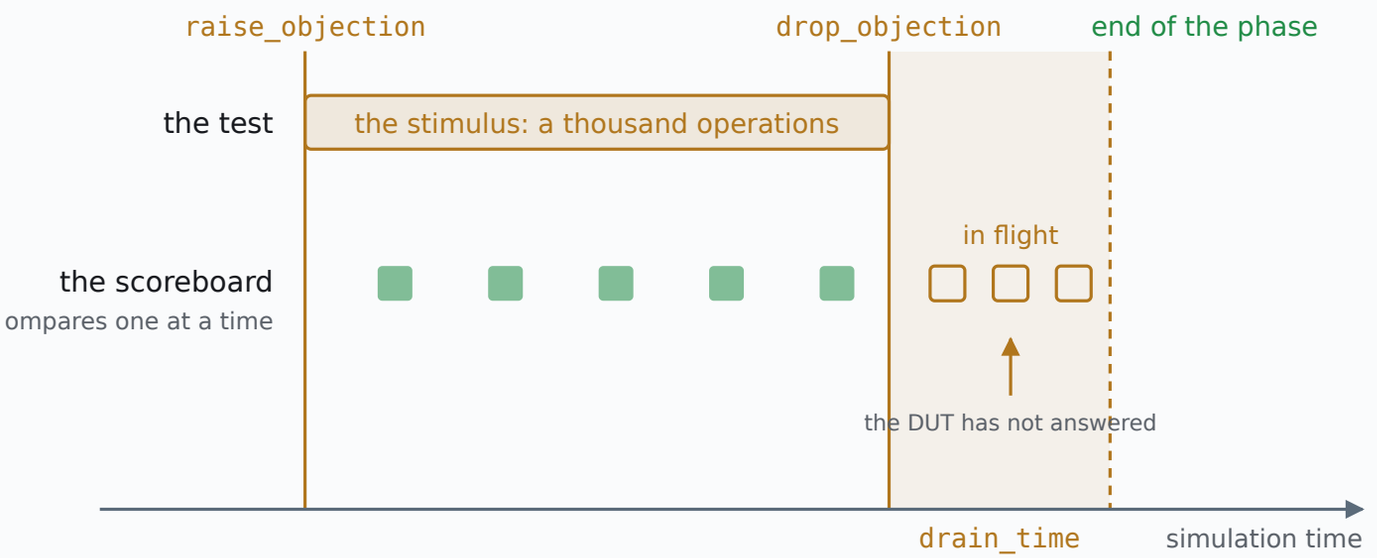
- The objection says *"I finished **sending**"*. It does not say *"I finished **comparing**"*: between the two there is a scoreboard with a FIFO inside and results the DUT has not answered yet
- When the last objection drops the phase cuts **at that instant**: what was left in flight does not arrive, nobody compares it, and the summary says `UVM_ERROR : 0`

```
task run_phase(uvm_phase phase);
   phase.get_objection().set_drain_time(this, 200ns);  // a cushion after the last drop
   phase.raise_objection(this);
   ...                                                 // the stimulus
   phase.drop_objection(this);
endtask
```

- The `drain_time` is a number picked by eye. The version with a criterion is `phase_ready_to_end()`: UVM calls it when the last objection dropped, and the component that still has work raises one more and the phase carries on
- It is **the** bug of the second week, and it is silent: no error, no hang, and the last transactions were never compared

Tests

The timeline, end to end



The objection measures the stimulus, not the analysis: when the last one drops, the phase cuts at thi
Without the cushion, those three never get compared and the summary says `UVM_ERROR : 0`.

- `raise` and `drop` mark **the stimulus**, and nothing but the stimulus
- What is left in flight is not in that count: the three empty boxes of the drawing
- The `drain_time` is the time given away so those answers arrive and get compared
- Without `raise` nothing sits between the marks; without `drop`, the line never ends

Tests

The second test: the same thing, with another tester

```
task run_phase(uvm_phase phase);
   add_tester add_tester_h;
   coverage   coverage_h;
   scoreboard scoreboard_h;

   phase.raise_objection(this);

   add_tester_h = new(bfm);
   coverage_h = new(bfm);
   scoreboard_h = new(bfm);

   fork
      coverage_h.execute();
      scoreboard_h.execute();
   join_none

   add_tester_h.execute();
   phase.drop_objection(this);
endtask : run_phase
```

- `add_test` is `random_test` with **one different line**: `add_tester` instead of `random_tester`. Everything else repeats exactly as it is
- That repetition is real and it is ugly, and it is the debt the components pay: when the testbench is a tree of components, the test is going to build an `env` and it is not going to touch the stimulus
- For now it is worth seeing it copied: it is the problem that motivates what follows

Tests

How it gets run, and what it prints

```
set -e
. "$(dirname "${BASH_SOURCE[0]}")/../../verilator/common.sh"
vlt_uvm top --coverage-user -Wno-fatal -f dut.f -f tb.f
run_sim +UVM_TESTNAME=random_test
run_sim +UVM_TESTNAME=add_test
cov_report
```

```
UVM_INFO @ 0: reporter [RNTST] Running test random_test...
UVM_INFO $UVM_HOME/src/base/uvm_report_server.svh(1009) @ 46340: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    1
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 46ns
```

- A single `vlt_uvm`, two `run_sim`: **one compilation, two tests**. That is all the unit promises, and here it is done
- `[RNTST] Running test random_test...` is UVM saying what it found in the factory under the name you passed it
- The *Report Summary* with `UVM_ERROR : 0` is the verdict: reporting takes it apart and explains where each row comes from

Tests

Summary of the unit

- The testbench gets compiled **once** and the test gets picked from the command line: `+UVM_TESTNAME=random_test`. That is the whole business of the section
- The arithmetic that justifies it: a thousand tests at five minutes of compilation each are **three and a half days** of waiting for it to compile
- The `top` is still a module. It publishes the BFM in the `uvm_config_db` **before** `run_test()`, because before that line there is not a single UVM object alive
- The **`uvm_config_db`** is a global and **typed** database: the `#(...)` is part of the key, and the scope is a **hierarchical path**
- A test gets registered with `` `uvm_component_utils ``. Without that line, the factory does not find it and UVM aborts
- The virtual interface gets **declared** in the class and **filled in** in the `build_phase`: they are two different moments, and confusing them leaves a `null`
- The objection says when the test finished **sending**, not when the scoreboard finished **comparing**. That difference is paid for with `drain_time`, or else with `phase_ready_to_end()`

Components and phases

A component is what is in the tree, and the tree is walked by UVM

- A testbench has three different things: the **structure** —what parts there are and how they connect—, the **sequences** —which commands, in what order— and the **data**. This unit is the first one; the data is day 5 and the sequences day 6
- In the tests the test did `new()` on three loose objects and called them itself. That is not a structure: it is a long function
- A `uvm_component` is a **node with a name and a parent**. UVM assembles the whole tree and then walks all of it calling **phases**, in an order it guarantees
- Turning a class into a component is four steps, always the same ones:
    1. extend `uvm_component` or one of its daughters
    2. register it with `` `uvm_component_utils() ``
    3. the minimum constructor: `(string name, uvm_component parent)`
    4. override the phases it needs

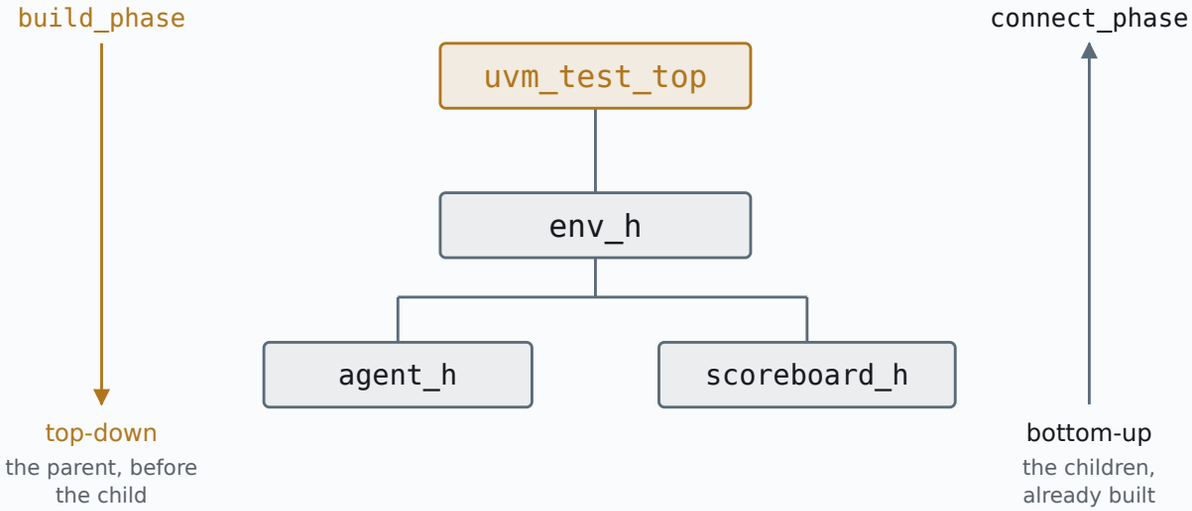
Components and phases

Intro UVM phases

- Every uvm component has these phase methods by inheritance
- UVM creates the TB and goes calling these methods in order
- The phases are *methods*: they get overridden like any other virtual method. Whether it is also worth calling `super` is the topic of the next slide
- *function void build_phase(uvm_phase phase):* UVM creates the TB (top-down). UVM components get instantiated in this method. If you try to instantiate them anywhere else it is an error
- *function void connect_phase(uvm_phase phase):* Connecting components
- *function void end_of_elaboration_phase(uvm_phase phase):* UVM calls it once it has created and connected every component
- *task run_phase(uvm_phase phase):* UVM runs this task in its own thread. Every run_phase runs simultaneously
- *function void report_phase(uvm_phase phase):* It runs when the last objection of the test drops. It shows Results

Components and phases

The tree, twice: one walk goes down and the other goes up



And inside that descent, the real order:



The child gets created inside the parent build, and its own build runs only once the parent one is don
That is why instantiating a component outside `build_phase` arrives late: the tree was already walkec

- `build_phase` goes down because a parent has to exist to create its children
- `connect_phase` goes up for the opposite reason: the other end has to be there
- The child gets **created** inside the parent build; its own build runs later
- That is why the `config_db` `set()` goes **before** the `create()` that builds it

Components and phases

And what about `super.build_phase()`?

```
// uvm_component.svh:2385-2393 -- uvm-core 2020.3.1, with the return; elided
function void uvm_component::build_phase(uvm_phase phase);
   build();            // -> apply_config_settings(): the automatic config
endfunction

function void uvm_component::connect_phase(uvm_phase phase);
   connect();          // -> return;  it is empty, like all the others
endfunction
```

- `build_phase` is the **only** phase that does anything in `uvm_component`: it applies the fields registered with `` `uvm_field_* ``. The course does not use those macros
- Careful, there are **two different `super`s**: that one, and the one of a base class of **yours**, which builds what you wrote. The second one the course **does** call
- Out there: **always put it in, unless you can name why in your case it is a no-op**. Adding it costs nothing; skipping it when it mattered —a `` `uvm_field_* ``, or inheriting from `uvm_agent` — leaves the field at its default and **nobody warns you**
- And the order matters: whatever has to be decided *before* the children get built goes before the `super`. The `set_type_override` of `add_test` is that

Components and phases

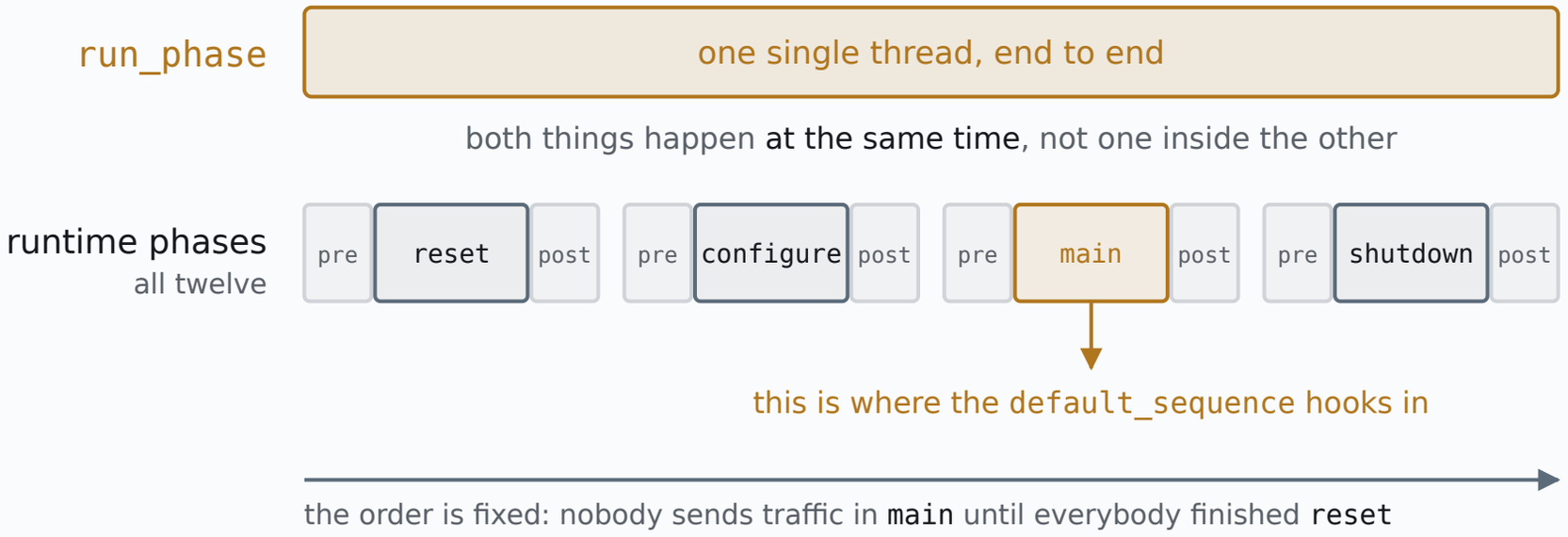
The full picture: there are nine, not five

Phase	What for	Order
`build_phase`	instantiate the components	**top-down**
`connect_phase`	connect the ports	bottom-up
`end_of_elaboration_phase`	hierarchy ready, before simulating	bottom-up
`start_of_simulation_phase`	last call before time 0	bottom-up
`run_phase`	**task**: this is where the simulation happens	one thread each
`extract_phase`	gather the data from the run	bottom-up
`check_phase`	decide whether it passed	bottom-up
`report_phase`	print the verdict	bottom-up
`final_phase`	close files and exit	**top-down**

- The course uses five. The other four exist, they are empty, and you are going to see them

Components and phases

And inside the `run_phase`, a schedule



The real name carries the suffix: `pre_reset_phase`, `reset_phase`, `post_reset_phase`...
Each one has its own objection, and a runtime phase without an objection ends as soon as it starts.

- The twelve are `task`s, they run **in parallel** with `run_phase`, and they are there to coordinate components from different teams without hand-made flags
- The course uses `run_phase` and nothing else — with a single agent there is nothing to coordinate. But the `default_sequence` of day 6 hooks into `main_phase`

Components and phases

The scoreboard, now as a component

```
// The scoreboard of the TB in objects, now as a uvm_component. The four steps
// --extend, register, constructor, phases-- are in the slides of the components section.
class scoreboard extends uvm_component;
   `uvm_component_utils(scoreboard);

   virtual vtalu_bfm bfm;

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

   // Self-sufficient: the component asks the config_db for the BFM. Before, the
   // test passed it in through the constructor, and the test had to receive it
   // only to hand it out.
   function void build_phase(uvm_phase phase);
      if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
         `uvm_fatal("SCOREBOARD", "Failed to get BFM")
   endfunction : build_phase
```

- The four steps, in order: it extends `uvm_component`, it registers itself, constructor `(name, parent)`, and an override of `build_phase`
- The underlying change does not show in the diff: before, the **test** received the BFM and handed it to the scoreboard through the constructor. Now the scoreboard **asks for it itself** from the config_db
- The `run_phase` is the `execute()` of the object-based testbench without touching a comma: the check did not change, what changed is who starts it

Components and phases

The test builds the tree and steps aside

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

   random_tester tester_h;
   coverage      coverage_h;
   scoreboard    scoreboard_h;

   // Components get instantiated in build_phase, never in the constructor.
   function void build_phase(uvm_phase phase);
      tester_h = new("tester_h", this);
      coverage_h = new("coverage_h", this);
      scoreboard_h = new("scoreboard_h", this);
   endfunction : build_phase

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

   // No run_phase: the test's job ends once the tree is built.

endclass
```

- The three `new("name", this)` are what creates the tree: `this` is the parent, and the string is the name that is going to come out in `print_topology` and in every message
- They go in `build_phase` **and nowhere else**. In the constructor there is no hierarchy yet; after `build_phase`, UVM has already moved on
- `random_test` **has no** `run_phase`: its job finished when the tree was assembled. The three children have theirs, and they run in parallel

Components and phases

The second test: it inherits, and it copies anyway

```
// add_test extends random_test, redeclares tester_h with another type and
// REWRITES the whole build_phase. That duplication is the problem the env section solves
// with the factory override.

class add_test extends random_test;
   `uvm_component_utils(add_test);

   add_tester tester_h;

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

   function void build_phase(uvm_phase phase);
      tester_h = new("tester_h", this);
      coverage_h = new("coverage_h", this);
      scoreboard_h = new("scoreboard_h", this);
   endfunction

endclass
```

- `add_test` extends `random_test`, so it inherits `coverage_h` and `scoreboard_h`. It only redeclares `tester_h`, with another type
- But the `build_phase` is **written out again in full**, and the three lines are identical except for one. Inheriting the class was not enough to inherit the structure
- And there is something worse hidden: `add_tester tester_h;` **shadows** the `tester_h` of the base class. They are two different variables with the same name
- It is the same problem as the tests, in disguise. The env solves it for real: **one** structure, and the factory substitutes the piece that changes

Components and phases

Summary of the unit

- A `uvm_component` is a **node with a name and a parent**. UVM assembles the tree with those two pieces of data and then walks it on its own
- Turning a class into a component is **four steps, always the same ones**: extend, register with the macro, the two-argument constructor, and the phases
- The **phases** are virtual methods that UVM calls **in order**: `build_phase` (top-down), `connect_phase`, `end_of_elaboration_phase`, `run_phase`, `report_phase`
- Components get instantiated **in the `build_phase` and nowhere else**
- Every `run_phase` runs **in parallel**, each one in its thread. None of them decides on its own when the simulation ends: that is the **objections**
- There are **two `super.build_phase()`**: `uvm_component`'s and your base class's. **Always put it in, unless you can name why in your case it is a no-op**

The env: structure and stimulus

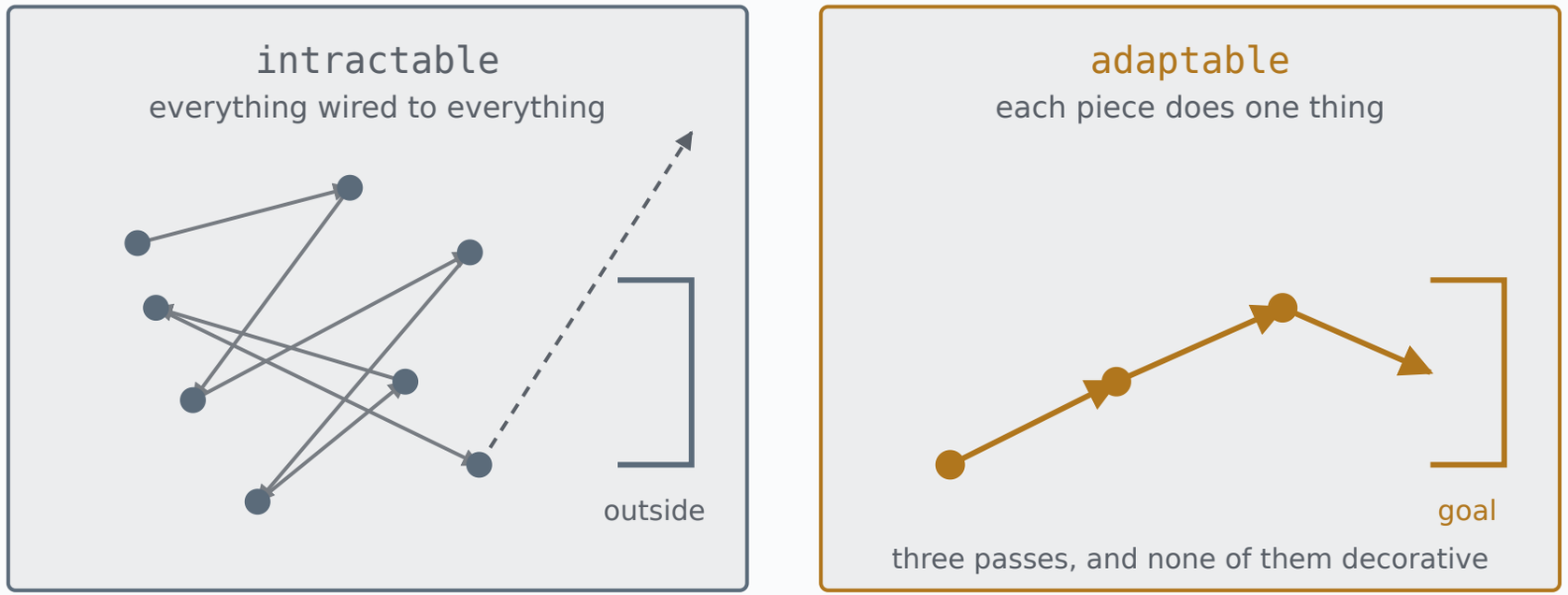
Two things that change for different reasons

- A testbench has **structure** —which components there are and how they connect— and **stimulus** —what gets sent to the DUT—
- They change for different reasons and at different rates: the structure changes when the design changes; the stimulus, every time you write a test
- Up to the components they are glued together: `add_test` is a copy of `random_test` with one line changed. Ten tests are ten copies, and one change in the structure is ten edits
- This section splits them into two classes: the **`uvm_env`** keeps the structure, the **`uvm_test`** keeps the stimulus
- And the piece that makes it possible is the **factory**: the `env` asks for a generic `base_tester`, and each test tells the factory which one to hand over

The env: structure and stimulus

Intractable vs. adaptable

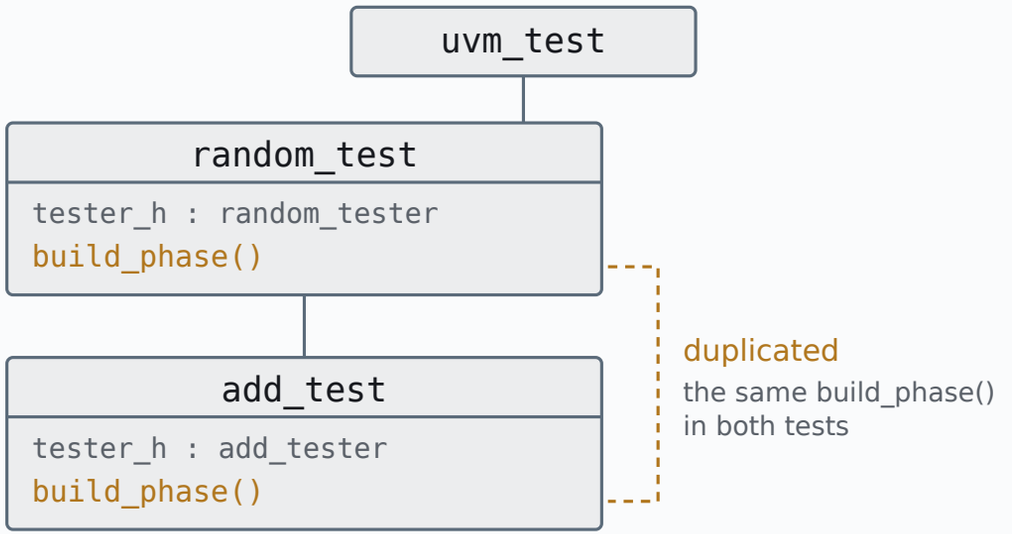
> *"Playing football is very simple, but playing simple football is the hardest thing there is." —* ***Johan Cruyff***



- *Intractable* is the code that comes out fast and afterwards will not let itself be touched. The criterion is not aesthetic: **somebody else is going to modify it**, and that somebody is going to be you a year from now, remembering nothing
- Three rules to turn the VTALU TB around: **one class, one thing**; **nothing hardcoded**; **program against the interface**, not against the implementation

The env: structure and stimulus

The problem: two tests, the same `build_phase()` twice

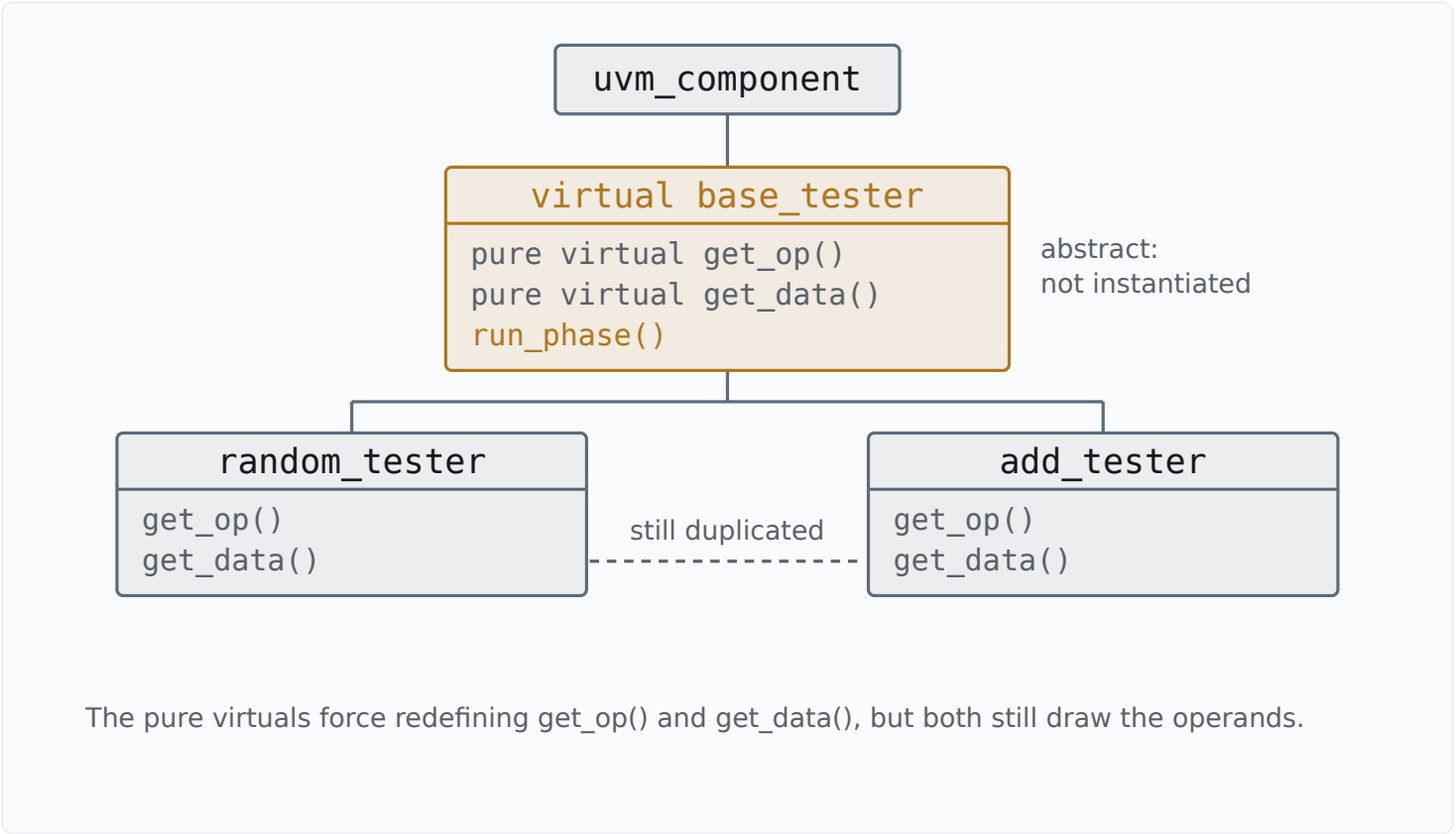


add_test inherits from random_test and rewrites its whole build_phase: the three new() calls are iden
The testers, on the other hand, duplicate nothing: add_tester extends random_tester and only overri
run_phase() is written once, in random_tester, and add_tester inherits it.

- Today there are two tests: `random_test` builds a `random_tester`, `add_test` an `add_tester`. Everything else —coverage and scoreboard— they build the same way
- `add_test` **inherits** from `random_test` and **rewrites its whole** `build_phase()`: the three `new()` calls are identical and written twice
- The testers, on the other hand, duplicate nothing: `add_tester extends random_tester` and only overrides `get_op()`. The `run_phase()` is written **once**, in `random_tester`, and it is inherited
- Which means: the inheritance ladder is already solved well below, in the testers, and solved badly above, in the tests. The next two slides show how it came out below; the rest of the section applies it above
- Duplicating the `build_phase` is not ugly, it is expensive: ten tests are ten copies, and one change in the structure is ten edits

The env: structure and stimulus

The way out: an abstract class, and the stimulus in the daughters

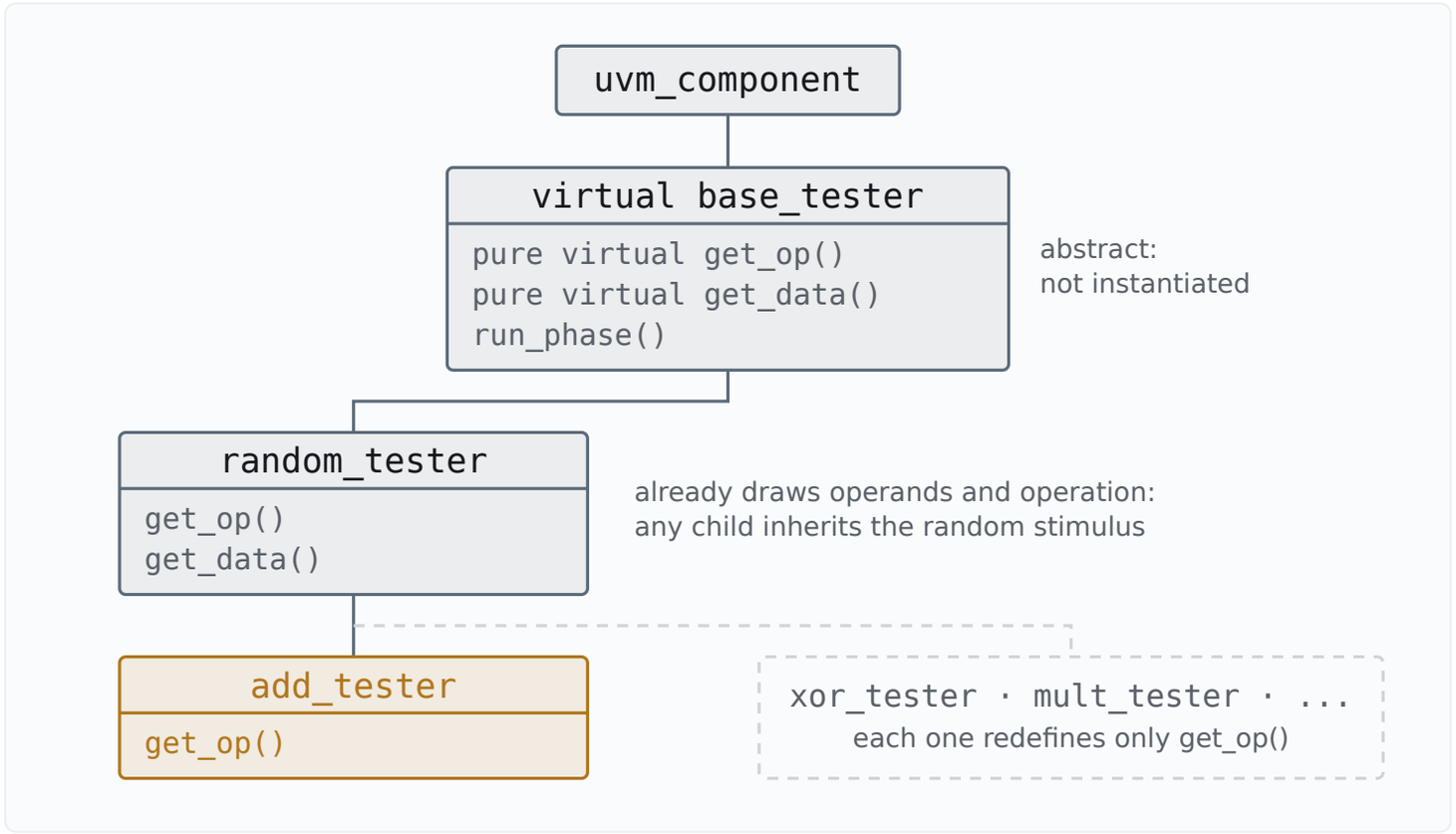


The pure virtuals force redefining get_op() and get_data(), but both still draw the operands.

- `base_tester` is an **abstract** class: it writes the `run_phase()` a single time —a thousand operations— and assembles it by calling `get_op()` and `get_data()`
- Those two methods are `pure virtual`, so the class cannot be instantiated and every daughter is **forced** to define them. The one that forgets does not compile
- But there is still repeated code: both testers randomize the operands the same way, and that is a sign that a step is missing in the hierarchy

The env: structure and stimulus

The result: the three-storey hierarchy



- `base_tester` puts in the `run_phase()` and declares `get_op()` and `get_data()` as `pure virtual`. Nobody instantiates it
- `random_tester` goes down one step and defines both: random operation, random operands
- `add_tester` **inherits from** **`random_tester`**, not from `base_tester`: it keeps the random operands and only redefines `get_op()`
- And the same step works for all the ones that come: `xor_tester`, `mult_tester`, `maxmult_tester`. Each one, one useful line

The env: structure and stimulus

add_tester

- The result is that *add_tester* is very simple: it inherits from *random_tester* and only redefines *get_op()*

```
class add_tester extends random_tester;
   `uvm_component_utils(add_tester)

   function operation_t get_op();
      return add_op;
   endfunction : get_op

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

endclass : add_tester
```

The env: structure and stimulus

The other half: who instantiates whom

- The testers are already well distributed. The other half is missing: **who instantiates them**
- The obvious way out is a family of `uvm_test`, one per tester. And it is the bad one: look at the `random_test` below
- It instantiates the three components **and** it picks the tester. Every new test copies that whole structure to change one line

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

   random_tester tester_h;
   coverage      coverage_h;
   scoreboard    scoreboard_h;

   // Components get instantiated in build_phase, never in the constructor.
   function void build_phase(uvm_phase phase);
      tester_h = new("tester_h", this);
      coverage_h = new("coverage_h", this);
      scoreboard_h = new("scoreboard_h", this);
   endfunction : build_phase

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

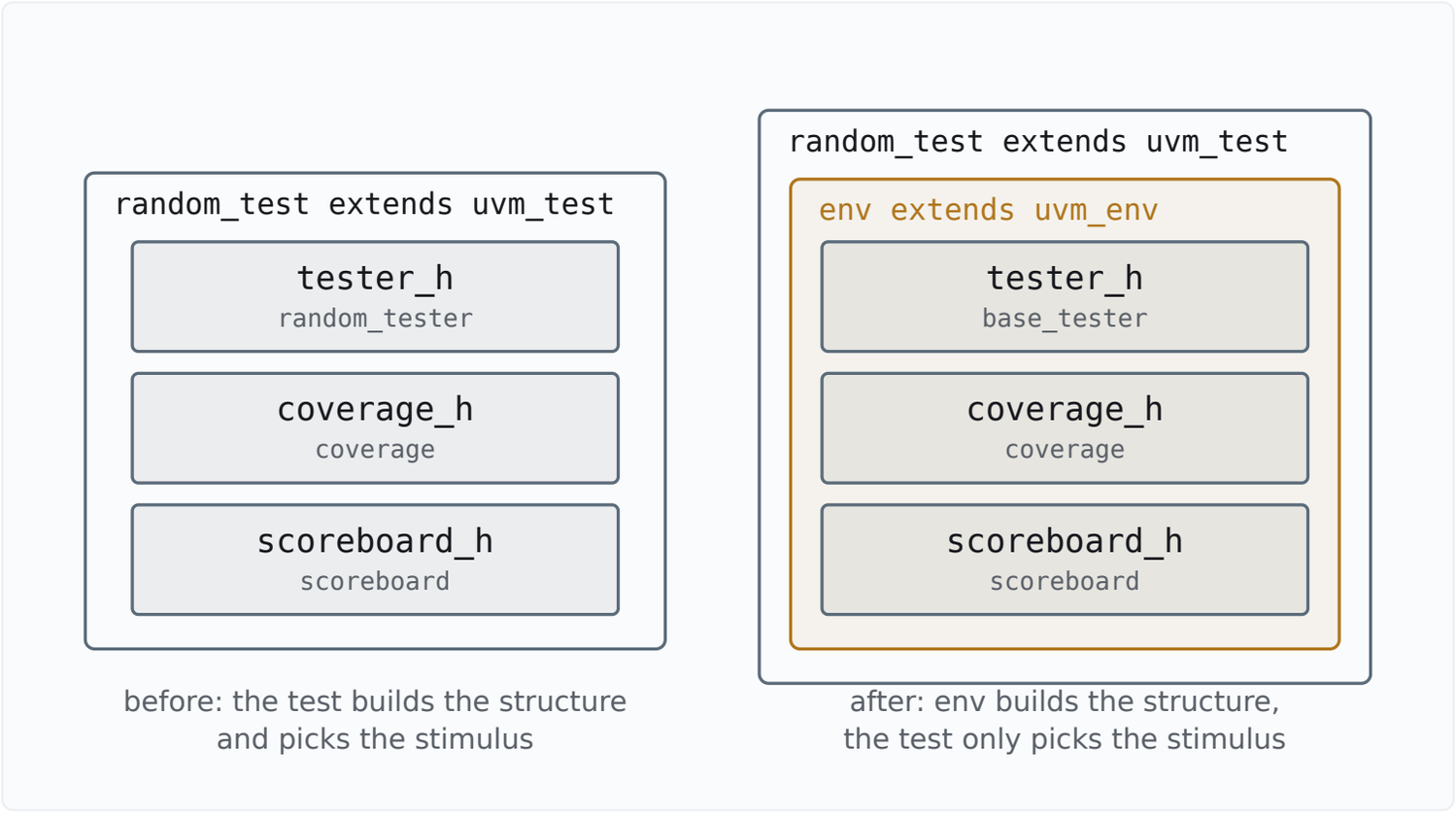
   // No run_phase: the test's job ends once the tree is built.
```

```
endclass
```

- The `random_test` does **two** things: it assembles the structure of the testbench *and* it picks which stimulus to send. They are two reasons to change in a single class
- UVM splits that in two: the structure moves to a `uvm_env`, and the test is left only with picking the stimulus
- A typical `env` defines **two** methods and nothing else: `build_phase` to instantiate and `connect_phase` to connect

The env: structure and stimulus

The map after splitting



before: the test builds the structure and picks the stimulus

after: env builds the structure, the test only picks the stimulus

- Two parallel hierarchies, and that is the whole section: on the left the tests —the stimulus—, on the right the `env` and its components —the structure—
- The `env` is **a single one** and it does not get touched. The tests are many and they change all the time
- The only thing that binds them is the factory: the test says *"when they ask for a `base_tester`, give them this one"*

The env: structure and stimulus

The env class: instantiate and connect, nothing else

- The `env` defines the structure of the testbench: it instantiates the components and then connects them. It is **a single one**, and it does not change from one test to another
- The tests, on the other hand, are many and they change all the time. That is the whole division
- The `build_phase()` does not use `new()`: each component gets asked for with `type_id::create()`, which is the factory formula of the next slide

```
// The structure of the TB, and nothing else: it instantiates the components and
// connects them. Which tester arrives is the test's call, with a factory override.
class env extends uvm_env;
   `uvm_component_utils(env);

   base_tester tester_h;
   coverage    coverage_h;
   scoreboard  scoreboard_h;

   function void build_phase(uvm_phase phase);
      // base_tester is abstract and this works anyway: create() does not build
      // it, it asks the factory what to hand out when somebody asks for a base_tester.
      tester_h = base_tester::type_id::create("tester_h", this);
      coverage_h = coverage::type_id::create("coverage_h", this);
      scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
   endfunction : build_phase
```

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

endclass
```

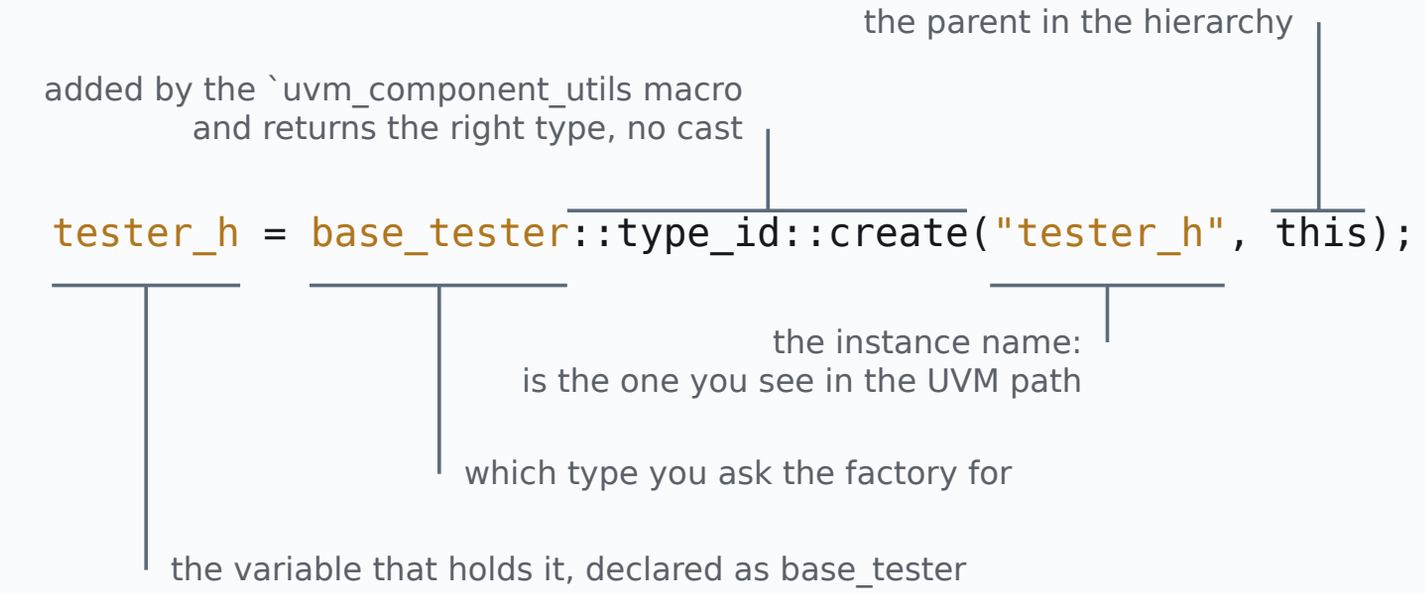
The env: structure and stimulus

UVM's factory: the class signs itself up

- The factory had two annoyances: the result had to be cast, and the `case` of the factory had to be edited to add a type
- UVM's has neither of the two. A class registers **itself**, with a macro on its first line:
  - `` `uvm_component_utils() `` for what goes in the tree
  - `` `uvm_object_utils() `` for what does not —the transactions—
- And it returns an object **of the right type**, without casting: `type_id` is a different class for every registered class, so the type is put in by the compiler

The env: structure and stimulus

The formula, read from left to right



In amber, the only thing that changes from one line to the next; the rest is always the same.

- It is the *static member/method* pattern of the static variables, exactly as it is: `type_id` is a static member of the class, and `create()` a static method of that member
- The two arguments are the usual ones: the **name** that is going to come out in the topology, and the **parent** that hangs it from the tree
- The misspelt class and the one that does not exist show at **compile time**. The one that forgot the `` `uvm_component_utils `` only if **no ancestor** is registered: if it extends a registered one, the `type_id` is **inherited** and there is no error

The env: structure and stimulus

The objection that comes on its own: `base_tester` is abstract

- Now the objection that comes on its own: `tester_h` is of type `base_tester`, which is an **abstract** class. And this compiles, runs and works. Why?

