Constrained random on Verilator

This page answers: *how do I get the random stimulus I asked for on this simulator, and how do I know I got it?*

Lab 02′s generator chose each packet's DST and length itself, with calls to `$urandom_range` written into its loop. In Lab 03 the generator works on a `rand_packet` instead, a class that adds `rand` fields and constraints to Lab 02′s `packet`. It calls `randomize()`, which hands the constraints to a solver and lets the solver choose the values. That moves the decisions about what the stimulus looks like out of the generator's code and into one class, where a test can change them. It also means that what comes out depends on how the simulator chooses among the values your constraints allow, and Verilator 5.050 doesn't always choose the way the SystemVerilog standard says it should.

This page explains what the standard says `randomize()` does, where Verilator 5.050 does the same and where it doesn't, and how to check your own constraints by counting what comes out. The examples are in `guide-examples/constrained-random-on-verilator/`. They're small classes, most of them built on SimProc's ALU operations, and they build with the course's `uvm.mk` like every lab. You'll run the same kind of experiment on the router's packet in Lab 03, Part 3.

rand fields, constraints and randomize()

A class marks the fields it wants chosen with `rand`, and states the rules they must follow in named `constraint` blocks. Here is an ALU vector:

```
typedef enum bit [2:0] {ADD, SUB, OR, NAND, SHL, SHR} alu_op_e;

class alu_vec;
  rand alu_op_e  op;
  rand bit [7:0] A, B;
  constraint c_op    { op inside {[ADD:SHR]}; }
  constraint c_shift { (op inside {SHL, SHR}) -> B < 8'd4; }
endclass
```

SimProc's shifts use only `B[1:0]`, so `c_shift` keeps `B` below 4 whenever the operation is a shift, and leaves it free otherwise. A call to `v.randomize()` asks for values of `op`, `A` and `B` that satisfy every constraint block that is switched on, all at once. It returns 1 if it found such values and 0 if it didn't. A call can add constraints of its own with `with { ... }`, which apply to that call only, on top of the class's own constraints. A block can be switched off with `constraint_mode(0)`. The standard also says that every combination of values that satisfies the constraints is equally likely to be chosen.

`cr_item_test` randomizes one `alu_vec` a few times:

```
[ITEM] randomize() returned 1: SHR A=cc B=02
[ITEM] randomize() returned 1: SHR A=43 B=03
[ITEM] randomize() returned 1: SHL A=2f B=00
[ITEM] randomize() returned 1: SHL A=65 B=03
[ITEM] with { op == SUB; A == B; } returned 1: SUB A=f4 B=f4
[ITEM] with { op == SUB; A == B; } returned 1: SUB A=35 B=35
[ITEM] before: SUB A=35 B=35
[ITEM] with { op == SHL; B == 5; } returned 0: SUB A=e5 B=35
[ITEM] the same with c_shift off returned 1: SHL A=10 B=05
```

The call with `{ op == SHL; B == 5; }` can't be satisfied, because `c_shift` says a shift has `B` below 4. It returned 0, and the run printed an `UNSATCONSTR` warning naming both constraints. The standard says a failed call leaves the

variables as they were, but here `A` changed from `35` to `e5`. So after the failed call the object held the old `op` and `B` with a new `A`, which is a combination that no call asked for. With `c_shift` switched off, the same call succeeds.

That's why every `randomize()` in Labs 01 to 07 and in the guide examples checks what it returned, and most of them stop the run when it's 0, as in `if (!p.randomize()) `uvm_fatal("RAND", ...)`. It's also why the course's `make run` calls a run BROKEN when any call failed, even if the test itself passed. This run's verdict was:

```
 BROKEN: randomize() failed   0.02 s   log: build/runs/cr_item_test-s1/run.log
```

The solver

Verilator doesn't solve constraints itself. It writes them out in the SMT-LIB language for a separate solver program, which the environment variable `VERILATOR_SOLVER` names. The course image has `yices-smt2`. The course's `uvm.mk` sets `VERILATOR_SOLVER` to `yices-smt2 --incremental` in the environment of every run, and it also builds that command into the simulator as the default, which the simulator uses when you start it by hand without the variable. Before it builds or runs anything, `make` checks that the program that `VERILATOR_SOLVER` names is on your PATH. Asked to use `z3`, which the image doesn't have, it stops with:

```
 VERILATOR_SOLVER is "z3 --in", and z3 is not on PATH.
======================================================================
 Without a solver every constrained randomize() returns 0, nothing is
 randomized, and the run still passes. Are you in the course image?
```

