# Standard Regression Testing Does not Work

Presented at DVCon Feb 2015, San Jose, CA

Daniel Hansson
Verifyter AB
Lund, Sweden
daniel.hansson@verifyter.com

***Abstract*— A popular approach to regression testing is to test every commit to the revision control system with a short test suite. The idea is that if this short test suite fails then we know which commit that caused the problem and the committer can be automatically notified about his or her mistake.**

**This paper shows that this seemingly simple idea does not work. The number of times a bug report is assigned to the wrong committer was measured in multiple ASIC projects to be 41%, which is shockingly bad. There is no point in using an automated system that produces such a bad result.**

**The reason this approach does not work is because it assumes only one failure is introduced per failing test and that this failure is promptly fixed before anything else bad happens. Unfortunately the reality is much more complex. Failures are continuously introduced during development and the failures overlap in time. Tests often fail for multiple reasons, not just one reason. Quality during development is basically too messy to use this simple approach.**

**At the end of the report we show how to fix this problem. We outline two alternative methods on how to setup a regression test system that produces 100% correct bug reports.**

***Keywords—regression testing; automatic debug***

## I. INTRODUCTION

### A. *Standard Regression Testing*

During product development of ASIC's or software, new bugs are continuously introduced by mistake causing the quality of the product to deteriorate. These bugs are called regression bugs. Regression bugs are captured by running regular test runs, so called regression tests, which typically are RTL simulations that are run once or several times per day. By comparing the test failures with the results from the previous time the regression tests were run, makes it possible to determine when the failure started to occur. It is possible to list all commits to the revision control system that have occurred since the tests passed and assume that the regression bug was introduced in one of the commits.

Ideally, if test times are not too long, then we are able to run regression tests on every commit, in which case we are able to say exactly in which commit a bug first appeared. See Figure 1 "Small Test Suite" where revision 6 and 12 are examples of commits where a failure first appears, because these are the first time the test fails (red). In this case a bug report can be sent to the committers of these commits and let them know that a test started to fail after they made a commit. This is as precise as it can be, in terms of blame. We know exactly who should fix the issue.

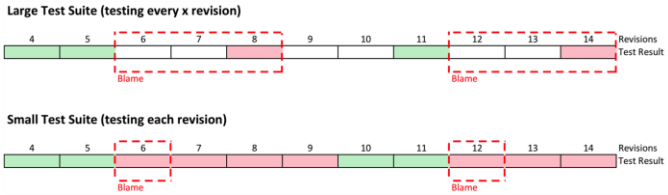



**Figure 1. Regression test setup – large and small test suites**

In reality it is impossible to run all tests on every commit in ASIC projects, because the total test time is too long. It is possible to run a small test on every commit, but the far majority of tests can only be run on every x revision (see Figure 1 "Large Test Suite" where white indicates that the tests have not been run on this revision). Here it is not possible to say exactly which commit by which committer that may have introduced a problem. Instead a range of commits can be generated which contains the bad commit, but also a range of innocent commits, but alas we cannot separate them. In Figure 1 "Larger Test Suite" a commit in the range from revision 6 to 8 introduced a test failure and a bit later a commit in revisions 12 to 14 introduced another failure. A common practice is to email all these committers even though it contains innocent committers as well guilty committers.

This paper is measuring both cases in Figure 1. It will test the idea that if we just know when a test started to fail then we know which commit that is the cause. This paper will show that this is a fallacy. This approach is too simplistic.

### B. *Regression Test Failure Scenarios*

The simple scenario (see Figure 2 – "1.Simple Scenario") consists of a regression bug that is introduced in a commit (revision 6) and this bug is still the only reason why the test fails at the latest revision (revision 9) some time later. However, this too simplistic and far from the only test failure scenario. The picture may be complicated because there are

multiple bugs or unreliable or non-existent test results. These more complicated scenarios builds up over time, between the first time the test fails and as long as the test still fails. Let's go through some common cases one by one.