```
function void build_phase(uvm_phase phase);
   // base_tester is abstract and this works anyway: create() does not build
   // it, it asks the factory what to hand out when somebody asks for a base_tester.
   tester_h = base_tester::type_id::create("tester_h", this);
   coverage_h = coverage::type_id::create("coverage_h", this);
   scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
endfunction : build_phase
```

- Because `base_tester` here is not a mould: it is an **order form**. The `env` does not build a `base_tester`, it asks the factory for one
- The factory returns an object of some **derived** class, and which one is decided by a `set_type_override` done **before** the `build_phase` runs
- That override is done by the **test**, which is the one that instantiates the `env`. The structure never finds out which tester it got

The env: structure and stimulus

The override: changing the part without touching the plan

- In the components, `add_test` was `random_test` copied with one line changed. The copy was the problem, not the solution
- An `add_tester` **is** a `base_tester`, so it works anywhere the code asks for a `base_tester` — that is polymorphism
- So there is no need to touch the `env`: it is enough to tell the factory *"when they ask you for a `base_tester`, hand over an `add_tester`"*
- That is the **factory override**, and it is the only thing a test has to do

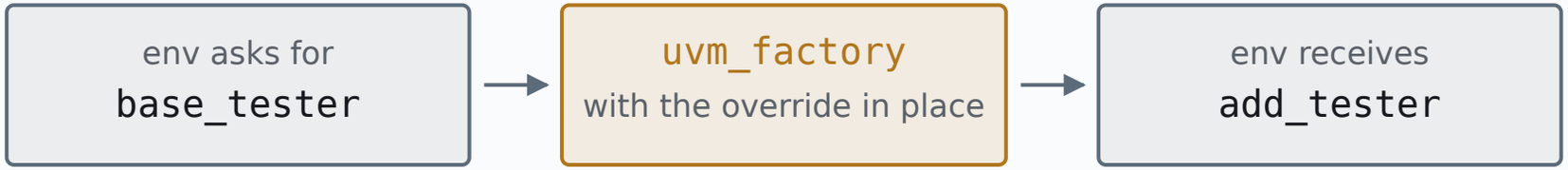
The env: structure and stimulus

The factory override

```
base_tester::type_id::set_type_override(add_tester::get_type());
```

the type the env asks the factory for

the type the factory returns



The override goes in the test build_phase, before the env build_phase runs.

- The static method *set_type_override()* tells the factory that when it sees a request for a *base_class_name* it should return an object of type *new_class_name*
- This feature is the one that lets us separate the structure of the TB (*uvm_env*) from the type of stimulus generated (*uvm_test*)

The env: structure and stimulus

The factory override, in the test

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

   env env_h;

   function void build_phase(uvm_phase phase);
      base_tester::type_id::set_type_override(random_tester::get_type());
      env_h = env::type_id::create("env_h", this);
   endfunction : build_phase

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

endclass
```

- We can see that this version is more adaptable
- Every test does an override of base_tester depending on the tester it needs for its stimulus
- The env class creates the object; running it is UVM's job
- Now the test class does one thing well (generating stimulus) and the env class does another thing very well (creating the structure)

The env: structure and stimulus

By instance, and the knob for seeing what got set

```
base_tester::type_id::set_type_override(add_tester::get_type());          // ALL of them
base_tester::type_id::set_inst_override(mult_tester::get_type(),
                                        "uvm_test_top.env_h.t2_h");      // just one

uvm_coreservice_t::get().get_factory().print(0);   // which overrides are in place
```

```
Instance Overrides:
  Requested Type  Override Path            Override Type
  base_tester     uvm_test_top.env_h.t2_h  mult_tester
  base_tester     env_h.t3_h               mult_tester   <- registered and did NOT apply
Type Overrides:
  base_tester     add_tester
```

- `set_inst_override` changes **one path**, not a type. And where the two coincide, the instance one wins
- The path is **absolute and starts at `uvm_test_top`**. Measured: `"env_h.t3_h"` gets registered all the same, gives no error, and `t3_h` comes out an `add_tester`
- `factory.print(0)` is the counterpart of `uvm_top.print_topology()`: one says what you **asked for**, the other what **came out**. When they do not match, the bug is in between
- With `*` it works: `"*.env_h.t4_h"` applies. It is the workaround when the test does not know where it is going to be instantiated

The env: structure and stimulus

Summary of the unit

- A testbench has **structure** —which components there are and how they connect— and **stimulus** —what gets sent—. They change for different reasons and at different rates
- Up to the components they were glued together: `add_test` was a copy of `random_test` with one line changed. Duplicating the loop is not ugly, it is **expensive**
- The `uvm_env` keeps the structure; the **test** only says which stimulus it wants. A new test stops touching the env
- The piece that makes it possible is the **factory**: the env asks for a `base_tester` and the test decides which one arrives, with `set_type_override`
- The abstract class `base_tester` has the `run_phase` **written a single time**; the only thing each tester redefines is `get_op()` and `get_data()`
- The three rules for writing adaptable code: **one class, one thing**; no copying and pasting structure; and programming against the **base class**

Reporting

47 % of the time goes here

- The day-1 trends chart said that almost half of a verification engineer's time goes into **debug**. This section is about that half
- A scoreboard that only says `FAIL` leaves you right there: you know something is wrong and you do not know which component, at what moment, or with what data
- The natural reaction is to fill the code with `$display`, and then delete them. And put them back next week
- UVM solves it the other way round: the messages **stay in** and get filtered. Each one brings along the time, the hierarchical path of the component and the file
- And there are two different knobs, which get confused all the time: **verbosity** filters `` `uvm_info ``; the **actions** control warnings, errors and fatals

Reporting

What `$display` cannot do

- A thousand operations per test, dozens of tests per regression: one `$display` per transaction and the log stops being readable before the first coffee
- And with `$display` the only way of lowering the noise is **deleting lines**, that is, losing exactly what you are going to need next time
- UVM replaces the `$display` with four macros that bring two things for free: context —time, component, file— and a **filter** that is driven from outside

Reporting

The reporting macros

- There are four and they differ by **severity**: `` `uvm_info ``, `` `uvm_warning ``, `` `uvm_error `` and `` `uvm_fatal ``. **All four** are counted in the *Report Summary*; what changes is the default **action** — `` `uvm_error `` comes with `UVM_DISPLAY|UVM_COUNT` and `` `uvm_fatal `` also kills the simulation
- The **ID** is the first argument: a string that says who is talking. It is what you filter with afterwards, so it is worth making it the name of the component and always the same one
- The **message** is the second, and it gets assembled with `$sformatf` when it carries data
- The **verbosity** is the third, and **only** `` `uvm_info `` has it: it is the one that decides whether the message comes out or not

```
// ILLUSTRATION -- not compiled. The signature of each macro, not a call.
`uvm_info   (Message ID String, Message String, Verbosity)
`uvm_warning(Message ID String, Message String)
`uvm_error  (Message ID String, Message String)
`uvm_fatal  (Message ID String, Message String)
```

Reporting

The reporting macros: what each one does

- This is how it looks in the scoreboard: a `` `uvm_error `` when the comparison fails, with the message assembled with `$sformatf`
- And this is how it comes out in the log. Look at what the line brings **without anybody asking for it**: the time, the file and the line, and the hierarchical path of the component that spoke

```
    data_str = $sformatf(
        " %2h %0s %2h = %4h (%4h predicted)",
        cmd.A,
        cmd.op.name(),
        cmd.B,
        t,
        predicted_result
    );

    if (predicted_result != t) `uvm_error("SCOREBOARD", {"FAIL: ", data_str})
    else `uvm_info("SCOREBOARD", {"PASS: ", data_str}, UVM_HIGH)
```

```
UVM_ERROR tb_classes/scoreboard.svh(39) @ 290: uvm_test_top.env_h.scoreboard_h [SCOREBOARD] FAIL:  5b add_op ff = 015a (015b predicted)
```

Reporting

The six levels, and why the message stays

- The usual cycle: you fill the code with `$display`, you find the bug, you delete everything — and next week you write it again
- With verbosity the messages **stay in place** and do not get in the way. It is two steps, and the second one does not touch the code:
  - put a verbosity on each `` `uvm_info ``, according to how much noise it is
  - set the **ceiling** of the run: global, or per branch of the tree
- A message comes out if its verbosity is **less than or equal to** the ceiling. That is why `UVM_NONE` always comes out

```
// ILLUSTRATION -- not compiled. uvm_object_globals.svh:382-390, uvm-core 2020.3.1,
// copied verbatim: code/.uvm/ is downloaded and gitignored, so the slide cannot
// pull the enum straight out of the library.
typedef enum {
   UVM_NONE   = 0,
   UVM_LOW    = 100,
   UVM_MEDIUM = 200,  // the default: with nothing set, UVM_MEDIUM is what you get
   UVM_HIGH   = 300,
   UVM_FULL   = 400,
   UVM_DEBUG  = 500
} uvm_verbosity;

// Any message above the verbosity that was set does not get printed
```

Reporting

The global ceiling: a plusarg, without recompiling

- It gets set in two ways, and the first one is the one used every day: a **plusarg** on the command line
- No code to touch and no recompiling. The same binary gives the log of the regression and the log of the debug
- The problem shows up with the team: if there are ten people with their own debug messages, raising the global ceiling brings you everybody's
- That is why it has to be possible to ask for it **per component**, which is the next slide

```
./obj_dir/top/sim +UVM_TESTNAME=random_test +UVM_VERBOSITY=UVM_HIGH
```

Reporting

`uvm_cmdline_processor`: the command line, whole

- `+UVM_VERBOSITY` and `+UVM_TESTNAME` are not read by magic: they are read by **a class**, and it is available for your own plusargs

```
uvm_cmdline_processor clp = uvm_cmdline_processor::get_inst();
string valor;

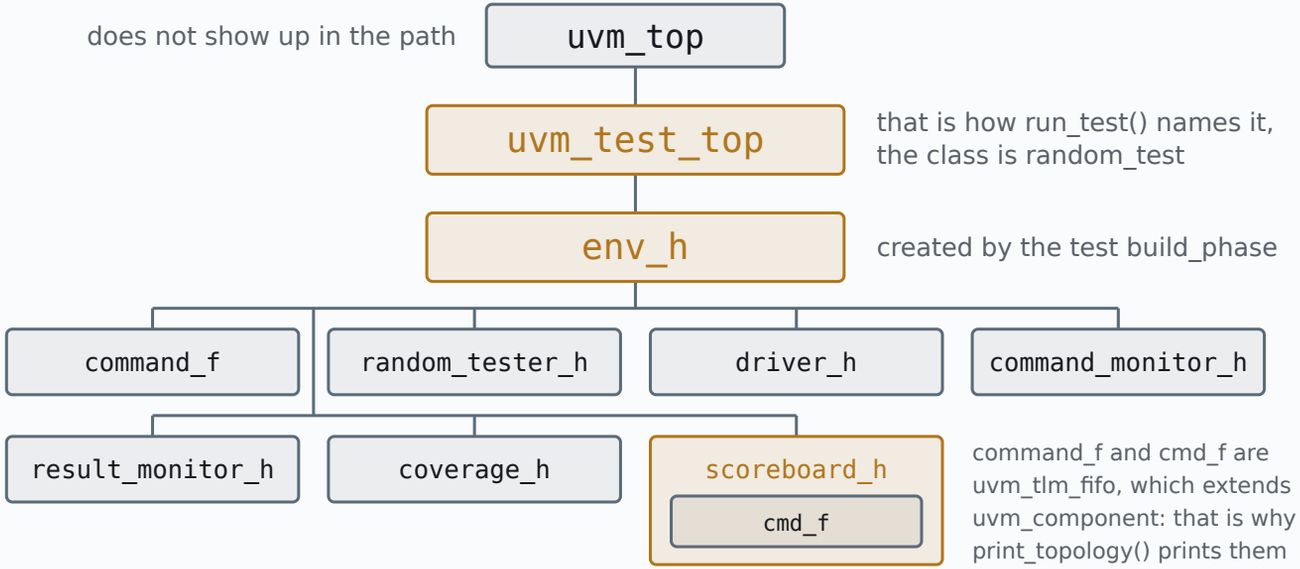
if (clp.get_arg_value("+COUNT=", valor)) count = valor.atoi();
```

- It is a **singleton**: `get_inst()` from any class, without building anything and without passing it through the `config_db`
- `get_arg_value` returns **how many times the argument appeared**, not the value: the value comes out through the `ref`. If it appeared twice you find out — `$value$plusargs` keeps quiet with the first one
- And there are `get_args()`, `get_plusargs()` and `get_uvm_args()`, which return the whole command line in a queue: it is how you print under what conditions a regression from three months ago ran

Reporting

The ceiling per branch: two methods and a tree

- Every `uvm_component` comes with methods to set its own ceiling, in two flavours: only itself, or itself **and everything hanging below** (`_hier`)
- Careful about which hierarchy it is: it is **not** the DUT's module one. It is the tree that UVM assembles by calling `build_phase()` from the top down
- It is the same tree the prefix of each message comes out of, and the same one the `uvm_config_db` uses as a scope




```
UVM_ERROR tb_classes/scoreboard.svh(39) @ 290:
uvm_test_top.env_h.scoreboard_h [SCOREBOARD] FAIL: 5b add_op ff
```

That prefix is the component path inside the tree.

Reporting

The ceiling per branch: where the call goes

- The call goes **after** the hierarchy exists —otherwise the component is not there yet— and **before** the simulation starts
- That window has a name and it is one of the nine phases: `end_of_elaboration_phase`
- `set_report_verbosity_level_hier()` reaches the component **and its children**; `set_report_verbosity_level()`, only it

```
// To use the UVM hierarchy to control verbosity, the reporting methods have to
// be called after the UVM hierarchy has been built, but BEFORE the simulation
// starts. That is what end_of_elaboration_phase is for
function void end_of_elaboration_phase(uvm_phase phase);
   //scoreboard_h.set_report_severity_action_hier(UVM_ERROR, UVM_NO_ACTION);
   scoreboard_h.set_report_verbosity_level_hier(UVM_HIGH);
endfunction : end_of_elaboration_phase
```

Reporting

Switching off warnings, errors and fatal messages

- A real situation: the DUT is fine, the scoreboard predicts the addition wrong, and the class is being fixed by somebody else. You need to carry on working in the meantime
- Lowering the verbosity ceiling **does not help**: it only affects `` `uvm_info ``, and what has to be silenced is a `` `uvm_error ``
- Another knob is needed

```
UVM_INFO @ 0: reporter [RNTST] Running test random_test...
UVM_INFO tb_classes/scoreboard.svh(40) @ 150: uvm_test_top.env_h.scoreboard_h [SCOREBOARD] PASS:  43 and_op ff = 0043 (0043 predicted)
UVM_ERROR tb_classes/scoreboard.svh(39) @ 290: uvm_test_top.env_h.scoreboard_h [SCOREBOARD] FAIL:  5b add_op ff = 015a (015b predicted)
...
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    4
UVM_WARNING :    0
UVM_ERROR :    2
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[SCOREBOARD]     5
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 820ps
```

Reporting

Switching off messages: where it gets done

- Warnings, errors and fatals are **immune** to the verbosity ceiling, and it is right that they should be: nobody wants to switch off an error by accident
- The knob that does reach them is called **actions**, and it answers another question: not *"does it get printed?"* but *"what gets done with this message?"*
- Printing is only one of the seven possible things

```
// ILLUSTRATION -- not compiled. uvm_object_globals.svh:354-364, uvm-core 2020.3.1,
// copied verbatim: code/.uvm/ is downloaded and gitignored, so the slide cannot
// pull the enum straight out of the library.
typedef enum {
   UVM_NO_ACTION = 'b0000000,
   UVM_DISPLAY   = 'b0000001,
   UVM_LOG       = 'b0000010,
   UVM_COUNT     = 'b0000100,
   UVM_EXIT      = 'b0001000,
   UVM_CALL_HOOK = 'b0010000,
   UVM_STOP      = 'b0100000,
   UVM_RM_RECORD = 'b1000000
} uvm_action_type;
```

Reporting

Switching off messages: the complete example, and the log you are left with

- `set_report_severity_action_hier(UVM_ERROR, UVM_NO_ACTION)` on the scoreboard: when an error shows up in there, do nothing
- It goes in the `end_of_elaboration_phase` of the `env`, for the same reason as the hierarchical verbosity
- And the log below is the result: same DUT, same broken scoreboard, zero errors in the *Report Summary*
- In `code/u4/reporting` the line is written and **commented out**: you uncomment it to reproduce this log, and comment it back. That it lives commented out in the repo is part of the lesson

```
// To use the UVM hierarchy to control verbosity, the reporting methods have to
// be called after the UVM hierarchy has been built, but BEFORE the simulation
// starts. That is what end_of_elaboration_phase is for
function void end_of_elaboration_phase(uvm_phase phase);
   //scoreboard_h.set_report_severity_action_hier(UVM_ERROR, UVM_NO_ACTION);
   scoreboard_h.set_report_verbosity_level_hier(UVM_HIGH);
endfunction : end_of_elaboration_phase
```

```
UVM_INFO @ 0: reporter [RNTST] Running test random_test...
--- UVM Report Summary ---
```

```
** Report counts by severity
UVM_INFO :    1
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 820ps
```

Reporting

`uvm_report_catcher`: demoting the error you were expecting

- `UVM_NO_ACTION` switches off **every** error of a branch. When the test verifies the error path on purpose —a `PSLVERR` that has to arrive— what has to be silenced is **one single one**, and the rest has to carry on shouting

```
class demotar_pslverr extends uvm_report_catcher;
   virtual function action_e catch();
      if (get_severity() == UVM_ERROR && get_id() == "PSLVERR")
         set_severity(UVM_INFO);      // stops counting as an error
      return THROW;                   // and carries on, already downgraded
   endfunction
endclass

demotar_pslverr c = new();
uvm_report_cb::add(null, c);          // null = the whole testbench; or a component
```

- The catcher gets in **before** the report: it sees every message and can change its severity, its ID, its text or its action
- `THROW` lets it carry on already modified; `CAUGHT` makes it disappear
- And the difference that matters: the summary prints *Number of demoted UVM_ERROR reports : 1*. **The blackout leaves a trace**, which is what `UVM_NO_ACTION` does not do

Reporting

Summary of the unit

- Two knobs, and they do not cross: **verbosity** filters `` `uvm_info ``; **actions** decides what gets done with warnings, errors and fatals
- Both can be set per component or per branch, in `end_of_elaboration_phase`
- And there is a third one for the **expected** error: the `uvm_report_catcher` demotes it and leaves it counted in the summary, instead of switching it off without a trace
- With that, debug messages get written once and stay in place for ever — which is the only thing that makes debug cheap
- And it arrives here, at the end of day 3, on purpose: it is the tool the exercises of days 4 and 5 get debugged with — the day 5 one is solved by **raising the verbosity**, and there is no other way of finding it

Review · Day 3

1 of 7 · uvm_test

What does `+UVM_TESTNAME=add_test` let you do that you could not do before?

- **a** Pick the test on an already compiled testbench ✓
- **b** Run the simulation faster
- **c** Change the DUT without recompiling: UVM re-elaborates on start-up
- **d** Lower the verbosity of the messages

Pick the test without recompiling — UVM reads that plusarg and asks the **factory** for the test by name. It is the difference between 1000 tests × 5 minutes of compilation and a single compilation.

Review · Day 3

2 of 7 · UVM Phases

In what order does UVM walk the hierarchy in `build_phase` and in `connect_phase`?

a Both from the top down

b Both from the bottom up

c `build_phase` top-down, `connect_phase` bottom-up ✓

d In the order the components were declared, from the top down

Build top-down, connect bottom-up — and it makes sense: you cannot connect a component that does not exist yet, so first the whole hierarchy gets built and only then does it get connected.

Review · Day 3

3 of 7 · Objections

What are `raise_objection()` / `drop_objection()` for in the `run_phase`?

a. For reporting scoreboard errors

b. For keeping the simulation alive while there is work ✓

c. For synchronizing two threads

d. So that the `run_phase`s of every component start at the same time

So that the phase does not end early — every `run_phase` runs in parallel, each one in its thread, and the phase ends when **the last objection drops**. Without raising it, the simulation ends on you at time 0.

Review · Day 3

4 of 7 · The env

What does each one get: `uvm_env` and `uvm_test`?

a `env` generates the stimulus; `test` assembles the structure

b Both do the same thing, `env` is optional

c `env` runs the DUT; `test` runs the scoreboard

d `env` assembles the structure; `test` picks the stimulus ✓

Structure vs. stimulus — each class does **one single thing well**. That is why the `env` almost always has only `build_phase` and `connect_phase`, and the test almost always has only a factory override.

Review · Day 3

5 of 7 · Factory override

`set_type_override()` replaces `base_tester` with `add_tester`. When does it have to be called?

- **a** Before the `build_phase` that creates the object runs ✓
- **b** At any moment: the factory applies it retroactively to what is already built
- **c** After the `connect_phase`
- **d** Inside the `run_phase` of the tester

Before the `build_phase` — the factory decides what to build **at the moment of the `create()`**. If the override arrives late, the `env` has already instantiated the base class and it does nothing.

Review · Day 3

6 of 7 · Verbosity

The verbosity ceiling (`+UVM_VERBOSITY=UVM_HIGH`), which macros does it act on?

a. On all four: info, warning, error and fatal

b. Only on `` `uvm_info `` ✓

c. On error and fatal only

d. On none: it only changes the format of the message

Only on `uvm_info` — warnings, errors and fatals are **immune** to the verbosity ceiling, and it is right that they should be: nobody wants to switch off an error by accident. For those you need the *actions* mechanism.

Review · Day 3

7 of 7 · Report actions

You want to silence the `` `uvm_error `` of a scoreboard somebody else is fixing. Where does the `set_report_severity_action_hier()` go?

a In the `build_phase` of the `env`

b In the `run_phase` of the test, before raising the objection

c In the constructor of the scoreboard

d In the `end_of_elaboration_phase` of the `env` ✓

In `end_of_elaboration_phase` — it has to be **after** the hierarchy is built (otherwise the component does not exist yet) and **before** the simulation starts. That phase is exactly that window.

Exercise · Day 3 · 1 of 2

A new test without touching the structure

```
$ cd code/ejercicios/d3 && bash run.sh
```

- Write `mult_tester` and `mult_test`, with `set_type_override`
- **Without touching `env.svh`**: the structure never finds out that the stimulus changed
- The `env` brings a `chequeo` component that marks the exercise on its own

Exercise · Day 3 · 2 of 2

The `uvm_error` that says nothing

```
$ cd code/ejercicios/d3b && bash run.sh
```

- The scoreboard is **right** —it catches every mismatch— and when it fails it says `FAILED` and nothing else: even less than the line of the day-1 waves exercise
- Make the `uvm_error` say **which one** failed: `A`, `B`, the operation, the DUT's result and the one you predicted
- And make the one that **passes** get printed too, at `UVM_HIGH`, so that it does not show up in the everyday log
- The checker watches the same bus you do and knows which was the first that failed

Yesterday we left...

Where the testbench stands

- UVM walked in: `run_test()`, the test, the phases, the `env` and the scoreboard, and the `+UVM_TESTNAME` that picks the class from the command line
- The **objection** is what keeps the simulation alive, and the reporting already tells verbosity from *action*
- And one bottleneck was left: the scoreboard is **one** and it is wired to the monitor. The day there are two listeners, the one that publishes has to be edited

Agenda

Day 4 · ≈ 4 h · unit 5 · how the components talk

- One producer, many listeners
- A single place that watches the wire
- When somebody has to wait
- Who waits for whom

By the end of the day you can:

- **Hang** a new subscriber off an analysis port **without touching the one that publishes**
- **Choose** between an analysis port and a TLM FIFO, and say which of the two blocks
- **Remove** the `#500` at the end of a test and replace it with the objection that belongs there

One producer, many listeners

Two ways of talking between objects

- The structural paradigm asks *which steps to follow*. OOP asks **what objects there are and how they talk to each other** — and that is where the single-`initial` testbench ends
- And talking, between objects, is exactly two things. The difference is whether the two run in **the same thread** or not
- **Same thread:** one calls the other's method and comes straight back. It is the `write()` of a `uvm_analysis_port` — the first two sections of the day
- **Different threads:** the two run at the same time and they have to be **coordinated**, which means somebody is going to have to wait. It is `put` / `get` and the TLM FIFOs — the last two
- The whole of day 4 is those two columns. We start with the left-hand one

One producer, many listeners

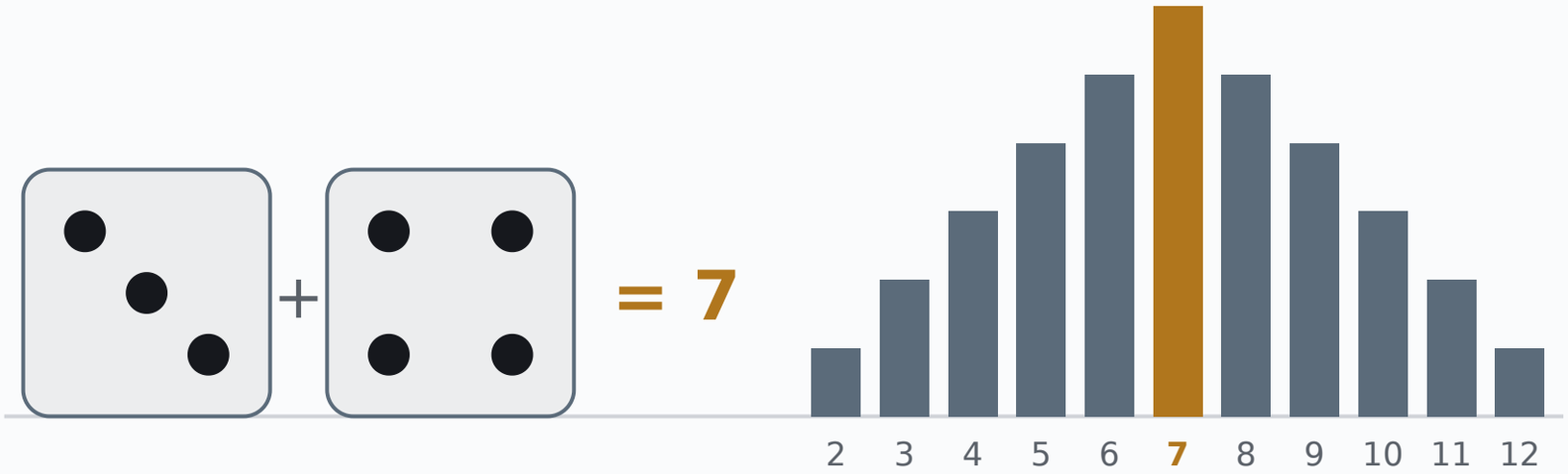
The one who publishes does not know who reads it

- A piece of testbench data almost never has **one** addressee: every command the monitor sees is wanted by the scoreboard, the coverage and, tomorrow, the log
- The obvious solution is for the producer to keep a handle to every consumer and call their method. It works, and adding a consumer forces you to **touch the producer**
- It is the same problem the BFM solved in interfaces and BFM, one level up: there we hid the wire, here we want to hide **the list of addressees**
- The answer is an old pattern: the producer publishes on a port and does not know —nor care— who is on the other side
- The section builds it by hand with dice first, and only then replaces it with `uvm_analysis_port`. The example is deliberately foreign to hardware

One producer, many listeners

The problem: two dice, three things to count

- Write a program that simulates rolling two dice (2d6) 20 times and reports the following
  - The average of the values the dice gave
  - A histogram with the frequency of values
  - A coverage report that shows whether every possible value from 2 to 12 came up



20 rolls of 2d6: average, histogram and coverage from 2 to 12.

The sum is never 1 or 13, and 7 comes up in 6 of the 36 combinations.

One producer, many listeners

The three observers: the same shape, three bodies

```
class average extends uvm_component;
   `uvm_component_utils(average);

   protected real dice_total;
   protected real count;

   function new(string name, uvm_component parent = null);
      super.new(name, parent);
      dice_total = 0.0;
      count = 0.0;
   endfunction : new

   function void write(int t);
      dice_total = dice_total + t;
      count++;
   endfunction : write

   function void report_phase(uvm_phase phase);
      `uvm_info("AVERAGE", $sformatf("DICE AVERAGE: %2.1f", dice_total / count), UVM_NONE)
   endfunction : report_phase

endclass : average
```

- `average` is the simplest one: a `write(int t)` that accumulates, and a `report_phase()` that divides and reports at the end
- That is the **shape** the three of them are going to have: they receive one roll at a time, and the result only exists once everything is over

One producer, many listeners

The other two: the body changes, not the shape

```
function void write(int t);
   rolls[t]++;
endfunction : write
```

```
function void write(int t);
   the_roll = t;
   dice_cg.sample();
endfunction : write
```

- `histogram` keeps the rolls in an associative array and draws the bars in the `report_phase`
- `coverage` copies the roll into `the_roll` and calls `sample()`: it is the covergroup of the conventional testbench, with the sampling tied to the data instead of to the clock
- The three `write()`s are different on the inside and **identical on the outside**: same name, same argument. That is what is going to let us treat them the same

One producer, many listeners

The producer: it rolls the dice and returns a number

```
class dice_roller extends uvm_component;
   `uvm_component_utils(dice_roller);

   rand byte die1;
   rand byte die2;

   constraint d6 {
      die1 >= 1;
      die1 <= 6;
      die2 >= 1;
      die2 <= 6;
   }

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

   function byte two_dice();
      byte the_roll;
      void'(randomize());
      the_roll = die1 + die2;
      return the_roll;
   endfunction : two_dice

endclass : dice_roller
```

- `dice_roller` randomizes two bytes with a `constraint` that keeps them between 1 and 6, and returns the sum
- That is all it does, and it is right that it should be all: **it produces the data and knows nothing about who uses it**
- Look at what it does **not** have: not a handle to `coverage_h`, nor to `histogram_h`, nor to `average_h`. That list lives somewhere else, and that is where the problem is

One producer, many listeners

The one handing it out: the test, and it should not be

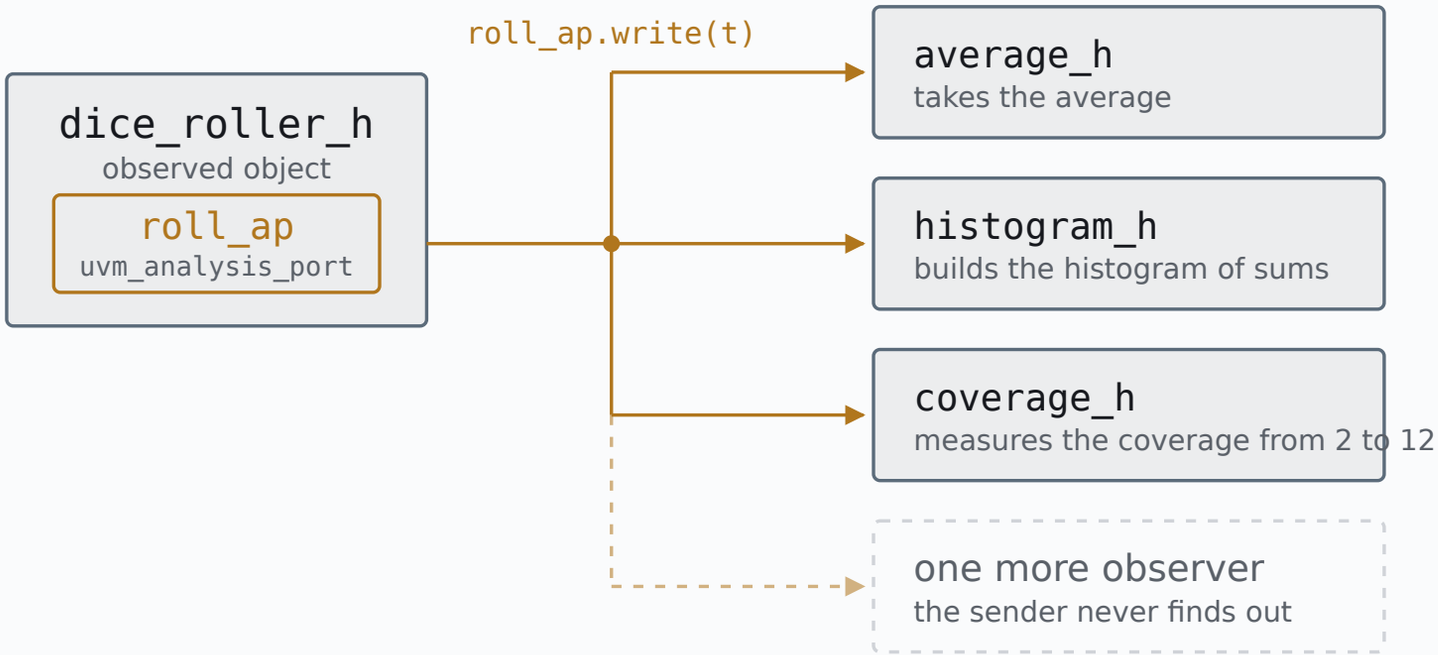
- The `dice_test` extends `uvm_test`, instantiates the four components and starts
- It works. But look at the `run_phase`, and in particular at what is inside the `repeat (20)`

```
task run_phase(uvm_phase phase);
   int the_roll;
   phase.raise_objection(this);
   // This part needs attention.
   // What belongs here is:
   //  the Observer design pattern
   //  a UVM analysis port
   // Otherwise none of the OOP magic is being used:
   // these are plain functions passing data along
   repeat (20) begin
      the_roll = dice_roller_h.two_dice();
      coverage_h.write(the_roll);
      histogram_h.write(the_roll);
      average_h.write(the_roll);
   end
   phase.drop_objection(this);
endtask : run_phase
```

One producer, many listeners

Observer Design Pattern

- Whoever publishes a tweet does not know who reads it or what they do with it, and yet it reaches everybody. That is the **Observer**
- An object produces a piece of data and publishes it. The ones who need that data subscribe. The producer **does not keep the list**
- Here the observed one is `dice_roller_h`; the observers —*subscribers*— are `coverage_h`, `histogram_h` and `average_h`
- The property that buys everything: adding a fourth observer **does not touch one line** of the one producing

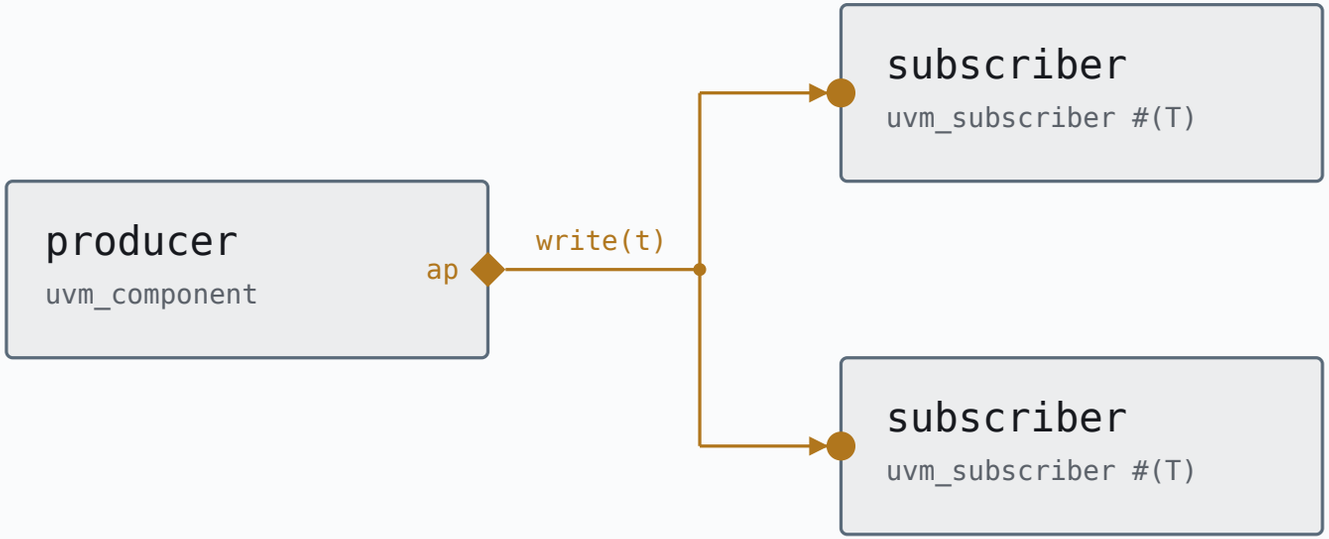


The arrows go one way only: the sender does not know who or how many listen.

One producer, many listeners

The pattern, ready-made: the two ends of the wire

- UVM brings the pattern ready-made, in two classes that are the two ends of the wire:
  - *uvm_analysis_port:* Sends data to a set of subscribers (observers)
  - *uvm_subscriber:* An extension of uvm_component that lets the component subscribe to a uvm_analysis_port



◆ uvm_analysis_port declared by the producer; publishes with .write()

● analysis_export comes with uvm_subscriber; tied with .connect()

In the example: dice_roller publishes the roll; coverage, histogram and average receive it.