As a last check, `dd105_base_test` randomizes a small probe class in its `build_phase` and stops the run if that fails. Here is the simulator run by hand, without `make`, with `VERILATOR_SOLVER` naming `z3`:

```
VlRProcess::open: execvp(z3): No such file or directory
UVM_INFO @ 0: reporter [RNTST] Running test cr_item_test...
%Warning: Subprocess command `z3 --in' failed: exit status 127
%Warning: Unable to communicate with SAT solver, please check its installation or specify a
different one in VERILATOR_SOLVER environment variable.
 ... Tried: $ z3 --in
UVM_FATAL labs/common/uvm/dd105_pkg.sv(51) @ 0: reporter [SOLVER] a constrained randomize()
failed, so this run cannot randomize anything. Verilator needs VERILATOR_SOLVER="yices-smt2
--incremental" in the environment. Run the test with make, which sets it, or export it before
running the simulator by hand.
```

To get random values out of the solver, Verilator's runtime (`verilated_random.cpp` in the image) picks a random target value for every bit of every `rand` variable, and asks the solver for a solution that matches all of those bits. If the targets break a constraint, it gives up one bit's target and asks again, and it keeps doing that until the remaining targets fit the constraints. A target that lands on an allowed value comes out as it is. A target that lands on a forbidden value comes out as an allowed value that differs from it in a bit or two, so the allowed values next to forbidden ones come out more often than the others. (A class with an array in it goes through a different method in the same file.)

`alu_op_e` has six names in three bits, so `op inside {[ADD:SHR]}` forbids two of its eight codes, `110` and `111`. With that constraint, 1,000 draws in `cr_enum_test` gave:

```
[ENUM] op inside {[ADD:SHR]}:             ADD  127  SUB  109  OR  148  NAND  124  SHL  234
SHR  258
```

The six counts add up to 1,000, so every draw obeyed the constraint and none came out as 6 or 7. The standard would give each operation about 167, but `SHL` (`100`) and `SHR` (`101`) got about twice the share of the others. You get those two codes from `110` and `111` by giving up the middle bit's target, so the draws that aimed at the forbidden codes landed there. A constraint decides which values can come out, and on this simulator you can't rely on it for how often each one does.

Shapes that come out differently here

Lab 03's experiments (`make traps`) and the examples here found five shapes of class and constraint for which Verilator 5.050 doesn't do what the standard says. Each section below gives a way around it.

A rand enum that no constraint names

The standard makes an unconstrained `rand` enum uniform over its names. On this simulator that holds for some enums and not others:

```
[ENUM] 3 names in 2 bits, no constraint:      317  347  336
[ENUM] 4 names in 2 bits, no constraint:      813   66   63   58
[ENUM] 6 names in 3 bits, no constraint:      160  167  179  163  146  185
[ENUM] 8 names in 3 bits, no constraint:      890   14   12   15   18   10   22   19
[ENUM] 4 names, v inside {[N4_A:N4_D]}:       245  237  247  271
```

Four names in two bits, and eight names in three, fill every code of their bits. Those two enums came out as their first name 813 and 890 times in 1,000, with no warning. Line 1 of Lab 03's experiments gives 809 for its four-name enum. The three-name and six-name enums came out even. So the course names every `rand` enum in a constraint, except the `kind` of Lab 03's starter packet, which you weight yourself in Part 3. For an enum whose names fill its bits, `inside` with every name is enough, as the last line above shows. For an enum whose names don't fill its bits, `inside` gives an uneven spread, as the previous section showed. A `dist` with one range and equal weights gave an even one, and that's how Lab 01's `alu_item` writes its `op`:

```
[ENUM] op dist {[ADD:SHR] := 1}:          ADD  161  SUB  166  OR  163  NAND  173  SHL  176
SHR  161
```

No lab writes a class that takes the enum's type as a parameter, but the same test tried one (`class free_param #(type T = e3_t); rand T v;`). In that class the three-name enum took the unused code 3 in 256 draws of 1,000:

```
[ENUM] 3 names, in a parameterized class:    235  247  262   and 256 values that aren't names
```

`randomize()` returned 1 every time, and the run printed no warning.

A dist written with enum names

A `dist` gives each value or range a weight, and under the standard it makes no difference whether you write the values as enum names or as numbers. Here are weights of 10, 20, 30 and 40 written both ways:

```
[DIST] weights 10/20/30/40 by name:    K_ZERO  229  K_ONES  256  K_SAME  267  K_ANY  248
[DIST] weights 10/20/30/40 by number:  K_ZERO   93  K_ONES  212  K_SAME  291  K_ANY  404
```

Written with the names, the weights had no effect, and each kind came out about a quarter of the time. Written as the numbers 0 to 3, the kinds came out close to 10, 20, 30 and 40 percent. In line 2 of Lab 03's experiments, weights of 25, 50 and 25 written with names went wrong in a different way, with the 50 on the wrong name. So wherever

a `dist` gives values different weights, the course writes the values as numbers, as in `kind dist {0 := w_one, 1 := w_mid, 2 := w_max}`.