**Figure 2. Regression Test Failure Scenarios**

Let's start with overlapping failures, where only one failure is still open, see Figure 2 – "2.Overlapping Failures – Only most recent failure is still open". In this example one failure was introduced in revision 6 and another failure was introduced in revision 7, both causes the same test to fail but for different reasons. The first failure was fixed in revision 8, so at revision 9 the only reason the test still fails is because of the second test failure. In this scenario it does not work to use the simple approach and accuse the first commit that caused a test to fail (revision 6) because that bug has been fixed. The test fails because of the second failure (introduced in revision 7).

Another case of overlapping failures is when both bugs are still open, see Figure 2 – "3.Overlapping Failures – Both are still open". The only difference here is that both bugs are still open. It is not completely wrong to use the simple approach here and accuse the first commit (revision 6), because it is still open, but it is an incomplete result. The person who committed revision 6 cannot make the test pass solely by fixing the issue in revision 6, because the second bug in revision 7 is still there and must also be fixed.

ASIC regression tests are often using constrained random tests, which is a method that have particular problems. First of all, if new seeds are generated in every run then it is not possible to compare the test results at all. This can be solved by running with a fixed seed number (which is bad from a coverage point of view). If the testbench has been modified then it may not be possible to reproduce the same problem, even if the same seed is used. This problem is called *random instability*. In Figure 2 – "4. Random Instability – Different testbench revisions may generate different behaviour" we explore this scenario. In this example the test is failing at revision 9 because the seed creates the Random Scenario No 2 which reveals a bug. At revision 8 a testbench update occurred which caused Random Scenario No 2 to be generated. In earlier revisions Random Scenario No 1 was generated using the same seed number. This confuses the analysis if you are using the simple approach. The first time the test failed was at revision 6 when Random Scenario 1 was still generated, but the failure at revision 9 is due to a bug introduced even earlier, at revision 5, and is triggered only by Random Scenario 2.

Intermittent failures is another complication. The first time a test fails it may be due to an intermittent failure e.g. due to some IT issue and not at all due to a real test failure. In Figure 2 –"5. Intermittent Failure" the simple approach would not work as revision 5 would be mistaken for the first commit that contained a faulty revision.

If a build does not compile then it is not possible to run a test on that revision, which means there are no test results available for that revision. In Figure 2 – "6. Build Failure preventing test to be run" this scenario is explained with an example. A test started to fail on revision 5, but there is no revision information available here. The simple approach does not work because there is no test result to use. The simple approach cannot distinguish if the test started to fail on revision 5 or 6.

Also, the hierarchical structure of a revision control system does not translate into a simple time lime, which is an assumption of the simple approach

## II. METHODOLOGY

### A. *How the measurement was done*

If a regression bug is only due to one commit and this commit is the one identified with the the simple approach by going back to the first time a test failed, then we will be able to make the test pass again by simply undoing all changes that were introduced in this commit. If the test still fails after this operation then the picture is more complicated. In that case there must be a second reason or alternatively a totally different commit which is the reason why the test fails. Only if the test can be made to pass was the debug analysis correct.

By checking each bug report and trying to remove the suspect bad commit listed in the bug report we are able to tell whether the debug analysis was correct or not. In order to be able to do this on large sets of data we used PinDown[1], an automatic debug tool for regression tests, because it has this test feature in-built and can be set to report whether the simple approach worked or not.

### B. *Problems with backing out commits*

There is one complication with backing out old commits. Sometimes several people do many changes within the same code lines which makes it hard to back out individual commits. In this case it is not possible to say whether the debug analysis was correct or not because we cannot do a clean removal of a commit.

However, whether it is possible to cleanly back out an old commit is clearly signaled as a failure to patch code or as a compilation error. This happened very rarely, in less than 1% of the cases, which meant we could ignore this aspect as it has no effect of the overall outcome. The reason this problem happen very seldom is that an ASIC project typically consists of a large number of large files and regression testing is run very often. In the short time period since the last regression test suite was run it is very rare that the exact same lines have been modified several times, which is the only time this problem occur.