One producer, many listeners

uvm_analysis_port

- We declare a variable of type *uvm_analysis_port* that states the type of data it is going to carry
- We instantiate the analysis port in *build_phase*
- We write data into the port through the *.write()* method
- Once we write data into the port, it goes to all of its subscribers
- We use the *connect()* method to connect the subscribers to the port. This method takes a single argument, and it is the subscriber's **export** (`analysis_export`), not another port

```
// Same as version 01, but publishing: instead of returning the number it writes
// it to an analysis port, and stops knowing who reads it.
class dice_roller extends uvm_component;
   `uvm_component_utils(dice_roller);

   rand byte die1;
   rand byte die2;

   constraint d6 {
      die1 >= 1;
      die1 <= 6;
      die2 >= 1;
      die2 <= 6;
   }
```

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

   // 1) declare the port, with the data type it will carry
   uvm_analysis_port #(int) roll_ap;

   // 2) instantiate it in build_phase, with new(): ports do not go through the factory
   function void build_phase(uvm_phase phase);
      roll_ap = new("roll_ap", this);
   endfunction : build_phase

   task run_phase(uvm_phase phase);
      int the_roll;
      phase.raise_objection(this);
      repeat (20) begin
         void'(randomize());
         the_roll = die1 + die2;
         // 3) write: the port calls write() on every subscriber
         roll_ap.write(the_roll);
      end
      phase.drop_objection(this);
   endtask : run_phase

endclass : dice_roller
```

One producer, many listeners

uvm_subscriber

- We extend the *uvm_subscriber* class parametrically, so that it knows what type of data it is going to read
- The class gives us an object called *analysis_export*
- The class requires us to write a method called *write()* that takes a single argument *t*, of the same type as the extension of the class
- In our example, we have 3 subscribers (average, coverage and histogram)

```
// uvm_subscriber:
//    extends the uvm_component class and lets you hook onto an analysis port
//    It is a parameterizable class!
//    The class gives you something and asks for something in return
//       * It gives you an object called analysis_export
//       * It asks you to write a write() method that handles the data

class coverage extends uvm_subscriber #(int);
   `uvm_component_utils(coverage);
   int the_roll;

   covergroup dice_cg;
      coverpoint the_roll {bins twod6[] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};}
   endgroup

   function new(string name, uvm_component parent = null);
      super.new(name, parent);
      dice_cg = new();
   endfunction : new
```

```
   function void write(int t);
      the_roll = t;
      dice_cg.sample();
   endfunction : write

   function void report_phase(uvm_phase phase);
      `uvm_info("COVERAGE", $sformatf("COVERAGE: %2.0f%%", dice_cg.get_inst_coverage()),
                UVM_NONE)
   endfunction : report_phase

endclass : coverage
```

One producer, many listeners

The producer, publishing now

```
// 1) declare the port, with the data type it will carry
uvm_analysis_port #(int) roll_ap;

// 2) instantiate it in build_phase, with new(): ports do not go through the factory
function void build_phase(uvm_phase phase);
   roll_ap = new("roll_ap", this);
endfunction : build_phase

task run_phase(uvm_phase phase);
   int the_roll;
   phase.raise_objection(this);
   repeat (20) begin
      void'(randomize());
      the_roll = die1 + die2;
      // 3) write: the port calls write() on every subscriber
      roll_ap.write(the_roll);
   end
   phase.drop_objection(this);
endtask : run_phase
```

- The three numbered steps are everything that changed with respect to the previous version: declare the port, instantiate it in `build_phase`, write with `write()`
- `roll_ap = new(...)`: ports are **not** created with the factory. They are the wiring of the testbench, not interchangeable pieces
- The `two_dice()` that returned a number is gone: now the roll gets published and the producer carries straight on

One producer, many listeners

connect_phase(): where the wiring gets assembled

- We need to link the producer with the consumer
- UVM provides a phase method (*connect_phase()*) to connect the objects
- UVM calls the build_phase method TOPDOWN, and once it is done with every object in the hierarchy, UVM calls the connect_phase method BottomUP
- The connection process has 2 fundamental parts
    - uvm_subscribers contain an object called analysis_export, which we must not instantiate since it comes along when we extend the uvm_subscriber class
    - The uvm_analysis_port class provides a method called connect()

One producer, many listeners

connect_phase(): where the wiring gets assembled

- A single line of `connect_phase()` hooks the subscriber up to the analysis port: `ap.connect(sub_h.analysis_export)`

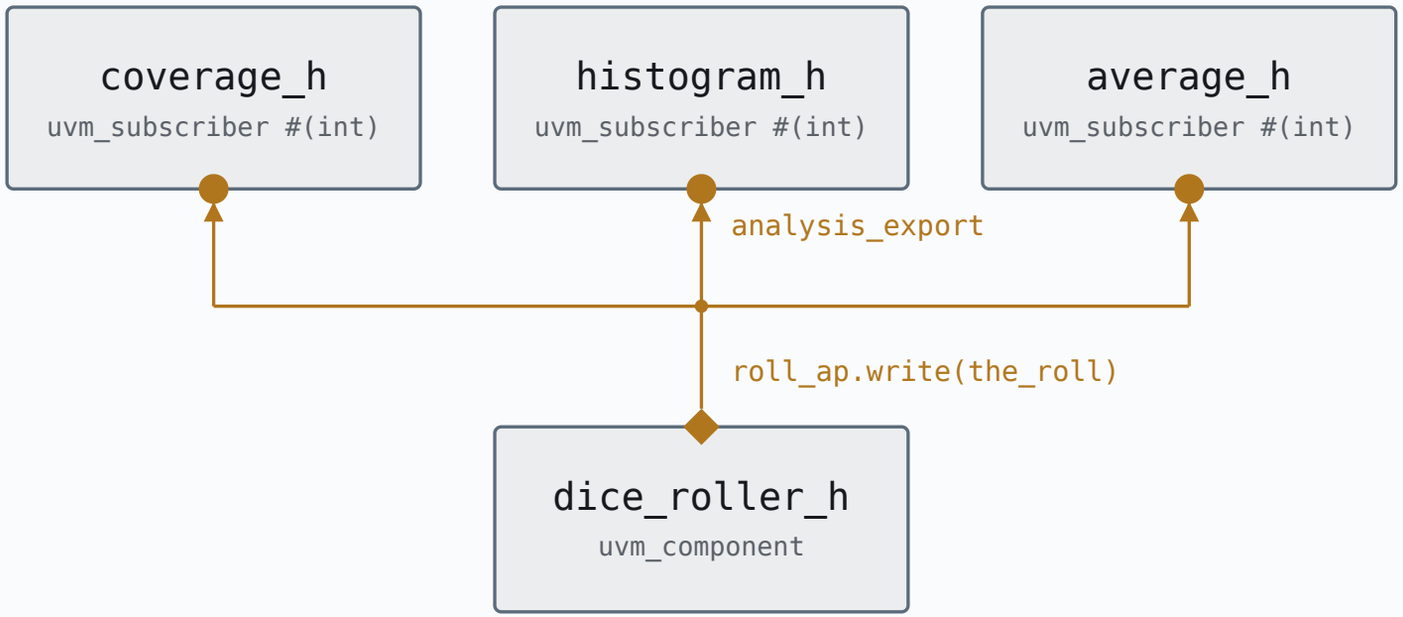
```
// build_phase: UVM calls it TOP-DOWN, the parent first.
function void build_phase(uvm_phase phase);
   dice_roller_h = new("dice_roller_h", this);
   coverage_h = new("coverage_h", this);
   histogram_h = new("histogram_h", this);
   average_h = new("average_h", this);
endfunction : build_phase

// connect_phase: BOTTOM-UP, and always port.connect(export).
function void connect_phase(uvm_phase phase);
   dice_roller_h.roll_ap.connect(coverage_h.analysis_export);
   dice_roller_h.roll_ap.connect(histogram_h.analysis_export);
   dice_roller_h.roll_ap.connect(average_h.analysis_export);
endfunction : connect_phase
```

One producer, many listeners

The solution, wired up

- In the connection diagram we can see that the dice_roller class has a uvm_analysis_port object called roll_ap
- Each subscriber has an analysis_export object
- The connection between the two is made through the connect() method



`connect_phase: roll_ap.connect(coverage_h.analysis_export)`

A single write() on roll_ap reaches all three: the producer does not know who listens.

One producer, many listeners

Summary of the unit

- Two objects talk to each other in **exactly two ways**, and the difference is whether they run in the same thread or not
- **Same thread**: one calls the other's method and comes straight back. It is a `function`, **it cannot block**. It is the `write()` of an analysis port
- **Different threads**: they have to be coordinated and somebody is going to wait. It is a `task`, **it can block**. It is `put`/`get`, and they are the last two sections
- The question in front of any UVM port is always the same: **can this block?** The signature tells you before you read the implementation
- The **analysis port** solves the underlying problem: the one who publishes **does not know who reads it**, and adding a reader does not touch one line
- It is the same old *observer* pattern, and it is the same move the BFM made in interfaces and BFM: one level up

A single place that watches the wire

A single place that watches the wire

- Today the scoreboard and the coverage each watch **the signals** on their own: both wait for `done`, both read `op_set`, both know the protocol
- That is the same code written twice, and the day the protocol changes you have to remember both places
- The missing piece has a name in the industry: the **monitor**. A single one watches the bus, assembles the transaction and publishes it
- The scoreboard and the coverage become **subscribers**, like the dice of the previous unit: they never touch a wire again
- In UVM jargon those two form the *analysis layer*, and that is where the name `uvm_analysis_port` comes from

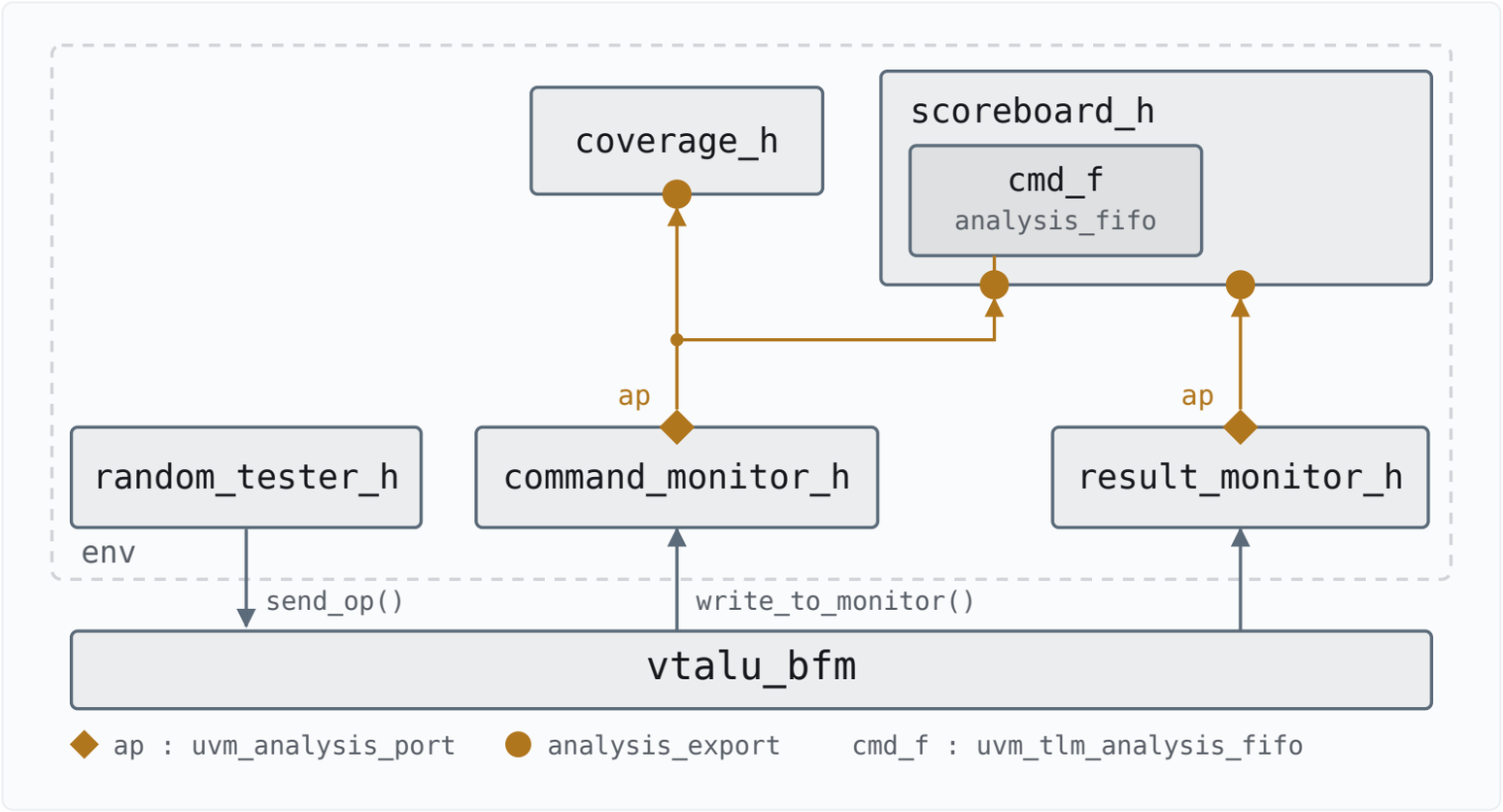
A single place that watches the wire

Which pieces show up

- `command_monitor` — watches the start of every operation and publishes what was asked for
- `result_monitor` — watches the `done` and publishes what the DUT answered
- The **BFM** stops being only signals: it gains three `always` and a handle to each monitor
- The **scoreboard** and the **coverage** stop waiting for edges and get a `write()` instead
- The `tester` never finds out about any of it: it goes on sending stimulus just like yesterday

A single place that watches the wire

Testbench diagram



A single place that watches the wire

The handles the BFM keeps

- Up to now the class read the interface. Here it flips around: **the interface keeps a handle to the class** and it is the interface that calls the method

```
interface vtalu_bfm;
   import vtalu_pkg::*;

   // These handles get assigned in the build_phases of the classes
   command_monitor        command_monitor_h;
   result_monitor         result_monitor_h;

   byte unsigned          A;
   byte unsigned          B;
   bit                    clk;
   bit                    reset_n;
   bit                    start;
   wire                   done;
   wire            [15:0] result;
   wire                   ovf;
   operation_t            op_set;
```

- These handles are set by each monitor in its `build_phase`, right after pulling the BFM out of the config_db

```
   function void build_phase(uvm_phase phase);
      virtual vtalu_bfm bfm;

      if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
         `uvm_fatal("COMMAND MONITOR", "Failed to get BFM")
```

```
    bfm.command_monitor_h = this;

    ap = new("ap", this);

endfunction : build_phase
```

A single place that watches the wire

Monitoring the VTALU commands

- Three `always` in the BFM: the command one fires when an operation starts, the reset one on the falling edge of `reset_n`, and the result one when the DUT raises `done`

```
// Here is the first monitor (commands)
// A very simple FSM. If start is up, check whether this is a new command
// If it is, send it to the TB with commands_monitor.write_to_monitor()
always @(posedge clk) begin : cmd_monitor
   bit new_command;
   if (!start) new_command = 1;
   else if (new_command) begin
      command_monitor_h.write_to_monitor(A, B, op_set);
      new_command = (op_set == 3'b000);  // handle no_op
   end
end : cmd_monitor

always @(negedge reset_n) begin : rst_monitor
   if (command_monitor_h != null)  //guard against VCS time 0 negedge
      command_monitor_h.write_to_monitor(A, B, rst_op);
end : rst_monitor

always @(posedge clk) begin : rslt_monitor
   if (done) result_monitor_h.write_to_monitor(result);
end : rslt_monitor
```

- And on the other side, the method they give notice to: it packs the `command_s` up and publishes it on the analysis port

```
function void write_to_monitor(byte A, byte B, bit [2:0] op);
   command_s cmd;
   cmd.A = A;
   cmd.B = B;
   cmd.op = op2enum(op);
   `uvm_info("COMMAND MONITOR", $sformatf("A:0x%2h B:0x%2h op: %s", A, B, cmd.op.name()
             ), UVM_HIGH)
   ap.write(cmd);
endfunction : write_to_monitor
```

A single place that watches the wire

The VTALU Coverage Class as a Subscriber

- With the `command_monitor` there is **a single place** in the whole testbench that knows what a command looks like on the wire
- Whoever needs to know which operation ran subscribes to its analysis port and implements `write()`. They never look at a signal again

```
// With the analysis port this is far simpler,
// because nothing here has to know about the ALU signals:
// what arrives is a command_s struct that wraps the command
//
function void write(command_s t);
      A = t.A;
      B = t.B;
      op_set = t.op;
      op_cov.sample();
      zeros_or_ones_on_ops.sample();
endfunction : write
```

A single place that watches the wire

Subscribing to multiple analysis ports

- The basic UVM analysis port mechanism lets a uvm_subscriber grab a single analysis_port
- However, in cases like a scoreboard, we need one and the same component to receive data from 2 analysis ports
- The reason for the limit: `uvm_subscriber` gives you **a single** `write()`, and that is the method the port calls. Two ports have nowhere to come in
- UVM solves it without splitting the component, with the *uvm_tlm_analysis_fifo* class, which is **unbounded by construction**: its constructor passes `size = 0` (`tlm1/uvm_tlm_fifos.svh:281`, with the comment "analysis fifo must be unbounded")
- And `write()` is a `function` that does `void'(this.try_put(t))`: it throws the return value away. A bounded FIFO would drop **without warning** — which is why this one is built with no cap
- It is parameterized and it has two faces: an *analysis_export* on one side —which gets connected like any subscriber— and a `try_get()` on the other
- `try_get()` takes an element out and returns 0 if the FIFO is empty, without blocking
- Which means: it turns *"they tell me when it arrives"* into *"I go and get it when I need it"*

A single place that watches the wire

Subscribing to multiple analysis ports

- Scoreboard class:

```
uvm_tlm_analysis_fifo #(command_s) cmd_f;

function void build_phase(uvm_phase phase);
   cmd_f = new("cmd_f", this);
endfunction : build_phase

// Every time the DUT hands a result to the analysis port, this method
// gets triggered
function void write(shortint t);
   shortint  predicted_result;
   command_s cmd;
   cmd.op = no_op;

   // Here is where the prediction happens: look in our FIFO for the command
   // that was sent, skipping the resets and the no_ops with the
   // do....while
   // since whenever there is a result there must be an operation,
   // a result with no operation behind it is an error
   do
   if (!cmd_f.try_get(cmd)) `uvm_fatal("SCOREBOARD", "No command in self checker")
   while ((cmd.op == no_op) || (cmd.op == rst_op));

   case (cmd.op)
      add_op: predicted_result = cmd.A + cmd.B;
      sub_op: predicted_result = cmd.A - cmd.B;
      and_op: predicted_result = cmd.A & cmd.B;
      xor_op: predicted_result = cmd.A ^ cmd.B;
      mul_op: predicted_result = cmd.A * cmd.B;
   endcase  // case (op_set)

   if (predicted_result != t)
      `uvm_error("SCOREBOARD", $sformatf(
```

```
                    "FAILED: A: %2h  B: %2h  op: %s actual result: %4h   expected: %4h",
                    cmd.A,
                    cmd.B,
                    cmd.op.name(),
                    t,
                    predicted_result
                    ))
    endfunction : write
```

A single place that watches the wire

The other way: `uvm_analysis_imp_decl

```
`uvm_analysis_imp_decl(_cmd)        // makes the class uvm_analysis_imp_cmd
`uvm_analysis_imp_decl(_result)

class scoreboard extends uvm_component;
   uvm_analysis_imp_cmd    #(command_s, scoreboard) cmd_imp;
   uvm_analysis_imp_result #(shortint,  scoreboard) result_imp;

   function void write_cmd(command_s t);    ... endfunction   // one write per port
   function void write_result(shortint t);  ... endfunction
endclass
```

- The macro **generates a class** for each suffix, and each one calls a different `write_`: that way a component receives from two ports without a FIFO
- It is what you are going to see in most production code, so **you have to be able to read it**
- The course uses the FIFO anyway, for an underlying reason: `write_cmd()` runs **when the command arrives**, and the scoreboard needs it **when the result arrives**. The FIFO gives it that control; the macro forces it to keep the data by hand

A single place that watches the wire

Subscribing to the monitors

- We connect the analysis ports to the monitors using the connect_phase() method in the env class:

```
class env extends uvm_env;
   `uvm_component_utils(env);

   random_tester   random_tester_h;
   coverage        coverage_h;
   scoreboard      scoreboard_h;
   command_monitor command_monitor_h;
   result_monitor  result_monitor_h;

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

   function void build_phase(uvm_phase phase);
      random_tester_h = random_tester::type_id::create("random_tester_h", this);
      coverage_h = coverage::type_id::create("coverage_h", this);
      scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
      command_monitor_h = command_monitor::type_id::create("command_monitor_h", this);
      result_monitor_h = result_monitor::type_id::create("result_monitor_h", this);

   endfunction : build_phase

   function void connect_phase(uvm_phase phase);

      result_monitor_h.ap.connect(scoreboard_h.analysis_export);
      command_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
      command_monitor_h.ap.connect(coverage_h.analysis_export);

   endfunction : connect_phase
```

```
endclass
```

A single place that watches the wire

When the DUT does not answer in order

- The scoreboard of this section does a `try_get()` from a FIFO: the first command that went in is the one that matches the first result that came out. That holds because **the VTALU answers in order**, one operation at a time
- A real bus does not. In AXI every transfer carries an **ID** and the responses can come back crossed over: the FIFO starts comparing the response of one with the request of another, and screams on every one

```
transaccion esperado [int];              // associative, indexed by whatever pairs them up

function void write_cmd(transaccion c);  esperado[c.id] = c;  endfunction

function void write_result(respuesta r);
   if (!esperado.exists(r.id)) `uvm_error("SB", "a response nobody asked for")
   else begin comparar(esperado[r.id], r); esperado.delete(r.id); end
endfunction
```

- The pattern is always the same: **a table indexed by whatever pairs them up**, a `delete` once it has been compared, and at the end of the run the table **has to be left empty**
- That check at the end is half the value: what stayed inside are the requests the DUT never answered

A single place that watches the wire

On one side of the monitor you speak in signals, on the other in transactions

- The **monitor** is the only class that knows a `clk` exists. Coverage and scoreboard receive data and have no idea where it came from
- The Observer of talking to several objects, applied to the DUT: the monitor publishes and does not keep the list. Adding an observer is one line in the `connect_phase`
- And the limit, which is the section that follows: all of this happens in **a single thread**. The `always` of the BFM calls `write_to_monitor()`, which calls `ap.write()`, which calls the `write()`s of the subscribers — a chain of functions, without time moving

A single place that watches the wire

Summary of the unit

- Up to yesterday the scoreboard and the coverage each watched **the signals** on their own: the same protocol code written twice
- The piece that was missing has a name in the industry: the **monitor**. A single one watches the wire and **publishes** what it saw
- There are two: `command_monitor` publishes what was asked for, `result_monitor` publishes what the DUT answered
- The scoreboard and the coverage become **subscribers**: they stop waiting for edges and only implement `write()`
- That pair —monitors that publish, subscribers that analyse— is the **analysis layer**, and that is where the name of the port comes from
- The `tester` **never finds out about any of it**. That is the proof the split came out right: changing the analysis does not touch the stimulus
- And the scoreboard with a FIFO holds **as long as the DUT answers in order**. When it does not, the pattern is a table by ID that has to be left empty at the end

When somebody has to wait

You were already doing this, with modules

- The name is scary and you have been using the mechanism forever: two modules with ports, each one with its `always`, passing data to each other

```
// Sends the data through the shared variable and signals it by toggling get_it
// Then the producer blocks on the put_it signal
module producer (
    output byte shared,
    input  bit  put_it,
    output bit  get_it
);
   initial
      repeat (3) begin
         $display("Sent %0d", ++shared);
         get_it = ~get_it;
         @(put_it);
      end
endmodule : producer

// The consumer blocks on the get_it signal, waiting for data from the producer
// when the producer toggles that signal, it processes the data and toggles put_it
// when done, to unblock the producer
module consumer (
    input  byte shared,
    output bit  put_it,
    input  bit  get_it
);
   initial
      forever begin
         @(get_it);
         $display("Received: %0d", shared);
         put_it = ~put_it;
```

```
    end
endmodule : consumer
```

- That **is** communication between threads. The only thing missing is doing it between objects, which have no ports

When somebody has to wait

An object has no port list

- What SystemVerilog does have is shared handles, semaphores and mailboxes
- That is enough to solve it, and that is the problem: **everybody solves it differently**, and afterwards nobody can read the testbench of the person next to them
- UVM standardizes a single way, in two pieces:
  - *Ports*: they get instantiated in a `uvm_component` so that its `run_phase()` can talk to another thread. *Put port* to send, *get port* to receive
  - *TLM FIFO*: the object that joins a put port to a get port. `new()` creates it with **size 1** by default — `new(name, parent, size = 1)` —, and with `0` it would be unbounded. The one in the example uses the default
- With size 1 the FIFO is not a buffer, it is a **meeting point**: the one putting waits for the other one to take out

When somebody has to wait

The producer as an object

- The producer declares a `uvm_put_port` and calls `put()`. That is all: no signal handshake, no shared flag, no semaphore

```
// UVM solution to interthread communication
//  It has 2 parts:
//    * Ports-Objects: instantiated in our uvm_components so that
//        run_phase can talk to other threads
//    * TLM Fifos (they hold only 1 element)

class producer extends uvm_component;
   `uvm_component_utils(producer);

   int shared;
   // uvm_put_port is a parameterized class
   // this class takes care of the whole synchronization mess around
   // get_it and put_it
   uvm_put_port #(int) put_port_h;

   function void build_phase(uvm_phase phase);
      put_port_h = new("put_port_h", this);
   endfunction : build_phase

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

   task run_phase(uvm_phase phase);
      phase.raise_objection(this);
      repeat (3) begin
         put_port_h.put(++shared);
         `uvm_info("PRODUCER", $sformatf("Sent %0d", shared), UVM_MEDIUM)
      end
```

```
      phase.drop_objection(this);
   endtask : run_phase
endclass : producer
```

When somebody has to wait

The consumer as an object

- On the other side, `uvm_get_port` and a `get()`. With the FIFO empty, `get()` **blocks** and the thread stays there
- When the producer puts a piece of data in, the `get()` unblocks and carries on with the data already read. Nobody wrote the synchronization: the FIFO gives it

```
class consumer extends uvm_component;
   `uvm_component_utils(consumer);

   // uvm_get_port handles all the interthread coordination
   uvm_get_port #(int) get_port_h;
   int shared;

   function void build_phase(uvm_phase phase);
      get_port_h = new("get_port_h", this);
   endfunction : build_phase

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

   task run_phase(uvm_phase phase);
      forever begin
         get_port_h.get(shared);
         `uvm_info("CONSUMER", $sformatf("Received: %0d", shared), UVM_MEDIUM)
      end
   endtask : run_phase
endclass : consumer
```

When somebody has to wait

Instantiating the three: two components and a FIFO

- The `uvm_tlm_fifo` takes **the same parameter** as the two ports: if they do not match, it does not compile
- The three get instantiated in the `build_phase` of the test, like any component. The FIFO is a `uvm_component` too
- And the order is the usual one: UVM calls every `build_phase` from the top down, and only once it has finished does it start the `connect_phase` from the bottom up

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

   producer producer_h;
   consumer consumer_h;
   uvm_tlm_fifo #(int) fifo_h;

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

   function void build_phase(uvm_phase phase);
      producer_h = new("producer_h", this);
      consumer_h = new("consumer_h", this);
      fifo_h = new("fifo_h", this);
   endfunction : build_phase
```

```
    // Connect phase: the two ports and the fifo already exist -- build_phase
    // runs first, top-down -- so this is where they get wired together.
    function void connect_phase(uvm_phase phase);
       producer_h.put_port_h.connect(fifo_h.put_export);
       consumer_h.get_port_h.connect(fifo_h.get_export);
    endfunction : connect_phase

endclass : communication_test
```

When somebody has to wait

Connecting them: `put_export` and `get_export`

- It is the same `connect()` as the analysis ports: on one side the **port**, on the other the **export** the FIFO provides
- `uvm_tlm_fifo` exposes twelve handles; the two that get used are `put_export` for the one that puts and `get_export` for the one that takes out —which is an alias of `get_peek_export`, and that is the name you are going to see in `print_topology()` —
- The rule that holds for all of TLM: **the port connects to the export**, never two ports to each other

```
// Connect phase: the two ports and the fifo already exist -- build_phase
// runs first, top-down -- so this is where they get wired together.
function void connect_phase(uvm_phase phase);
   producer_h.put_port_h.connect(fifo_h.put_export);
   consumer_h.get_port_h.connect(fifo_h.get_export);
endfunction : connect_phase
```

```
UVM_INFO @ 0: reporter [RNTST] Running test communication_test...
UVM_INFO producer.svh(28) @ 0: uvm_test_top.producer_h [PRODUCER] Sent 1
UVM_INFO consumer.svh(19) @ 0: uvm_test_top.consumer_h [CONSUMER] Received: 1
UVM_INFO producer.svh(28) @ 0: uvm_test_top.producer_h [PRODUCER] Sent 2
UVM_INFO consumer.svh(19) @ 0: uvm_test_top.consumer_h [CONSUMER] Received: 2
```

```
UVM_INFO producer.svh(28) @ 0: uvm_test_top.producer_h [PRODUCER] Sent 3
UVM_INFO consumer.svh(19) @ 0: uvm_test_top.consumer_h [CONSUMER] Received: 3
UVM_INFO $UVM_HOME/src/base/uvm_report_server.svh(1009) @ 0: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    7
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[CONSUMER]     3
[PRODUCER]     3
[RNTST]     1
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 0s
```

When somebody has to wait

NON-blocking communication

- Blocking is fine as long as the one waiting has nothing better to do. With a clock in the middle, standing there waiting means losing an edge
- That is what the non-blocking versions are for: `try_put()` and `try_get()`
- They return 1 if they could and **0 if they could not**, and carry on. That is why they are `function` and not `task`: they consume no time

```
class consumer extends uvm_component;
   `uvm_component_utils(consumer);

   uvm_get_port #(int) get_port_h;
   virtual clk_bfm clk_bfm_i;
   int shared;

   function void build_phase(uvm_phase phase);
      get_port_h = new("get_port_h", this);
      clk_bfm_i = example_pkg::clk_bfm_i;
   endfunction : build_phase

   // Consumer and producer run on different clocks
   // The consumer has its first positive edge at 7ns
   //    and then checks the data every 14ns, because it uses bfm.clk (= 14ns)
   // The producer produces one datum every 17ns
   task run_phase(uvm_phase phase);
      forever begin
         @(posedge clk_bfm_i.clk);
         if (get_port_h.try_get(shared))
            `uvm_info("CONSUMER", $sformatf("Received: %0d", shared), UVM_MEDIUM)
      end
```

```
   endtask : run_phase

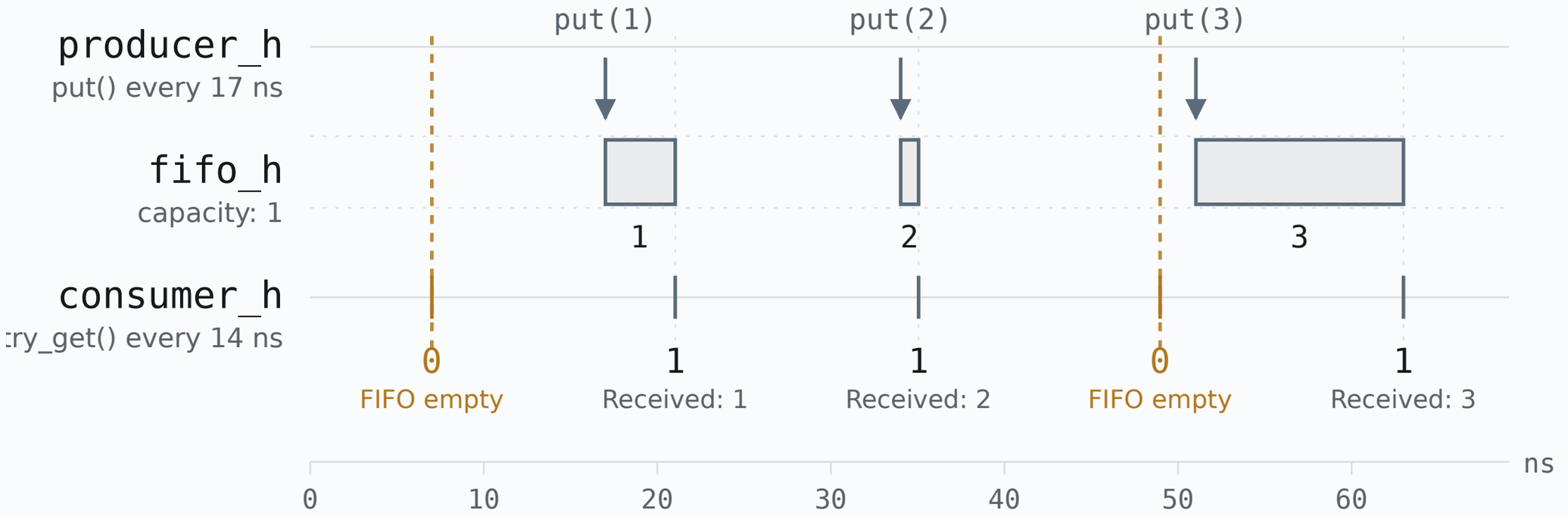
   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction : new

endclass : consumer
```

```
UVM_INFO @ 0: reporter [RNTST] Running test communication_test...
UVM_INFO producer.svh(18) @ 17: uvm_test_top.producer_h [PRODUCER] Sent 1
UVM_INFO consumer.svh(21) @ 21: uvm_test_top.consumer_h [CONSUMER] Received: 1
UVM_INFO producer.svh(18) @ 34: uvm_test_top.producer_h [PRODUCER] Sent 2
UVM_INFO consumer.svh(21) @ 35: uvm_test_top.consumer_h [CONSUMER] Received: 2
UVM_INFO producer.svh(18) @ 51: uvm_test_top.producer_h [PRODUCER] Sent 3
UVM_INFO consumer.svh(21) @ 63: uvm_test_top.consumer_h [CONSUMER] Received: 3
UVM_INFO $UVM_HOME/src/base/uvm_report_server.svh(1009) @ 68: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :    7
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[CONSUMER]     3
[PRODUCER]     3
[RNTST]     1
- $UVM_HOME/src/base/uvm_root.svh:633: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.052 2026-09-05
- Verilator: $finish at 68ns
```

When somebody has to wait

NON-blocking communication: the timeline

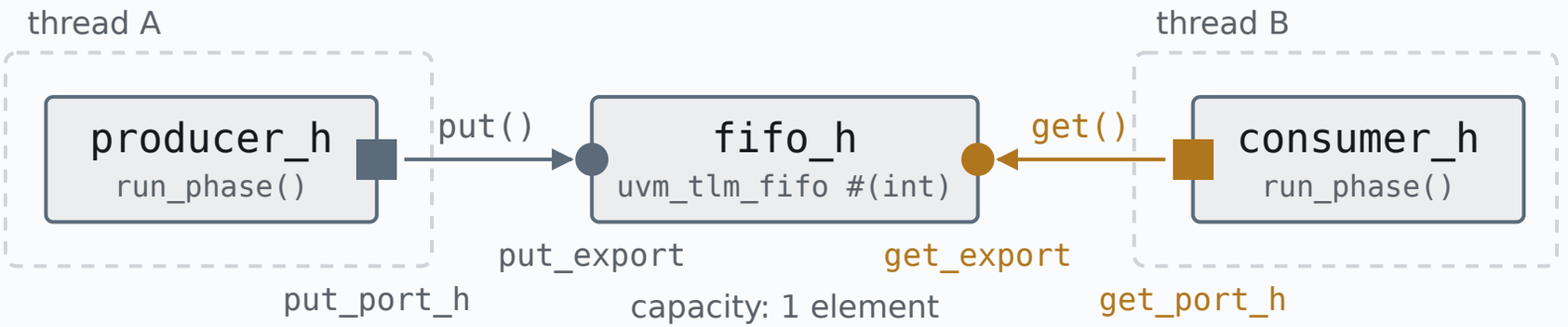


With `get()` blocking, the consumer was left waiting on both edges at amber.

- The producer puts a piece of data in every 17 ns and the consumer looks every 14 ns: the clocks run differently, and on two edges the FIFO is empty
- There `try_get()` returns 0 and the consumer carries on. With `get()` it stayed waiting

When somebody has to wait

How TLM diagrams are read



- `get()` blocks the `consumer_h` until the FIFO has a datum
- `put()` blocks the `producer_h` if the FIFO already holds its only element
- `try_get()` and `try_put()` do not block: they return 0 if they could not

- The convention holds for all the UVM material you are going to read afterwards:
  - *Square:* put port / get port — the one that **starts** the call
  - *Circle:* export — the one that **receives** it and implements it
  - *Diamond:* analysis port — the one from the two previous sections

When somebody has to wait

fork: the three ways of starting in parallel

```
fork  wait_done();  count_cycles();  join        // goes on when BOTH have finished
fork  wait_done();  count_cycles();  join_any    // goes on with THE FIRST; the other stays alive
fork  wait_done();  count_cycles();  join_none   // goes on NOW; both stay running
```

Variant	The parent goes on...	What it is used for
`join`	when **all** of them finish	two checks that both have to close
`join_any`	when **the first** one finishes	response against a timeout
`join_none`	**right away**	launching threads that live for the whole test

- The difference between the three is not how they start —all three start everything at once— but **when the one that started them goes on**
- You have used `join_none` already: it is the `fork` of the day 2 `testbench` class, and it is what UVM does on its own with one `run_phase()` per component
- The branch left alive after a `join_any` **does not die on its own**. It goes on until it finishes — or until somebody kills it, which is the next slide

When somebody has to wait

disable fork and wait fork: turning off what stayed on

```
// The response-against-timeout idiom. The outer fork ISOLATES
fork begin
   fork
      begin  wait_done();          `uvm_info("BFM", "it arrived", UVM_LOW)  end
      begin  repeat (100) @(posedge clk);  `uvm_error("BFM", "timeout")        end
   join_any
   disable fork;      // kills the sister that lost, and nobody else
end join

wait fork;            // kills nobody: it waits for ALL the children of this thread
```

- `disable fork` kills **every child process of the thread that executes it**. That is why the outer `fork begin ... end join`: without it, it also takes down whatever was already running
- `wait fork` is the opposite: it kills nothing, it **waits** for the children to finish. It is what a test uses so as not to close with transactions in flight
- The `join_any` + `disable fork` pair is the timeout of every production BFM. It gets written once and copied forever
- Whichever finishes first wins: if `done` arrives before the 100 edges, the timeout `uvm_error` is not printed; if it does not, the `uvm_info` of `"it arrived"` is not printed. The `disable fork` kills the loser

When somebody has to wait

⚠ *The trap: the `for` index inside the `fork`*

```
// A -- a single variable, shared by the three threads. A join_none runs
// nothing: it schedules. By the time the three run, the for is over.
for (i = 0; i < 3; i++)
  fork got = {got, $sformatf("%0d", i)}; join_none
wait fork;
check("int i outside the for, no copy", "333");

// B -- declared inside the for: one per pass, and nothing to copy.
for (int j = 0; j < 3; j++)
  fork got = {got, $sformatf("%0d", j)}; join_none
wait fork;
check("int j declared in the for", "012");

// C -- the slide's one: the automatic as the first line of a begin/end.
// Here the copy is NOT made when the fork is scheduled, so k reads the same
// i that A does and gives the same answer as A. This is the finding.
for (i = 0; i < 3; i++)
  fork begin
    automatic int k = i;
    got = {got, $sformatf("%0d", k)};
  end join_none
wait fork;
check("fork begin automatic k = i; end  (the slide's)", "333");

// D -- the LRM 9.3.2 one: the automatic declared in the fork, no begin/end.
// Same workaround, written where the LRM puts it, and this one does copy.
for (i = 0; i < 3; i++)
  fork
    automatic int k = i;
    got = {got, $sformatf("%0d", k)};
  join_none
wait fork;
check("fork automatic k = i;  no begin/end (LRM 9.3.2)", "012");
```

- A `join_none` **does not execute anything yet**: it leaves the thread ready and goes on. By the time the thread runs, the `for` has already finished and the variable holds the last value
- If the variable is declared **inside** the `for`, each pass has its own and there is no problem. It is what the LRM says and what Verilator does
- If it comes from outside —an `int` of the `run_phase`, a field of the class— it has to be **copied** with an `automatic`, and **where** the copy is written changes the result: inside a `fork begin ... end` it **does not copy**; declared in the `fork` itself, with no `begin`/`end` —which is the LRM 1800-2017 §9.3.2 form— it does
- Measured on Verilator 5.052, and the file ends in `$fatal` the day any of it changes (`make repros`): the four variants print `3 3 3`, `0 1 2`, **`3 3 3`** and `0 1 2`

When somebody has to wait

Summary of the unit

- **Inter**-thread communication between objects is the equivalent of what ports do between modules: passing data from one thread to another
- UVM provides it with *uvm_put_port*, *uvm_get_port* and *uvm_tlm_fifo*
- Any object that wants to communicate with another thread has to instantiate a port and connect it to a FIFO
- Careful with the pair of names: the *analysis ports* of the two previous sections are **intra**-thread —`write()` is a `function` and runs in the thread of the one publishing—; this is **inter**-thread, and that is why `put()` and `get()` are `task`
- And underneath it all there is `fork`: `join` waits for all, `join_any` for the first, `join_none` for none. `disable fork` kills the children and `wait fork` waits for them
- Now we have to use this to wire our TB: we are going to separate the stimulus generation from the driver of the DUT

Who waits for whom

What to send and how to send it

- The analysis ports split the analysis: the monitor watches, the subscribers interpret. The same is still missing on the stimulus side
- Today the `base_tester` does **two** things: it picks the operation *and* it applies it on the BFM cycle by cycle. They are two jobs that change for different reasons
- The **what** changes with every test. The **how** changes only if the protocol of the DUT changes — which means, almost never
- Separating them leaves the protocol written in a single place: the **driver**. Every new test writes what to send and never looks at an `@(negedge clk)` again
- And since the two run in different threads, the joint is the one from inter-thread communication: `uvm_put_port`, `uvm_get_port` and a `uvm_tlm_fifo` in the middle

Who waits for whom

The tester does two jobs, and one of them is not its own

- This is how the `base_tester` of the analysis ports was left, and it does **two** jobs:
  - it picks which operation to send
  - and it applies it on the BFM, cycle by cycle
- The second one is the one that should not be there: it is protocol, and the protocol does not change when the test changes
- The class that takes that job away is called a **driver**: it takes a piece of data from the testbench and turns it into signals

```
task run_phase(uvm_phase phase);
   byte unsigned iA;
   byte unsigned iB;
   operation_t   op_set;

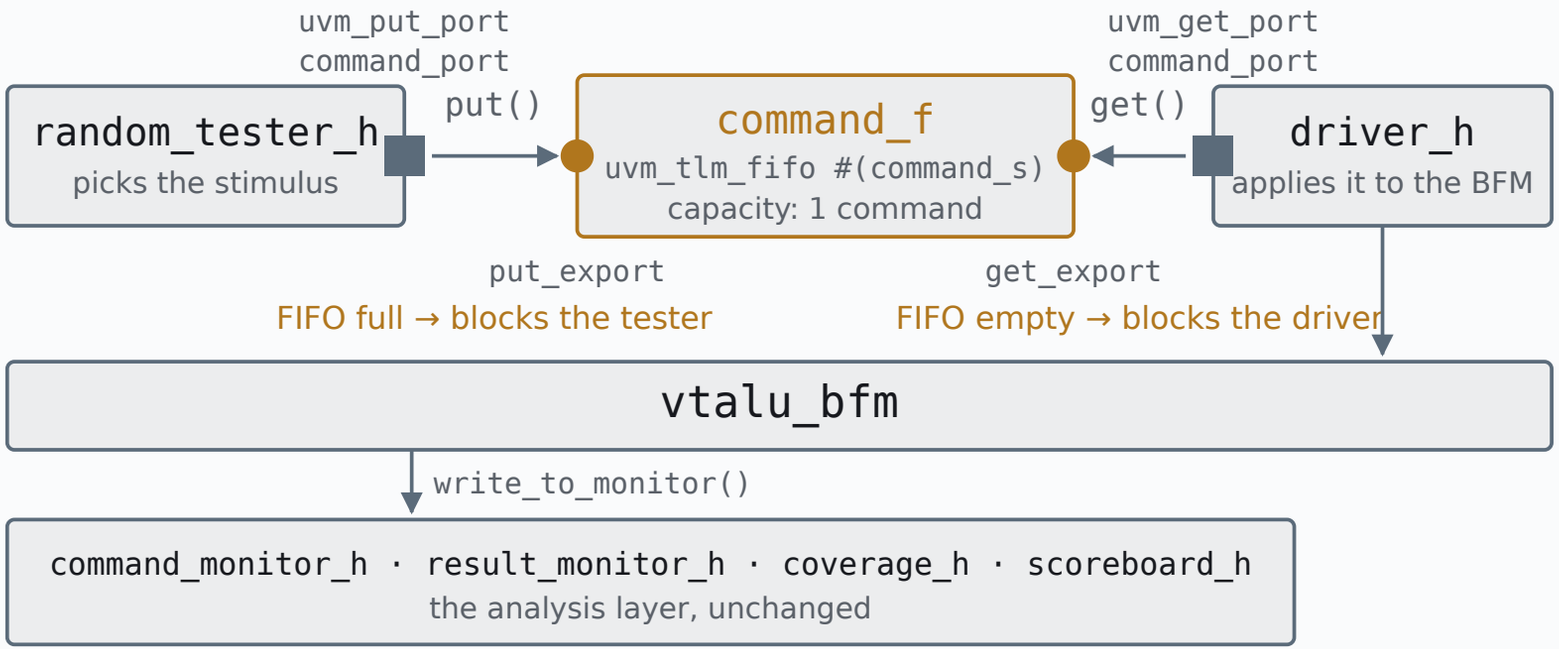
   phase.raise_objection(this);
   bfm.reset_alu();
   repeat (1000) begin : random_loop
      op_set = get_op();
      iA = get_data();
      iB = get_data();
      bfm.send_op(iA, iB, op_set);
   end : random_loop
   #500;
```

```
    phase.drop_objection(this);
  endtask : run_phase
```

Who waits for whom

How it ends up split

- The result: one class decides **what** to send and another one knows **how** to send it, and between the two there is a FIFO instead of a method call


uvm_put_port
command_port
uvm_get_port
command_port
random_tester_h
picks the stimulus
put()
command_f
uvm_tlm_fifo #(command_s)
capacity: 1 command
get()
driver_h
applies it to the BFM
put_export
FIFO full → blocks the tester
get_export
FIFO empty → blocks the driver
vtalu_bfm
write_to_monitor()
command_monitor_h · result_monitor_h · coverage_h · scoreboard_h
the analysis layer, unchanged

Who waits for whom

The env: seven objects and five `connect()`

- Mantra: "Ports connect to exports"

```
function void build_phase(uvm_phase phase);
   command_f = new("command_f", this);
   random_tester_h = random_tester::type_id::create("random_tester_h", this);
   driver_h = driver::type_id::create("driver_h", this);

   coverage_h = coverage::type_id::create("coverage_h", this);
   scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
   command_monitor_h = command_monitor::type_id::create("command_monitor_h", this);
   result_monitor_h = result_monitor::type_id::create("result_monitor_h", this);
endfunction : build_phase

function void connect_phase(uvm_phase phase);
   // PORTS CONNECT TO EXPORTS!
   driver_h.command_port.connect(command_f.get_export);
   random_tester_h.command_port.connect(command_f.put_export);

   result_monitor_h.ap.connect(scoreboard_h.analysis_export);

   command_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
   command_monitor_h.ap.connect(coverage_h.analysis_export);
endfunction : connect_phase
```

Who waits for whom

The `base_tester`, without a single signal

- Only one thing changes: where there was a handle to the BFM there is now a `uvm_put_port #(command_s)` called `command_port`
- `random_tester` and `add_tester` **do not get touched**. They only define `get_op()` and `get_data()`, and neither of the two knew how the stimulus was applied
- That is the return on having separated things properly in the env: the change stopped at the base class

```
virtual class base_tester extends uvm_component;

   `uvm_component_utils(base_tester)

   // the handle to the BFM is replaced by a uvm_put_port
   uvm_put_port #(command_s) command_port;

   function void build_phase(uvm_phase phase);
      command_port = new("command_port", this);
   endfunction : build_phase

   pure virtual function operation_t get_op();

   pure virtual function byte get_data();

   task run_phase(uvm_phase phase);
      byte unsigned iA;
```

```
    byte unsigned iB;
    operation_t   op_set;
    command_s     command;

    phase.raise_objection(this);
    command.op = rst_op;
    command_port.put(command);
    repeat (1000) begin : random_loop
      command.op = get_op();
      command.A = get_data();
      command.B = get_data();
      command_port.put(command);
    end : random_loop
    #500;
    phase.drop_objection(this);
  endtask : run_phase
```

Who waits for whom

The driver: the only one that touches the wire

- A `forever` with two lines: `get()` a command —which blocks until there is one — and apply it with `send_op()`. The whole protocol lives here

```
class driver extends uvm_component;
   `uvm_component_utils(driver)

   virtual vtalu_bfm bfm;

   uvm_get_port #(command_s) command_port;

   function void build_phase(uvm_phase phase);
      if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
         `uvm_fatal("DRIVER", "Failed to get BFM")
      command_port = new("command_port", this);
   endfunction : build_phase

   task run_phase(uvm_phase phase);
      command_s command;

      forever begin : command_loop
         // Blocking: if there are no commands to send, it waits
         command_port.get(command);
         bfm.send_op(command.A, command.B, command.op);
      end : command_loop
   endtask : run_phase
```

Who waits for whom

Summary of the unit

- The testbench ended up split into three layers that change for different reasons: the **stimulus** (tester), the **protocol** (driver and monitors) and the **analysis** (scoreboard and coverage)
- None of them knows the insides of the others: between them only commands and results travel
- What is still missing is for the testbench to know how to **tell what is going on**, which is the unit that follows

Review · Day 4

1 of 5 · Observer Pattern

In the Observer pattern, what does the observed object know about its observers?

a How many there are and of what type

b Only the first one that subscribed

c It knows them because they get handed to it in the constructor, one by one

d Nothing: not how many there are, not who they are ✓

It knows nothing — and that ignorance is the whole point. Adding a fourth subscriber does not force you to touch one line of the one publishing.

