# Regression Testing with Random Tests Cannot Identify Regressions. - What to do about it.

by Daniel Hansson, CEO Verifyter

## ABSTRACT

Most ASIC companies use random tests not only to verify new designs but also for regression testing. Using random tests for regression testing is a great idea for coverage as the randomness over time will ensure that the total coverage will improve. Instead of running the same tests every night, each night's regression test suite is slightly different with different seeds. However improving coverage is not what the specific topic of regression testing is about. The purpose of regression testing is to quickly identify dips in quality, i.e. regressions, in order to address them and keep the quality high. And here random tests have one downside – they cannot identify regressions. But there are ways to address this issue.

## 1. IS IT BETTER OR WORSE?

To be more precise, random tests cannot distinguish between a dip in quality and increased coverage. A random test that fails may do so because it hit a new and never before tested corner case which reveals a bug in a module that was designed by professors and PhD's long time ago in a completely different project. It is great news to stumble upon such a corner case in order to iron it out, hopefully before the customer will notice it. Alternatively the random test may fail because John accidentally sat on his keyboard while checking in his code update (he is very agile). This caused some unexpected behavior in same functions he was not even working on (sitting on keyboards often do). This is a classical case of a regression. In the first case you have great news to report, an old corner case has been identified, you are a hero. In the second case, you have to hit the panic button and hold the release. Distinguishing between good and bad news is always welcome, not only in the world of regression testing, but alas random tests cannot help you with this. The random test just tells you that something failed, but cannot say whether it is a new or old problem.

## 2. FAST FIXES FOR HIGH QUALITY

Another difference between regression bugs and new test that covers a new corner case is that regression bugs are comparatively easy to fix. If you point out that a developer made an error in an update then it is often quite easy to fix. Identifying problems as regressions, and even better, linking the problem to the revision(s) where the problem was introduced, results in faster fixes. The faster you fix regression bugs the better quality you have of the design during development, which in turn leads to earlier time to market, as the developers jobs are not hampered by quality dips. So separating regression bugs from failures due to new test scenarios also leads to a substantial productivity gain.

## 3. DIFF DOES NOT WORK

A directed test, as opposed to a random test, is good at identifying regressions. If a directed test passed earlier, but it fails now then you most probably have identified a regression. Comparing the revision database today, when the test fails, with some time back when the test fails, makes it possible to narrow down when the problem occurred. You can basically do a diff between good result and bad result both in terms of log files and the revision database and draw some conclusions. The cause of the quality dip, i.e. the regression, is one of the updates to the revision database in this time window. You don't know exactly which one, but you have a list of changes and limited set of people you can blame. Directed tests are great at identifying regressions. They are not great at providing good and steadily improving coverage over time as random tests are, but in terms of being able to identifying regressions directed tests are great and doing a diff between pass and failure gives you lots of useful information.

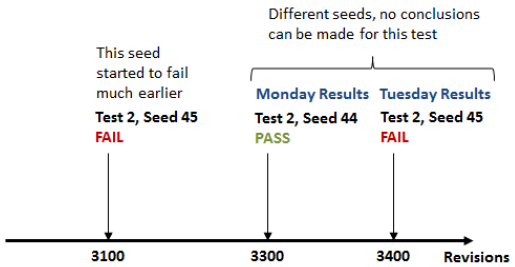


**Figure 1 Diff Cannot be Used For Random Tests**

Diff doesn't work at all with random tests (refer to Fig 1). If a random test passed yesterday, but failed today, but with a different seed, then this can be due to either a regression in quality, increased coverage or the test may even be illegal. An illegal test will probably lead to a constraint being set to eliminate this type of test, whereas in the case of regressions and coverage

improvements will lead to fixes in the design under test. For random tests we must find a different solution.

## 4. BACKTRACKING IS THE WAY FORWARD

In order to draw conclusions why a random test failed we must retest the very same test on older revisions. This means rerunning the failing test, using the same seed, on older revisions, in order to identify when the problems started to arise. This is the only way to be able to compare the test results on older revisions with the test results on the latest revision. Once you have rerun the same test on an old revision then you will be able to do the same comparison as you would with a directed test. If the same test with the same seed passes in an older revision then you are able to identify that a regression has occurred. If the same test and seed has always failed then you know this is a new test. This new test may in turn either be catching a new corner case, or alternatively it may be an illegal test. Either way you are now able to distinguish between new tests and regression in quality.