## III. RESULTS

### A. *Result of Measurements*

We checked whether 916 bug reports in real ASIC projects errors had a correct debug analysis using the simple approach and found that 376 bug reports were incorrect (see Table 1). The bug reports contained regression setups using both cases as described in Figure 1.

**Table 1. Correct and Incorrect Bug Reports**

| Bug Reports | Correct Bug Reports | Incorrect Bug Reports |
|---|---|---|
| 916 | 540 | 376 |

This means that 41% of all bug reports blamed the wrong commit or blamed just one of several commits that together caused this problem (see Figure 3).

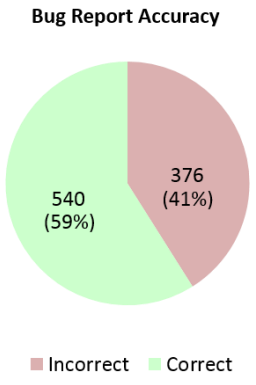



**Figure 3. Bug Report Accuracy**

The bug reports contains both cases

Please note that we are measuring the accuracy of bug reports. One bug report may correctly report which commit that contained the bug, but the next bug report which also points to the same commit may be wrong, because that bug report has subsequently been fixed. There is now a newer commit that is causing the second bug report to fail and consequently the second bug report is wrong.

It is possible to have more bug reports than bugs, which is the case when engineers don't fix the problems before the next round of testing is launched. This was the case in our measurements.

### B. *Accurate/Inaccurate Blame*

In our measurements bug reports were always correct if only one commit had been made since the last time the same test produced a pass. In this simple case (described in Figure 4, "1 Accurate Blame") there is no other possible commit that may explain a failure (ignoring temporary failures).

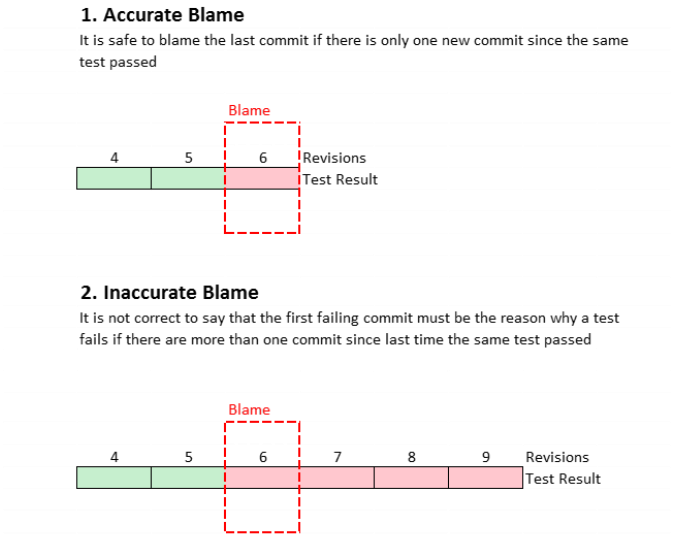



**Figure 4. Accurate and Inaccurate Blame**

It is only when there are more than one commit since the last registered pass for the same test that we get into a more complex situation (see Figure 4, "2. Inaccurate Blame"). In this case we are certain that a bug was introduced in revision 6, but that does not mean this bug is still there in revision 9. It may have been fixed and another failures is causing the failure at revision 9. Just because all revisions since revision 6 have caused this test to fail, does not mean it is for the same reason as when the problem started.

To sum up the results, we cannot blame the first revision that caused a test to fail to be forever responsible, because too often (in 41% of cases) it is a different reason why the test fails on the latest revision.

## IV. HOW TO SOLVE THE PROBLEM

There are two ways to avoid this problem. Either you have to prevent the complex regression test failure scenarios as

described in Figure 2 from ever appearing or you have to use a tool that can handle these complex scenarios.