A dist and another constraint on the same variable

To the standard, a `dist` is one more constraint, solved together with all the others. Verilator 5.050 behaves as if it first picks one entry of the `dist` list by weight, and only then solves the rest. If the other constraints rule out the entry it picked, the call fails:

```
[DIST] op dist {0 := 5, 1 := 1, ...}:   ADD  513  SUB   96  OR   92  NAND  102  SHL   94
SHR  103
[DIST] op dist {0 := 5, 1 := 1, ...}, then with { op == SUB; }: randomize() failed 921 times
in 1000
[DIST] the same with c_op.constraint_mode(0):             randomize() failed 0 times in 1000
[DIST] op dist {[ADD:SHR] := 1}, then with { op == SUB; }:  randomize() failed 0 times in 1000
```

The weights came out right. But asking for `SUB` with the `dist` switched on failed 921 times in 1,000, which is about the nine draws in ten that picked an entry other than `SUB`. Lab 01′s `dist` is a single range, which counts as one entry. Whatever it picks allows `SUB`, so asking for `SUB` failed 0 times in 1,000.

The same thing happens when the call constrains a different variable, one that depends on the variable with the `dist`. This class puts the `dist` on a selector, `kind`, and uses implications to choose `A` from `kind`, which is the pattern the labs use:

```
class vec_kind;
  rand kind_e    kind;
  rand bit [7:0] A, B;
  constraint c_kind { kind dist {0 := 10, 1 := 20, 2 := 30, 3 := 40}; }
  constraint c_A    { (kind == K_ZERO) -> A == 8'h00;
                      (kind == K_ONES) -> A == 8'hFF;
                      (kind == K_SAME) -> A == B; }
endclass
```

```
[DIST] kind dist, then with { A < 8'h10; }:                  randomize() failed 199 times
in 1000
[DIST]   the calls that worked: K_ZERO 114  K_ONES 0  K_SAME 305  K_ANY 382
[DIST] the same with c_kind.constraint_mode(0):                randomize() failed 0 times
in 1000
```

`A < 8'h10` rules out `K_ONES`, which has a fifth of the weight, and a fifth of the calls failed. A selector is still the better place for a `dist`, because a call then fails only when its constraint rules out a whole kind. In line 3 of Lab 03′s experiments, `len != 7` made 72 calls in 1,000 fail with the `dist` on `len` itself, and none with the `dist` on `kind` and `len` chosen from it. For a call whose constraint does rule out a kind, switch the `dist` off with `constraint_mode(0)`, or build the item without the solver. Lab 03′s `close_holes()` builds its fill packets with `rand_packet::make()`, because with the packet's weights on, `randomize() with { len == 3; }` failed 478 times in 1,000 (also line 3 of its experiments).

solve...before

`solve big before len` asks the solver to choose `big` first, which changes how often each value of `big` comes out without changing which values are legal. In Lab 03′s experiment the standard gives `big` about 500 times in 1,000

with it and about 125 without, and Verilator gave 33 and 41, as if the line weren't there. `cr_solve_test` tries a different shape, where `is_shift` chooses between the two shifts and the four other operations:

```
[SOLVE] solve is_shift before op:  is_shift in 665 of 1000;   ADD   51  SUB   77  OR   74
NAND  133  SHL  223  SHR  442
[SOLVE] without solve...before:    is_shift in 512 of 1000;   ADD   69  SUB  178  OR   58
NAND  183  SHL   64  SHR  448
```

For `is_shift`, the standard gives about 500 with `solve...before` and about 333 without, because two of the six operations are shifts. Verilator's runtime has a separate pass for `solve...before`, and in this shape it moved the count from 512 to 665, but neither experiment came out at the standard's numbers. So don't use `solve...before` to shape a distribution on this simulator. If you want `is_shift` half the time, ask for it with a `dist`, which gave 502 in 1,000:

```
[SOLVE] is_shift dist {0:=1, 1:=1}: is_shift in 502 of 1000;   ADD   56  SUB   53  OR  196
NAND  193  SHL   59  SHR  443
```

The operations within each half still aren't even (`SHR` 443 against `SHL` 59). As in "The solver" above, this simulator doesn't choose evenly among the values a constraint allows, although giving up one bit's target at a time doesn't by itself explain why `SHR` wins over `SHL` here. If the spread of the operations matters to your test, count it.

unique and sum() on arrays

Verilator accepts `unique {a}` on an array but doesn't enforce it. In line 6 of Lab 03′s experiments, four random 4-bit values repeated a value in 315 draws of 1,000, which is about the one draw in three that you'd get with no constraint at all. On four separate variables, as `unique {w, x, y, z}`, it works, and no value repeated in 1,000 draws. `lens.sum() == 8` on a dynamic array can't be solved at all and failed 20 times in 20, while the same constraint on a fixed-size array works. That's why Lab 03′s `fill_plan` has `lens[8]`, with a 0 meaning "no packet".