Review · Day 4

2 of 5 · Analysis Ports

A `uvm_subscriber` needs data from two different analysis ports. How does that get solved?

a. By implementing the `write()` method twice

b. With a `uvm_tlm_analysis_fifo` for the second port ✓

c. By registering the component twice in the factory, once per port

d. By connecting both ports to the same `analysis_export`

With a `uvm_tlm_analysis_fifo` — a `uvm_subscriber` has a single `write()`, so it can only listen to one port. The FIFO gives an `analysis_export` on one side and a `try_get()` on the other. It is what the VTALU scoreboard does. The other way is `` `uvm_analysis_imp_decl ``, which manufactures one `write_` per suffix: they are not two `write()` in the same class, which SystemVerilog does not allow.

Review · Day 4

3 of 5 · Intra vs. inter thread

When the monitor publishes and the `write()`s of the subscribers run, how many threads are involved?

a. One per subscriber

b. Two: the publisher's and the subscriber's

c. Only one: `write()` is a function call ✓

d. One per subscriber plus the monitor's, and UVM syncs them at the end of the delta

Only one, and it is intra-thread communication — `write()` is a `function`, not a `task`: it consumes no time and runs in the thread of the one publishing. That is precisely why **another** mechanism (put/get + FIFO) is needed to talk between threads.

Review · Day 4

4 of 5 · Put and get ports

The consumer calls `get()` and the `uvm_tlm_fifo` is empty. What happens?

- **a** It blocks until the producer puts a piece of data in ✓
- **b** It returns 0 and carries on
- **c** A UVM fatal error
- **d** It returns the last piece of data read, which stays in the FIFO until overwritten

It blocks — `get()` is blocking, and that is why it is declared `task` and not `function`. That is the whole synchronization: no hand-written signal handshake is needed.

Review · Day 4

5 of 5 · try_get()

And `try_get()` with the FIFO empty?

- **a** It blocks just like `get()`
- **b** It returns 1 with a garbage value
- **c** It waits one clock cycle and tries again, up to the timeout of the phase
- **d** It returns 0 immediately, without blocking ✓

It returns 0 and carries on — it is the non-blocking version, and that is why it can be a `function`. Notice that the scoreboard uses it in a `do ... while` to skip over the `no_op`s and the `rst_op`s.

Exercise · Day 4 · 1 of 2

One more observer, without touching the ones already watching

```
$ cd code/ejercicios/d4 && bash run.sh
```

- A `uvm_subscriber #(command_s)` that counts the commands
- Instantiate it and connect it in the `env`, without touching the connections that are already there
- It is marked by cross-checking: your count has to come out the same as the `command_monitor`'s

Exercise · Day 4 · 2 of 2

The #500 is a patch

```
$ cd code/ejercicios/d4b && bash run.sh
```

- The *put and get* testbench with **one line changed**: the `env`'s FIFO comes unbounded
- The tester empties its thousand commands at `t = 0`, waits the usual `#500`, drops the objection — and the simulation ends **in green with 13 commands on the bus**
- It gets fixed in the driver: an objection **while there is a command in flight**
- `env.svh` does not get touched: the unbounded FIFO is the DUT of this exercise

Yesterday we left...

Where the testbench stands

- The monitor **publishes** and knows nobody: the scoreboard and the coverage hang off it, and adding a third does not touch the one that publishes
- The TLM FIFOs settled who waits for whom, and `fork`/`join` who runs with whom
- But the data is still **three loose fields**: `A`, `B` and `op` travel separately, they get copied by hand, and comparing them means writing the whole comparison

Agenda

Day 5 · ≈ 4 h 30 · unit 6 · the data

- Copying an object that contains another
- Transactions
- Constrained random

By the end of the day you can:

- **Write** a constraint that reaches a corner case, and **measure** whether the `dist` does what it says
- **Close** a bin that was left open, with a directed case asked for through `randomize() with {}`
- **Explain** the difference between copying the handle and copying the object, and what happens when the object contains another

Copying an object that contains another

The handle is not the object

- From the transactions on, the testbench data stops being a `struct` and becomes **objects that travel**: the monitor creates them, the scoreboard compares them
- And there a problem appears that a `struct` did not have. A `struct` gets copied when you assign it; an object does **not**: `obj1_h = obj2_h` copies the handle, not the data
- Which means two parts of the testbench can be looking at the same object without knowing it. If one modifies it, the other sees the change halfway through
- Copying it for real has to be written, and written **at every level of the hierarchy**, or the fields from above get lost in silence
- This section is the OOP that is missing so that the transactions can be copied, compared and printed without lying

Copying an object that contains another

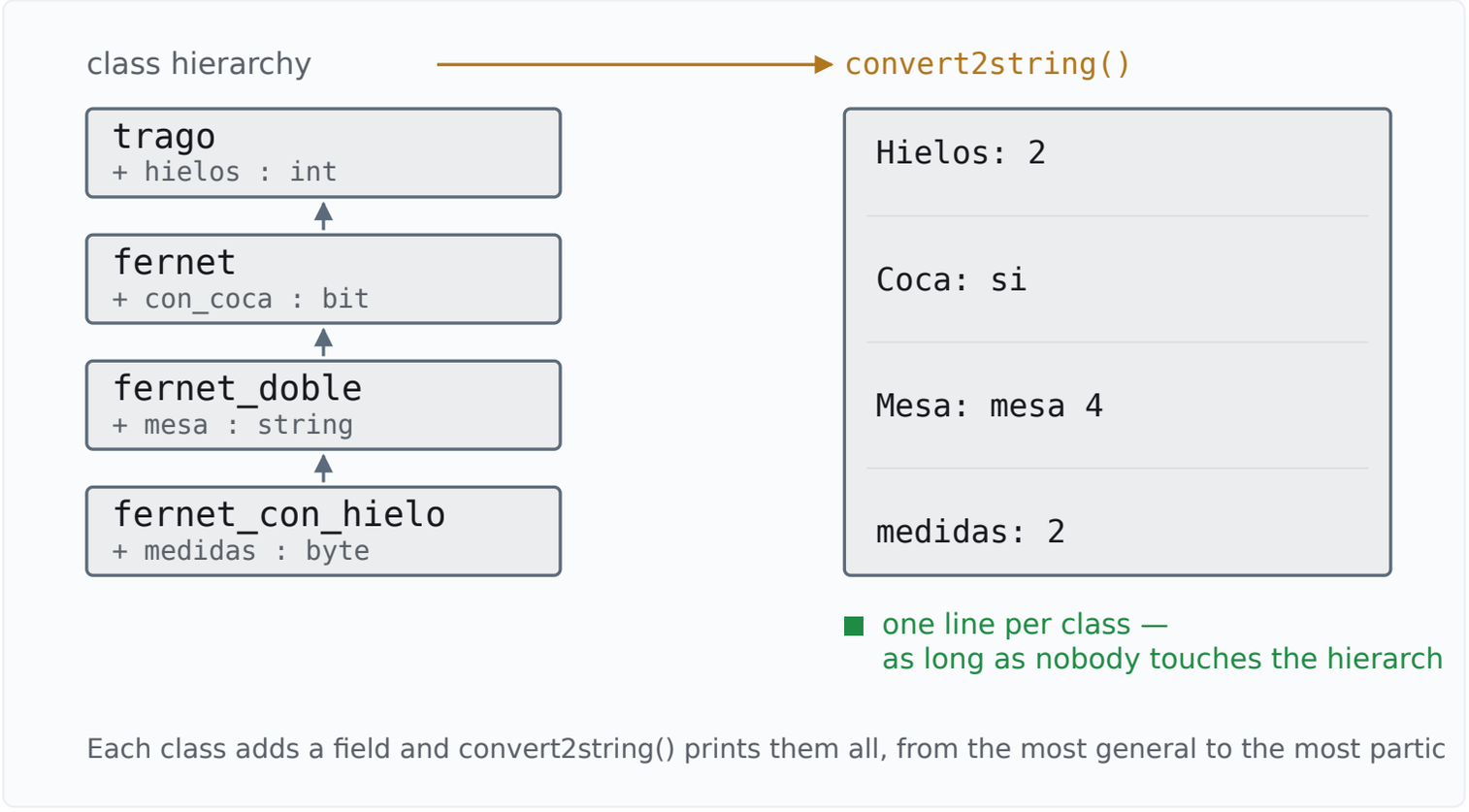
Never duplicate code

> *"If we find ourselves copying code and making small modifications to it inside our program, we are doing things wrong. Code must never be duplicated."*

In OOP, instead of copying code...

Copying an object that contains another

convert2string(): the problem



Each class adds a field and convert2string() prints them all, from the most general to the most partic

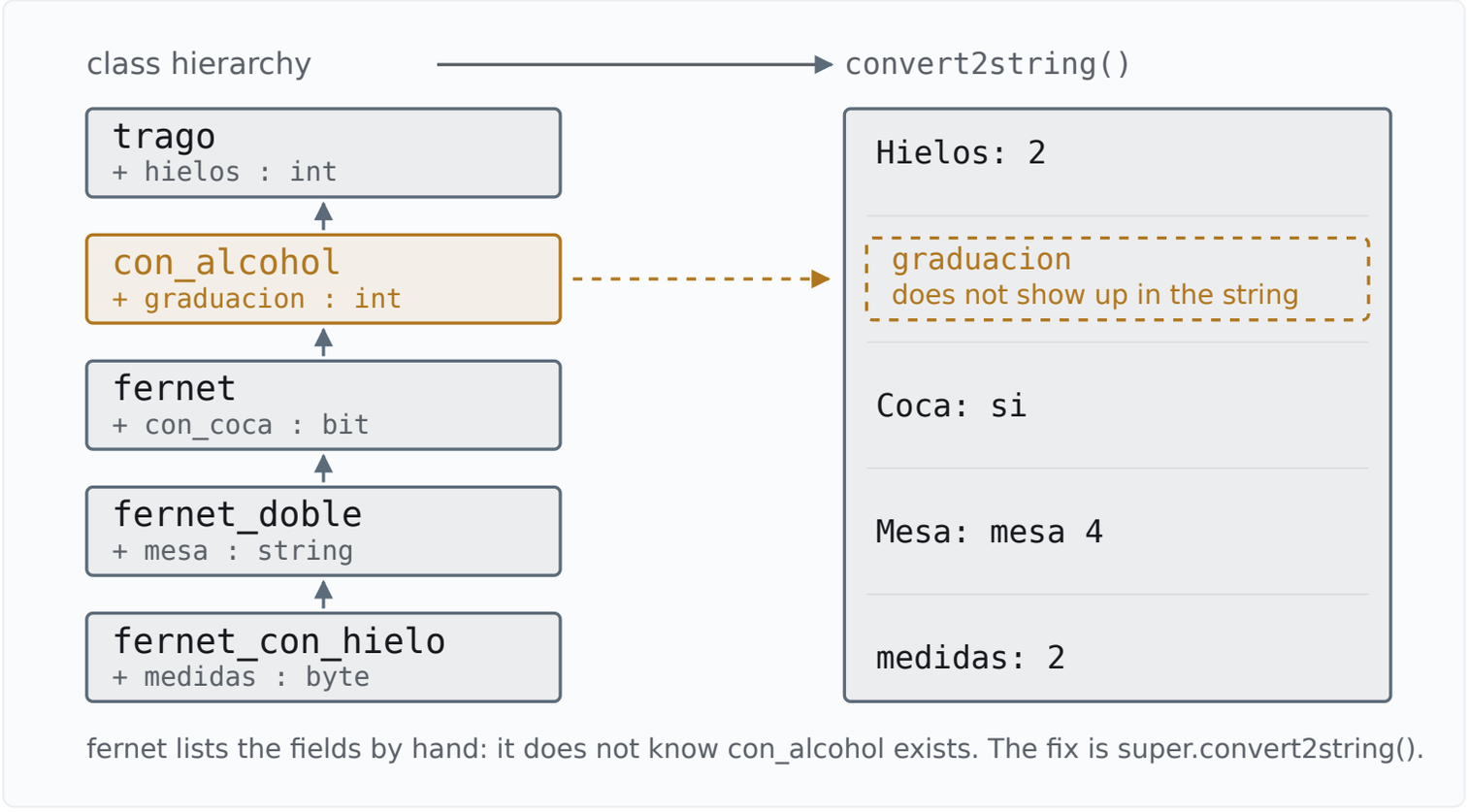
- The hierarchy goes from the general to the particular, and each class adds fields of its own
- We want two things from it: to be able to **copy** a whole object and to **print** it whole, without anything from the levels above getting lost
- This first version works. Count how many times the name of a field of the base class shows up

```
// ILLUSTRATION -- not compiled. The WRONG way to write convert2string!
// The classes are drawn as comments: only the methods are on screen.
//class trago;
   virtual function string convert2string();
      return $sformatf("Hielos: %0d", hielos);
   endfunction : convert2string
//class fernet extends trago;
   function string convert2string();
      string coca_s;
      coca_s = (con_coca) ? "si" : "no";
      return $sformatf("Hielos: %0d  Coca: %s", hielos, coca_s);
   endfunction : convert2string

//class fernet_doble extends fernet;
   function string convert2string();
      return {$sformatf("hielos: %0d  coca: %0b  mesa: %s",
                  hielos, con_coca, mesa) };
   endfunction : convert2string
```

Copying an object that contains another

convert2string(): the deep version



fernet lists the fields by hand: it does not know con_alcohol exists. The fix is super.convert2string().

- The day somebody puts a class **in the middle**, the ones below go on printing what they knew and the one in the middle does not show up. And there is no error: the method still returns a string
- The rule that fixes it: each class prints **only what is its own** and asks `super` for the rest
- Nobody knows what is further up, so inserting a level breaks nothing

```
// ILLUSTRATION -- not compiled. The same three methods written deep, with the
// classes drawn as comments. The version that does compile is deep.sv.
//virtual class trago;
   virtual function string convert2string();
      return $sformatf("Hielos: %0d", hielos);
   endfunction : convert2string

//class con_alcohol extends trago;

   function string convert2string();
      return {super.convert2string(),
              $sformatf("\ngraduacion: %0d",graduacion)};
   endfunction : convert2string

//class fernet extends con_alcohol;

   function string convert2string();
      string coca_s;
      coca_s = (con_coca) ? "si" : "no";
      return {super.convert2string(), "\nCoca: ",coca_s};
   endfunction : convert2string
```

Copying an object that contains another

The equals sign copies nothing

- `obj1_h = obj2_h` **copies nothing**: it leaves two handles pointing at the same object
- If you modify it through one, the other sees it. There is no notice, there is no warning
- With a `struct` that did not happen: it gets copied when you assign it. It is the difference to keep in mind from here to the end of the course

Copying an object that contains another

Copying for real is two steps

Copying for real is two steps, and they have to be written: **instantiate** a new object and **hand it the data** field by field

```
// ILLUSTRATION -- not compiled. The initial block of deep.sv:110-118, on its own.
initial begin
   fernet_hielo1_h =
       new(.hielos(2), .con_coca(1), .graduacion(2), .mesa("mesa 4"), .medidas(2));
   $display("\n--- Fernet 1 ---\n", fernet_hielo1_h.convert2string());
   fernet_hielo2_h = new();
   $display("\n--- Fernet 2 before the copy ---\n", fernet_hielo2_h.convert2string());
   // do_copy is a method that copies member by member
   // it suffers from the same problem already seen with convert2string()
   // Each class can copy the variables defined in that class
   // super.do_copy() has to be used to copy the inherited ones
   fernet_hielo2_h.do_copy(fernet_hielo1_h);
   $display("\n--- Fernet 2 after the copy ---\n", fernet_hielo2_h.convert2string());
end
```

- The `new()` is the first step and it is not optional: without a new object on the other side, there is nowhere to copy to
- The second step is by hand, field by field. SystemVerilog has no built-in deep copy
- And field by field means **the fields of the classes above too**, which is exactly what gets forgotten

Copying an object that contains another

do_copy(): each class touches what is its own

do_copy(): Every class in the hierarchy has to call its parent class, and therefore every do_copy() method needs the same type of argument. But we have different classes... We use *polymorphism*

```
class fernet_con_hielo extends fernet_doble;
   byte medidas;

   function new(int hielos = 0, bit con_coca = 0, int graduacion = 0,
                string mesa = "", byte medidas = 0);
      super.new(hielos, graduacion, con_coca, mesa);
      this.medidas = medidas;
   endfunction : new

   function string convert2string();
      return {super.convert2string(), "\n", $sformatf("medidas: %0d", medidas)};
   endfunction : convert2string

   //do_copy passes the argument up to its parent class
   function void do_copy(trago copia);
      fernet_con_hielo copia_fernet_con_hielo;
      super.do_copy(copia);
      // Then the copy gets cast to a variable of its own class type
      $cast(copia_fernet_con_hielo, copia);
      this.medidas = copia_fernet_con_hielo.medidas;
   endfunction : do_copy
```

```
endclass : fernet_con_hielo
```

Copying an object that contains another

Summary: the terrible solution

```
function string convert2string();
   return {
      $sformatf(
          "hielos: %0d  coca: %0b  mesa: %s  medidas: %0d",
          hielos,
          con_coca,
          mesa,
          medidas
      )
   };
endfunction : convert2string
```

- It is the `convert2string()` of `fernet_con_hielo`, the bottom-most class: it prints **all four fields by hand**, including the three it inherited
- It works, and that is why it is dangerous. The day `fernet` gains a field, this method goes on compiling and goes on printing — less than it should
- The whole file is in `code/u6/jerarquias/wrong.sv`. The three `do_copy()`s in there do call `super`; the one with the vice is `bad_copy()`, at the very bottom

Copying an object that contains another

Summary: the deep solution

```
function string convert2string();
   string coca_s;
   coca_s = (con_coca) ? "si" : "no";
   return {super.convert2string(), "\nCoca: ", coca_s};
endfunction : convert2string

function void do_copy(trago copia);
   fernet copia_fernet;
   super.do_copy(copia);
   $cast(copia_fernet, copia);
   this.con_coca = copia_fernet.con_coca;
endfunction : do_copy
```

- The same two methods, written the way they should be: each one calls `super` and then touches **a single field**, its own
- `convert2string()` **concatenates** what the one above returned; `do_copy()` **calls the one above first** and then copies what is its own
- With that shape, putting a new class in the middle of the hierarchy works on its own: the chain of `super` includes it without anybody editing anything

Copying an object that contains another

Summary of the unit

- `obj1_h = obj2_h` **copies nothing**: it leaves two handles pointing at the same object. If you touch it through one, the other sees it, and there is not even a warning
- With a `struct` that did not happen —it gets copied when you assign it—, and that is the difference to keep in mind until the end of the course
- **Copying for real is two steps**: instantiate a new object and hand it the data. That is where the `clone()` of the transactions comes from
- In a hierarchy, each class has to touch **a single field, its own**, and call `super`. Writing the four fields by hand is the terrible solution
- For that to work, the `do_copy()` / `do_compare()` all take **the same type of argument**, the base class. It is polymorphism, once again
- The day somebody puts a class **in the middle** of the hierarchy, the deep version goes on working by itself. The other one you have to go out and fix

Transactions

The testbench is well divided up, and the data is not

- The testbench is already well divided up, but **the data is still a `struct`**: `command_s` for the commands and a bare `shortint` for the results
- A `struct` does nothing by itself. Printing a command is a `$sformatf` written by hand, and it is written **three times**: the two monitors and the scoreboard
- The same with randomizing and with comparing: every component builds its own, and the day the `struct` gains a field you have to remember all three
- And the result is not even a `struct`: it is a **bare** `shortint`, so the `ovf` of the VTALU —the second output— has nowhere to travel
- A class, on the other hand, has methods: the data and what gets done with it live together, and the rest of the testbench shrinks

Transactions

A transaction is data with the methods that operate on that data

- It is the word the industry uses, and there is no magic class behind it: a **transaction** is that and nothing more
- The methods are four, and they are always the same:
  - `randomize()` — it fills itself in, respecting its `constraint`s
  - `convert2string()` — it prints itself
  - `do_copy()` — it copies itself
  - `do_compare()` — it compares itself
- What changes is not the data: it is **who knows things about it**. The `tester` used to know which values were legal; now it only says *"randomize yourself"*
- And the rest of the testbench shrinks: the three hand-written `$sformatf` become one call to `convert2string()`

Transactions

Defining a transaction: which class it extends

- Transactions are defined by extending the *uvm_sequence_item* base class and writing the following methods:
  - do_copy()
  - do_compare()
  - convert2string()
- What follows is turning the old `command_s` structure into a class, `command_transaction`, method by method

```
// ILLUSTRATION -- not compiled. The struct as it is in code/u5/put-get/vtalu_pkg.sv,
// the one this section turns into a class.
typedef struct {
   byte unsigned A;
   byte unsigned B;
   operation_t   op;
} command_s;
```

Transactions

The randomized fields: the "before"

- Today the stimulus gets built with two hand-written functions, `get_op()` and `get_data()`, which draw the values with `$random` — the Verilog-95 one, with a global seed
- The legal range of `A` and `B` and the bias to the edges are properties **of the data**, not of the test — and yet they live in the tester
- SystemVerilog already brings this ready-made, and there is no need to write a single `$urandom`

```
// ILLUSTRATION -- not compiled. The two functions as they are in
// code/u5/put-get/tb_classes/random_tester.svh, the section this one replaces.
//
// This function handles the fact that
// the operation bus has 8 possible values,
// but only 6 of them are valid operations
virtual function operation_t get_op();
  bit [2:0] op_choice;
  op_choice = $random;
  case (op_choice)
          3'b000 : return no_op;
          3'b001 : return add_op;
          3'b010 : return sub_op;
          3'b011 : return and_op;
          3'b100 : return xor_op;
          3'b101 : return mul_op;
          3'b110 : return rst_op;
```

```
          3'b111 : return rst_op;
  endcase // case (op_choice)
endfunction : get_op

// This function was meant to spread out
// the probability of getting all zeros (prob = 1/3),
// biasing the randomness a little so the prob is not 1/256,
// since that would make our coverage goal hard to reach
virtual    function byte get_data();
  bit [1:0]   zero_ones;
  zero_ones = $random;
  if (zero_ones == 2'b00)
    return 8'h00;
  else if (zero_ones == 2'b11)
    return 8'hFF;
  else
    return $random;
endfunction : get_data
```

Transactions

The randomized fields: the "after"

- Every SystemVerilog class brings an implicit `randomize()`, which picks values for the fields marked `rand`
- `get_op()` disappears: an `enum` randomizes itself over its legal values
- `get_data()` too, and it gets replaced by a `constraint` with `dist`, which is where the bias to the edges gets declared

```
class command_transaction extends uvm_sequence_item;
   `uvm_object_utils(command_transaction)
   rand byte unsigned A;
   rand byte unsigned B;
   rand operation_t   op;

   // :/ and NOT :=  ---  this is the line that reproduces the bias towards the
   // corner cases that get_data() of the Functional coverage section did by hand (1/4 at 00, 1/4 at FF,
   // 1/2 in the middle). With ":=" the weight applies to EVERY value of the range, so the
   // middle takes 254 out of 256 and 00 comes up once every 256: the distribution
   // ends up almost uniform and the corner bins never fill.
   // The Constrained Random unit has the experiment with the numbers.
   constraint data {
      A dist {
         8'h00 :/ 1,
         [8'h01 : 8'hFE] :/ 2,
         8'hFF :/ 1
      };
      B dist {
         8'h00 :/ 1,
         [8'h01 : 8'hFE] :/ 2,
         8'hFF :/ 1
```

```
    };
}
```

Transactions

The constructor: a `uvm_object` has no parent

- *uvm_sequence_item* extends *uvm_transaction*, and that one *uvm_object* — which means a transaction is **not** a *uvm_component*. It therefore has a simpler constructor
- A transaction **does not live in the tree of the testbench**: that tree is made of *uvm_components*, and it is the one UVM walks in `build_phase`. With no place in the tree there is no parent, and that is why the constructor only asks for a `name`
- Objects get created and destroyed all the time while the simulation runs; components get built once, before it starts, and they stay
- It is still worth giving them a name: it is the one that shows up in the UVM messages

```
// Constructor of a uvm_object: dead simple, it only takes a name. A
// transaction has no parent because it does not live in the component tree.
function new(string name = "");
   super.new(name);
endfunction : new
```

Transactions

do_copy(): copy the object, not the handle

- `uvm_object` brings two ready-made methods: `copy()`, which pours the data onto another object **that already exists**, and `clone()`, which also creates the new object
- Both work only if the class implements `do_copy()`: the library knows how to copy, but it does not know **which fields** your transaction has
- It is the same pattern as the rest of the section: `copy()` is the one that gets called, `do_copy()` is the one that gets written
- The only difference is the argument: it has to be a `uvm_object` —hence the `$cast`—. UVM calls it *rhs* (*right hand side*) by convention, not by requirement

```
function void do_copy(uvm_object rhs);
   command_transaction copied_transaction_h;

   // Checks that make DEBUG easier
   if (rhs == null)
      `uvm_fatal("COMMAND TRANSACTION", "Tried to copy from a null pointer")

   if (!$cast(copied_transaction_h, rhs))
      `uvm_fatal("COMMAND TRANSACTION", "Tried to copy wrong type.")

   super.do_copy(rhs);  // copy all parent class data
```

```
    A = copied_transaction_h.A;
    B = copied_transaction_h.B;
    op = copied_transaction_h.op;

  endfunction : do_copy
```

Transactions

The MOOCOW rule: sharing yes, modifying no

- Several components can see the same piece of data if they have handles to the same object. It is cheap and it works — as long as nobody modifies it
- The rule that holds it up is called **MOOCOW** —*Manual Obligatory Object Copy On Write*—, and that is how *The UVM Primer*, by Ray Salemi, christens it
- It says this: sharing a handle is fine **as long as you do not touch the object**. Whoever wants to modify, copies first
- It is a discipline of the team, not something the language enforces. UVM only puts up the tool: `clone()`, which returns a `uvm_object` and has to be cast

Transactions

`clone_me()`: the `$cast` written a single time

- Since `clone()` returns a `uvm_object`, every component that uses it ends up writing its own `$cast` — the same one, repeated
- The *UVM Primer* convention is to add a `clone_me()` to the transaction that does both things and returns the right type already
- UVM does not bring it: it is four lines written once in the data, so that none of the ones who use it have to remember the cast

```
function command_transaction clone_me();
   command_transaction clone;
   uvm_object tmp;

   tmp = this.clone();
   $cast(clone, tmp);
   return clone;
endfunction : clone_me
```

Transactions

`do_compare()`: comparing two objects without writing an `if` per field

- `compare()` comes with `uvm_object` and returns 1 if the two objects are equal. It is the one that gets called; the one that gets **written** is `do_compare()`
- It takes two arguments: the `rhs` —the other object— and a `uvm_comparer`, which is the comparison policy: how many differences to report and with what verbosity
- Almost nobody touches the comparer, but it has to be declared because the signature asks for it

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
   command_transaction compared_transaction_h;
   bit same;

   if (rhs == null)
      `uvm_fatal("RANDOM TRANSACTION", "Tried to do comparison to a null pointer");

   // First check whether we are comparing the same type
   if (!$cast(compared_transaction_h, rhs)) same = 0;
   else
      // Deep comparison: what the parent compares, plus the three fields of
      // this class
      same = super.do_compare(
          rhs, comparer
      ) && (compared_transaction_h.A == A) && (compared_transaction_h.B == B) &&
          (compared_transaction_h.op == op);
```

```
    return same;
endfunction : do_compare
```

Transactions

`convert2string()`: the transaction prints itself

- It is the method that returns the object as text. It gets written **once**, and it is used by the scoreboard, the monitor and anybody who has to report
- `$sformatf()` builds the string with the same format specifiers as always
- And the detail that makes a log readable: an `enum` has `name()`, which returns `add_op` instead of `3'b001`. Without that the error says a number and nobody reads it

```
function string convert2string();
   string s;
   // op.name() can be add_op, no_op ...
   s = $sformatf("A: %2h  B: %2h op: %s", A, B, op.name());
   return s;
endfunction : convert2string
```

Transactions

The field macros: reading them even if you do not write them

```
class command_transaction extends uvm_sequence_item;
   `uvm_object_utils_begin(command_transaction)
      `uvm_field_int (A,  UVM_DEFAULT | UVM_HEX)   // an integral field
      `uvm_field_enum(operation_t, op, UVM_DEFAULT)
      `uvm_field_object(payload, UVM_DEFAULT | UVM_REFERENCE)  // handle, not a copy
   `uvm_object_utils_end
endclass
```

- It is what you will find in **every VIP** you open. They generate `do_copy`, `do_compare`, `do_print`, `do_pack` and the automatic configuration, out of one declaration per field
- The second argument is a mask: `UVM_DEFAULT` turns on every operation, and `UVM_NOCOMPARE` / `UVM_NOPRINT` turn one off. The radix (`UVM_HEX`, `UVM_DEC`) is only for printing
- `UVM_REFERENCE` on a `uvm_field_object` copies **the handle**; without it, the copy is deep. It is the same decision as `do_copy()`, taken in a constant
- The course writes them by hand for the reason already given: that gets debugged, this does not. But reading them you do have to know

Transactions

The seven steps, and why there are seven

What it costs to change the data type of a TB that already exists. We look at **2, 3, 6 and 7**; **1, 4 and 5** are a direct port and are in the repo:

#	What	Where
1	a `result_transaction` for the trip back	`result_transaction.svh`
2	an `add_transaction`, additions only	`add_transaction.svh`
3	the two testers merge into one	`tester.svh`
4	the `command_monitor` publishes objects	`command_monitor.svh`
5	the `result_monitor`, the same	`result_monitor.svh`
6	the `scoreboard` uses `compare()`	`scoreboard.svh`
7	`add_test` overrides the **data**	`add_test.svh`

Transactions

Step 2 · an `add_transaction` that only adds

- `add_transaction` extends `command_transaction` and adds **a single** constraint to it: `op == add_op`
- That is enough to change the stimulus, and the `tester_h` never finds out: it uses the child class exactly as it used the mother
- Constraints **are inherited and they accumulate**: the bias to the edges of `A` and `B` is still there, because the one in the base class does not go away

```
class add_transaction extends command_transaction;
   `uvm_object_utils(add_transaction)

   constraint add_only {op == add_op;}

   function new(string name = "");
      super.new(name);
   endfunction
endclass : add_transaction
```

Transactions

Step 3 · two testers become one

```
task run_phase(uvm_phase phase);
   command_transaction command;

   phase.raise_objection(this);

   // ALL transactions come out of the factory, the directed ones too.
   // A new() here would compile and work just the same, but it would skip
   // add_test's set_type_override: the factory never hears about what you
   // build by hand. See the slide "the new() that eats the override".
   command = command_transaction::type_id::create("command");
   command.op = rst_op;
   command_port.put(command);

   repeat (1000) begin : random_loop
      command = command_transaction::type_id::create("command");
      // randomize() returns 0 if the constraints have no solution, and aborts
      // nothing. A bare assert() lets an unrandomized transaction through in
      // silence: the TB carries on, sending garbage.
      if (!command.randomize()) `uvm_fatal("TESTER", "randomize() failed")
      command_port.put(command);
   end : random_loop

   // Directed: the multiplier overflow case, which 1000 random operations
   // might never touch. It comes out of the factory like the others, but
   // since it is not randomized, add_transaction's constraint does not run:
   // the factory picks the TYPE, constraints only act inside randomize().
   command = command_transaction::type_id::create("command");
   command.op = mul_op;
   command.A = 8'hFF;
   command.B = 8'hFF;
   command_port.put(command);

   #500;
```

```
    phase.drop_objection(this);
  endtask : run_phase
```

- The testbench of the env had `base_tester` and `add_tester`: two classes for two kinds of stimulus
- Now there is **only one**. The `repeat` creates a transaction and tells it `randomize()`; what comes out of that is decided by the **data**, not by the tester
- That is the line that deletes a class: the decision moved from the structure to the type of the transaction

Transactions

Step 6 · the scoreboard compares objects, not numbers

- Both sides of the comparison are now `result_transaction`: the one that arrives from the `result_monitor` and the one the predictor builds
- The `write()` takes the result `t`, looks for the command that corresponds to it in the `command_monitor` queue, and asks `predict_result()` for the prediction
- The comparison comes down to one line — `predicted.compare(t)` — and the error message builds itself out of the `convert2string()` of the three transactions

```
function void write(result_transaction t);
   string data_str;
   command_transaction cmd;
   result_transaction predicted;

   do
   if (!cmd_f.try_get(cmd))
      `uvm_fatal("SELF CHECKER", "Missing command in self checker")
   while ((cmd.op == no_op) || (cmd.op == rst_op));

   predicted = predict_result(cmd);

   data_str = {
      cmd.convert2string(),
```

```
        " ==>  Actual ",
        t.convert2string(),
        "/Predicted ",
        predicted.convert2string()
    };

    if (!predicted.compare(t)) `uvm_error("SELF CHECKER", {"FAIL: ", data_str})
    else `uvm_info("SELF CHECKER", {"PASS: ", data_str}, UVM_HIGH)

endfunction : write
```

Transactions

Step 7 · the override, now on the data

```
class add_test extends random_test;
   `uvm_component_utils(add_test);

   function void build_phase(uvm_phase phase);
      command_transaction::type_id::set_type_override(add_transaction::get_type());
      super.build_phase(phase);
   endfunction : build_phase

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

endclass
```

- The whole test is **two lines**: the override and the `super.build_phase()`
- And the override no longer changes a **component** —the tester— but a **piece of data** —the transaction—. The structure of the testbench stopped having an opinion about the test
- `super.build_phase(phase)` goes **after** the override, and that order is not negotiable: the env gets built in there

Transactions

The new() that eats the override

```
command = new("command");                                    // NO
command.op = rst_op;

command = command_transaction::type_id::create("command");  // YES
command.op = rst_op;
```

- `set_type_override()` talks **to the factory**, and the factory only finds out about what goes through `type_id::create()`
- A `new()` builds the type that line says and nothing else: the override **does not reach it**. It compiles, it runs, and it does not fail — that object simply got left out
- It is the worst kind of bug: it does not break, it **lies**. With one test and one transaction it never shows; the day the override really mattered, it is already too late
- The rule, short: **every `uvm_object` that travels around the testbench comes out of `type_id::create()`** — the transactions of the tester, and also the ones the monitors build and the `predicted` of the scoreboard
- The *ports* and the *exports* are the exception: they are not in the factory and they get instantiated with `new()`, as we saw in one producer, many listeners. The *TLM FIFO* is registered, but the course builds it with `new()` anyway because the `size` goes through the constructor and `create()` cannot pass it

Transactions

And `randomize()` never goes on its own

```
assert (command.randomize());                                  // NO
if (!command.randomize()) `uvm_fatal("TESTER", "randomize() failed")  // YES
```

- `randomize()` returns **0** when the constraints have no solution. It does not abort, it prints nothing on its own: it returns 0 and carries on
- A bare `assert()` reports outside UVM: it does not go into the *Report Summary* —the same one we learned to read in reporting— and in many simulators execution carries on with the transaction **unrandomized**
- And if the project compiles with assertions turned off, there are simulators that do not even evaluate the expression: `randomize()` simply **never gets called**
- With `` `uvm_fatal `` the error brings component, time and file, and it stops the run right there

Transactions

Summary of the unit

- We saw how to use *transactions* to move data around the TB
- That lets us simplify the components of the TB and even remove a class from our initial TB (add_tester)
- The unit that follows groups the classes that talk to one and the same interface into a single reusable block: the `uvm_agent`

Constrained random

The other half of the pincer

- The functional coverage of the conventional testbench tells you **what is missing**. It does not tell you how to get there: that is the stimulus
- Writing a directed test per bin does not scale — there are 76 of them in the VTALU, and a real chip has thousands
- The industry recipe is the other way round: **random for the bulk, directed for the holes**. And the random has to be *legal*, or the DUT complains about things that are never going to happen in silicon
- *Constrained random* is writing the rules of the stimulus **once, in the transaction**, and letting the solver build the cases

Constrained random

`rand` and `randomize()`: what you have already seen

```
class command_transaction extends uvm_sequence_item;
   rand byte unsigned A;      // rand  -> it enters the draw
   rand byte unsigned B;
   rand operation_t   op;     // an enum randomizes over its values

   constraint data { ... }    // the rules, in a block of their own
endclass
```

- `randomize()` comes from SystemVerilog, not from UVM: **every** class has it
- It picks values for the `rand` fields that **satisfy every active constraint at once**
- A `constraint` **is not sequential code**: it does not run from top to bottom, it has no order. It is a set of relations the solver satisfies together
- That is why two constraints that contradict each other do not give a compilation error: they give a `randomize()` that returns 0

Constrained random

The other three words: randc and the two hooks

```
class command_transaction extends uvm_sequence_item;
   rand  byte unsigned A;       // it can repeat: 3 3 6 1 3 4 6 2 ...
   randc bit [2:0]     indice;  // it walks the 8 and only then repeats

   function void pre_randomize();   // runs BEFORE the draw
      if (!primera) A = A_anterior; // …with the values still old
   endfunction

   function void post_randomize();  // runs AFTER, with the new values
      checksum = A ^ B;             // what is not drawn gets computed here
   endfunction
endclass
```

- `randc` is **cyclic**: it exhausts every value before repeating one. Measured on Verilator 5.052: `4 2 1 7 5 6 3 0` and only then does it start over
- It is for walking a small space —an opcode, a register index— without waiting for chance to cover it. Only on **small** integral types: the LRM guarantees 8 bits and simulators usually stop at 16
- `pre_randomize()` / `post_randomize()` are the two hooks `randomize()` calls on its own. The **derived** field —a checksum, a parity— gets computed in the second one, it does not get drawn
- And a warning: whatever gets computed in `post_randomize()` does **not** take part in the solver. If it has to satisfy a constraint, it is a `rand` field

Constrained random

`dist :=` is not the same as `:/`