A methodology called *continuous integrati*on can be used to avoid complexity. The second solution is PinDown, the automatic debug tool for regression failures, which can handle this complexity. Let's start with continuous integration.

### A. *Strict Continuous Integration – Avoiding Complexity*

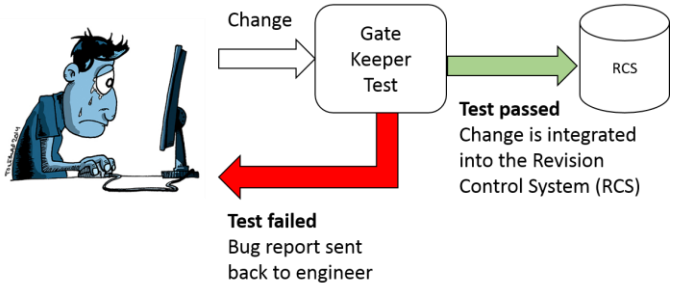



**Figure 5. Strict Continuous Integration**

Let's start with the first solution alternative: avoiding the problem in the first place. This is done by setting up a strict continuous integration [2] framework where each commit will only be allowed to be checked in to the main branch after it has passed a smoke test (thus I call this *strict*). The latest state of the main branch plus the commit-in-waiting (see Figure 5) are tested together which means there is no complexity. If the smoke-test passes then the commit is allowed to move into the revision control system. If the smoke-test fails then it is stopped from entering the revision control system and an email bug report is sent back to the engineer who wanted to commit this change.

There are never any complex scenarios in the revision control system for this smoke-test because the smoke-test has passed for each commit before it is being allowed to be integrated.

However, you have to run this on every commit so the smoke test must be very short. The second draw back, is that it does not work with constrained random testing, something which is commonly used in practically any ASIC project. The reason a random test does not work as a gate keeper is because a failing random test may have generated a new scenario which has never been tested before and may not say anything about the quality of the commit-in-waiting.

There are a lot of continuous integration frame works, of which Jenkins [3] is one of the most common.

To sum it up, a strict continuous integration setup with a smoke test as a gate keeper solves the problem for short and non-random test suites.

Software developers rarely use random tests and in many software segments (e.g. web and apps) test suites are very short. Consequently this is a very popular solution in the software industry.

### B. *PinDown – Handling Complexity*

The second alternative is to go for an automatic debug tool for regression failures such as PinDown. This tool does not act as a gate keeper. Instead it analyses any test failure, random or non-random by re-testing older revisions until it finds the bad commit (or combinations of bad commits) that causes the test to fail.

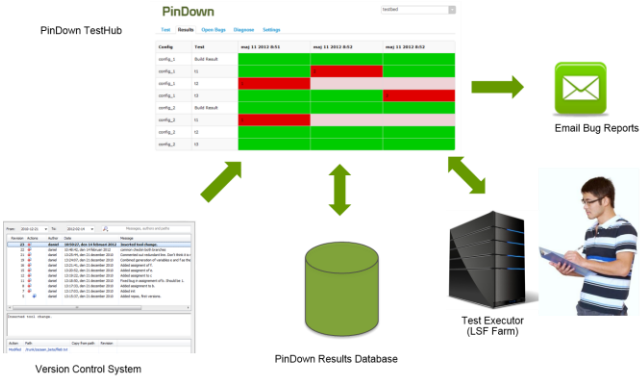



**Figure 6. PinDown Setup**

PinDown sits as a server on-site in the user's network (see Figure 6). It is directly connected to the version control system, the computer farm (e.g. LSF) and its own SQL database. The results are reported on its web page (which is only available internally in the user's network). Bug reports are also sent by email to the engineer that does the bad commits.