Seeds

Every run starts the random number generators from a seed, which `make` passes as `+verilator+seed+N` from its `SEED` variable (1 unless you set it). The same seed gives the same values. We ran `cr_item_test` twice with seed 1 and got the same nine lines both times, and seed 2 gave a different first line, `SHR A=2d B=02`. So you can rerun a failing test and watch the same failure happen again, and `make seeds` can run a test on seeds 1 to 10.

A seed reproduces a run only when the testbench is the same, because a different class asks the solver different questions. That's why Lab 03 says your own constraints may give different counts from ours with the same seed.

Counting what came out

The four-name enum that came out as its first name 813 times in 1,000 broke no constraint and printed no warning, and its run passed. In a testbench with a design in it, the scoreboard wouldn't have noticed either, because it checks the design's answers and not how often each input was sent. The tests on this page found it by counting the values in arrays of their own. In Lab 03 you count with a covergroup instead, and its first run shows about a third of the router's packets 1 beat long and a third 8 beats long (260 and 270 of 800), because the packet's length comes from a three-name enum, `kind`, that no constraint weighted. Functional coverage you can read explains how to read those counts.

Check yourself

1. A transaction has `rand bit [1:0] dst;` and `rand mode_e mode;`, where `mode_e` has four names in two bits, and no constraint mentions `mode`. What do you expect 1,000 draws to give, and what do you add?

   Answer: A1, in Answers at the end.

2. An ALU item has `constraint c_op { op dist {0 := 2, 1 := 1, 2 := 1, 3 := 1, 4 := 1, 5 := 1}; }`. A directed sequence calls `req.randomize() with { op == NAND; }`. What happens, and what are two ways to fix it?

   Answer: A2, in Answers at the end.

3. A run of your Lab 03 test passes, and the log has one `UNSATCONSTR` warning in it. Why does `make run` call it BROKEN rather than ok?

   Answer: A3, in Answers at the end.

Further reading

ChipVerify's SystemVerilog pages cover each construct here with longer examples. Their output comes from commercial simulators, so where Verilator 5.050 differs, trust the counts on this page and in Lab 03.

- SystemVerilog Randomization
- Constraint Blocks
- Inline Constraints. On Verilator 5.050, an inline constraint on a variable that has a `dist` fails whenever the `dist` picked an entry that the inline constraint rules out (see "A dist and another constraint on the same variable" above).
- Constraints. One of its `dist` examples names a variable `priority`, which is a SystemVerilog keyword, so that example doesn't compile. Another gives its weights to enum names, and on Verilator you should write those values as numbers.
- solve before explains correctly what the standard means, but Verilator 5.050 doesn't produce the distributions it shows.
- Soft Constraints. It says that the solver decides which of two conflicting soft constraints wins. The standard fixes the order: a later one beats an earlier one, a derived class beats its base, and an inline `with` beats the class.
- Disable Constraints, for `constraint_mode()`.

Next: Functional coverage you can read.

Answers

A1. Check yourself

Expect the first name about four times in five (813 here and 809 in Lab 03), with no warning. Add a constraint that names `mode`. Because its four names fill its two bits, `mode inside {...}` with all four names is enough, and that gave an even spread above. For an enum whose names don't fill their bits, use a `dist` with one range and equal weights, as Lab 01's `alu_item` does.

A2. Check yourself

About six calls in seven fail. The simulator picks an entry of the `dist` first, and only the entry for 3 (`NAND`), which has a seventh of the weight, satisfies `op == NAND`. Either switch the `dist` off for the call with `req.c_op.constraint_mode(0)`, or set `op` by hand without calling `randomize()`, as Lab 01's directed sequence does.

A3. Check yourself

A failed `randomize()` means that at least one item wasn't what the test asked for. On this simulator the failed call can also change some of the item's fields (`A` went from `35` to `e5` above), so the item may hold a mix of old and new values. A pass then says only that the router handled whatever was sent, and that isn't the stimulus the test describes.