```
A dist {8'h00 := 1, [8'h01 : 8'hFE] := 1, 8'hFF := 1};   // the weight goes to EACH value
A dist {8'h00 :/ 1, [8'h01 : 8'hFE] :/ 2, 8'hFF :/ 1};   // the weight gets SPLIT
```

- `dist` sets **weights**: it does not change which values are legal, it changes how often they come up
- With `:=` the weight is applied **to each value of the range**: the middle range weighs 254 and the edges 1 each. `A=00` comes up **1 time in 256**
- With `:/` the weight belongs to **the whole range**: 1 – 2 – 1 gives 25 % on `00`, 50 % in the middle, 25 % on `FF`. Which is exactly the bias the `get_data()` of the conventional testbench did by hand
- Both lines compile, both run, and **one of the two never fills the edge bins**

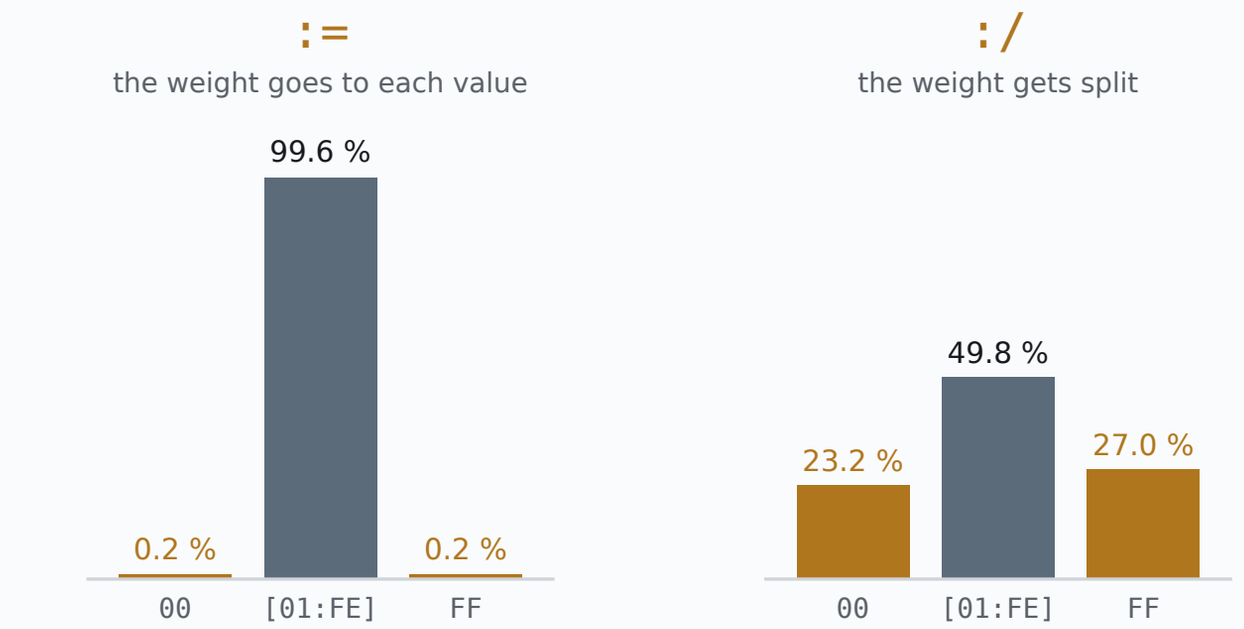
Constrained random

Do not assume it: measure it

```
// The weight goes to EVERY value of the range: the middle takes 254 of 256.
class por_valor;
   rand byte unsigned A;
   constraint data {A dist {8'h00 := 1, [8'h01 : 8'hFE] := 1, 8'hFF := 1};}
endclass

// The weight is SPLIT inside the range: 1/4 at 00, 1/2 in the middle, 1/4 at FF.
// It is the same bias get_data() of the Functional coverage section did by hand.
class por_rango;
   rand byte unsigned A;
   constraint data {A dist {8'h00 :/ 1, [8'h01 : 8'hFE] :/ 2, 8'hFF :/ 1};}
endclass
```

```
$ cd code/u6/transactions/constraints && bash run.sh
400 randomizations of each version
  :=   A=00   0.2%   A=FF   0.2%   the weight goes to each value
  :/   A=00  23.2%   A=FF  27.0%   the weight gets split
```



400 randomizations of each version, measured. Both compile, both run, neither one warns. The edge bins are the same, and with `:=` they take 64 times longer to fill.

Constrained random

inside and randomize() with {}

```
class comando;
   rand byte unsigned A;
   rand byte unsigned B;
   rand operation_t   op;

   constraint data {
      A dist {8'h00 :/ 1, [8'h01 : 8'hFE] :/ 2, 8'hFF :/ 1};
      B dist {8'h00 :/ 1, [8'h01 : 8'hFE] :/ 2, 8'hFF :/ 1};
   }

   // inside is a set of legal values. Without it the random also asks for
   // no_op and rst_op, which compute nothing.
   constraint utiles {op inside {add_op, and_op, xor_op, mul_op};}
endclass
```

```
   // The portable workaround meanwhile: turn off the split, which for a
   // directed case makes no sense anyway.
   c.data.constraint_mode(0);
   if (!c.randomize() with {
         op == mul_op;
         A  == 8'hFF;
         B  == 8'hFF;
       })
      $fatal(1, "randomize() with failed even with the constraint turned off");
   $display("with with: 1 try,      A=%2h %s B=%2h", c.A, c.op.name(), c.B);
```

- `inside {a, b, c}` is the set of legal values: without it, the random also asks for `no_op` and `rst_op`, which compute nothing. Here it also leaves out `sub_op`, so that the draw comes out uniform among four
- `randomize() with { ... }` adds constraints **for that call only**: the directed case gets asked for at the point of use, without touching the class and without writing a new tester. It is the tool of *coverage closure*

Constrained random

`soft`: the default a `with {}` can override

```
class command_transaction extends uvm_sequence_item;
   rand byte unsigned A;
   constraint por_defecto { soft A == 8'h00; }   // "zero, unless somebody asks otherwise"
endclass

t.randomize();                          // A = 00
t.randomize() with { A inside {[1:10]}; };   // A between 1 and 10: the soft drops on its own
```

- A `soft` constraint is a **wish**, not a rule: if it clashes with another, the solver **drops** it instead of returning 0
- It is the modern way of writing a default value in the transaction without forcing anybody to call `constraint_mode(0)` to override it
- The **hard** constraint wins, and between two clashing `soft` ones the one declared **later** wins — the order does matter here, and only here
- Measured on Verilator 5.052: with the `with {}` it comes out between 1 and 10; without it, `00`

Constrained random

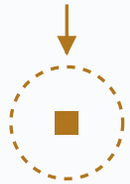
The order of resolution biases without warning

```
rand bit           es_reset;
rand byte unsigned A;
constraint c {es_reset -> A == 8'h00;}   // "if I ask for a reset, the operands at zero"
```

- It looks harmless. **It is not**: the solver picks among the *solutions*, not among the values of each field
- With `es_reset = 1` there is **one**; with `es_reset = 0`, **256**. You asked for a reset half the time and you will see it **1 time in 257**
- The *"any operation after a reset"* bin of the plan does not get filled, and the report does not tell you why
- The answer from the language is `solve es_reset before A`: pick the control field first

One cell, one solution. The draw is uniform over all of them:

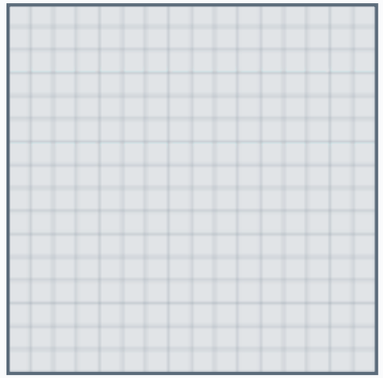
`es_reset = 1`



1 solution

`A == 8'h00`

`es_reset = 0`



256 solutions

A is left free

The solver picks among the 257 solutions, not between the two values of `es_reset`.
You asked for reset half the time and you will see it 1 in 257. Measured: 0.3 %.

Constrained random

Measured, with the workaround that Verilator does honour

```
class sesgada;
   rand bit           es_reset;
   rand byte unsigned A;
   constraint c {es_reset -> A == 8'h00;}
endclass

// The language's answer is "solve es_reset before A": pick the control field
// FIRST and then solve the rest. Verilator 5.052 accepts the syntax and does
// NOT honour it -- it leaves es_reset stuck at 0, which is worse than
// ignoring it. Repro in code/verilator/repro-solve-before.sv.
//
// The workaround that IS portable: ask explicitly for the split of the
// control field. It says the same and does not rely on the solver ordering well.
class pareja extends sesgada;
   constraint reparto {es_reset dist {1'b0 :/ 1, 1'b1 :/ 1};}
endclass
```

```
2000 randomizations of each version
  as written             es_reset=1 in   0.3%   (1 in 257)
  with dist on es_reset  es_reset=1 in  51.7%
```

- Verilator 5.052 **accepts `solve ... before` and does not honour it**: it leaves the field pinned at 0, which is worse than ignoring it. Repro in `code/verilator/repro-solve-before.sv`
- The portable workaround: ask for the spread of the control field with a `dist`. It says the same thing and it does not depend on the solver getting the order right

Constrained random

When there is no solution

```
class comando;
   rand byte unsigned A;
   constraint chico {A < 8'h10;}
   constraint grande {A > 8'hF0;}  // contradicts the previous one: no solution
endclass

initial begin
   comando c;
   c = new();

   // 1. Both constraints active: there is no A that satisfies the two.
   if (!c.randomize()) $display("1. randomize() returned 0: the constraints do not close");
   else $display("1. randomize() gave %2h  (should not get here)", c.A);

   // 2. constraint_mode(0) turns a constraint off at run time.
   c.grande.constraint_mode(0);
   if (!c.randomize()) $display("2. randomize() returned 0  (should not get here)");
   else $display("2. with 'grande' turned off: A=%2h, and it honours 'chico'", c.A);

   // 3. rand_mode(0) takes the field out of the draw: it keeps its current value.
   c.A.rand_mode(0);
   if (!c.randomize()) $display("3. randomize() returned 0  (should not get here)");
   else $display("3. with A out of the draw: A=%2h, the same as before", c.A);

   $finish;
```

```
1. randomize() returned 0: the constraints do not close
2. with 'grande' turned off: A=06, and it honours 'chico'
3. with A out of the draw: A=06, the same as before
```

- `constraint_mode(0)` turns **one constraint** off at run time; `rand_mode(0)` takes **one field** out of the draw and leaves it with the value it had — and if that value does not satisfy the constraints that name it, `randomize()` returns 0
- They are for the test that needs to break a rule on purpose — injecting an illegal opcode, for instance — without touching the class everybody else uses

Constrained random

How a constraint that does not close gets debugged

```
// 1. Make the testbench say so. Without this there is no bug: there is silence
if (!t.randomize()) `uvm_fatal("RAND", "the constraint did not close")

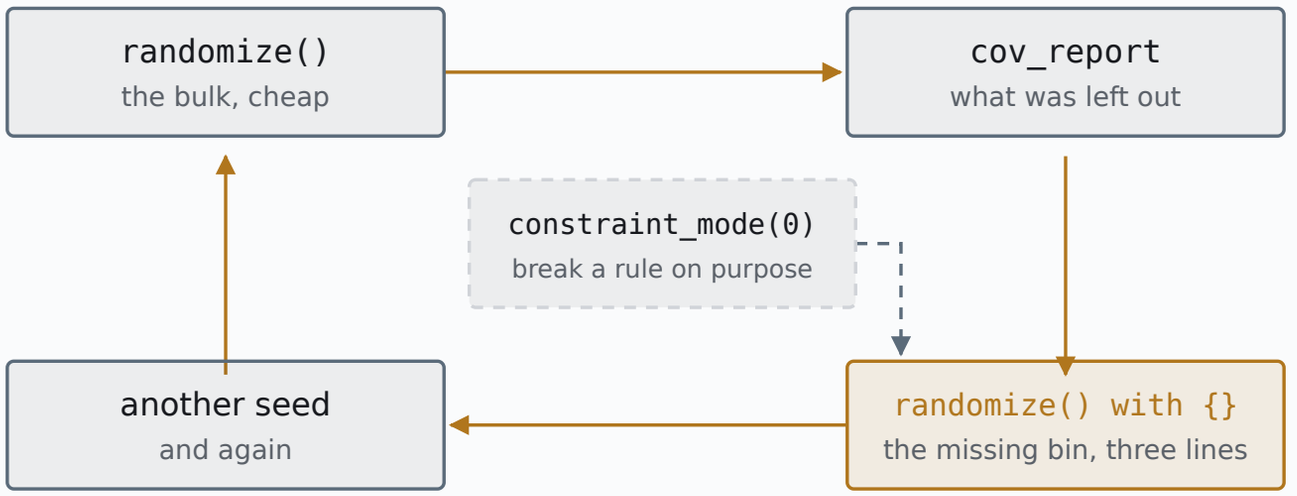
// 2. Bisection: turn them off one at a time until it closes. The last one you turned off is it
foreach (nombres[i]) begin
   t.chico.constraint_mode(0);
   if (t.randomize()) `uvm_info("RAND", "it closes without 'chico'", UVM_LOW)
   t.chico.constraint_mode(1);
end

// 3. Take the field out of the draw and see which value was blocking it
t.A.rand_mode(0);
```

- The symptom is never *"it does not close"*: it is a `randomize()` that returns 0 and a transaction with **the previous one's values**. First it has to be made noisy
- The method is **bisection**, just like with `git bisect`: `constraint_mode(0)` one at a time until it closes. The one you had not turned off yet is the culprit
- The other cut is by field: `rand_mode(0)` leaves the value it had and shows whether the conflict was that one
- The cause is almost always the same: a constraint of the **base** class that the test's `with {}` contradicts. There the design answer is `soft`

Constrained random

This is how coverage gets closed



Closing coverage is this cycle, and it is what a verification engineer spends the day on. What it is not: writing one test per bin, or looking at the total percentage.

- No step asks for a new class: `dist` and `inside` in the transaction, three lines of `with {}`, and `rand_mode(0)` or `constraint_mode(0)` to break a rule
- That is *coverage closure*, and it is what a verification engineer spends the day on
- What it is **not**: writing one test per bin. Nor looking at the total percentage
- The stimulus still comes out of a `tester` you wrote yourself. In modern UVM that is a `uvm_sequence` — and that is day 6

Constrained random

"Another seed, and again": what it moves and what it does not

```
$ SEED=7 bash run.sh    # u6/transactions/constraints, 400 randomizations
    seed: 7
  :/   A=00  25.0%   A=FF  22.0%   # with SEED=8: 20.8% and 27.0%
$ SEED=7 bash run.sh    # u2/convencional, and again with SEED=8
  covergroup : 86.8% (66/76)       # both times. And the merge, the same
```

- `SEED=N` passes `+verilator+seed+N`, and `run_sim` **prints the one it used**: a failure that comes up once every ten runs does not get debugged, it gets repeated
- It **does** move the percentages of a `dist` measured with few samples: 400 randomizations give 25 % or 21 % depending on the draw. It is a sample, not the distribution
- It does **not** move the 10 bins that are missing. With 1000 operations the random already got as far as it can, and another seed is tossing the coin expecting a different result
- That is why *"another seed"* is the **last** step of the previous cycle and not the first: the `with {}` comes first

Constrained random

Summary of the unit

- The coverage tells you **what is missing**; constrained random is **how to get there**. They are the two halves of the same pincer
- Writing a directed test per bin does not scale: there are **76 bins** in the VTALU. The recipe is the other way round — **random for the bulk, directed for what is left**
- A `constraint` **is not sequential code**: it does not run from top to bottom. It is a system of equations the solver resolves all together
- That is why two constraints that contradict each other **do not give a compilation error**: they give a `randomize()` that returns 0
- `dist` sets **weights**, not legality. And `:=` is not `:/`: the first weighs **each value** of the range, the second **the whole range**
- `randomize()` returns 0 and carries on. That is why it **always** goes inside an `if` with `` `uvm_fatal `` — it is the most expensive silent trap of the course

Review · Day 5

1 of 8 · Deep copy

`obj1_h = obj2_h`. What did I copy?

- **a** Nothing: both handles point at the same object ✓
- **b** Every field of `obj2_h` into `obj1_h`
- **c** Only the `rand` fields
- **d** A shallow copy: the first level is copied and the handles inside get shared

Nothing — there is no copy, there is a second name for the same object. If you modify it through one handle, the other one sees it. Out of that comes the *MOOCOW* rule: if you are going to modify, copy first.

Review · Day 5

2 of 8 · super.do_copy()

Why does every `do_copy()` in the hierarchy have to call `super.do_copy()`?

a Because UVM demands it in order to register the class

b So that the object ends up registered in the factory

c Because otherwise the fields of the classes above do not get copied ✓

d Because `do_copy()` is `pure virtual` and the base class has no implementation

Otherwise the fields above get lost — it is the same problem as `convert2string()` : the day somebody puts a new level in the middle, the method stops seeing half the data and nobody warns you.

Review · Day 5

3 of 8 · clone()

`clone()` returns a `uvm_object`. Why is it recommended to also write a `clone_me()`?

- **a** To wrap the `$cast` in a single place ✓
- **b** Because `clone()` does not copy the data
- **c** Because `clone()` is deprecated in IEEE 1800.2 and `copy()` replaces it
- **d** To be able to clone components as well as transactions

So as not to repeat the `$cast` — without it, every place that clones ends up writing its own cast. It is the same idea as always: if you are going to repeat it, wrap it.

Review · Day 5

4 of 8 · dist: := against :/

`A dist {8'h00 := 1, [8'h01:8'hFE] := 1, 8'hFF := 1};` — how often does `A = 8'h00` come out?

a A third of the time: they are three entries with the same weight

b Half: the edges share it between the two of them

c Never: `:=` only takes single values, not ranges, and the range is dropped

d Once every 256: with `:=` the weight goes to **each value** ✓

1 in 256 — `:=` gives weight 1 to *each one* of the 254 values in the middle, so the range weighs 254 against the 1 and 1 of the edges. To spread the weight *inside* the range you need `:/`. Both spellings compile and run: the difference only shows up in the coverage that does not go up.

Review · Day 5

5 of 8 · randomize() with {}

`cmd.randomize() with { A == 8'hFF; }` — what happens to the constraints the class already had?

a. They get replaced: for that call only what is inside the braces counts

b. They add up: both have to be satisfied ✓

c. They stay switched off until the next `randomize()` without a `with`

d. It depends on the declaration order: the one lower in the file wins

They add up — the `with {}` adds constraints **for that call only**, and erases nothing. That is why it is the tool for closing a bin without writing a new class: three lines at the point of use, and the rest of the stimulus stays what it always was. And that is also why it clashes with a `dist` that already biases the same field: both have to hold at once.

Review · Day 5

6 of 8 · constraint_mode()

`randomize() with { A == 8'hFF; }` on a field the class spreads with a `dist` returns 0 three times out of four. What is the workaround?

a Retry in a `do ... while` until it returns 1

b `constraint_mode(0)` on the `dist` constraint, for that object ✓

c Raise the weight of the `8'hFF` bin in the `dist` of the class

d `rand_mode(0)` on the field: it takes the `dist` out and lets the `with` rule

Switch off the constraint that gets in the way — Verilator solves the `dist` by **picking a value first** and only then checking the rest, so the probability of success is the probability of the bin: measured, `with {A == 8'hFF}` solves 25 % of the time. `constraint_mode(0)` switches it off for that object alone and touches nobody else. It is exactly the line the `d5c` exercise asks for.

Review · Day 5

7 of 8 · rand_mode()

What does `cmd.A.rand_mode(0)` do?

a It switches off every constraint that mentions that field

b It randomizes it once and freezes it afterwards

c It makes the solver resolve it last, after all the other fields

d It takes the field out of the draw and leaves it the value it had ✓

It stops being `rand` — they are the two run-time knobs and they get mixed up often: `rand_mode(0)` takes **a field** out of the draw, `constraint_mode(0)` switches off **a constraint**. One picks *what gets drawn*, the other *which rules hold*. It is what you use to pin one operand by hand and go on randomizing the rest.

Review · Day 5

8 of 8 · The order of resolution

`rand bit es_reset; rand byte unsigned A;` with `constraint c { es_reset -> A == 8'h00; }`. How often does `es_reset = 1` come out?

a Half the time: it is a `rand` bit and nothing forbids it

b Never: the implication forces it to 0

c 1 in 257: the solver draws among the **solutions** ✓

d It depends on the seed, and over enough runs it averages out to half

1 in 257 — the solver picks uniformly among the *solutions*, not among the values of each field: `es_reset=1` leaves a single combination (`A = 00`) and `es_reset=0` leaves 256. The *"any operation after a reset"* bin of the plan does not get filled, and the report does not say why. The answer in the language is `solve es_reset before A`; Verilator accepts it and **does not honour it**, so the portable workaround is to ask for the spread of the control field with a `dist`.

Exercise · Day 5 · 1 of 3

The scoreboard screams and the DUT is healthy

```
$ cd code/ejercicios/d5 && bash run.sh
```

- The TB of the transactions reports `FAIL` on almost every operation
- The DUT is the same one that has been passing since the spec. The bug belongs to the TB
- The messages that give it away are `UVM_HIGH`: turn the verbosity up
- If you get stuck, the **debug toolbox** appendix has the row for this symptom; the **silent traps** one has the cause

Exercise · Day 5 · 2 of 3

Measure your `dist`

```
$ cd code/ejercicios/d5b && bash run.sh
```

- One class, one `dist` and a histogram. No UVM, no DUT, no testbench
- The weights **already say 10, 80 and 10**. Run it first, and look at what comes out
- What is asked is that the three buckets give 10 / 80 / 10, with ±2 points of tolerance
- The difference between what is there and the solution is **one character**

Exercise · Day 5 · 3 of 3

Closing a bin

```
$ cd code/ejercicios/d5c && bash run.sh
```

- The test sends a reset and **60 random operations**, and the report shows the bin *"both legs at `FF`, multiplying"* empty
- Write the directed case that closes it. But **ask for it with `randomize() with {}`**, not by assigning the fields, which is what this section's `tester` does three lines above
- It is graded by comparing the coverage before and after: it has to go up
- It is **row 3** of the verification plan of day 1: the only one of the twelve whose stimulus column says *directed case*

Yesterday we left...

Where the testbench stands

- The data is an **object**: `command_transaction` with `do_copy`, `do_compare` and `convert2string`, and the scoreboard compares objects, not fields
- The stimulus is asked for instead of written: `randomize()`, `constraint`, `dist`, and the histogram that says whether the distribution does what it promises
- And **two** things are left stuck together that do not belong together: the `tester` is structure and stimulus at once, and the `env` names the monitor and the driver one by one

Agenda

Day 6 · ≈ 4 h 45 · unit 7 · the reusable testbench

- Agents
- Callbacks
- Sequences

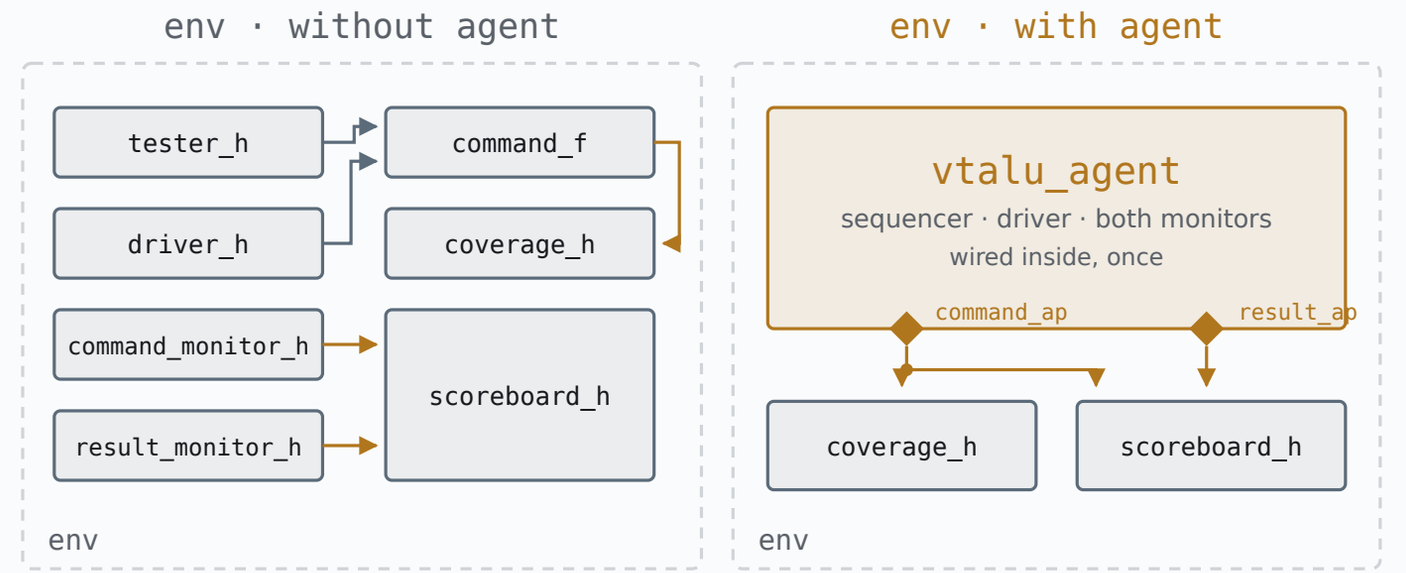
By the end of the day you can:

- **Say** what an agent with `is_active = UVM_PASSIVE` builds and what it does **not**
- **Explain** why a `uvm_sequence` does **not** show up in `print_topology()`
- **Remove** the last hard-wired line of a test, with `default_sequence` through `uvm_config_db`

Agents

The problem: a testbench that cannot be copied

- The `env` we left in the transactions works, but it is a **welded block**: seven components and five `connect()` written by hand
- Every one of those components exists for a single reason: **it drives or watches the same interface**, the `vtalu_bfm`
- Ask what happens if the design brings *two* VTALU: you have to copy the seven objects, the five connections, and rename everything
- Copying and pasting structure is exactly what we had been avoiding since the `env`
- UVM puts a name on the solution: **`uvm_agent`**



7 components and 5 connections, written by hand

Second VTALU: copy the whole block

1 agent and 2 subscribers

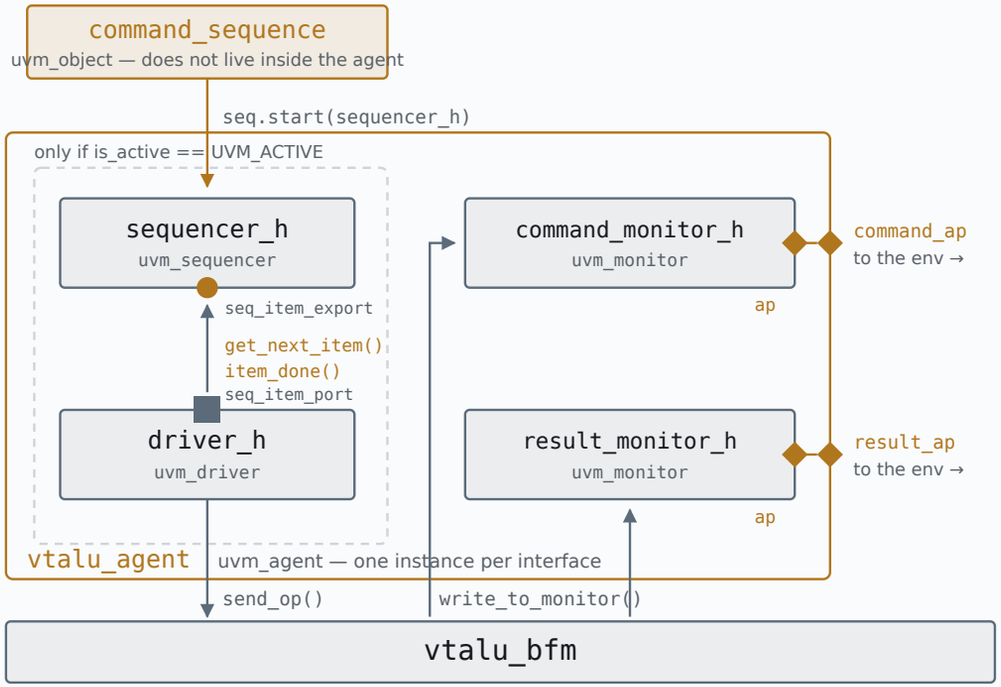
Second VTALU: one more line, with another config

The agent is not just another class: it is the unit instantiated once per interface.

Agents

What is inside an agent

- A `uvm_agent` is one more `uvm_component`: same tree, same phases, same factory
- What changes is **what it puts inside**, always the same:
  - a *sequencer*, which hands out stimulus
  - a *driver*, which turns it into signals of the BFM
  - the *monitors*, which watch the bus and publish transactions
- And it exposes outwards **two** `uvm_analysis_port`: the one that publishes commands and the one that publishes results
- The scoreboard and the coverage **do not go inside**: they are analysis, and they live in the `env` hanging off those two ports
- From this section on every piece uses **its** base class: `uvm_driver #(T)`, `uvm_sequencer #(T)` and `uvm_monitor` instead of a plain `uvm_component`



The scoreboard and the coverage live in the env, hanging off those two diamonds.

Agents

The jump: from the tester to the sequence

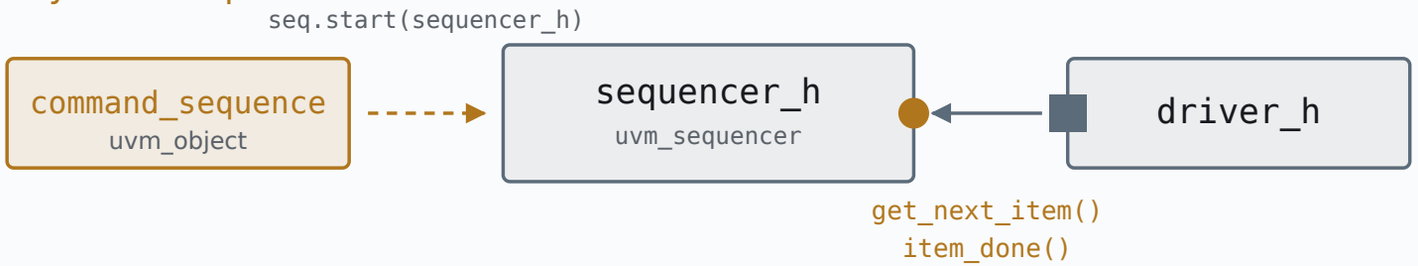
- In the put and get ports we split the stimulus in two: the `tester` chose **what** to send, the `driver` knew **how**
- We joined them with `uvm_put_port` / `uvm_get_port` and a `uvm_tlm_fifo` in the middle: three components in the tree and two `connect()`
- UVM already brings that done, and with a name: the **`uvm_sequencer`**
- The `driver` stops being a generic `uvm_component` and becomes `uvm_driver #(T)`, which **already comes with a `seq_item_port`**
- And the `tester` disappears: a `uvm_sequence` replaces it, and that **is not a component** — it is an object, it is not in the tree and it does not get connected

day 4 · put / get



three components in the tree, two connect(), and no warning back

day 6 · sequencer



The sequencer replaces the FIFO and adds the `item_done()`: the driver says when it finished.

The sequence is not a component: not in the tree, not connected — it gets started.

Agents

The handshake: `get_next_item()` / `item_done()`

- The driver no longer does `get()`: it does `get_next_item()`, which **blocks** until the sequence has something
- When it has finished driving the signals, it calls `item_done()`
- That second call is what the `put`/`get` pair **did not have**: it is the driver saying *"that is it, send me the next one"*
- Without `item_done()` the sequence hangs forever in its `finish_item()`, and the simulation stops advancing without a single error message
- The pair always goes complete, and in that order

```
task run_phase(uvm_phase phase);
   command_transaction command;
   forever begin : command_loop
      seq_item_port.get_next_item(command);
      bfm.send_op(command.A, command.B, command.op);
      seq_item_port.item_done();
   end : command_loop
endtask : run_phase
```

Agents

The driver, now a `uvm_driver`

- `uvm_driver #(command_transaction)` brings the `seq_item_port` for free: it is not declared nor instantiated with `new()`
- The `build_phase` no longer looks for the BFM straight in the `config_db`: it asks the **agent config** for it
- The `run_phase` is **three lines**: ask for the item, `bfm.send_op(...)`, say it is done. The driver does not touch a wire
- The protocol stays where we left it in interfaces and BFM: **inside the BFM**. That is why the driver of an agent is so short, and why changing protocol does not touch it

```
class driver extends uvm_driver #(command_transaction);
   `uvm_component_utils(driver)

   virtual vtalu_bfm bfm;

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

   function void build_phase(uvm_phase phase);
      vtalu_agent_config cfg;
```

```
      // The BFM is not asked for on its own: ask for the agent config, and it comes from there.
      if (!uvm_config_db#(vtalu_agent_config)::get(this, "", "config", cfg))
         `uvm_fatal("DRIVER", "Failed to get agent config")
      bfm = cfg.bfm;
   endfunction : build_phase

   task run_phase(uvm_phase phase);
      command_transaction command;
      forever begin : command_loop
         seq_item_port.get_next_item(command);
         bfm.send_op(command.A, command.B, command.op);
         seq_item_port.item_done();
      end : command_loop
   endtask : run_phase

endclass : driver
```

Agents

The sequencer gets declared with a `typedef`

- `uvm_sequencer` is a parameterized class: it is parameterized with the type of item it is going to hand out
- There is no need to extend it for anything: it is used as is
- A `typedef` in the package gives it a short name, and that name is the one used by the agent and the `create()`

```
typedef uvm_sequencer #(command_transaction) sequencer;
```

- The `command_transaction` of the transactions already extends `uvm_sequence_item`, so **it does not have to be touched**: that is why we chose that base class and not `uvm_transaction`
- The `typedef` goes **after** the `` `include `` of the transaction and **before** the driver and the agent

Agents

The minimal sequence

- A `uvm_sequence #(T)` has a task `body()`: that is all there is to write
- Every item goes between `start_item()` and `finish_item()`
- `start_item()` blocks until the sequencer gives us the turn; `finish_item()` blocks until the driver called `item_done()`
- The `randomize()` goes **between the two**: that way the constraints get resolved at the moment the item is about to be handed over, not before
- This is the direct translation of the `tester` of the transactions, and nothing more than that — the sequences section is about what can be done in here

```
class command_sequence extends uvm_sequence #(command_transaction);
   `uvm_object_utils(command_sequence)

   function new(string name = "command_sequence");
      super.new(name);
   endfunction : new

   task body();
      command_transaction command;

      // All transactions come out of the factory, the directed ones too:
      // a new() would skip add_test's set_type_override.
      command = command_transaction::type_id::create("command");
      start_item(command);
```

```
    command.op = rst_op;
    finish_item(command);

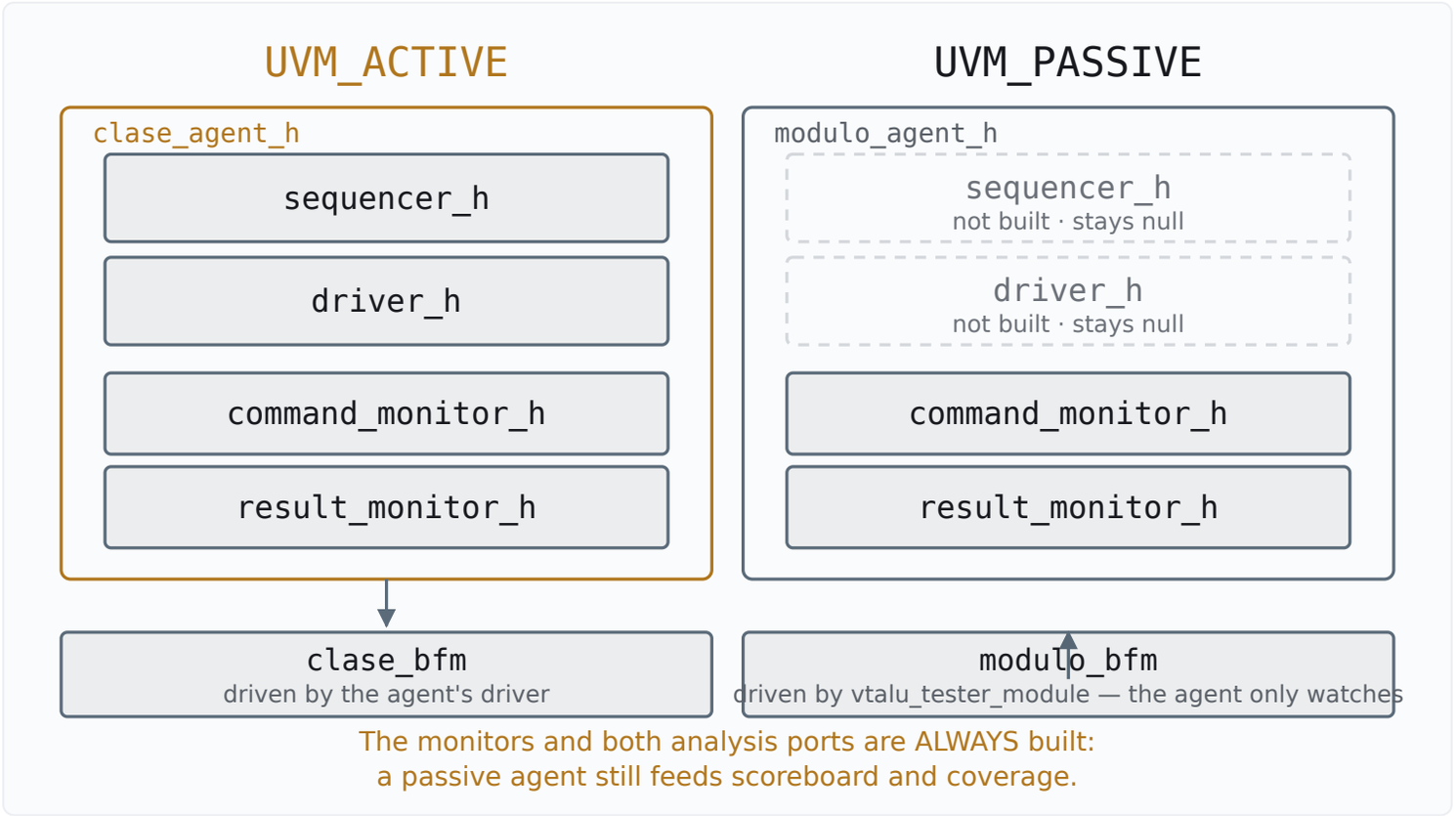
    repeat (1000) begin : random_loop
       command = command_transaction::type_id::create("command");
       start_item(command);
       // The randomize() goes BETWEEN start_item and finish_item: that way the
       // constraints are solved when the item is about to be handed over, and
       // not before. Today it makes no difference; the day a constraint
       // depends on the DUT state, it will.
       if (!command.randomize()) `uvm_fatal("COMMAND SEQ", "randomize() failed")
       finish_item(command);
    end : random_loop

    // Directed: the multiplier overflow, which 1000 random operations might
    // never touch. It comes out of the factory like the others, but since it
    // is not randomized, add_transaction's constraint does not run.
    command = command_transaction::type_id::create("command");
    start_item(command);
    command.op = mul_op;
    command.A  = 8'hFF;
    command.B  = 8'hFF;
    finish_item(command);
  endtask : body
```

Agents

`is_active`: active and passive

- `uvm_agent` brings a field `is_active` of type `uvm_active_passive_enum`, with two values: `UVM_ACTIVE` and `UVM_PASSIVE`
- **Active**: the sequencer and the driver get built. The agent drives the interface
- **Passive**: they do not get built. The agent **only watches** an interface that somebody else drives
- The monitors and the two analysis ports get built **always**: a passive agent keeps feeding scoreboard and coverage
- It is read with `get_is_active()` and decided in the `build_phase`, with an `if`


UVM_ACTIVE
UVM_PASSIVE
clase_agent_h
sequencer_h
driver_h
command_monitor_h
result_monitor_h
modulo_agent_h
sequencer_h
not built · stays null
driver_h
not built · stays null
command_monitor_h
result_monitor_h
clase_bfm
driven by the agent's driver
modulo_bfm
driven by vtalu_tester_module — the agent only watches
The monitors and both analysis ports are ALWAYS built:
a passive agent still feeds scoreboard and coverage.

Agents

And it can be seen

- `bash run.sh +TOPOLOGY` prints the tree of components UVM built

```
clase_agent_h          vtalu_agent
  command_ap           uvm_analysis_port
  command_monitor_h    command_monitor
  driver_h             driver
    seq_item_port      uvm_seq_item_pull_port
  result_ap            uvm_analysis_port
  result_monitor_h     result_monitor
  sequencer_h          uvm_sequencer
    seq_item_export    uvm_seq_item_pull_imp
modulo_agent_h         vtalu_agent
  command_ap           uvm_analysis_port
  command_monitor_h    command_monitor
  result_ap            uvm_analysis_port
  result_monitor_h     result_monitor
```

- Same class, two instances, and one has four components fewer
- There too are the `seq_item_port` and the `seq_item_export` that the driver and the sequencer bring already fitted, without anybody declaring them — against the put and get ports, where the `uvm_put_port` had to be declared and `new()`ed
- `print_topology()` is besides the debug tool for when a `connect()` points at a component that does not exist

Agents

The config object of the agent

- The agent needs two pieces of data from outside: **which BFM** and **whether it is active**
- Both travel together in a configuration object, which is the public interface of the agent
- It is a **bare class**, not a `uvm_object`: that is why the constructor can **demand** both pieces of data, and whoever forgets one does not compile
- A `uvm_object` is created with `type_id::create()`, which only takes a `name`: the guarantee of the constructor is lost
- The field `is_active` goes `protected` with a getter: nobody changes it after the agent has been built

```
// The agent config is NOT a uvm_object: it is a plain class. That is why the
// constructor can DEMAND both values, and whoever forgets one does not compile.
// A uvm_object is built with type_id::create(), which only takes a name, and
// that guarantee is lost.
class vtalu_agent_config;

   virtual vtalu_bfm bfm;

   // protected + getter: nobody changes it after the agent was built.
```

```
   protected uvm_active_passive_enum is_active;

   function new(virtual vtalu_bfm bfm, uvm_active_passive_enum is_active);
      this.bfm = bfm;
      this.is_active = is_active;
   endfunction : new

   function uvm_active_passive_enum get_is_active();
      return is_active;
   endfunction : get_is_active

endclass : vtalu_agent_config
```

Agents

The agent class: `build_phase`

- It asks the `uvm_config_db` for its config and copies the `is_active` **by hand**: without `super.build_phase()`, `uvm_agent`'s, the one that would read it, never runs
- It builds sequencer and driver **only if it is active**; the monitors and the two analysis ports, always
- The analysis ports are ports: they get instantiated with `new()`, not with the factory

```
class vtalu_agent extends uvm_agent;
   `uvm_component_utils(vtalu_agent)

   vtalu_agent_config cfg;

   sequencer       sequencer_h;
   driver          driver_h;
   command_monitor command_monitor_h;
   result_monitor  result_monitor_h;

   uvm_analysis_port #(command_transaction) command_ap;
   uvm_analysis_port #(result_transaction)  result_ap;

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

```
function void build_phase(uvm_phase phase);
   if (!uvm_config_db#(vtalu_agent_config)::get(this, "", "config", cfg))
      `uvm_fatal("AGENT", "Failed to get agent config")
   is_active = cfg.get_is_active();

   if (get_is_active() == UVM_ACTIVE) begin : estimulo
      sequencer_h = sequencer::type_id::create("sequencer_h", this);
      driver_h    = driver::type_id::create("driver_h", this);
   end : estimulo

   command_monitor_h = command_monitor::type_id::create("command_monitor_h", this);
   result_monitor_h  = result_monitor::type_id::create("result_monitor_h", this);

   command_ap = new("command_ap", this);
   result_ap  = new("result_ap", this);
endfunction : build_phase
```

Agents

The agent class: `connect_phase`

- Inside: the `seq_item_port` of the driver against the `seq_item_export` of the sequencer — **a single line**, and only if it is active
- Outwards: the `ap` of each monitor gets connected to the analysis port **of the agent**
- That second `connect()` is the one that makes the agent a closed box: the `env` never touches `command_monitor_h`, it talks to `command_ap`
- The mantra *ports connect to exports* still holds, with one more twist: a port also connects **to another port of the same type, outwards**. That is `command_monitor_h.ap.connect(command_ap)`, and it is called *port forwarding*

Agents

How the env ends up

- Four lines of structure instead of seven objects and five connections
- The `env` no longer knows there is a driver, or a sequencer, or a FIFO: it knows there is an agent and that it has two analysis ports
- The scoreboard and the coverage stay just as in the analysis ports: they are subscribers, and they find out about nothing

```
class env extends uvm_env;
   `uvm_component_utils(env)

   vtalu_agent        agent_h;
   vtalu_agent_config agent_cfg_h;

   scoreboard scoreboard_h;
   coverage   coverage_h;

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

   function void build_phase(uvm_phase phase);
      virtual vtalu_bfm bfm;

      if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "bfm", bfm))
         `uvm_fatal("ENV", "Failed to get BFM")

      agent_cfg_h = new(.bfm(bfm), .is_active(UVM_ACTIVE));
      uvm_config_db#(vtalu_agent_config)::set(this, "agent_h*", "config", agent_cfg_h);
```

```
   agent_h      = vtalu_agent::type_id::create("agent_h", this);
   scoreboard_h = scoreboard::type_id::create("scoreboard_h", this);
   coverage_h   = coverage::type_id::create("coverage_h", this);
endfunction : build_phase

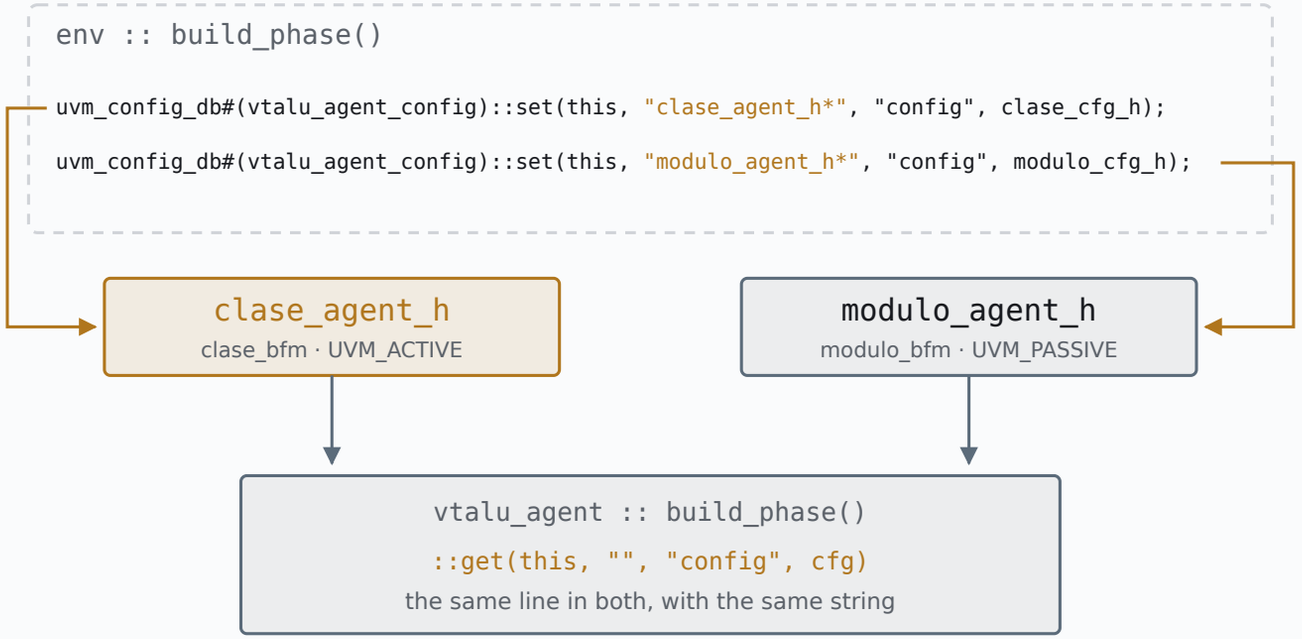
function void connect_phase(uvm_phase phase);
   agent_h.command_ap.connect(scoreboard_h.cmd_f.analysis_export);
   agent_h.command_ap.connect(coverage_h.analysis_export);
   agent_h.result_ap.connect(scoreboard_h.analysis_export);
endfunction : connect_phase
```

- That is the `env` of a single VTALU. With two, it changes less than it looks

Agents

Two agents, one single string

- The two agents execute **the same line**: `uvm_config_db#(vtalu_agent_config)::get(this, "", "config", cfg)`
- And yet they have to receive **different objects**
- The second argument of the `set()` resolves it: the **scope**, which is the path of the component that is going to read
- `"clase_agent_h*"` reaches that agent and everything hanging off it — that is why the driver and the monitors find the same config without anybody passing it to them
- The `this` of the `set()` is the starting point of that path: the `env`



The scope of the `set()` is the path of the component that will read, not a free name.

