Writing the verification report

This page answers: *you've run the regression and closed the holes your plan found. What do you hand over, so that someone else can check your claim without redoing your work?*

DD104's capstone ended with a bug report about one bug, built around a test that reproduces it and the command that runs that test. This course's capstone ends with a **verification report**, which claims that a whole design follows its specification in every situation that a written plan describes and that your environment checks. The report gives the evidence for that claim and says where the evidence stops. You'll write one for the router in Lab 07. In Part 4 you copy `labs/lab-07-capstone/REPORT-template.md` into your repository as `REPORT.md` and fill in its hunt table, and in Part 8 you fill in the rest.

A report like this is read by someone who has to rely on the design without having run your tests, such as whoever builds the design into a chip or reviews it before that happens. That reader can rely only on what they can check, so everything in the report is either a line that a tool printed, with the command that printed it beside it, or a statement with its limits written next to it.

A short report for the counter

The router's report runs to eight sections and a few pages, so it's easier to see how a report works on something small first. Regressions and seeds uses a 4-bit counter that is meant to stop at 15. With the planted bug `wrap` switched on, the counter wraps from 15 to 0 instead. The example's test, `CountTest`, drives random `inc` and `clr` for 300 cycles and checks the count against a model in every cycle. Here is most of a report for it, in the template's sections, with every number taken from our runs of `guide-examples/regressions-and-seeds/`:

```
## 1. Summary

The counter in satcnt.sv was verified against three rules: clr sets it to 0, inc adds
one, and at 15 it stays at 15. CountTest checks the count in every cycle against a
model. It passed on seeds 1 to 10, and it caught the planted wrap bug, but only five of
the ten seeds incremented the counter at 15, so the rule about 15 was tried on those
five only.

## 4. Regression

make seeds TEST=CountTest SEEDS="1 2 3 4 5 6 7 8 9 10"

    10 ok  (10 runs)

## 5. Coverage

    top                                        100.00 %   4 of 4 bins

The four bins are the merge of all ten runs. Run by run (the [COUNT] line at the end
of each log), the counter was incremented at 15 on seeds 4, 6, 7, 9 and 10, and never
on seeds 1, 2, 3, 5 and 8.

## 6. Bugs

make seeds TEST=CountTest SEEDS="1 2 3 4 5 6 7 8 9 10" BUG=wrap

    5 MISSED, 5 caught  (10 runs)
```

```
caught on seeds 4, 6, 7, 9 and 10, MISSED on seeds 1, 2, 3, 5 and 8. The first error
on seed 4:

      1675.00ns ERROR    ..ions-and-seeds/test_seeds.py(69) [uvm_test_top]: [COUNT] count is
0, expected 15

## 7. What this doesn't show

- On half the seeds the counter was never incremented at 15, so the rule that matters
  most was tried on five seeds of ten. A run on one of the other five would have passed
  the wrap bug.
- The coverage line is 100%, but it is the union of ten runs, and five of them never
  hit the bin for an increment at 15.
- Nothing here resets the counter after time 0, so a clear and a reset are never seen
  in the same cycle, and nothing checks the count while rst_n is 0.
- The simulator has two states, so nothing here says anything about X.

## 8. Sign-off

This counter holds at 15, clears on clr and counts on inc, in every cycle of CountTest
on seeds 1 to 10, which incremented it at 15 on seeds 4, 6, 7, 9 and 10. To check it:

    make seeds TEST=CountTest SEEDS="1 2 3 4 5 6 7 8 9 10"   # 10 ok  (10 runs)
```

Sections 2 and 3, which describe the environment and the plan's status, are left out here, because the counter's environment is one test and the example has no written plan. A plan for it would have three rows, one for each rule in the summary.

Numbers and the commands behind them

Every number in that report comes from the output of a command that the report names, and the lines in the indented blocks are pasted character for character, so a reader can check each one. A summary in your own words can't be checked like that. "The regression passed" could mean any number of tests on any seeds, with or without a run that crashed. The line `10 ok  (10 runs)`, under the command that printed it, says that there were ten runs and that all ten passed. `make seeds` marks a run that crashed, or that raised an exception that isn't a check, as BROKEN and counts it separately on the same line, so the line also tells you that none of the ten runs crashed. A reader can run the same command and compare that one line with yours.