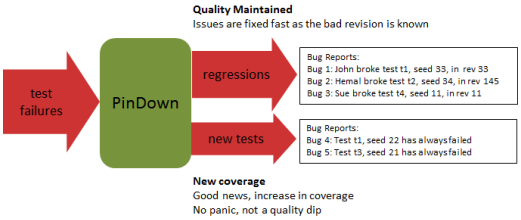


**Figure 2 Random Testing with PinDown**

Backtracking through older revisions used to be a manual process, consuming expensive engineering time, but this has now been automated in PinDown, the automatic debug tool. PinDown can automatically debug any test failure, both random and directed, down to the exact revision that caused the failure and send the developers who cause the failure a bug report before the night's regression has even finished.

Figure 2 shows how PinDown operates on the flow of random test failures. The stream of random failures are split into *regressions* and *new tests*, where the regressions are diagnosed down to the exact revision that caused the problem and a bug report is sent to the person who committed each the error. This allows regression errors to be fixed fast and thus allows the device and testbench to maintain high quality.

The other category is *new tests*, i.e. tests that have always failed and are consequently covering a new test scenario. These are not failing due to a sudden regression in quality, which may lead to panic and holding the release, but is new test coverage which is overall positive news.

This setup solves the problem with using random tests in regression testing. It allows you to keep running random testing with the upside of getting good coverage without the downside of not being able to identify regressions.

## 5. CHALLENGES WITH BACKTRACKING

There are challenges with backtracking, it is not as straight forward (or backward) as it may sound.

The first challenge is random stability, a topic widely discussed as it affects any debugging with random tests. Random stability is the art by which the same seed should always give you the same test even if the testbench has been updated. When you debug a test failure you want to reproduce the same scenario by providing the same seed number, and not get a new scenario where the test may not even fail, just because the testbench was updated. In one end of the spectrum, the EDA vendors often claim that they have perfect random stability, but at the other end of the spectrum it is impossible to make such guarantees for major changes of the testbench.

Random stability of the commercial tools has improved in recent years. Some years ago a vendor, who shall not be named, could not handle any changes to the testbench, not even comments, without losing the random stability, as the randomness was based on the number of characters in a file. Luckily those days are over. These days limited changes to the testbench does not affect random stability unless you fiddle with the random generation itself or change the structure of the entire testbench, .e.g instantiate more modules with random generators or change the dependencies between modules.

How does random stability affect back tracking? Well, if you encounter a pass in an older revision this is probably because you have reached a point before that error was introduced (which allows you to point at the faulty revision), but the pass may also be because of testbench changes which have changed the test to test something else. Capturing the impact of limited testbench changes is as important as capturing design bugs, but there always the risk that the test with the same seed passed on an earlier revision was producing a different test back then as random stability is not guaranteed for major changes. This problem is

bigger when the testbench is undergoing major design changes and is reduced at the later stages of the project when the testbench is updated with smaller changes, such as constraint changes. The fact that backtracking can help you narrow down the problem is still very useful, especially using automatic backstracking such as PinDown, as it can point to the exact revision in the testbench when the test started to fail. If the commit message for the faulty revision says something like "changed constraints to solve an issue" then this revision probably introduced a real error and the debug analysis was correct. However if the commit message on the other hand says "Changed the random generation for one module" then this revision may not have introduced an actual error, just changed the test to test something completely different.

How often do bigger changes occur? Most changes are minor changes, like constraints update, whereas major revamps or new designs come less frequent. Every big change is followed by a number of small fixes. According to one paper 90% of updates is less than 10 lines of code. Depending how well designed the testbench is the more it will be randomly stable. But in most systems the far majority of the testbench changes will be minor and easily debuggable by back tracking. But what happens if the debug goes wrong because the exact same test is not reproducible on older revisions due to a major change? Well, if the automated debug failed, you are back to where you are now: manual debug. Automatic back tracking is about improving productivity, and no damage is done if there are cases where you still have to do manual debug. As long as the far majority of all issues can be automatically debugged then you get the much sought after overall productivity improvement.

A second challenge with backtracking is that it consumes time. All debugging takes time, a lot of time, so this is nothing unique with backtracking. However, the smarter you make the selection of older revisions and tests the faster you can backtrack through the revision history. PinDown has an algorithm (patent pending of course) which does a very good job at this, but if you do backtracking manually you should use your knowledge of the design to carefully select the fastest test on some good older revisions to get to a conclusion fast.

## 6. CONCLUSION

Random tests are great to use in regression testing to get good coverage, but they cannot distinguish regressions, i.e. dips in quality, from improvement in coverage. This can be solved by backtracking through older revisions and retest the failing test using the same test and the same seed on older revisions in order to separate regressions from tests that fail because it contains a new test scenario. This process can be done manually or automated with a tool such as PinDown. The issue of random stability means some updates of the testbench will still need to be manually debugged, but the far majority of all test failures can be automatically analysed. Identifying regressions quickly and automatically allows you to maintain high quality, which in the end leads to an earlier release.

**Formatted:** Font: Times New Roman, 9 pt, Not Bold