With `"*"` in both, the second overwrites the first and both agents start passive.

Agents

When the `config_db` does not find: `+UVM_CONFIG_DB_TRACE`

```
$ bash run.sh +UVM_CONFIG_DB_TRACE
UVM_INFO ... Configuration 'config' (type vtalu_agent_config) set by env_h
UVM_INFO ... Configuration 'config' read by
             accessor=uvm_test_top.env_h.clase_agent_h.driver_h
UVM_INFO ... Configuration 'config' read by
             accessor=uvm_test_top.env_h.modulo_agent_h.command_monitor_h
```

- The `config_db` is a database by strings: when it fails, **it fails in silence** or with a `uvm_fatal` that does not say why
- `+UVM_CONFIG_DB_TRACE` makes UVM print **every `set()` and every `get()`**, with the complete hierarchical path of the component that asked for it
- It works for the two questions one asks: *did the `set()` reach this component?* and *which components actually exist?*
- It is a plusarg, not code: nothing has to be touched or recompiled

Agents

The example: two VTALU and the legacy module

- The case that justifies all of this: two **sources of stimulus** have to be compared on the same design
- The `top` instantiates two VTALU and two `vtalu_bfm`
- The first one is driven by our agent, in `UVM_ACTIVE`
- The second one is driven by `vtalu_tester_module`, an ordinary module without one line of UVM, and we hang an agent on it in `UVM_PASSIVE` to watch it
- Two scoreboards and **two coverages**: that is what answers which of the two stimuli covers more

```
module top;
   import uvm_pkg::*;
   import vtalu_pkg::*;
   `include "vtalu_macros.svh"
   `include "uvm_macros.svh"

   // The VTALU driven by the class testbench
   vtalu_bfm clase_bfm ();
   vtalu clase_dut (
       .A(clase_bfm.A), .B(clase_bfm.B), .op(clase_bfm.op_set),
       .clk(clase_bfm.clk), .reset_n(clase_bfm.reset_n), .start(clase_bfm.start),
       .done(clase_bfm.done),
```

```
        .ovf(clase_bfm.ovf), .result(clase_bfm.result)
    );

    // The VTALU driven by the legacy module
    vtalu_bfm modulo_bfm ();
    vtalu modulo_dut (
        .A(modulo_bfm.A), .B(modulo_bfm.B), .op(modulo_bfm.op_set),
        .clk(modulo_bfm.clk), .reset_n(modulo_bfm.reset_n), .start(modulo_bfm.start),
        .done(modulo_bfm.done),
        .ovf(modulo_bfm.ovf), .result(modulo_bfm.result)
    );

    vtalu_tester_module stim_module (modulo_bfm);

    initial begin
       // The top knows nothing about agents: it leaves the two interfaces with a
       // name and walks away. Handing them out is the test's job.
       uvm_config_db#(virtual vtalu_bfm)::set(null, "*", "clase_bfm", clase_bfm);
       uvm_config_db#(virtual vtalu_bfm)::set(null, "*", "modulo_bfm", modulo_bfm);
       run_test();
    end

endmodule : top
```

- The module calls **the very same** `bfm.send_op()` as the driver: it receives the interface through its **port** instead of through a virtual interface, but the protocol it executes is exactly the same code
- That is what makes the comparison fair: both stimuli speak identically to the DUT. The only thing being compared is **what** each one sends, not how

Agents

The test, and what is still missing

- The `test` takes the two BFM out of the `config_db`, builds the `env_config` and passes it down
- Then it starts the sequence **by hand**, on the sequencer of the active agent:

```
seq = command_sequence::type_id::create("seq");
seq.start(env_h.clase_agent_h.sequencer_h);
```

- It works, but look at what that line does: the test **goes across the hierarchy** to reach the sequencer, and with that it knows the inside of the env again
- Everything we have just encapsulated leaks out in one line
- The sequences section fixes it: the sequence is handed to the sequencer **through the `config_db`**, and the test stops knowing where it is

```
function void build_phase(uvm_phase phase);
   virtual vtalu_bfm clase_bfm, modulo_bfm;
   env_config env_config_h;

   // The top left the two interfaces loose; the test packs them up.
   if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "clase_bfm", clase_bfm))
      `uvm_fatal("DUAL TEST", "Failed to get clase_bfm")
   if (!uvm_config_db#(virtual vtalu_bfm)::get(this, "", "modulo_bfm", modulo_bfm))
      `uvm_fatal("DUAL TEST", "Failed to get modulo_bfm")
```

```
    env_config_h = new(.clase_bfm(clase_bfm), .modulo_bfm(modulo_bfm));
    uvm_config_db#(env_config)::set(this, "env_h*", "config", env_config_h);

    env_h = env::type_id::create("env_h", this);
  endfunction : build_phase

  // +TOPOLOGY prints the component tree. It is how to SEE that the passive
  // agent built neither a sequencer nor a driver.
  function void end_of_elaboration_phase(uvm_phase phase);
    if ($test$plusargs("TOPOLOGY")) uvm_root::get().print_topology();
  endfunction : end_of_elaboration_phase

  task run_phase(uvm_phase phase);
    command_sequence seq;

    phase.raise_objection(this);

    // The only thing left wired by hand: the test reaches across the
    // hierarchy to get to the sequencer. The sequences section takes it out of here.
    seq = command_sequence::type_id::create("seq");
    seq.start(env_h.clase_agent_h.sequencer_h);

    #500;
    phase.drop_objection(this);
  endtask : run_phase
```

Agents

What this section does differently

- **The book does not use a sequencer here.** In *The UVM Primer* the agent of this section still has the `tester` and the `uvm_tlm_fifo` inside, and the sequencer only shows up with sequences — where besides the `tester` disappears
- We build the agent **complete in one go**: it is the way it is written today, and it makes the sequences section be only about sequences
- **The scoreboard and the coverage stay outside the agent**, against what the *UVM Primer* does. A passive agent should not drag a scoreboard along
- **The config of the agent is not a `uvm_object`**: it is a bare class, so that the constructor forces you. The industry usually makes it a `uvm_object`
- The protocol lives in the BFM, the same as in the *UVM Primer*. Up to Verilator 5.051 this could not be done: an interface task called through a virtual interface did not propagate. It was fixed in 5.052 — `docs/en/verilator.md`

Agents

Summary of the unit

- An **agent** packs everything that knows how to speak one interface: sequencer, driver and monitors, wired inside a single time
- It gets instantiated **once per interface**, and it is configured with a **configuration object**, not with loose handles
- `is_active` decides whether, besides watching, it drives
- The `env` went from seven components wired by hand to **one agent and two subscribers**
- The `uvm_config_db` stops being a global mailbox: the **scope** is a path in the tree, and with more than one agent it is what makes each one receive its own
- One single thing was left hard-wired: the test still looks for the sequencer by hand to start the sequence. **That is the sequences section**

Callbacks

The third hook

- On day 1 we promised **three** ways of writing a new test without touching the env: **constraints**, **factory override** and **callbacks**. Two are done
- The three are the same idea at different heights: the constraint changes **values**, the override changes **a whole class**, the callback changes **what a class does at one point**
- The case that separates them: you are handed a VIP agent —from another team, or bought— and you need the driver to send a broken piece of data **once in a while**
- With an override you have to rewrite the whole driver to change one line. With a callback you hook that line and nothing else
- And there is a difference that is not a matter of style: **callbacks stack**. The override picks *one* class; the callbacks all run, **every one of them**, in order

Callbacks

The hook: one class, one empty virtual method

```
class driver_callback extends uvm_callback;
   `uvm_object_utils(driver_callback)

   function new(string name = "driver_callback");
      super.new(name);
   endfunction : new

   // A task, not a function: a callback that only edits the transaction could
   // be a function, but one that adds delay needs to consume time. Making the
   // hook a task costs nothing and leaves both open.
   virtual task pre_send(driver drv, command_transaction cmd);
   endtask : pre_send

endclass : driver_callback
```

- A `uvm_callback` **is not a component**: it is not in the tree, it has no phases and it does not show up in `print_topology()` — just like a sequence
- The base method is **empty on purpose**: with nobody registered, the testbench runs exactly as before
- It is a `task` and not a `function` because a *delay* callback has to be able to consume time. One that only edits the transaction could be a function

Callbacks

Two lines in the driver, and not one more

```
class driver extends uvm_driver #(command_transaction);
   `uvm_component_utils(driver)
   // Declares "this component accepts callbacks of this type". Without it the
   // add() below still hooks up and the callback still runs, but with a CBUNREG
   // warning and without the type check: outside the contract, and nearly silent.
   `uvm_register_cb(driver, driver_callback)

   virtual vtalu_bfm bfm;
```

```
   task run_phase(uvm_phase phase);
      command_transaction command;
      forever begin : command_loop
         seq_item_port.get_next_item(command);
         // Every callback registered on this driver, in the order they were
         // added. With none registered this is a loop over an empty queue.
         `uvm_do_callbacks(driver, driver_callback, pre_send(this, command))
         bfm.send_op(command.A, command.B, command.op);
         seq_item_port.item_done();
      end : command_loop
   endtask : run_phase
```

- `` `uvm_register_cb `` declares the type/callback pair. Without it the `add()` hooks up all the same and the callback **runs** — with a `UVM_WARNING CBUNREG` lost in the log, and without the type check or `add_by_name`
- `` `uvm_do_callbacks `` is the extension point: it walks the queue of **that instance** of driver, in order

Callbacks

Putting them in: the test, and nothing but the test

```
// end_of_elaboration and not build: the driver is a grandchild of the env,
// and build_phase runs top-down -- when this test's build_phase returns, the
// driver does not exist yet.
function void end_of_elaboration_phase(uvm_phase phase);
   super.end_of_elaboration_phase(phase);

   flip_h   = flip_bit_cb::type_id::create("flip_h");
   jitter_h = jitter_cb::type_id::create("jitter_h");

   // Both on the same driver, and both run: the queue is per component
   // instance, in insertion order. The passive agent has no driver, so there
   // is nothing to hook there.
   uvm_callbacks #(driver, driver_callback)::add(env_h.clase_agent_h.driver_h, jitter_h);
   uvm_callbacks #(driver, driver_callback)::add(env_h.clase_agent_h.driver_h, flip_h);

   if ($test$plusargs("CALLBACK_TRACE"))
      uvm_callbacks #(driver, driver_callback)::display(env_h.clase_agent_h.driver_h);
endfunction : end_of_elaboration_phase
```

- The test **does not touch** the env, nor the agent, nor the driver, nor the sequence. Just like the override, one level further down
- `end_of_elaboration_phase` and not `build_phase`: the driver is a grandchild of the env, and when the `build_phase` of the test finishes **it does not exist yet**
- The two callbacks go on the **same** instance and **both** run, in the order in which they were added

`$ bash run.sh +CALLBACK_TRACE` prints the queue of the driver

Callbacks

The flipped bit the scoreboard does not see

```
$ cd code/u7/callbacks && bash run.sh
```

- The `flip_bit_cb` flips a bit of `A` in one out of every eight transactions. The scoreboard **says nothing**, and both runs close at 0 `UVM_ERROR`
- It is not a hole in the checker: it is the proof that **the monitor watches the wire**, not what the driver thought it was going to send
- A monitor that rebuilt the transaction from the driver —or that copied it from the sequencer— would have given *PASS* on data the DUT never saw
- What can be seen of the `jitter_cb` is the **time**: same stimulus, more cycles

The hook	Changes	How many at a time
Constraint	the values	as many as you like
Factory override	the whole class	**one**
Callback	what a class does at one point	as many as you like

Sequences

The only thing left hard-wired

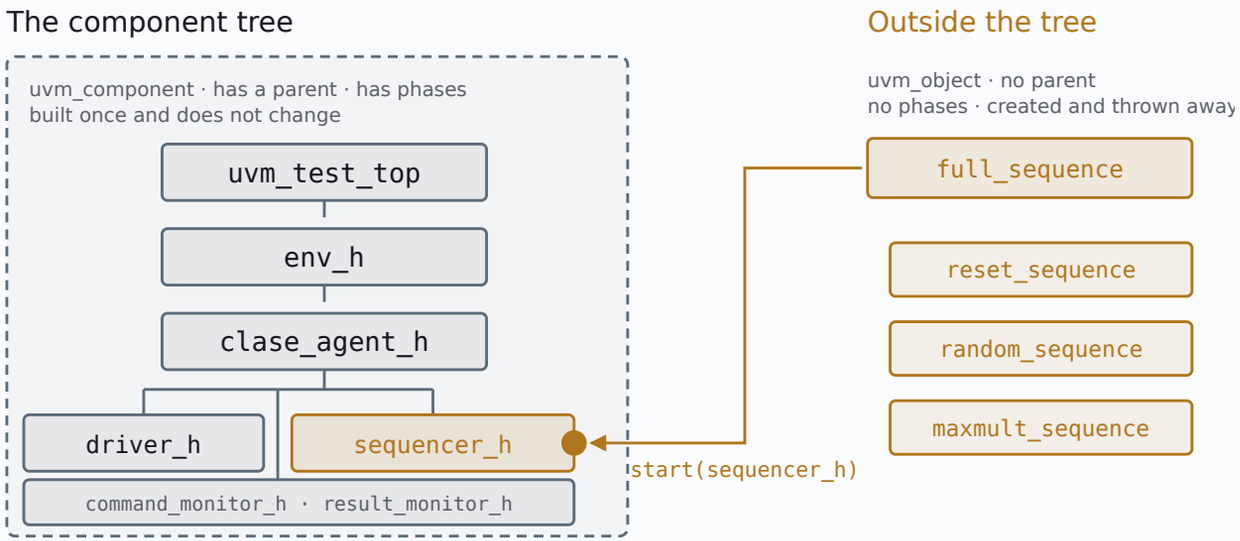
- The agents left the testbench encapsulated: one `agent` per interface, the analysis in the `env`, and nobody naming somebody else's components
- Except for one line, the one in the `run_phase` of the test:

```
seq.start(env_h.clase_agent_h.sequencer_h);
```

- The test **goes across the hierarchy** to reach the sequencer, and along the way it gets to know the inside of the `env` we have just closed
- And there is something bigger behind it: that `command_sequence` is **one** class doing **three** things — resetting, sending a thousand random operations and one directed
- This section is about what happens inside `body()`, and about how a sequence gets started without wiring it

Sequences

An object, not a component



Even though it is not in the tree, UVM reports it under the sequencer, with `@@` in between:

`uvm_test_top.env_h.clase_agent_h.sequencer_h@@full_seq.random_seq`

- The `uvm_sequencer` is a `uvm_component`: **it is in the tree**, it has a parent, it has phases, the `build_phase` of the agent builds it
- The `uvm_sequence` is a `uvm_object`: **it is not in the tree**, it has no parent, it has no phases, and it gets created with a single argument — just like the transactions
- Practical consequence: it gets created, runs, finishes and is thrown away. You can run another one right after, or two at once, without touching the structure

Sequences

The smallest sequence there is

```
// The smallest sequence there is: a single item.
//
// It is a uvm_object, not a uvm_component: it is not in the tree, it has no
// parent, it has no phases. It gets created, it runs, and it gets thrown away.
class reset_sequence extends uvm_sequence #(command_transaction);
   // `uvm_object_utils, NOT `uvm_component_utils: the constructor takes a single
   // argument and there is no parent to pass.
   `uvm_object_utils(reset_sequence)

   function new(string name = "reset_sequence");
      super.new(name);
   endfunction : new

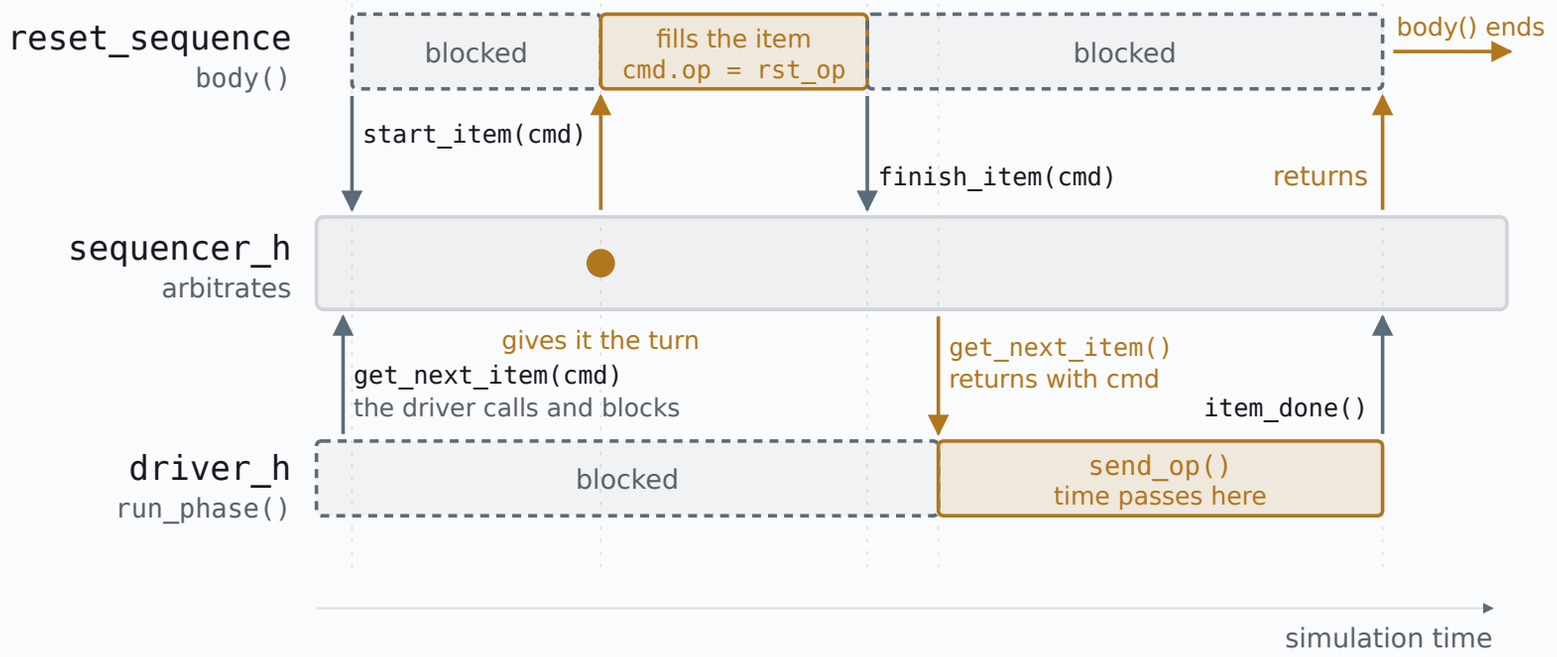
   // body() is a TASK, not a function: it blocks inside. Nobody calls it by
   // hand; UVM calls it when somebody starts the sequence.
   task body();
      command_transaction command;
      command = command_transaction::type_id::create("command");
      start_item(command);    // blocks until the sequencer gives us our turn
      command.op = rst_op;    // only now do we decide what to send
      finish_item(command);   // blocks until the driver calls item_done()
   endtask : body

endclass : reset_sequence
```

- It extends `uvm_sequence #(T)`, parameterized with the item it is going to send
- It gets registered with `` `uvm_object_utils `` — **not** with `` `uvm_component_utils ``
- Constructor with a single argument: `name`, without `parent`
- All the work lives in `body()`, which is a **task**: it can block
- Nobody calls `body()` by hand: UVM calls it when somebody starts the sequence

Sequences

The handshake, now from the sequence side



Time only advances on the amber driver bar: filling the item costs no edge.
Without the `item_done()`, the sequence stays in `finish_item()` forever.

- The same four calls as the agents, looking at the other lane: `start_item()` blocks the **sequence** until the sequencer gives it the turn, and `finish_item()` blocks it until the `item_done()` of the driver
- Simulation time advances **only** in the amber bar: filling the item does not cost a clock edge
- Two threads, four calls and no `#delay` in the middle

Sequences

What happens between `start_item()` and `finish_item()`

```
task body();
   command_transaction command;

   repeat (count) begin : random_loop
      // Through the factory, same as in the Transactions section: that is what lets
      // add_test change the TYPE without touching a line of this sequence.
      command = command_transaction::type_id::create("command");

      start_item(command);

      // Late randomization: the values are picked AFTER getting the
      // sequencer's turn, not when the object was created.
      if (!command.randomize())
         `uvm_fatal("RANDOM SEQUENCE", "randomize() failed")

      finish_item(command);
   end : random_loop
endtask : body
```

- When `start_item()` comes back, the sequence **already has the turn** of the sequencer: nobody else is going to beat it to the driver
- Only then do the fields get filled, or the randomize happen. That is **late randomization**: the values get chosen at the last possible moment
- It is useful because by then it is already known what state the DUT is in, and a constraint can depend on that
- `randomize()` still goes with its `else`: it returns 0 and aborts nothing — it is the same thing they saw in the Constrained random unit

Sequences

Directed late randomization, and the Verilator hole

```
start_item(command);
command.data.constraint_mode(0);                        // 1. I switch off the dist
if (!command.randomize() with {A inside {[1:10]};})     // 2. I ask for the directed one
   `uvm_fatal("SEQ", "randomize() failed")
finish_item(command);
```

- Measured with Verilator 5.052, on a class with the constraints of `command_transaction`, 100 randomizations of each form:

```
with {op == mul_op}                     100 of 100   op has no dist
with {A inside {[1:10]}}                  1 of 100   A does have one
constraint_mode(0) + with {A inside ...}  100 of 100   the workaround
```

- The rule, in one line: **if the `with {}` touches a field that has a `dist`, switch that constraint off first**
- It is not the language, it is the tool: `repro-dist-with.sv` in `code/verilator/`

Sequences

The item comes back with the result inside

```
task run_phase(uvm_phase phase);
   command_transaction command;
   shortint unsigned   alu_result;
   forever begin : command_loop
      seq_item_port.get_next_item(command);  // blocks until there is an item
      bfm.send_op(command.A, command.B, command.op, alu_result);
      command.result = alu_result;  // the way back
      seq_item_port.item_done();  // unblocks the finish_item()
   end : command_loop
endtask : run_phase
```

- The driver writes `command.result` **before** calling `item_done()`
- The sequence still holds the handle to that same object: when `finish_item()` comes back, the result is there
- It is the way back, and that is why `command_transaction` gains a field `result` in this unit — the only change to the transaction since it was created
- It breaks *MOOCOW* on purpose: the driver modifies an object it did not create. It is agreed between the two parties, and it is the only place in the testbench where it is done

Sequences

The formal way back: `get_response()`

```
// The sequence declares TWO types, and the response is no longer the request
class fib_sequence extends uvm_sequence #(command_transaction, result_transaction);
   start_item(cmd);
   finish_item(cmd);
   get_response(rsp);              // blocks until the driver answers

// And the driver, on the other side:
   seq_item_port.get_next_item(req);
   rsp = result_transaction::type_id::create("rsp");
   rsp.set_id_info(req);           // without this the response does not find its sequence
   seq_item_port.item_done(rsp);   // or item_done() now and put_response(rsp) later
```

- The shared handle of the previous slide holds as long as the driver is **blocking and serves one at a time**. The course's one is
- A *pipelined* driver calls `item_done()` as soon as it pushes the command and the response arrives N cycles later: by then the sequence has already sent another item, and writing into `req.result` writes **to the wrong item**
- `set_id_info(req)` copies the sequence and transaction id onto the response. It is what makes `get_response()` know who to answer
- The other two cases where it is needed: a VIP that **clones** the item, and a protocol with **more than one response** per request

Sequences

Fibonacci: when the stimulus depends on the result

```
task body();
   byte unsigned n_menos_2 = 0;
   byte unsigned n_menos_1 = 1;
   command_transaction command;

   command = command_transaction::type_id::create("command");
   start_item(command);
   command.op = rst_op;
   finish_item(command);

   `uvm_info("FIBONACCI", "Fib(01) = 000", UVM_MEDIUM)
   `uvm_info("FIBONACCI", "Fib(02) = 001", UVM_MEDIUM)

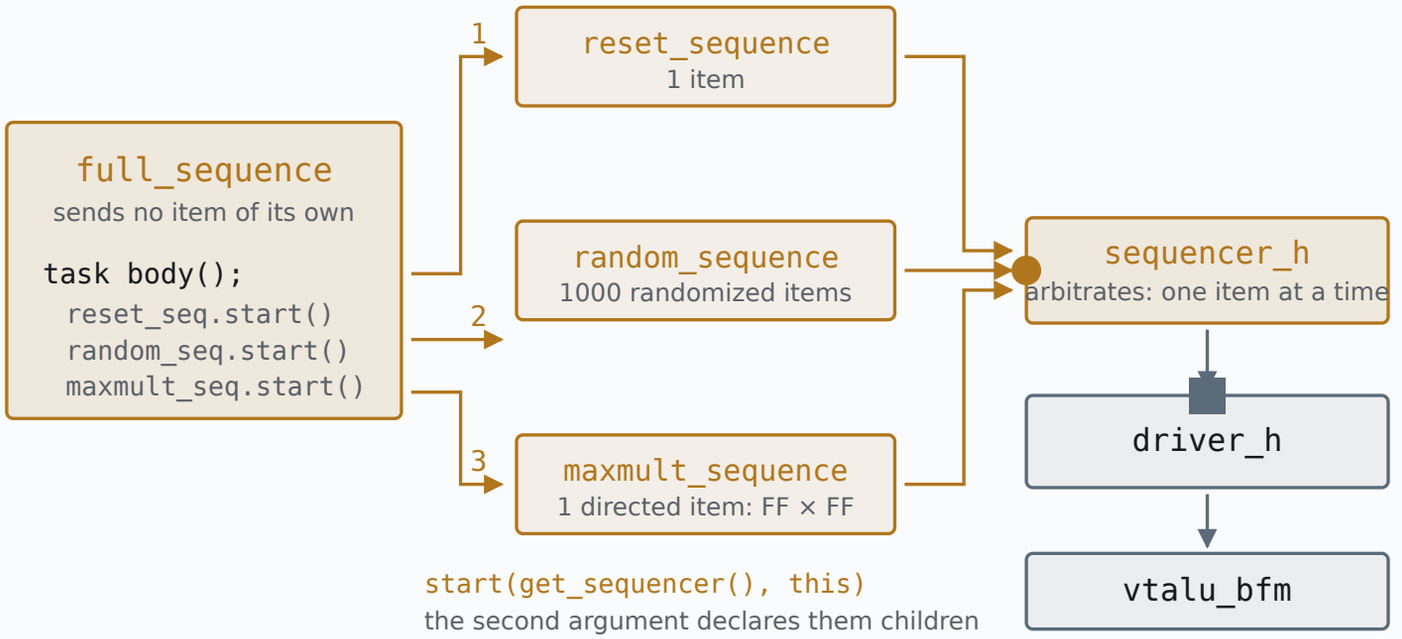
   // Up to 14 and no further: Fib(14) = 233 and Fib(15) = 377, which does not
   // fit in the 8 bits of A and B.
   for (int ff = 3; ff <= 14; ff++) begin : fib_loop
      command = command_transaction::type_id::create("command");
      start_item(command);
      command.A  = n_menos_2;
      command.B  = n_menos_1;
      command.op = add_op;
      finish_item(command);        // returns when the driver called item_done()

      n_menos_2 = n_menos_1;
      n_menos_1 = command.result;  // the driver wrote it inside the item
      `uvm_info("FIBONACCI", $sformatf("Fib(%02d) = %03d", ff, n_menos_1),
                UVM_MEDIUM)
   end : fib_loop
endtask : body
```

- Every addition needs the result of the previous one: without a way back this cannot be written
- `finish_item()` comes back **after** the ALU has finished, so `command.result` is already valid
- It is the same `env`, the same agent, the same driver and the same BFM as the random test. Only the sequence changes

Sequences

Sequences that call sequences



In series, as drawn.
With `fork` / `join` around the `start()`, the sequencer interleaves the items.

- A sequence can start others: it is the way of composing stimulus without duplicating code
- The `command_sequence` of the agents, split into three pieces that now get combined however you like
- The sequencer arbitrates: whatever it receives, it hands the driver **one item at a time**

Sequences

full_sequence: three pieces, one single sequencer

```
task body();
   reset_sequence   reset_seq;
   random_sequence  random_seq;
   maxmult_sequence maxmult_seq;

   reset_seq   = reset_sequence::type_id::create("reset_seq");
   random_seq  = random_sequence::type_id::create("random_seq");
   maxmult_seq = maxmult_sequence::type_id::create("maxmult_seq");

   random_seq.count = count;

   // get_sequencer() returns the sequencer start() handed us: the children run
   // on the same one. The second argument declares them CHILDREN -- without it
   // they compete with us in the arbitration instead of inheriting our turn.
   reset_seq.start(get_sequencer(), this);
   random_seq.start(get_sequencer(), this);
   maxmult_seq.start(get_sequencer(), this);
endtask : body
```

- `get_sequencer()` returns the sequencer that `start()` handed to **this** sequence: the daughters run on the same one
- The second argument, `this`, declares them **daughters**. Without it they compete with the mother in the arbitration instead of inheriting her turn
- `random_seq.count = count` is the advantage of the sequence being an object: it gets configured between the `create()` and the `start()`
- To run two in parallel, `fork` / `join` around the `start()`, and the sequencer interleaves the items

Sequences

Starting a sequence, form 1: `start()`

```
task run_phase(uvm_phase phase);
   full_sequence full_seq;
   full_seq = full_sequence::type_id::create("full_seq");

   // THE TEST raises the objection, not the sequence: the sequence knows
   // nothing about phases, and has to be able to run inside another sequence.
   phase.raise_objection(this);
   full_seq.start(sequencer_h);   // does not return until body() finished
   phase.drop_objection(this);
endtask : run_phase
```

- `start(sequencer)` hands the sequence the sequencer it is going to run on and **does not come back until `body()` has finished**
- The test raises the objection before and drops it after: the run lasts exactly as long as the stimulus lasts
- It is explicit and reads top to bottom. For a test that runs a single sequence, it is what suits
- And it still has the problem of the first slide: `sequencer_h` came out of `env_h.clase_agent_h.sequencer_h`

Sequences

Form 2: default_sequence through uvm_config_db

```
function void build_phase(uvm_phase phase);
   full_sequence full_seq;

   // super.build_phase() of base_test, which is a class of OURS: it builds the
   // env. What the course never calls is uvm_component's build_phase.
   super.build_phase(phase);

   full_seq = full_sequence::type_id::create("full_seq");
   full_seq.count = 200;

   // WITHOUT this line main_phase ends at t=0: nobody raises the objection,
   // and a phase without an objection lasts zero. The test PASSES with no stimulus.
   full_seq.set_automatic_phase_objection(1);

   // The instance name carries the "_phase" suffix: that is the phase the
   // sequencer will start it in. Notice the test no longer names the sequencer
   // as a handle: it names it as a PATH, and that is a config, not code.
   uvm_config_db#(uvm_sequence_base)::set(
       this, "env_h.clase_agent_h.sequencer_h.main_phase", "default_sequence",
       full_seq);
endfunction : build_phase
```

- The test **has no** `run_phase`: the sequencer starts the sequence on its own, when the phase begins
- The instance name carries the suffix `_phase`: `"...sequencer_h.main_phase"` means *"in the `main_phase` of that sequencer"*
- It can be configured per **instance** (as here, with `uvm_config_db #(uvm_sequence_base)`) or per **type** (`uvm_config_db #(uvm_object_wrapper)` + `get_type()`). If both are there the one with the **higher precedence** wins, and at equal precedence the **last** `set()`
- The sequencer stopped being a handle and became a **path**: that is already configuration, not code

Sequences

When to use which

	`seq.start(sqr)`	`default_sequence`
Where it is written	`run_phase` of the test	`build_phase` of the test, or the `top`
Objection	the test handles it	`set_automatic_phase_objection(1)`
It reads	top to bottom	you have to go looking for it
Several sequences in order	trivial	one per phase
Changing it without recompiling	no	yes, it is a config
Knows the inside of the env	yes, through the handle	no, only the path
Reusable third-party agent	not always possible	it is the intended mechanism

- For the tests of the course: `start()`, because it shows
- For an agent you hand over to another team: `default_sequence`, because it lets them change the stimulus without touching your code

Sequences

The `uvm_do macro and why the course does not use it

```
`uvm_do(command)                          // one line

command = command_transaction::type_id::create("command");   // four
start_item(command);
if (!command.randomize()) `uvm_fatal("SEQ", "randomize() failed")
finish_item(command);
```

- `` `uvm_do `` does exactly those four things: create, `start_item`, randomize, `finish_item`
- What it hides: the `randomize()` **without an else**, so a constraint without a solution goes by in silence
- What it prevents: filling fields by hand between the two calls — that is, everything on the previous slides
- What it breaks: the error message points at the expanded macro, not at your code
- There is a whole family (`` `uvm_do_with ``, `` `uvm_create ``, `` `uvm_send ``) and the same argument holds for all of them

Sequences

The override still works, and no sequence found out

```
// The same add_test as the Transactions section, word for word. No sequence finds
// out: random_sequence still asks the factory for a command_transaction,
// and the factory hands it back an add_transaction.
class add_test extends full_test;
   `uvm_component_utils(add_test)

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

   function void build_phase(uvm_phase phase);
      command_transaction::type_id::set_type_override(add_transaction::get_type());
      super.build_phase(phase);
   endfunction : build_phase

endclass : add_test
```

- It is the same `add_test` as the transactions, word for word
- `random_sequence` still creates `command_transaction::type_id::create()`; the factory hands it back an `add_transaction`
- Measured result: 1000 operations, **all** `add_op`, 0 `UVM_ERROR`
- `maxmult_sequence` still sends `mul_op` under `add_test` — because it does not randomize, and **the constraint only acts on `randomize()`**

Sequences

Two sequences on the same sequencer: the arbitration

- Up to here every sequencer has had **one** sequence at a time. In a real testbench there are several in parallel on the same driver —background traffic, a directed test, an error injector— and somebody has to hand out the turns

```
fork
   fondo_seq.start(sqr);                 // priority 100, the default one
   urgente_seq.start(sqr, null, 500);    // the third argument is the priority
join
```

The mode, in `sqr.set_arbitration(...)`	**How it hands out**
`UVM_SEQ_ARB_FIFO`	order of arrival, and **the priority is not looked at** (default)
`UVM_SEQ_ARB_STRICT_FIFO`	the priority rules; at equal priority, order of arrival
`UVM_SEQ_ARB_WEIGHTED`	at random, with the priority as the weight
`UVM_SEQ_ARB_RANDOM`	at random, all equal

The mode, in `sqr.set_arbitration(...)`	**How it hands out**
`UVM_SEQ_ARB_STRICT_RANDOM`	at random, but only among the highest priority ones
`UVM_SEQ_ARB_USER`	`user_priority_arbitration()`, you write it

- The default **ignores the priority**: putting a 500 in and not changing the mode is the mistake of this slide, and nobody gives you a warning

Sequences

`lock()` and `grab()`: when the arbitration is not enough

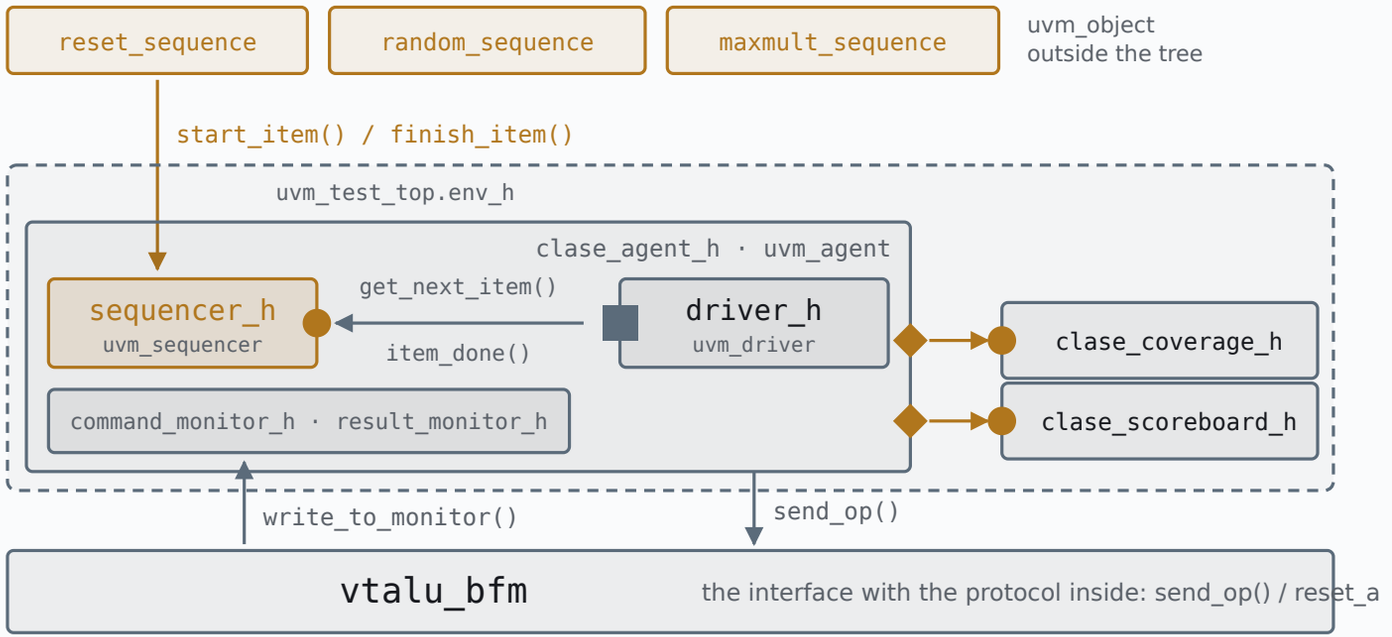
- There are scenarios that **cannot be interleaved**: a configuration sequence, an atomic burst, a read-modify-write. If another sequence slips an item in the middle, the scenario stopped being the one you wrote

```
task body();
   grab();                  // cuts ahead of everyone and closes the door
   ... the items of the scenario, with nobody in between ...
   unlock();                // ungrab() is the same method, under another name
endtask
```

- `lock()` **queues up**: it waits its turn like anybody else and only then closes the door. `grab()` **cuts in**: it goes ahead of everyone, even of the higher priority ones
- Both are released with `unlock()`. Without the `unlock()`, if the sequence is **still alive** the sequencer stays closed forever and without a single error; if its `body()` finishes, UVM takes it away and shouts `SEQFINERR`
- The rule: `grab()` for reset and error recovery —what cannot wait—, `lock()` for everything else, and the `unlock()` in the same `body()`

Sequences

The testbench, complete



Everything in the dotted box is built once and does not change.
What changes between one test and another is the row above.