The comment at the top of the template asks for the same thing ("paste the line the tool printed, not your summary of it"). So that you have those lines when you write the report, Lab 07 has you save the output of the runs that the report quotes as you go. In Part 6 you send the regression's output through `tee` into `results/`, in Part 7 you do the same with the three `make check-all` runs, and in Part 8 you copy that folder into your repository beside the report. The last step of Part 8 is to open the repository on GitHub and check that a reader who has only that page could find the command behind every number in your report.

The template's "Environment" line asks for the commit of your repository that the report describes, because a seed repeats a run only on the same code. (Regressions and seeds shows that one more random draw anywhere in a test changes every random choice made after it, in any coroutine, and that renaming a test changes its stimulus too.)

The sign-off sentence

Section 8 of the template asks for one sentence that starts "This router…" and keeps going until it's true, so the sentence tends to come out long. For the counter you'd like to write "This counter saturates at 15", but your evidence is `CountTest` on seeds 1 to 10, and only five of those seeds incremented the counter at 15, so the report's sentence names the test, the seeds and those five. Every condition that your evidence depends on has to go into the sentence, and a reader can then decide whether those conditions are good enough for what they need.

Section 7: the limits

"What this doesn't show" is the section people leave short, and it's the one a reader who has to rely on your claim needs most, because it's the only place where they can find out where your evidence stops. Without it they would have to take the claim on trust or check it from scratch. In the counter's report, section 7 says that on half the seeds the counter was never incremented at 15, so the rule that the wrap bug breaks was never tried on them. For the router, the items come from your own work in Lab 07:

- the rows of your plan that nothing checks, or that nothing makes happen, which you put in the plan's Open items in Part 2;
- coverage bins that stayed empty, and why;
- the seeds you ran, and the seeds you didn't run;
- anything that depends on X, which Verilator never shows, because it has only two states; for example, the specification says that the router's state before its first reset is undefined, but on Verilator the router happens to start at all zeros;
- the fact that catching the fifteen planted bugs shows only that your environment catches those fifteen, and says nothing about a bug unlike all of them.

The hunt table

Section 6 has a table with one row for each mystery bug that you were dealt in Part 4, and its columns are in the order in which you fill them in. You copy the first check that fired from `make hunt`, and then write your diagnosis, which is the rule that the router broke and the file in `labs/common/rtl/` that you'd look in. Only after that do you run `make verdict`, and then you fill in the bug's name and the sequence that made it show. You write the diagnosis before you see the answer so that it records how well a failure message from your environment led you to the fault. It's also practice for the times when there is no `make verdict` to ask, and the log is all you have to work from.

A hunt that comes back `quiet` gets a row too. `quiet` means that nothing `RouterTest` drives tells that design from a correct one, so in that row you write which rows of your plan could be hiding the bug. In our runs, a hunt of B15 with the starter environment came back quiet, because the starter resets the router only at time 0, when every buffer is empty. With the assertions and the resets with traffic in flight that you write in Part 5, `make check-all` catches it.

Check yourself

1. A report's summary says "the regression passes and coverage is 100%". Name three things a reader can't tell from that sentence.

   Answer: A1, in Answers at the end.

2. Your Lab 07 report's section 7 is empty. Name two things from the lab that belong there.

   Answer: A2, in Answers at the end.

3. Why does the hunt table ask for your diagnosis before the verdict?

   Answer: A3, in Answers at the end.

Further reading

- Writing the bug report, in DD104: the other kind of report, for one bug, built around a test that reproduces it.
- Bug Lifecycle: what a useful bug report contains (the seed, the commit, expected against observed) and how a bug moves from open to closed. Its command lines are for one commercial simulator; in this course the seed is `COCOTB_RANDOM_SEED`, which `make run SEED=n` sets.
- Sign-Off Criteria: the gates a report usually has to show were met.
- Verification Metrics: coverage, bug rates and escapes as measures of how far verification has got.

Next: Lab 07: Capstone.

Answers

A1. Check yourself

Any three of: which tests and seeds ran, and how many runs that was; which command produced the result, so that they could run it themselves; which commit of the code the claim is about; and whether any single run hit every bin, or which bins only some seeds reached. The counter's coverage was 100% over ten seeds while five of them never incremented it at 15.

A2. Check yourself

Any two of: the plan's open items (rows that nothing checks or nothing makes happen), the seeds you didn't run, the bins that stayed empty and why, the fact that a two-state simulator says nothing about X, and the fact that catching fifteen planted bugs says nothing about a sixteenth unlike them.

A3. Check yourself

If you wrote it after the verdict, it would only repeat the bug's name, and it would say nothing about how directly your environment's first failure message led you to the rule the design broke and the block it's in.