- It is the diagram drawn today in any verification project, with the names used today
- Everything inside the `env` is built once and does not change. What changes between one test and another is the cloud of sequences at the top

Sequences

Summary of the unit · the sequence and the item

- The stimulus came out of the component tree: it is a `uvm_object`, not a `uvm_component`, and that is why it gets created, configured, run and thrown away
- Everything happens inside `body()`, between `start_item()` and `finish_item()`: that is where the item gets filled, that is where the `randomize()` goes, and that is where `pre_do` and `mid_do` hook on
- The driver writes the result **inside the item** before `item_done()`: it is the way back, and it is what makes a stimulus that reacts possible
- Two ways of starting: explicit `start(sequencer)`, or `default_sequence` through `uvm_config_db` — which besides takes the last hard-wired line out of the testbench

Sequences

Summary of the unit · how they compose

- Sequences compose: one calls others, in series or with `fork`/`join`
- And when there is **more than one sequencer**, the one that composes is a **virtual sequence**, which comes just as day 7 starts ↪ day 7
- With **two sequences on the same sequencer** the one who hands out is the sequencer: priority and `set_arbitration(...)`, and `lock()`/`grab()` for the scenario that cannot be interleaved
- With this the testbench of the course is complete —it is missing one single piece, tomorrow's— and it is the way it is written today in the industry

Review · Day 6

1 of 10 · is_active

An agent in `UVM_PASSIVE`, what does its `build_phase` build?

- **a** The monitors and the analysis ports; the driver and the sequencer, no ✓
- **b** Nothing: a passive agent is an empty shell
- **c** Everything the same as an active one, but without connecting the driver to the sequencer
- **d** Only the sequencer, so it can receive sequences from another agent of the same env

The monitors, always; the sequencer and the driver are left at `null` — watching is never optional: a passive agent goes on feeding scoreboard and coverage. What gets skipped is what *drives* the interface, because there somebody else is already driving it.

Review · Day 6

2 of 10 · The handshake of the driver

The driver calls `get_next_item()`, drives the signals and forgets the `item_done()`. What happens?

a Compilation error: UVM demands the complete pair

b The sequencer hands over the next item just the same, with a warning

c The item gets discarded and the scoreboard reports a mismatch on the next comparison

d The sequence stays waiting in `finish_item()` and does not advance any more ✓

It hangs inside `finish_item()`, and without saying anything — it is the classic mistake of the first week, and the symptom misleads: there is no error and no warning, time stops advancing and the objection never gets dropped. `get_next_item()` is a loan; `item_done()` is giving it back.

Review · Day 6

3 of 10 · Scope of the config_db

The `env` instantiates two agents and does both `set()` with the scope `"*"`. What does each one receive?

- **a** Both fail with `uvm_fatal`: the string `"config"` is duplicated
- **b** Each one receives its own, in order of creation
- **c** Both receive the same one: the second `set()` overwrites the first ✓
- **d** The first receives its config and the second is left with `cfg == null`

Both receive the same thing — the `uvm_config_db` does not match by order nor by type: it matches by **path**. With `"*"` both entries describe the same components, so the last one wins. The scope is a path in the tree, not a label.

Review · Day 6

4 of 10 · The sequencer

`typedef uvm_sequencer #(command_transaction) sequencer;` compiles without touching the transaction of the transactions. Why?

a Because it is registered in the factory with `` `uvm_object_utils ``

b Because `uvm_sequencer` accepts any `uvm_object`

c Because the driver does the `$cast` internally

d Because `command_transaction` extends `uvm_sequence_item` ✓

Because of the base class we chose in transactions — `uvm_sequencer #(T)` requires that `T` derive from `uvm_sequence_item`. If that day the transaction had extended a plain `uvm_transaction`, this `typedef` would not compile today. A decision of one section enabling the next.

Review · Day 6

5 of 10 · super.build_phase()

Instead of the config object, you put `is_active` straight into the `uvm_config_db`. In this course the agent starts up active all the same. Why?

a Because `is_active` is `protected` and the `config_db` cannot write it

b Because the `config_db` does not accept enumerated types, only `int` and `string`

c Because `is_active` is fixed in the constructor and `build_phase` arrives late

d Because the one who reads it is `uvm_agent::build_phase`, and nobody calls `super` ✓

That mechanism is switched off — `uvm_agent::build_phase` looks for `is_active` in the resource pool (it is in `code/.uvm/src/comps/uvm_agent.svh`, it can be opened). Without `super.build_phase()` that line never runs, and nobody warns you. Both ways are valid; what does not work is half of each one.

Review · Day 6

6 of 10 · Object, not component

What is the practical difference of a `uvm_sequence` being a `uvm_object` and not a `uvm_component`?

- **a** That it gets created, runs and is thrown away ✓
- **b** That it cannot be registered in the factory nor overridden
- **c** That it cannot have `rand` fields nor constraints
- **d** That UVM builds it in `build_phase`, like any other class of the tree

It gets created, runs and is thrown away — you can start several, one after the other, on the same sequencer. A component is built once in `build_phase` and lives to the end. That is why the stimulus cannot be a component: it changes from test to test, and sometimes within the same test. Out of that comes as well that it can be configured between the `create()` and the `start()`, the way `full_seq.count = 200` does.

Review · Day 6

7 of 10 · start_item()

`start_item(command)` has just come back. What is it that this guarantees?

a. That the driver has already received the item and is driving the signals

b. That the sequencer gave the turn to this sequence ✓

c. That the `randomize()` has already been resolved with the constraints of the class

d. That the objection of the phase is already raised by the sequencer

You have the turn —nobody else is going to beat you to the driver— and you have not handed anything over yet — and that is why the `randomize()` goes *after*: it is the last possible moment to choose the values, when you already know what state the DUT is in. That is late randomization, and it is where `pre_do()` and `mid_do()` hook on.

Review · Day 6

8 of 10 · The way back

At what moment does `command.result` have a value that can be read?

a When `finish_item()` came back, because the driver wrote it before calling `item_done()` ✓

b As soon as `start_item()` came back

c When the `result_monitor` publishes it through its analysis port

d Never: to receive a response you have to use the REQ/RSP pair of `uvm_sequence #(REQ, RSP)`

After `finish_item()` — there is no return channel at all: there is a shared handle and an agreement between the two parties. The REQ/RSP pair exists and is the formal mechanism, but almost nobody uses it: writing the result into the request is enough. This is what makes Fibonacci possible.

Review · Day 6

9 of 10 · default_sequence

You configure a `default_sequence` through `uvm_config_db` and forget the `set_automatic_phase_objection(1)`. What happens?

a `uvm_fatal` in `build_phase`: the sequence does not find the sequencer

b It runs just the same: when there is a `default_sequence`, the sequencer raises the objection on its own

c The `main_phase` finishes at t=0 and the test passes with 0 errors ✓

d The simulation hangs waiting for an objection nobody drops

It passes in zero seconds without sending a single piece of stimulus, and it lies — a phase without an objection finishes as soon as it starts. It is the same kind of bug as the `new()` that eats the override: it does not break, it deceives. And in a regression of a thousand tests, the one that passes in zero seconds is one nobody looks at.

Review · Day 6

10 of 10 · Sub-sequences

`full_sequence` starts its daughters with `reset_seq.start(get_sequencer(), this)`. What is the second argument for?

a — To pass it the sequencer, because `get_sequencer()` only returns the type

b — To declare the daughter a sub-sequence of the mother ✓

c — So the daughter runs in a separate thread, in parallel with the mother

d — To register the daughter in the factory under the name of the mother, so it can be overridden

It declares the mother-daughter relationship — without it, the sequencer treats the two as independent sequences and they compete for the turn. With it, the daughter inherits the mother's turn and her priority in the arbitration. And for the parallel that is not enough: a `fork` / `join` around the `start()` is needed.

Exercise · Day 6 · 1 of 3

The agent that only watches

```
$ cd code/ejercicios/d6-agents && bash run.sh
```

- The `env` has a single agent. Add the **passive** one on the second VTALU
- And make `is_active` actually count: today the agent builds driver and sequencer always
- It is graded with `+UVM_CONFIG_DB_TRACE`: the census of the tree is written by UVM, not by you

Exercise · Day 6 · 2 of 3

A sequence that only multiplies

```
$ cd code/ejercicios/d6-sequences && bash run.sh
```

- Write `mult_sequence` and the `mult_test` that starts it
- The driver, the agent and the `env` are already there: adding stimulus **does not touch the structure**, which is precisely what the section came to demonstrate
- It is graded crosswise against what the `chequeo` sees on the bus, not against your count

Exercise · Day 6 · 3 of 3

Three planted bugs, and none of them looks like the others

```
$ cd code/ejercicios/d6-debug && bash run.sh
```

	File	How it shows up
1	`driver.svh`	**it hangs**: it dies on `[PH_TIMEOUT]`
2	`default_seq_test.svh`	**it ends at `t=0`** and says PASS
3	`random_sequence.svh`	**it lies in green**: `add_test` passes with the wrong stimulus

- The checker says **which** of the three is still broken. Which line it is, is the exercise

Yesterday we left...

Where the testbench stands

- The testbench was left **reusable for one agent**: agents with `is_active`, the analysis in the `env`, and the stimulus outside the tree, in sequences. The last piece —coordinating two— is this morning
- No line of the test names a component from inside the `env`: the last one went with the `default_sequence` through `uvm_config_db`
- And the other half of the verification plan is missing: the rows **no scoreboard can close**, because they are not about the result but about the protocol

Agenda

Day 7 · ≈ 5 h 15 · unit 8 · the other half

- Warm-up: five seeds and a merge, with `make regresion`
- Virtual sequences: the last piece of the reusable testbench
- Assertions (SVA)
- The capstone: an APB testbench, from scratch
- From the VTALU to a real bus
- The debug toolbox · The 21 silent traps
- Glossary, references and wrap-up
- Self-assessment: fifteen things you should be able to do by now
- *On Monday*: what to do with this once the screen goes off

*And afterwards, **day 8**: optional, and for whoever has already handed in the capstone*

By the end of the day you can:

- **Write** a property that sees a protocol violation, and say why `--assert` is not optional
- **Coordinate** two interfaces from a single virtual sequence
- **Hand in** a whole testbench for a DUT you have not seen before

Exercise · Day 7 · 1 of 2

The same sequence, another seed

```
$ cd code/ejercicios/d7-semillas && bash run.sh
```

- The only exercise of the course without SystemVerilog: what gets written is the **regression**, six lines of shell
- Run the same test with seeds 1 to 5 and **merge** the five coverages with `verilator_coverage --write`
- It is graded by checking that the merge covers more than the best of the five on its own

Virtual sequences

The problem: two interfaces and a single `start()`

- A **virtual sequence** is a sequence that does not send items of its own: it only starts other sequences, on **several sequencers**
- It shows up when the DUT has more than one interface —a configuration bus and a data stream, for example— and the test needs to coordinate them: *configure first, then send traffic*
- It is called "virtual" because it is not tied to a type of item nor to a sequencer
- The `u7/agents` already left us the scenario built: **two VTALU**. There the second agent was passive; with both active there are **two sequencers**, and something to coordinate
- `full_sequence` is already half a virtual sequence: it sends no items of its own, it only starts others. The only thing it lacks is the second sequencer

Virtual sequences

The virtual sequencer: handles, not items

```
class virtual_sequencer extends uvm_sequencer;
   `uvm_component_utils(virtual_sequencer)

   sequencer clase_sequencer_h;
   sequencer modulo_sequencer_h;

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

endclass : virtual_sequencer
```

```
      // The handles are passed HERE and not looked up by string: in connect_phase
      // both agents already exist, and if somebody renames a component this does
      // not compile instead of returning null at simulation time.
      virtual_sequencer_h.clase_sequencer_h  = clase_agent_h.sequencer_h;
      virtual_sequencer_h.modulo_sequencer_h = modulo_agent_h.sequencer_h;
```

- It has no queue, it does not arbitrate, it does not talk to any driver. It is a `uvm_component` that exists in order to **hold the handles** and to live in the tree with a name
- It extends `uvm_sequencer` **without parameterizing it**: the default item is `uvm_sequence_item` and none ever gets sent
- The handles get assigned in the `connect_phase`, where the agents already exist

Virtual sequences

The virtual sequence: `p_sequencer` and two branches

```
// `uvm_declare_p_sequencer declares `p_sequencer` with the type above and casts
// it on its own inside m_set_p_sequencer. Without it, get_sequencer() returns a
// uvm_sequencer_base and every use would need a cast by hand.
class coordinada_sequence extends uvm_sequence;
   `uvm_object_utils(coordinada_sequence)
   `uvm_declare_p_sequencer(virtual_sequencer)
```

```
      // --- 1. Both ALUs get reset at the same time -----------------------------
      // fork/join over TWO sequencers. Each branch blocks on its own driver, and
      // the join waits for both: that cannot be written inside a normal
      // sequence, which only knows the sequencer that started it.
      fork
         reset_a.start(p_sequencer.clase_sequencer_h);
         reset_b.start(p_sequencer.modulo_sequencer_h);
      join

      // --- 2. Parallel traffic, different on each interface -------------------
      // A does the directed overflow case while B sends at random.
      fork
         maxmult.start(p_sequencer.clase_sequencer_h);
         random_b.start(p_sequencer.modulo_sequencer_h);
      join
```

- `` `uvm_declare_p_sequencer `` declares `p_sequencer` **with the type of the virtual sequencer** and casts it on its own. Without it, `get_sequencer()` returns a `uvm_sequencer_base` and you have to cast by hand at every use
- The `fork`/`join` is over **two sequencers**: each branch blocks on its driver and the `join` waits for both
- That cannot be written inside a normal sequence, which knows a single sequencer

Virtual sequences

What no sequence on its own can do

```
// --- 3. What no single sequence can do ---------------------------------
// The result of VTALU A is the operand of B. It is a dependency BETWEEN
// interfaces: the second one cannot even start being built until the
// first one answered.
primera.A  = 8'h0F;
primera.B  = 8'h07;
primera.op = mul_op;
primera.start(p_sequencer.clase_sequencer_h);

`uvm_info("COORDINADA", $sformatf(
          "A returned %0d: it goes as an operand into B", primera.result), UVM_LOW)

segunda.A  = primera.result[7:0];
segunda.B  = 8'h01;
segunda.op = add_op;
segunda.start(p_sequencer.modulo_sequencer_h);
```

- The result of VTALU **A** goes in as an operand of **B**: a dependency **between interfaces**, and in series. The second one cannot even be assembled until the first has answered
- The `result` comes back inside the item, which is the way back of this unit
- `0F × 07 = 105`, and B answers `106`. Run it: `code/u7/sequences/virtual/run.sh`

Virtual sequences

And if branch B has to wait for A?

```
// uvm_event: a notice, from one to many. It comes from the global pool, nobody builds it
uvm_event listo = uvm_event_pool::get_global("dut_configurado");

listo.trigger();          // the branch that configured gives notice
listo.wait_trigger();     // the one waiting blocks here
listo.wait_ptrigger();    // the same, but if it already happened, it walks on

// uvm_barrier: a meeting point between N branches
uvm_barrier arranque = uvm_barrier_pool::get_global("arranque");
arranque.set_threshold(2);
arranque.wait_for();      // neither of the two goes on until both have arrived
```

- The `join` synchronizes **at the end**. When branch B has to wait for A **in the middle**, a synchronization object is needed
- `uvm_event` is **one to many**: one gives notice, those waiting move on. And `trigger(data)` can carry a `uvm_object` attached
- `uvm_barrier` is **symmetric**: nobody goes on until everybody has arrived. It is the "start together" of two agents that have to begin on the same cycle
- A shared `bit` with `wait(flag)` is not enough: it **loses the pulse** if the one waiting arrived late, and it carries no data

Assertions (SVA)

The hole the scoreboard left

- The scoreboard of this course checks **the result**: `A op B` against what the DUT gave back. A thousand operations, a thousand comparisons, zero errors
- Nobody checks **the protocol**. A DUT that gives back the right `result` but drops `done` one cycle early, or raises it on a `no_op`, **passes every test of the course**
- The rule of the protocol is written in three places: in prose in the spec, inside the BFM, and in the head of whoever wrote it. In **none** of the three is it checked
- An **assertion** is that same sentence, in executable form, running on its own the whole simulation

Assertions (SVA)

Immediate and concurrent: the one you know and the new one

```
// IMMEDIATE: it is a STATEMENT. It runs when the thread goes past it,
// once, and that is all. The one from the transactions is of this family.
assert (cmd.op inside {add_op, sub_op, and_op, xor_op, mul_op})
   else `uvm_error("SEQ", "invalid operation");

// CONCURRENT -- the new thing. It is a DECLARATION with a clock: it gets
// plugged in when the simulation starts and evaluated on every edge, forever.
a_done : assert property (@(posedge clk) start |=> ##[0:4] done);
```

- The **immediate** one lives inside a `begin/end`, it is procedural, and it only exists while the thread is standing there
- The **concurrent** one lives in no thread: it is a declaration, like an `always`. It gets written once and checks every edge of the whole simulation
- That is why a concurrent one can describe something that **lasts several cycles**, and an immediate one cannot

Assertions (SVA)

Anatomy of a property

```
a_done_llega :                      // <- the label: NOT optional
assert property (
   @(posedge clk)                   // <- the clock: when it gets sampled
   disable iff (!reset_n)           // <- when NOT to check
   start && (op_set != no_op)       // <- the antecedent
   |->                              // <- the implication
   ##[1:5] done                     // <- the consequent
) else `uvm_error("SVA", "...");    // <- what to do if it fails
```

- It reads like a sentence: *"on every edge of `clk`, as long as there is no reset, if `start` is up with a real operation, then `done` arrives within five cycles"*
- The **label** comes out in the error message and in the coverage report. A property without a label is an anonymous error at three in the morning
- The seven parts are always the same. The rest of the section is learning what can go in each one

Assertions (SVA)

|-> against |=>, mistake number one

```
// |->  overlapping: the consequent starts on THE SAME edge
a : assert property (@(posedge clk) start |-> !done);

// |=>  non-overlapping: the consequent starts on the NEXT edge
b : assert property (@(posedge clk) start |=> !done);
```

- `a |=> b` is exactly `a |-> ##1 b`. There is no more mystery than that
- Which one goes in? It depends on whether the response is **combinational** (the same edge) or **registered** (the next one). In the VTALU, `done` comes out of an `always_ff`: it is always the next one
- Getting it wrong does not have a single symptom: `start |-> done` with a registered `done` **fails on every transaction** —it sees the old `done`—, and a property whose antecedent never occurs passes vacuously. That is why it always goes with its `cover`

Assertions (SVA)

Looking at the past: $rose, $fell, $stable, $past

```
$rose(start)      // 0 -> 1 between the previous edge and this one
$fell(done)       // 1 -> 0
$stable(A)        // the value is the same as on the previous edge
$past(result, 3)  // the value it had 3 edges ago
```

- The four of them compare **against the previous sample**, not against the previous instant. In SVA time is counted in edges of the property's clock, not in nanoseconds
- `$stable` is the one that writes *"it was not touched"*, which is half a library of protocol rules
- `$past(x, n)` is a free delay line: it is good for checking a fixed latency without writing a state machine

Assertions (SVA)

The property that is worth the section

```
// --- The stimulus ---

// The rule from slide 1 of day 1, executable at last: while start is up, the
// operands and the operation are not touched.
property p_operandos_estables;
   @(negedge clk) start |=> $stable(A) && $stable(B) && $stable(op_set);
endproperty : p_operandos_estables

a_operandos_estables :
assert property (p_operandos_estables)
else
   `uvm_error("SVA", $sformatf(
              "%m: operand changed with start up: A=%0d B=%0d op=%s", A, B, op2enum().name()))
```

- It is **the rule of *The VTALU spec*, day 1**: while `start` is up, the operands are not touched. It sat written in prose for six days
- Eleven lines, and they check every transaction of every test, on both agents, without anybody connecting them to anything
- The scoreboard **cannot** write this rule: by the time the transaction reaches it, the protocol has already been erased

Assertions (SVA)

⚠ *Two clocks: an assertion is worth what its sampling is worth*

```
// TWO CLOCKS, and it is not an implementation detail: the BFM drives on
// negedge and the DUT registers on posedge. An assertion samples in the
// preponed region of the edge it is given, so a signal written ON the negedge
// is not seen on that negedge: it is seen on the next one. With everything on
// posedge, two consecutive no_op -- start goes down at t=111 and back up at
// t=120, between two posedges -- read as ONE transaction with the operands
// changing: 145 false positives every 1000 operations. With everything on
// negedge, the false positives move to the done properties.
//
//   stimulus (start, A, B, op_set)  -> negedge, where the BFM writes
//   response (done, result)         -> posedge, where the DUT registers

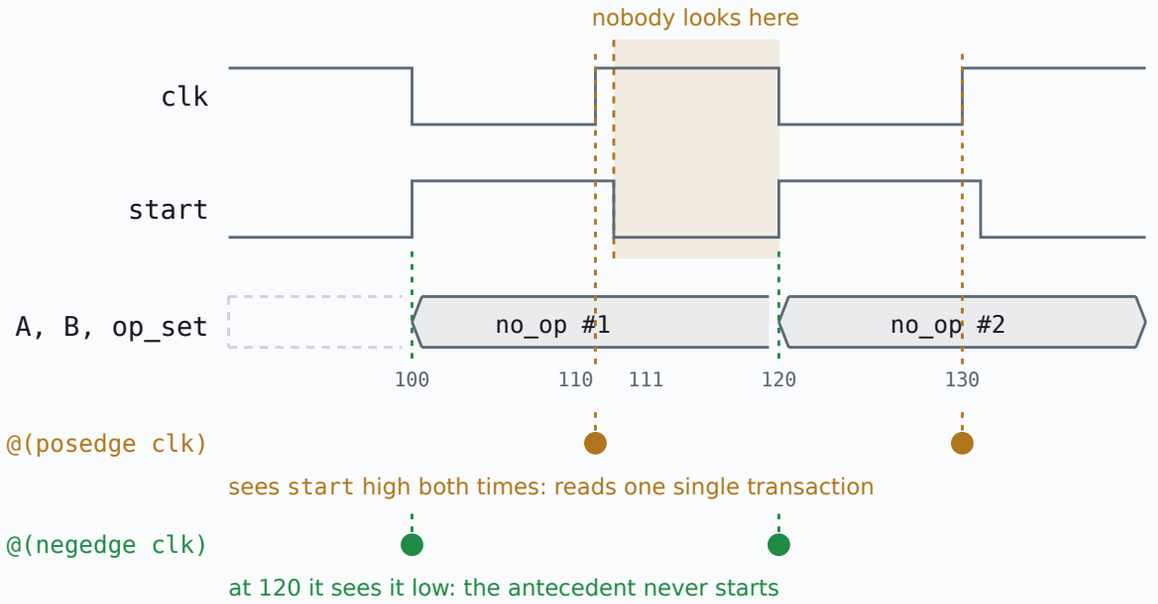
default disable iff (!reset_n);
```

With a single clock	**Over 1000 operations**
everything on posedge	**145 false positives** on the stability of operands
everything on negedge	false positives on the done properties
stimulus on negedge, response on posedge	**0 errors** ✅

- The BFM drives on `negedge` and the DUT registers on `posedge`. **There is no single edge that makes all four properties correct**
- SVA sampling is the *preponed* region: a signal written **on** the edge is not seen on that edge, it is seen on the next one

Assertions (SVA)

The pulse the sampling does not see



Two `no_op` in a row: `start` goes down at 111 and back up at 120, between two `posedge`.
The sampling never sees it go down: the two read as one, with the operands changing in the middle.
145 false positives every thousand operations, from a single character in the `@()`.

- Between two `posedge` a whole `start` pulse fits that the sampling never sees
- All on `posedge`: the two `no_op` read as **one**, and out come 145 screams
- Stimulus on `negedge`: at `t=120` it is already low, the antecedent does not start

Assertions (SVA)

Variable latency, in one line

```
// The variable latency from the VTALU spec section, in one line: one cycle for the
// one-cycle ops, four edges for the multiplication.
property p_done_llega;
   @(posedge clk) start && (op_set != no_op) |-> ##[1:5] done;
endproperty : p_done_llega

a_done_llega :
assert property (p_done_llega)
else `uvm_error("SVA", $sformatf("%m: start con op=%s y done no llego en 5 ciclos", op2enum().name()))

// The fine print: no_op is the only operation that does not answer.
property p_no_op_sin_done;
   @(posedge clk) start && (op_set == no_op) |=> !done;
endproperty : p_no_op_sin_done

a_no_op_sin_done :
assert property (p_no_op_sin_done)
else `uvm_error("SVA", $sformatf("%m: done subio para una no_op"))
```

- `##[1:5] done` says *"between one and five edges later"*. The VTALU takes **one** on `add`/`sub`/`and`/`xor` and **four** on the multiplication: one property covers both
- `##n` is an exact delay, `##[a:b]` a window, `##[1:$]` *"at some point"* — which is almost never what you want: a property without an upper bound can never fail by timeout
- The antecedent **excludes** `no_op` on purpose: it is the only operation that does not answer, and putting it inside would turn the property into a generator of false positives

Assertions (SVA)

`disable iff` and the reset

```
default disable iff (!reset_n);   // once, for the whole interface

property p;
   @(posedge clk) disable iff (!reset_n)   // or property by property
   start |=> done;
endproperty
```

- During a reset **everything is wrong on purpose**: `done` drops, `start` drops, the transactions halfway through get abandoned. Without `disable iff`, every reset is a batch of false positives
- `disable iff` **aborts** the evaluations in flight, it does not make them fail. The property forgets what it was waiting for and starts from zero
- The `default disable iff` at the top of the interface applies it to all of them: it is one line against twenty repetitions

Assertions (SVA)

Where they live: inside the interface

- They go **with the signals**, in the `interface`, not in the class testbench. There they see `clk`, `start`, `A`, `B` and `done` without anybody passing them along
- They do not have to be connected, nor instantiated, nor built in a `build_phase`. They exist because the interface exists
- The `top` of the agents instantiates **two** BFM: the same properties check both VTALU, including the one driven by the legacy module nobody wrote
- A **passive agent** gets them thrown in for free: watching an interface is now also checking it

Assertions (SVA)

`bind`: when the interface is not yours

```
// The checker lives outside. The RTL is not touched -- often it cannot be.
module apb_checks (input bit PCLK, PSEL, PENABLE, PREADY);
   a_setup : assert property (@(posedge PCLK) PSEL && !PENABLE |=> PENABLE);
   c_setup : cover  property (@(posedge PCLK) PSEL && !PENABLE);
endmodule

// And in the top, one line per module you want to watch over:
bind apb_regs apb_checks chk (.*);
```

- The previous slide holds when the interface **is yours**. Somebody else's RTL, the bought IP and the legacy module do not get edited
- `bind` puts an instance **inside** another module from the outside: the checker sees the DUT's internal signals as if it had been written there
- The `.*` connects by name. One `bind` covers **every** instance of that module — or `bind top.dut ...` for a single one
- You can also bind to an `interface`, and Verilator supports both forms

Assertions (SVA)

The integration with UVM: the `else` that makes it count

```
// The default: $error, and in Verilator also $stop — it cuts off on the first failure
a : assert property (p);

// What you want in UVM: the failure counts and the simulation goes on
a : assert property (p)
    else `uvm_error("SVA", $sformatf("%m: operand changed"));
```

- Without an `else`, the default action of an assertion that fails is `$error` —and in several simulators, `$stop`. The simulation gets cut off and the *Report Summary* **does not get printed**
- With an `else` that calls `` `uvm_error ``, the failure comes in through the same channel as everything else: it counts in the summary, it respects `+UVM_MAX_QUIT_COUNT`, and the `run.sh` detects it without changing a line
- The `%m` of the message prints **which instance** of the interface failed: `top.clase_bfm` or `top.modulo_bfm`

Assertions (SVA)

Every assertion goes with its `cover property`

```
// And the cover that goes with it: without this, a regression where the random
// never subtracted too much leaves the assertion green having checked nothing.
c_ovf : cover property (@(posedge clk) done && ovf);
c_sub_sin_borrow : cover property (@(posedge clk) done && (op_set == sub_op) && !ovf);

c_mult_3ciclos : cover property (@(posedge clk) $rose(start) && (op_set == mul_op) ##3 done);
c_mult_4ciclos : cover property (@(posedge clk) $rose(start) && (op_set == mul_op) ##4 done);
c_un_ciclo : cover property (@(posedge clk) $rose(start) && (op_set inside {add_op, sub_op, and_op, xor_op}) ##1 done);
```

```
covergroup : 86.8% (66/76)
user       : 80.0% ( 4/ 5)
```

- A property whose **antecedent never occurs** passes. Without assertion coverage, a switched-off check looks exactly like a green check
- `c_mult_3ciclos` stays at **zero forever**: the chain of the multiplier is `done3 → done2 → done1 → done_mult`, that is **four** edges, not three. The cover at 0 gives away that the mental model of the latency was wrong
- In Verilator the `cover property` land in the **same** `coverage.dat` as the covergroups of the conventional testbench: code, functional and assertion coverage, one single report

Assertions (SVA)

Assertion or scoreboard: the same rule, two places

	Assertion	**Scoreboard**
What it checks	the **protocol**: who moves, and when	the **transformation**: `A op B`
Where it lives	in the `interface`, with the wires	in the testbench, with the transactions
When it screams	on the exact edge on which it breaks	when the result arrives
What it needs	nothing: it plugs itself in	monitor, analysis port, `connect_phase`

- Short rule: **protocol → assertion. Data → scoreboard.** Writing a protocol check in the scoreboard is rebuilding the time the monitor has just taken care of erasing
- And the other way round: writing the reference model of a multiplier in SVA is possible and it is a bad idea

Assertions (SVA)

The example of the section: the bug the scoreboard does not see

```
// +BUG=1: changes B halfway through the multiplication. The
// multiplier already latched A and B on the first edge, so the
// RESULT DOES NOT CHANGE: the scoreboard stays green. The only
// thing that sees it is the assertion. That is the whole section.
if (bug_operandos && iop == mul_op) begin
   @(negedge clk);
   @(negedge clk);
   B = ~iB;
end
do @(negedge clk); while (done == 0);
oresult   = result;
start = 1'b0;
```

```
** Report counts by severity
UVM_INFO :    2
UVM_WARNING :    0
UVM_ERROR :  154
UVM_FATAL :    0
** Report counts by id
[MAXMULT]     1
[RNTST]     1
[SVA]   154
```

- `+BUG=1` changes `B` **halfway through the multiplication**. The multiplier has already latched the operands on the first edge: **the result comes out right all the same**
- The scoreboard compares 1000 operations and does not find a single difference: in the summary there is **not one** `[SELF CHECKER]`. It is doing its job properly — this bug is not a data bug
- The assertion catches it 154 times, on the exact edge. It is the same example as reporting, turned inside out: over there the scoreboard was broken to teach reporting; here the protocol gets broken to show who sees it

Assertions (SVA)

The traps of this section

The symptom	The cause	How to head it off
The property **always passes**	the antecedent never occurs	one `cover property` per assertion
False positives on every transaction	you sample it with the edge it is written on	stimulus and response go on different edges
False positives **at the start** of every test	the `disable iff (!reset)` is missing	`default disable iff` once, at the top
The log says **PASS** and the properties did not run	`--assert` is missing at compile time	the `cover` at 0 is the one that warns you

- The four share the shape of the appendix: **they compile, they run, and they lie**
- The fourth is the cheapest to commit and the most expensive: without `--assert`, Verilator compiles the properties and does not evaluate them. All green, zero checking

Assertions (SVA)

Summary of the unit

- A **concurrent assertion** is a declaration with a clock: it gets written once and checks every edge of the simulation, on its own
- The anatomy never changes: `label : assert property (@(clock) disable iff (reset) antecedent |-> consequent) else action;`
- `|->` is the same edge and `|=>` **is** `|-> ##1`. The question is not which of the two, it is how many edges later the spec promises it
- The properties live **in the** `interface`, with the signals. They do not get connected, they do not get built, and they travel with it
- In UVM the action is `` `uvm_error ``, so the failure **counts** in the *Report Summary* instead of killing the simulation
- **Every assertion goes with its** `cover property`: it is the only check of the check
- And the one that takes the section: **an assertion is worth what its sampling is worth**. The stimulus gets sampled where the stimulus gets written

Assertions (SVA)

What this section does differently

- **The *UVM Primer* has no SVA.** It teaches the class testbench end to end and leaves the assertions out: they belong to the other half of SystemVerilog
- This section exists because the question comes anyway —in the interview, and in the first real block one gets to verify
- The two clocks, the table of the 145 false positives and the `cover property` that never gets covered **did not come out of a tutorial**: they came out of writing this section on the VTALU of the course and looking at why it did not add up
- Everything here runs on Verilator, without licences, with the same `run.sh` as the other thirty-seven examples

Exercise · Day 7 · 2 of 2

The legacy module violates the protocol and nobody knew

```
$ cd code/ejercicios/d7-sva && bash run.sh
```

- The "boss's tester" of the agents drives its VTALU **by hand** for the multiplication, and it goes about getting the operand of the next one ready while it waits for `done`. It has been in production for years
- The scoreboard compares a thousand operations and does not find a single difference: the multiplier latched `A` and `B` on the first edge. **The bug is not a data bug**
- Write the property that does see it. The two `done` ones are already there, as a template
- The checker asks for three things: that it fire on `modulo_bfm`, that it **not** fire even once on `clase_bfm`, and that the scoreboard stay green

Review · Day 7

1 of 5 · Immediate and concurrent

What is the underlying difference between `assert(x.randomize())` and `assert property (@(posedge clk) …)`?

- **a** One is a **statement**; the other, a **declaration with a clock** ✓
- **b** None: the second is syntactic sugar for the first
- **c** The first can be switched off from the command line and the second cannot
- **d** The first is only valid inside a class and the second only inside a module, because of the scheduler

Statement against declaration — the immediate one runs when the thread goes past it; the concurrent one gets evaluated on its own, on every edge of its clock. It is to an `if` what the concurrent one is to an `always_ff`: one executes, the other gets instantiated. That is why only the concurrent one can describe something that lasts several cycles.

Review · Day 7

2 of 5 · `|->` against `|=>`

The `done` of the VTALU comes out of an `always_ff`. Which implication goes in `start |?? done`?

a — `|->`, because the antecedent and the consequent belong to the same transaction

b — `|=>`, because what gets written with `<=` is read one edge later ✓

c — Either of the two: the difference is a matter of style

d — Neither: for registered signals you have to use `$past()` on the antecedent

`|=>` when the consequent comes out of a `<=` — because `|=>` *is* `|-> ##1`, and the question to ask is how many edges later the spec promises it. With `|->` against a registered signal the property does not pass vacuously: it **fails on every transaction**, because it compares against the old `done`. The one that passes quietly is the one whose antecedent never occurs — which is why it always goes with its `cover property`.

Review · Day 7

3 of 5 · The sampling edge

All the properties of the VTALU sampled on `@(posedge clk)` give 145 errors over 1000 operations, and the DUT is healthy. Why?

- **a** The `disable iff (!reset_n)` is missing
- **b** The `posedge` is too fast: the clock has to be divided
- **c** Covergroups and assertions cannot share the clock without a `clocking block`
- **d** The stimulus is written on the `negedge` and the sampling does not see it ✓

An assertion is worth what its sampling is worth — the BFM writes the stimulus on the `negedge`, and on two consecutive `no_op` `start` goes down and comes back up between two `posedge`: the sampling does not see it go down. The stimulus gets sampled where the stimulus gets written. Stimulus on `negedge`, response of the DUT on `posedge`: zero errors. With a single clock there is no way.

Review · Day 7

4 of 5 · The assertion that checks nothing

An `assert` property reports 0 failures during the whole regression. What do you know?

- **a** Nothing yet: it may never have been evaluated at all ✓
- **b** That the rule it describes holds
- **c** That the DUT is free of protocol bugs
- **d** That the property has a badly written `disable iff` and stayed off the whole time

Zero failures and zero evaluations look the same — its antecedent may never have occurred, or `--assert` may be missing and it is not even being evaluated. That is why every assertion goes with its `cover property`: it is the only check of the check. In the section, `c_mult_3ciclos` stays at 0 and gives away that the real latency is four edges, not three.

Review · Day 7

5 of 5 · Assertion or scoreboard

The DUT gives back the right `result` but drops `done` one cycle earlier than the specification says. Who catches it?

a. The scoreboard, when it compares the result

b. The functional coverage, because the `done` bin is left empty

c. An assertion in the interface: it is a **protocol** bug ✓

d. The `uvm_fatal` of the `command_monitor`, which would stop seeing commands on the bus

Protocol → assertion. Data → scoreboard — and it is not a preference: by the time the transaction reaches the scoreboard, the protocol is no longer there. Writing the check there would be rebuilding by hand the time the monitor has just erased.

Capstone · Day 7

The whole testbench, from a blank sheet

```
$ cd code/ejercicios/d7-final && cat spec.en.md
```

- The previous exercises gave you a file with a hole in it. This one gives a **DUT, a spec and nothing else**
- The DUT is not the VTALU: it is an **APB3 slave** with four registers. It has addresses, two phases per transfer, a *wait state* and an error response
- Everything gets written: the interface with the protocol, the transaction, the driver, the monitor, the agent, the scoreboard, the covergroup, the sequences and the tests
- The checker goes **in stages**, and each one prints its `STAGE N OK`. It can be finished one at a time — and that is the way to do it

Capstone · Day 7

The DUT: four registers and the small print

Addr	Name	Access	Contents
0x00	CTRL	RW	bit 0 = EN · bit 1 = CLR, **autoclear**
0x04	SCRATCH	RW	32 bits, and **adds to ACC if EN**
0x08	ACC	RO	the accumulator
0x0C	STATUS	RO	bit 0 = EN · bit 1 = OVF, sticky

```
// The handshake: the write does not wait, the read inserts ONE wait state.
// rd_wait re-arms as soon as PENABLE drops, that is between one transfer and
// the next, with no need to count cycles.
logic rd_wait;
always_ff @(posedge PCLK or negedge PRESETn) begin
   if (!PRESETn) rd_wait <= 1'b1;
   else if (!PENABLE) rd_wait <= 1'b1;
   else if (access && !PWRITE) rd_wait <= 1'b0;
end

assign PREADY  = !PSEL ? 1'b1 : (PWRITE ? 1'b1 : !rd_wait);
assign PSLVERR = access && !mapped;
```

- The read puts in **one wait state**: the driver waits for the handshake, it does not count cycles
- From `0x10` upwards, `PSLVERR`. Writing an RO, on the other hand, is **not** an error

Capstone · Day 7

The order: monitor, driver, scoreboard, coverage

- **1 · The monitor.** The `top.sv` brings **two** slaves: one for your testbench and another driven by an ordinary module, without UVM. A **passive** agent on that one, and let us see the eight transfers. Without a driver
- **2 · The driver.** The protocol inside the interface, and a directed sequence that writes and reads the four registers
- **3 · The scoreboard.** The DUT modelled in software. The checker runs it twice: against the healthy DUT it has to keep quiet, and with `+BUG=1` it has to scream
- **4 · The coverage.** The covergroup with the nine rows of the verification plan of the spec: at least 20 points, 90 % covered. And **two rows come empty**
- **5 · The properties.** The APB protocol inside `apb_if.sv`, with `--assert`. With `+BUG=2` there is a bug the scoreboard cannot see

Capstone · Day 7

Stage 5: the bug the scoreboard cannot see

```
// In apb_if.sv, with the signals. The payload does not move until it ends
property p_payload_estable;
   @(posedge PCLK) PSEL && !(PENABLE && PREADY) |=> $stable({PADDR, PWRITE, PWDATA});
endproperty

a_payload_estable : assert property (p_payload_estable)
   else `uvm_error("SVA", $sformatf("%m: the payload moved before PREADY"))
```

- `+BUG=2` does to the usual module what `d7-sva` did to the legacy tester: **it moves `PADDR` in the middle of ACCESS**
- The DUT answers at the new address, the monitor reconstructs **eight spotless transfers**, and the scoreboard has nothing to compare wrongly
- Measured: **8 `UVM_ERROR [SVA]`** and the same 8 transfers in the monitor's log. The only one that notices is the assertion
- And rows **8 and 9** of the plan come empty: *back to back* and unaligned address. The bins are called `back_to_back` and `unaligned`

Appendix · From the VTALU to a real bus

What does not change

- The DUT of the course has a three-signal handshake. On Monday you are going to run into **AXI, APB or AHB**, and the honest question is how much of this is any use
- The tree, all of it: `test` → `env` → `agent` → sequencer, driver and monitors, with the analysis hanging off the analysis ports. **It is the same diagram**
- The agent is still **one per interface**. A DUT with a configuration APB and a data AXI is two agents, not one bigger agent
- The configuration still comes down through **one object per level**, and the scope of the `config_db` is still a path
- The sequence is still an object that gets created, runs and is thrown away

Appendix · From the VTALU to a real bus

What does change

- **The driver stops being a linear task.** An operation of the VTALU is raising `start` and waiting for `done`. A bus has phases —address, data, response— and in AXI the five channels run **independently**: the driver goes from a `send_op()` to several threads with `fork`
- **The monitor has to reconstruct.** Watching one edge is no longer enough: the address of one channel has to be paired with the data of another, by ID. It is the hardest class of the agent, and the one most underestimated
- **The transaction grows**: address, burst type, length, strobes, response. And the constraints stop being decorative — in AXI a burst cannot cross a 4 KB boundary, and that is a `constraint`, not a comment
- **Virtual sequences stop being a luxury**: configure over APB, then send traffic over AXI with what the configuration gave back. The shape is the one from sequences — `virtual_sequencer`, `p_sequencer`, one `fork` per interface—; the only thing that changes is that the two sequencers are of different types
- **RAL shows up.** If the DUT has registers —and it does—, `uvm_reg` gives you the model and the register tests already done

Appendix · From the VTALU to a real bus

Vertical reuse: the block env inside the system env

```
class soc_env extends uvm_env;
   apb_env apb_env_h;     // the block env, without touching a line of it
   axi_env axi_env_h;
   soc_scoreboard sb_h;   // the system one watches both ends, not the transfers

   function void build_phase(uvm_phase phase);
      // What changes is not the env's code: it is its CONFIGURATION
      apb_cfg.is_active = UVM_PASSIVE;   // here the APB is driven by the SoC's CPU
      uvm_config_db #(apb_env_config)::set(this, "apb_env_h*", "cfg", apb_cfg);
      apb_env_h = apb_env::type_id::create("apb_env_h", this);
   endfunction
endclass
```

- The reuse of the agents section is **horizontal**: two instances of the same agent in the same env. This one is **vertical**, and it is the one that pays off in a SoC
- The block env does not get edited. What changes is its config: the agent becomes **passive** because at system level the bus is driven by the DUT, not the testbench
- That is why `is_active` and the config object are not ceremony: they are the hinge that makes the same code work at both levels
- What does **not** move up is the block scoreboard: the system one compares the SoC's input against its output, and the block one goes on measuring its bus

Appendix · From the VTALU to a real bus

Where to start

- **Look for a VIP before writing an agent.** Almost nobody writes an AXI agent from scratch: the vendors sell them and there are free ones. Writing your own is a project of months and it is not the one you were asked for
- **The protocol before the methodology.** 90 % of the bugs of a new agent are about a badly read protocol, not about UVM. The spec of the bus first
- **The first thing that gets written is the monitor**, not the driver. If you cannot *see* the bus you cannot verify anything — not even somebody else's stimulus
- And there the **passive agent** is your first real deliverable: watch before driving
- **SVA in parallel.** A bus has rules that get checked where they happen, not in the scoreboard, and on a bus with overlapping transactions that is the difference between half an hour and two days. You already have the tool: **the assertions**, with the advantage that the properties travel inside the interface of the VIP

Appendix · The debug toolbox

The tools we saw, all together

Tool	What for	Where it came up
`VLT_TRACE=1` + GTKWave	see the real signals	The VTALU spec
`+UVM_VERBOSITY=UVM_HIGH`	see the monitors' messages	Reporting
`+UVM_CONFIG_DB_TRACE`	who put what in the `config_db`, and who read it	Agents
`+TOPOLOGY` → `print_topology()`	the tree UVM **actually** built	Agents
`+UVM_OBJECTION_TRACE`	who raised and who dropped the objection	Tests
`+UVM_TIMEOUT=N,NO`	a ceiling for the hang, instead of waiting	Tests
`+UVM_MAX_QUIT_COUNT=N`	kill the run at the Nth error	Reporting

- The bottom six are **plusargs**: no code gets touched and nothing gets recompiled. On a testbench with UVM that is minutes of difference per attempt

Appendix · The debug toolbox

Which one to use, according to the symptom

The symptom	The first suspect	What to look at it with
It ends at **t=0** and says PASS	nobody raised the objection	`+UVM_OBJECTION_TRACE`
It **never ends**	an `item_done()` that was not called, or an objection that does not get dropped	`+UVM_TIMEOUT=5000000,NO` and then the trace
The scoreboard **screams on every one**	the monitor samples badly — the DUT hardly ever is the problem	`+UVM_VERBOSITY=UVM_HIGH`
The `config_db` **does not find**	the scope of the `set()`, not the `get()`	`+UVM_CONFIG_DB_TRACE`
The coverage gives **0 %**	the `new()` or the `sample()` of the covergroup is missing	read the `.dat` with `verilator_coverage`

The symptom	The first suspect	What to look at it with
The tree **is not the one you drew**	a `create()` without the factory, or a late override	`+TOPOLOGY`

- The golden rule is in the second column: **suspect number one is never the DUT.** It is the testbench watching it

Appendix · The 21 silent traps

It compiles, it runs, and it lies

The symptom	The cause	How to head it off	Section
The coverage gives **0 %**	the `new()` or the `sample()` of the covergroup is missing	one `sample()` per transaction, in the subscriber	Functional coverage
The daughter runs the method of **the mother**	without `virtual`, SV resolves by the **type of the variable**, at compile time	`virtual` on every method somebody is going to extend	Polymorphism
The component reads **somebody else's config**	`get(null, "*", …)` gives back the first thing that matches	`get(this, "", …)`, and `+UVM_CONFIG_DB_TRACE`	Tests
The test passes with	the `set_type_override()`	the override first, always. `+TOPOLOGY` gives it away	The env

The symptom	The cause	How to head it off	Section
the wrong stimulus	arrived **after** the `create()` of the env		
The register model goes **green** and the field was never tested	the access starts with `W0`: `bit_bash` skips it and `do_check` drops it from the mask	if the field is readable, `WC` or `W1C`. `W0*` only if it really is not	RAL · d8

- None of these five gives a warning. None leaves the regression in red. All of them get discovered weeks later, or do not get discovered

Appendix · The 21 silent traps

The one publishing and the one listening

The symptom	The cause	How to head it off	Section
The subscriber counts **0** and the monitor prints 1000	the `connect()` in the `connect_phase` is missing	cross-check it: against what the one that already worked says	Analysis ports · d4
The monitor **never publishes**, and the scoreboard screams	`bfm.command_monitor_h = this` is missing in the `build_phase`	the `uvm_fatal` shows up in the class that is **not** to blame	Analysis ports
The copy **loses the fields of the mother**	a `do_copy()` that does not call `super.do_copy(rhs)`	every `do_copy()` calls the one above first	Hierarchies
Two different transactions **compare equal**	`super.do_compare()` called on a loose line and thrown away	chained with &&, never loose	Transactions

- The four have the same shape: **the one that fails is not the one that got it wrong.** That is why the first reflex —opening the file that screamed— is the wrong reflex

Appendix · The 21 silent traps

Stimulus, scope and objections

The symptom	The cause	How to head it off	Section
The edge bins **never get filled**	`dist` with `:=`: the weight goes to **each** value of the range	measure the histogram **once**, with `:/`	Constrained random
The `randomize()` **does not run** with the asserts switched off	`assert(x.randomize())` — `assert` is a simulation directive	`if (!x.randomize()) uvm_fatal(…)`	Constrained random
Under the override, **some** transactions are of the old type	a `new()` where a `type_id::create()` belonged	what goes through the factory is what gets created with `create()`	Transactions

The symptom	The cause	How to head it off	Section
Both agents start up the same	two `set()` with scope `"*"`: the second overwrites the first	the scope is the **path** of the one that reads, with `*` at the end	Agents · d6
The agent starts up **active** even though you set `is_active` in the `config_db`	the missing `super.build_phase()`: the one that reads it is `uvm_agent`	either a config object, or `super` — never half of each	Agents
The simulation **never ends**	an `item_done()` that was not called	`+UVM_TIMEOUT=5000000,NO` first, the trace after	Agents
It ends at **t=0** and says PASS	nobody raised the objection around the sequence	`+UVM_OBJECTION_TRACE`	Sequences

Appendix · The 21 silent traps

The four from the assertions

The symptom	The cause	How to head it off	Section
The log says **PASS** and the properties did not run	`--assert` is missing at compile time	the `cover property` at 0 is the only one that warns you	Assertions
The property **always passes**	the antecedent never occurs, or the implication is the one that is wrong	one `cover property` per `assert property`	Assertions
145 false positives and the DUT is healthy	you sample it with the edge it is written on	stimulus on `negedge`, response of the DUT on `posedge`	Assertions · d7
False positives **at the start** of every test	the `disable iff (!reset_n)` is missing	`default disable iff`, once, at the very top	Assertions

- The four are of the same family as the ones above, with one aggravating factor: a broken assertion **looks exactly the same** as one that works. There is no output to look at

Appendix · The 21 silent traps

And the one the clocking block brings

The symptom	The cause	How to head it off	Section
The scoreboard fails **once every twenty** and the waveform looks fine	half the signals get read through `cb.sig` and the other half through `sig`	if it went into the clocking block, **the whole** protocol reads it through there	Interfaces and BFM

- It is the only one on the list that **a tool adds**: without a clocking block it does not exist. Badly used it is worse than not using it
- Two names for the same wire, and they differ by one cycle: `code/u2/clocking/mezcla.sv` prints `3` and `4` at the same instant
- The variant of the same mistake: waiting for `@(posedge clk)` and then reading `cb.sig`. If you adopt the clocking block, you adopt **its event too**

Glossary · the words of the trade

What it is called on the job, and not in the textbook

What a textbook says	What gets said at work
coverage of the functionality / of the code	*functional* / *code coverage*
the test plan	*verification plan*, *coverage plan*
reaching full coverage	*coverage closure*
edge cases	*corner cases*
hand-written / random stimulus	*directed* / *constrained-random stimulus*
signal plots	*waveforms*
the random number seed	*seed*
the nightly test run	*regression*
the first chips back · redoing the masks	*first silicon* · *respin*

What a textbook says	What gets said at work
a check written for the simulator	*assertion* — and `assert property` never gets translated
reading a signal at a given moment	*sampling* — and the edge it happens on, the *sampling edge*
test bed, test harness	**testbench**, always — nobody says anything else

- The concepts are the ones in the course; these are the exact words that the LRM, the *User Guide* and every answer you are going to find use. This table is the bridge

Glossary · The pieces

One line each, to have them together

Word	What it is
handle	the name of an object — it is **not** the object
factory	the dictionary of types: you ask for the base class and you get the derived one
override	changing the type the factory hands out, without touching whoever asks for it
transaction	a piece of data that travels through the testbench: a `uvm_sequence_item`
agent	everything that knows how to speak one interface, in a box
sequence / *sequencer*	the stimulus (object) and the arbiter that hands it out (component)
virtual sequence	the one that sends no items of its own: it coordinates **several** sequencers
driver	turns a transaction into signals

Word	What it is
monitor	watches the bus and publishes transactions
scoreboard	compares what came out against what should have come out
subscriber	anybody listening to an analysis port
objection	the *"I still have work to do"* that keeps the phase alive
assertion	a rule of the protocol written so the simulator checks it
property	the rule itself: clock, antecedent, implication, consequent
antecedent / *consequent*	the *"if"* and the *"then"* of a property
cover property	counts **how many times a property actually held**. The check of the check

Glossary · Three words that mislead

- `virtual` — in SystemVerilog it is **three unrelated things**: a *virtual method* (resolved by the object), a *virtual class* (abstract, it does not get instantiated) and a *virtual interface* (a handle to an interface). That they share the word is an accident of the language
- `assert` — in SystemVerilog it is **two things**: the *immediate* one, a statement that runs where the thread goes past, and the *concurrent* one (`assert property`), a declaration with a clock that gets evaluated on its own the whole simulation — the assertions. The `assert(randomize())` that is seen everywhere is the first one, and it is a poor way of checking a return value: that is why the course uses `` `uvm_fatal ``
- ***coverage*** — without a surname it means nothing. If somebody tells you "we are at 90 %", the question is always *code or functional?*
- And one of scale: a **test** in UVM is a **class**, not a file nor a run. The run is a *run*; the set of runs, a *regression*

References

- **IEEE 1800-2017** — SystemVerilog Language Reference Manual. The final arbiter of every discussion about syntax.
- **IEEE 1800.2** / **Accellera UVM** — the methodology, and the *UVM User Guide*. You have the code of `uvm-core` on disk: `make uvm`.
- **Verification Academy** — Siemens EDA · verificationacademy.com
- **Salemi, Ray.** *The UVM Primer* — Boston Light Press, 2013. The book I learned UVM with, and where several of the examples in `code/` came from.
- **2024 Siemens EDA / Wilson Research Group Functional Verification Study**. Source of the **data** of Trends. The charts are our own: they are generated with `make figs` from `res/trends/data.json`.
- **On RAL**: the *Using the Register Layer Classes* chapter of the **UVM 1.2 User's Guide** from Accellera is the long explanation, and the table of the twenty-five field accesses —`RW`, `WOC`, `W1C`...— is in the **IEEE 1800.2**, under `uvm_reg_field`. What the unit does not use lives in the same place: `add_hdl_path` for the backdoor, and **IP-XACT** (IEEE 1685) or **SystemRDL** (Accellera) to generate the model instead of writing it.
- **On clocking blocks**, which is the liveliest discussion of the course: the threads by **Dave Rich** on Verification Academy (2014, 2022 and 2024), his paper *The Missing Link: The Testbench to DUT Connection*, and the **DV Coding Style Guide from lowRISC/OpenTitan**, which prescribes them as a rule. The summary, with

If you want to keep reading



THE BOOK I LEARNED WITH

The UVM Primer

Ray Salemi · Boston Light Press, 2013 · ISBN 978-0974164939

This course is not that book: it has another structure, another DUT, another simulator and four units of material the book does not cover. What it does share is credited in `NOTICE`, and the complete list of differences is in `docs/en-que-se-diferencia.md` (in Spanish).

Credits

- Part of the examples in `code/` derives from those of the **UVM Primer**, published by its author under **Apache-2.0**. Here they stay under Apache-2.0, with the `NOTICE` the licence asks for and the list of changes.
- The course — `slides/` and `docs/` — is **CC BY 4.0**. The tools in `tools/`, **MIT**.
- The trend charts are our own, made from the published percentages of the Wilson Research Group / Siemens EDA study.
- **reveal.js** and **forkit.js** by Hakim El Hattab (MIT); the typefaces Chakra Petch and IBM Plex under SIL OFL 1.1.

This is as far as we got

What you take with you

- You know **why** verification exists as a discipline, and what the numbers of the industry say
- You wrote a conventional testbench, you measure it with **functional coverage**, and you know how to read the number that comes out
- You can read and write **OOP in SystemVerilog**: classes, inheritance, polymorphism, parameterized classes, the factory
- You built a **complete UVM testbench** —test, env, components, monitors, driver, scoreboard, transactions, agents and sequences— piece by piece, understanding what replaces what
- You wrote **assertions**: the other half of the plan, the one no scoreboard can close because it is not about the result but about the protocol
- And you handed it in: the **capstone** is a DUT you had not seen before, its spec, and the whole testbench from a blank sheet — which is literally what gets asked for at work
- And all of that runs on your machine, with **free tools**, without asking anybody for a licence

What was left out

and where to go on

Topic	Why you are going to run into it
`uvm_event`, `uvm_barrier`	The synchronization that is left when the objection is not enough
`uvm_heartbeat`	The one that warns you a component stopped showing signs of life in a regression of hours
`uvm_sequence_library`	A bag of sequences the sequencer picks from at random: the stress test without writing it
`uvm_pool`, `uvm_queue`	The collections of UVM. An associative array is enough until it has to be shared
TLM2, phase jumps	The limit of the course, and it is explicit
UPF, gate-level, PSS	Three worlds apart: low power, the netlist with delays, and generating the tests from a model

- The three places to go on: the **LRM IEEE 1800-2017**, the *UVM User Guide* from **Accellera**, and **Verification Academy**
- And one almost nobody uses: the **code of** `uvm-core`. It is on your disk, in `code/.uvm/src/`, and by this point you can read it
- And if you get stuck: **Discussions**, with one category per day. An example that does not run or an explanation that does not get understood are **issues of the course**, not problems of yours
- And if you want to go on right here: **day 8** is optional and starts after the self-assessment and *On Monday* — RAL, the reference model in C, and a second capstone

Self-assessment

Fifteen things you should be able to do by now · 1 of 3

Can I…?	Where it is
1 · Explain why **47 %** of a verification engineer's time goes on debug, and what this course does about it	Trends
2 · Write a five-column **verification plan** for a DUT I have not seen before	The verification plan
3 · Say why **100 % code coverage** does not mean the DUT is verified	Functional coverage
4 · Say what to look at **first** when a `covergroup` reports 0 %	Functional coverage
5 · Explain which race a `clocking block` avoids — and why they are **optional** anyway	Interfaces and BFM

Self-assessment

Fifteen things you should be able to do by now · 2 of 3

Can I…?	Where it is
6 · Extend a class and change its behaviour **without copying it**, and say what `super` does	Classes and extensions
7 · Change the class a testbench uses **without touching the env**, with a `set_type_override`	The factory pattern
8 · Explain why a UVM simulation **ends at t=0** if nobody raises an objection	Tests
9 · Say why `build_phase` runs top down and `connect_phase` the other way round	Components and phases
10 · Make a scoreboard that fails say **something more** than "it failed"	Reporting

Self-assessment

Fifteen things you should be able to do by now · 3 of 3

Can I…?	Where it is
11 · Hang a new subscriber off an analysis port **without touching the one publishing**	A single place that watches the wire
12 · Write a constraint that reaches a corner case, and **measure** whether the `dist` does what it says	Constrained random
13 · Say what an agent with `is_active = UVM_PASSIVE` builds and what it does **not**	Agents
14 · Explain why a `uvm_sequence` **does not show up** in `print_topology()`	Sequences
15 · Write a property that sees a protocol violation, and say why `--assert` is not optional	Assertions (SVA)

- All fifteen? Then what is left is the **capstone**: an APB slave, its spec, and the whole testbench from a blank sheet

On Monday

What to do with this once the screen goes off

- **First thing: a testbench of your own, running.** The capstone is the minimum. If it is already handed in, the next one is a DUT of yours — the one at work, the one from your thesis, whichever. A finished course proves nothing; a testbench does
- **Keep two pages and throw away the rest:** the one-page cheat sheet and *The 21 silent traps*. They are the two you open with the simulator running
- **If you get stuck, ask here:** Discussions has one category per day. An example that does not run is a **course issue**
- **And if something confused you, fix it.** It is CC BY 4.0 and the whole repo is there: the slides, the examples, the graders and the linters. A PR that clears up a slide is worth as much as one that adds a section

Thanks for getting this far.

Yesterday we left...

Where the testbench stands

- The capstone handed in: an APB slave, its spec, and a whole testbench written from a blank sheet — monitor, driver, scoreboard, coverage and properties
- The scoreboard of that testbench **predicts by hand** the value of every register, and the coverage of the bus was written row by row
- And that is done the same way at work... until the DUT has sixty registers. That is when both things get modelled, and that is the whole of day 8

Agenda

Day 8 · ≈ 4 h · optional · what comes after the capstone

- RAL, unit 9 — the register model over the same APB
- The reference model in C, through DPI
- The second capstone: a FIFO with backpressure

By the end of the day you can:

- **Read and write** a register by name, without writing a single `PADDR`, and say what the adapter and the predictor do
- **Plug in** a reference model written in C, and prove it with a mutation
- **Write** a scoreboard for a DUT where a table is not enough

RAL

Unit 9 · optional · the register stops being an address

- Every chip has configuration registers, and a small SoC has **two hundred**
- Without RAL, the testbench writes them like this: `addr = 8'h04`, `wdata = ...`, and the scoreboard keeps a `logic [31:0]` for each one
- The day the architect moves `SCRATCH` from `0x04` to `0x24`, you have to look for `8'h04` all over the testbench — and `0x04` was also the value of something else
- **RAL** —*Register Abstraction Layer*— is the layer that puts a name on that: `model.SCRATCH.write(status, 'h1000_0000)`
- It is optional and it goes at the end on purpose: **it is only understood once you have written the testbench it replaces**

RAL

Three pieces, and only one knows what bus is underneath

Piece	What it is	How many times it gets written
`uvm_reg_block`	the map: registers, fields, addresses	one per **DUT**
`uvm_reg_adapter`	translates access ↔ bus transaction	one per **protocol**
`uvm_reg_predictor`	what the monitor saw, said in registers	none: it comes done

- The model **does not know** there is an APB underneath. You change the adapter and the same model drives AHB
- The adapter is the file that comes with a VIP you bought. It is twenty lines
- And underneath everything, without a single change: the driver, the monitor and the agent of the capstone

RAL

The model: the table of the spec, in code

```
// 0x00 -- CTRL. EN is an ordinary bit. CLR is not: writing a 1 clears ACC and
// OVF, and the bit always reads back 0.
class ctrl_reg extends uvm_reg;
   `uvm_object_utils(ctrl_reg)

   rand uvm_reg_field EN;
   rand uvm_reg_field CLR;

   function new(string name = "ctrl_reg");
      super.new(name, 32, UVM_NO_COVERAGE);
   endfunction : new

   // configure(parent, size, lsb_pos, access, volatile, reset, has_reset,
   //           is_rand, individually_accessible)
   virtual function void build();
      EN = uvm_reg_field::type_id::create("EN");
      EN.configure(this, 1, 0, "RW", 0, 'h0, 1, 1, 0);

      // "WC" is the name UVM has for exactly this: the write clears the field,
      // the mirror drops to zero, and the read returns that zero. Modelling it as
      // "RW" is the classic mistake, and it is what +MAL does. "WOC" also passes,
      // for the wrong reason: any WO* access takes the field out of bit_bash and
      // out of do_check, so nobody looks at it. See the RAL unit slides.
      CLR = uvm_reg_field::type_id::create("CLR");
      CLR.configure(this, 1, 1, $test$plusargs("MAL") ? "RW" : "WC", 0, 'h0, 1, 1, 0);
   endfunction : build

endclass : ctrl_reg
```

- A field gets declared with `configure()`, and the argument that matters is the **access string**: `"RW"`, `"RO"`, `"WC"`…
- That string **is the specification**. It is not documentation: it is what UVM uses to predict, and to generate the tests of *The tests you did not write*

RAL

The adapter: the only thing that knows there is an APB

```
// Downstream: the model asks for a register access, this builds the bus item
// the sequencer will hand to the driver.
virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
   apb_transaction t = apb_transaction::type_id::create("t");
   t.write = (rw.kind == UVM_WRITE);
   t.addr  = rw.addr[7:0];
   t.wdata = rw.data[31:0];
   return t;
endfunction : reg2bus

// Upstream: a bus item comes back -- from the driver's response, or from the
// monitor when prediction is explicit -- and this fills the generic operation
// the model understands.
//
// PSLVERR is what makes status meaningful: an access to an unmapped address
// comes back UVM_NOT_OK, and the write()/read() call reports it.
virtual function void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
   apb_transaction t;
   if (!$cast(t, bus_item)) `uvm_fatal("REG ADAPTER", "Not an apb_transaction")
   rw.kind   = t.write ? UVM_WRITE : UVM_READ;
   rw.addr   = t.addr;
   rw.data   = t.write ? t.wdata : t.rdata;
   rw.status = t.slverr ? UVM_NOT_OK : UVM_IS_OK;
endfunction : bus2reg
```

- `reg2bus` goes down: the model asks for a generic access, this builds the transaction of the bus
- `bus2reg` goes up: the transaction comes back —from the driver, or **from the monitor**— and this counts it in terms of an access
- `PSLVERR` turns into `UVM_NOT_OK`, and there the `status` of every `write()` stops being decorative

RAL

Hooking it up: three lines and a `connect_phase`

```
function void connect_phase(uvm_phase phase);
   // The three lines that turn a model into a testbench component: which
   // sequencer carries the accesses, which adapter translates them, and where
   // the truth about the DUT comes from.
   model.default_map.set_sequencer(agent_h.sequencer_h, adapter);
   model.default_map.set_auto_predict(0);

   predictor.map     = model.default_map;
   predictor.adapter = adapter;
   agent_h.ap.connect(predictor.bus_in);
endfunction : connect_phase
```

- `set_sequencer` is what makes `model.CTRL.write()` end up in the driver: the model sends items through **the sequencer of the agent that already existed**
- `set_auto_predict(0)` switches off the automatic prediction, and the `uvm_reg_predictor` replaces it by listening to the **analysis port of the monitor**
- And the `model.build()` of the `build_phase`: a `uvm_reg_block` is a `uvm_object`, not a component. **Nobody calls it for you**

RAL

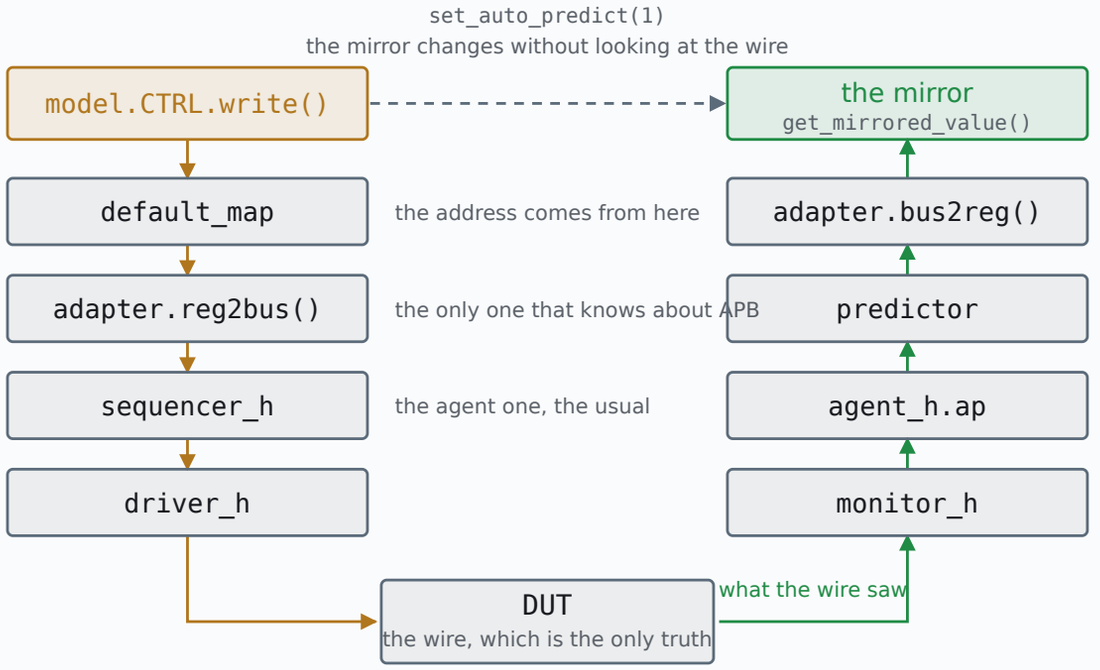
Prediction: who tells the model what happened

	Who predicts	What happens if the DUT does something else
automatic	the `write()` call itself	the model **does not find out**: it believes what it meant to send
explicit	the `uvm_reg_predictor`, from the monitor	the model believes **what happened on the wire**

- The automatic one is one line fewer and it is the one that comes up in the tutorials
- The explicit one is the one to use, and for the same reason the scoreboard of the capstone looks at the monitor and not at the driver
- And it is free: the monitor and its analysis port **were already there**

RAL

The whole path, from the call to the mirror



A single piece knows there is an APB below, and it is written twice: `reg2bus` goes down, `bus2reg` goe
With explicit prediction the mirror believes what happened on the wire, not what the model meant to

- The way down is written; the way up —monitor, analysis port and `bus2reg` — **was there**
- The `uvm_reg_predictor` is not written: instantiate it and plug in the map
- The dashed one is `set_auto_predict(1)`: the shortcut not to take

RAL

The stimulus, without a single PADDR

```
// --- the frontdoor -------------------------------------------------------
// write() puts the value on the wire through the sequencer. There is no
// second argument for the address: the address is in the map.
model.CTRL.write(status, 'h1);              // EN = 1
model.CTRL.read(status, data);
`uvm_info("RAL", $sformatf("CTRL read back 0x%08h (status %s)", data, status.name()),
          UVM_MEDIUM)

// --- prediction ----------------------------------------------------------
// Nothing above updated the mirror by hand. The monitor saw the two
// transfers, the predictor turned them into predictions, and the model
// knows -- with set_auto_predict(0), which is the honest setting: what the
// model believes comes from the BUS, not from what the test meant to send.
if (model.CTRL.EN.get_mirrored_value() != 1)
   `uvm_error("RAL", "the predictor did not update CTRL.EN")
`uvm_info("RAL", $sformatf("mirrored CTRL.EN = %0d, and nobody assigned it",
          model.CTRL.EN.get_mirrored_value()), UVM_MEDIUM)
```

- There is no address in any call: the address is **in the map**
- Nobody assigned the mirror. `get_mirrored_value()` gives back 1 because the monitor saw the transfer and the predictor counted it
- Underneath, the driver of the capstone is doing SETUP, ACCESS and the *wait state* of the read, the same as yesterday

```
$ bash run.sh
```

·

```
$ code/u9/ral/
```

RAL

The tests you did not write

```
rst_seq = uvm_reg_hw_reset_seq::type_id::create("rst_seq");
rst_seq.model = model;
rst_seq.start(null);

bash_seq = uvm_reg_bit_bash_seq::type_id::create("bash_seq");
bash_seq.model = model;
bash_seq.start(null);
```

- Both come from `uvm-core`. **They know nothing about this DUT**: they read the model and generate the stimulus and the check out of it
- `uvm_reg_hw_reset_seq` reads everything after the reset and compares it with the reset value declared
- `uvm_reg_bit_bash_seq` writes and reads **every writable bit of every register**, one by one, in both directions

RAL

And that is why the model is the spec

```
CTRL.CLR as "WC"   0 UVM_ERROR, and bit_bash did write bit 1
CTRL.CLR as "RW"   2 UVM_ERROR, from uvm_reg_bit_bash_seq
> Writing a 1 in bit #1 of register "model.CTRL" ...
  yielded 'h0000000000000000 instead of 'h0000000000000002
```

- One word changed in a string, and a sequence nobody wrote catches it with the number of the bit
- It is **trap 2** of the small print of the capstone — `CLR` never reads as 1— and there a whole scoreboard was needed to see it
- The register model is the only document of a project that is the spec **and** the code at the same time

`$ bash run.sh` runs both versions and compares them

RAL

What RAL does not do, and it has to be said

```
// The honest line of the whole unit. ACC and STATUS have an address, but
// they are not registers: their value is produced by writes to OTHER
// addresses. A register model predicts "what I wrote is what I will read",
// and for these two that sentence is false, so mirror(UVM_CHECK) is turned
// off on them. Reads still update the mirror -- a read is a prediction --
// and the accumulator itself stays where it was in the capstone: in the
// scoreboard, which is the thing that does know how to add.
ACC.VALUE.set_compare(UVM_NO_CHECK);
STATUS.EN.set_compare(UVM_NO_CHECK);
STATUS.OVF.set_compare(UVM_NO_CHECK);
```

- `ACC` and `STATUS` have an address, but **they are not registers**: their value gets produced by a write to *another* address
- A model predicts *"what I wrote is what I am going to read"*, and for these two that sentence is false. `set_compare(UVM_NO_CHECK)` says so in the model
- The accumulator goes on being checked where it already was: **in the scoreboard**, which is the one that knows how to add
- The read updates the mirror all the same — a read *is* a prediction

RAL

Summary of the unit

- The **model** is the map of the spec in code, and the access string of each field is the part that verifies
- The **adapter** is the only thing that knows about the bus: twenty lines, once per protocol
- The prediction goes **through the monitor**, not through the call — the same rule as the scoreboard
- It gives you `hw_reset_seq` and `bit_bash_seq` for free, and it charges you knowing **what is not a register**
- And all of this runs on **Verilator 5.052**, which until today nobody had shown

Exercise · Day 8 · 1 of 2

Modelling the register map

```
$ cd code/ejercicios/d8-ral && cat README.md
```

- It is **literally** the first thing you get asked for in a project with registers: they give you the table of the spec and you give back the model
- The DUT and the testbench are the ones from the capstone. The adapter, the predictor and the tests come from the unit. **The only thing you write is `apb_reg_block.svh`**
- Four registers, six fields, forty lines
- The checker goes in stages: the **map** —printed and compared with the table, without simulating anything—, the **accesses** —the two sequences of the library— and **what the model cannot predict**

The reference model in C

The scoreboard that rewrites the spec

- The scoreboard of the course predicts **in SystemVerilog**: a `case` with the arithmetic of the VTALU. For five eight-bit operations it is perfect
- For a DSP, a codec or a crypto engine, that arithmetic **already exists, and in C**: the algorithm team wrote it, and it is against that model that the spec was signed off
- Rewriting it in SystemVerilog is maintaining **two** models and spending the project debugging the difference between them. That is not verifying: it is having two specs
- The way out is not to rewrite it. Verilator compiles to C++, so the model gets **linked inside the simulation**: no socket, no separate process, no IPC

The reference model in C

import "DPI-C": one declaration on each side

```
   // The reference model lives in vtalu_golden.c and is called from here. This
   // is all that is needed on the SystemVerilog side: one declaration per
   // function, with the LRM types that cross the boundary.
   //
   //   input int  -> int          output int -> int *
   //   function   -> returns      task       -> may consume time (not here)
   //
   // The name after "DPI-C" is the symbol the linker looks for, so it has to
   // match the one in the .c letter by letter.
   import "DPI-C" function int  vtalu_golden(input int op, input int a,
                                             input int b, output int ovf);
   import "DPI-C" function void vtalu_golden_bug(input int on);
```

```
/* op, a and b arrive as SystemVerilog `int` (32-bit signed). The result goes
 * back as the return value and the borrow through a pointer, which is how an
 * `output int` argument crosses the boundary. */
int vtalu_golden(int op, int a, int b, int *ovf) {
   int r;

   a &= 0xff;
   b &= 0xff;
   *ovf = 0;

   switch (op) {
      case ADD_OP: r = a + b; break;
      /* 8-bit operands into a 16-bit result: a - b wraps, and the borrow is
       * the second half of the answer the scoreboard has to check. */
      case SUB_OP: r = a - b; *ovf = (a < b); break;
      case AND_OP: r = a & b; break;
      case XOR_OP: r = a ^ b; break;
      case MUL_OP: r = mutar ? (a * b) & 0xff : a * b; break;
      default:     r = 0; break;   /* no_op and rst_op do not answer */
   }
```

```
    return r & 0xffff;
}
```

- The name after `"DPI-C"` is **the symbol the linker looks for**: it has to match letter by letter with the one in the `.c`
- An `input int` travels as an `int`; an `output int` arrives as an `int *`. A `function` cannot consume time; a DPI `task` can, but only declared `context` and calling back into an `export "DPI-C" task`
- The `.c` goes on the `verilator` line as one more source. **There is no DPI flag**

The reference model in C

The scoreboard: the prediction gets asked for, not written

```
// The ONLY difference with the sequences scoreboard, and it is the whole
// section: there is no case with the arithmetic of the spec here. The
// prediction is asked of the C reference model, the one that signed it.
//
// What does not change: the monitor, the analysis port, the FIFO, the
// do_compare and the uvm_error. The golden model replaces SIX LINES -- the
// ones that predict -- and no other piece of the testbench finds out.
function result_transaction predict_result(command_transaction cmd);
   result_transaction predicted;
   int                ovf;

   predicted = result_transaction::type_id::create("predicted");

   predicted.result = vtalu_golden(int'(cmd.op), int'(cmd.A), int'(cmd.B), ovf);
   predicted.ovf    = (ovf != 0);

   return predicted;

endfunction : predict_result
```

- It is **the only** difference from the scoreboard of the sequences. The monitor, the analysis port, the FIFO, the `do_compare` and the `uvm_error` do not change
- The golden model replaces the six lines of predicting, and no other piece of the testbench finds out there is C on the other side
- And the traffic goes both ways: the result comes back from C, and `vtalu_golden_bug(1)` goes from SystemVerilog to **change the state** of the model

The reference model in C

What you gain, what you pay

You gain	You pay
One single model, the one that signed off the spec	the opcodes get synchronized **by hand**: one `enum` on each side
The model gets tested with the tools of C	an `undefined reference` does not say which of the two declarations is wrong
Arithmetic that in SystemVerilog is painful — floating point, matrices, crypto—	the C model runs in **zero time**: it cannot wait for an edge
Performance: it is a function call, not an IPC	a `segfault` in the model takes the whole simulation down with it

- The rule for deciding: if the prediction **already exists in C**, DPI. If you write it yourself from scratch and it is five lines, a `case` in SystemVerilog and that is it
- It runs: `make u8/dpi` — two tests against the healthy model, and one against the mutated one

Review · Day 8

1 of 8 · The access string

`CLR` clears itself when you write a 1 to it. You declare it `"WOC"`, and `bit_bash` and `mirror(UVM_CHECK)` give zero errors. What happened?

a. Nothing: `WOC` is the access that describes a field that clears on a write

b. `WOC` is not in the LRM, so UVM treats it as a plain `RW`

c. Both skip `WO*` accesses: that bit never got tested ✓

d. The predictor leaves the mirror at `x`, and a comparison against `x` always passes

Green is not checked — `uvm_reg_bit_bash_seq.svh:129-133` skips every field whose access starts with `WO` (*"you are not supposed to read them"*), and `do_check` takes it out of the comparison mask (`uvm_reg.svh:2782-2788`). The only trace is in the timing: bashing `CTRL` takes four transfers less. The right access is `WC` (or `W1C`), and with it the bit does get bashed. A `WO*` access is the cheapest way there is of switching a check off without noticing.

Review · Day 8

2 of 8 · Automatic or explicit prediction

What changes between `set_auto_predict(1)` and hooking a `uvm_reg_predictor` to the monitor?

- **a** With auto-predict the model believes what it **meant** to send, not what happened ✓
- **b** Nothing: the predictor is the internal implementation of auto-predict
- **c** The predictor is faster: it does not build the bus transaction
- **d** Auto-predict only works frontdoor, and the predictor works backdoor too

The one who checks cannot believe the one who drives — with `set_auto_predict(1)`, `model.CTRL.write()` updates the mirror at the moment of the call, before the bus has done anything: if the driver sends the transfer wrong, the model stays convinced. With the predictor, the mirror only changes once the monitor has seen the wire. It is the same drawing as the capstone scoreboard, with a library piece instead of a hand-written class. And it is free: the monitor and its analysis port **were already there**.

Review · Day 8

3 of 8 · volatile

`STATUS` changes on its own, without anybody writing it. What are you saying when you declare it `volatile` in `configure()`?

- **a** That the mirror is not evidence: the value can change without going through the bus ✓
- **b** That UVM is going to re-read it from the DUT before every comparison, to be safe
- **c** That the field drops out of the map and stops having an address
- **d** That it has to be read backdoor, because the frontdoor does not get there in time

The mirror stops being evidence — a model predicts *"what I wrote is what I am going to read"*, and for a `volatile` field that sentence is false. The two `UVM_WARNING GET_MIRRORED_VAL/VOL` of the example are the library saying exactly that, and they are in the output on purpose. What does have to be checked gets checked where it is known: in the **scoreboard**, and in the model goes `set_compare(UVM_NO_CHECK)` on that field so that RAL does not invent an error.

Review · Day 8

4 of 8 · mirror() against read()

`model.CTRL.read(status, data)` and `model.CTRL.mirror(status, UVM_CHECK)`. How do they differ?

a — `read` goes over the bus and `mirror` stays in the mirror, with no transfer at all

b — `mirror` writes the mirror into the DUT, to leave the two of them equal

c — `read` updates the mirror and `mirror` does not touch it, so as not to hide an error

d — Both read from the DUT; `mirror` also compares against what the model believed ✓

`mirror` **is a register scoreboard in one word** — both read from the DUT; the difference is that `mirror` compares against what the model believed **before** the read, and if it does not match it reports a `uvm_error` without anybody writing a check. And a read *is* a prediction: both update the mirror afterwards.

Review · Day 8

5 of 8 · Who knows about the bus

What does the `uvm_reg_block` know about the APB?

a. The base address and the width of `PADDR`, which reach it through the `uvm_reg_map`

b. Nothing: the only one that knows about the bus is the adapter ✓

c. Everything: that is why there is one register model per protocol

d. The SETUP and ACCESS timing, to predict the wait state of the read

The model does not know there is an APB underneath — swap the adapter and the same model drives an AHB. The adapter is twenty lines, once per protocol, and it is the file that comes with a VIP you bought. Watch out for one thing: `bus2reg` is called **by the predictor too**, with the item the monitor saw, so an adapter that depends on something the driver put there works in one direction and fails in the other.

Review · Day 8

6 of 8 · DPI and time

Can an `import "DPI-C" function` of the golden model wait for a clock edge?

a Yes, by putting a `#1` inside the `.c`

b No: a `function` runs in zero time ✓

c Yes, as long as the `.c` is compiled with Verilator's timing support

d Yes: the simulator suspends the C thread for the duration of the call

Zero time, like any `function` — to consume time you need `import "DPI-C" task`, and with a clarification almost every tutorial skips: C cannot block by itself. A DPI task consumes time **only** if it is declared `context` and from the C it calls back into an `export "DPI-C" task` of SystemVerilog, which is the one that waits for the edge. And at that point it is no longer a golden model, it is a bus model.

Review · Day 8

7 of 8 · undefined reference

The `.c` is written and it compiles, and the link fails with *undefined reference* to the golden model function. What do you look at first?

- **a** Whether the DPI flag is missing from the `verilator` command line
- **b** Whether the `.c` is in its own `-f` and not mixed in with the `.sv`
- **c** The name, and the `extern "C"`: the symbol may have come out mangled ✓
- **d** Whether the SystemVerilog `function` is declared `virtual` so the symbol gets exported

The compiler does not cross the two declarations: the one that joins them is the linker — hence an error about something that is written, right there, in plain sight. It is almost always one of two things: the name does not match letter for letter, or the file got compiled as C++ and the symbol came out mangled. That last one is literally the case here: Verilator hands the user's sources to the C++ compiler, so the `.c` needs its `extern "C"`. And there is no DPI flag: the `.c` goes on the `verilator` command line like any other source.

Review · Day 8

8 of 8 · The scoreboard that never shouted

The scoreboard with the golden model in C runs a thousand operations and reports not one. Is that enough?

a Yes: a thousand comparisons without a single difference is the definition of verified

b Yes, as long as the functional coverage closed at 100 % on top of that

c No, because both sides share the opcode `enum` and cancel each other out

d No: a scoreboard that has never seen an error is not tested ✓

The model has to be broken on purpose — `vtalu_golden_bug(1)` mutates the multiplication and `run.sh` demands that the scoreboard shout. If it does not shout, the testbench is comparing against itself and nobody was going to find out: it is the same mutation test the capstone grader does with `+BUG=1`, from the other side of the wire. And the duplicated `enum` is real but it is a different symptom: if **every** comparison fails at once, it is the mapping; if one in six fails, it is the DUT.

Exercise · Day 8 · 2 of 2

The reference model in C, and the two mutations that prove it

```
$ cd code/ejercicios/d8-dpi && bash run.sh
```

- The only file of the exercise is `vtalu_golden.c`. Not one line of the testbench gets touched
- Two operations are missing: the **subtraction with its borrow** —which goes out through a pointer— and the product with the mutation hook
- The checker runs three times: healthy DUT against your model, **mutated DUT** (`VTALU_BUG=1`), and your mutated model (`+GOLDEN_BUG`)

Capstone 2 · Day 8

The second final: a FIFO with backpressure

```
$ cd code/ejercicios/d8-fifo && cat spec.en.md
```

- For whoever has already handed in the APB. **The protocol is simpler** —there are no addresses, no wait states, no error response— and even so it is harder
- The APB one is a slave **with no useful memory**: an address and a value, and the scoreboard can be a table of four rows. A FIFO has **order** and **occupancy**, and both have to be carried
- The bug of `+BUG=1` is not a data bug: `almost_full` goes up **one place late**. The data still comes out right and in order, so a scoreboard that only compares `rd_data` passes green
- The small print: the flags describe the state **before** the edge, `rd_data` arrives **one cycle late**, writing with the FIFO full is **not an error**, and a simultaneous read **makes room** for the write