The user typically has a script that checks out the latest revision and kicks off a number of tests on a Linux farm. PinDown will read the results from these tests and analyze the failures. After some thinking it will kick off a number of tests, typically 10, which are run on older revisions. This is what is unique about PinDown; it does not only run on the latest revision, but also on some older revisions in order to understand when each test failed.

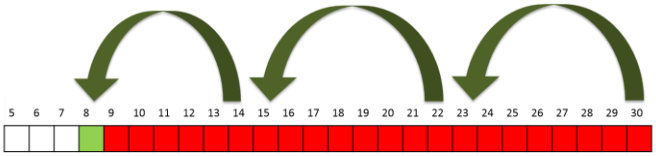



**Figure 7. PinDown Debug Testing**

The first thing PinDown does when it sees a number of failures is to group them, taking into account failure signatures and historical test results. This is sometimes called *bucketing*

and is not a unique PinDown feature. However all the steps from here are unique to PinDown.

Pindown selects the fastest test in each bug group and re-runs the test on some older revisions (see Figure 7). The goal is to find the first time this test and seed failed (for random tests the same seed number is used throughout). In this example the tests were initially run on revision 30. PinDown debugged the failure to be due to revision 9, which is the first time the test failed. For the PinDown binary-like algorithm this requires 1-4 sequential re-runs of the same test, which means the result is available in 1-4 hours. if the test takes one hour to run (including checkout and build time).

How does PinDown know that this failure is still open? The answer is that Figure 7 is a simplified illustration of the algorithm. PinDown does not traverse the history of the version control system in a linear way. It actually creates combinations of code which may have never existed, but where the result helps PinDown sorting bad from good commits. The end result is always 100% correct.

PinDown will run more tests on older revisions on the farm, typically never more than 10 tests. This means that 10 extra slots on the farm and 10 extra simulation licenses will be used. This is normally quite small compared to how many tests a typical user kicks off in a regression test script.

In Figure 8 we show an example of how this may look. The user script kick off 200 tests with a script out of which 10 tests fail. PinDown then starts debugging by kicking off 10 parallel debug tests. After some time PinDown sends away bug report no 1 and after some additional time bug sends away the second bug report. It turns out that all 10 test failure are due to these two bugs. As you can see the PinDown farm usage is very small compared to the large number of tests in the user's regression script.

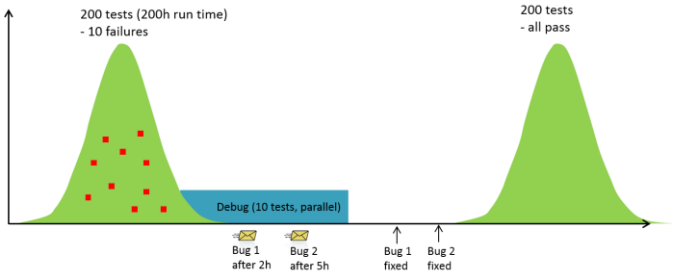



**Figure 8. PinDown Farm Usage**

However, it is worth taking in another piece of fact into this calculation. Bugs are fixed much faster when the debugging of regression failures is automated (see next chapter C. Advantage of Automated Debugging of Regression Failures). The two engineers that got the two bug reports will fix them faster and consequently there is less of a chance that the next regression run is performed with the exact same state (i.e. the two bugs are still open), which would be a complete waste of farm resources. It is better to spend few farm resources to quickly close issues than to keep re-running and finding the same failures.

### *C. Advantage of Automated Debugging of Regression Failures*

So far this paper has shown that it is better to use a strict continuous integration setup or PinDown, than a standard regression test setup, because then you will automatically know who caused a test to fail and with which exact commit. But how much will you actually gain from knowing this?

In a paper [4] we presented at the Microprocessor Test and Verification Conference 2013 we showed that automatic debug of regression failures reduce the time it takes to fix bugs. It is quite a dramatic improvement. Note that the results from this paper is valid both for PinDown and for a strict continuous integration setup.

Bugs were fixed 4 times faster (see Figure 9) with automatic debug compared to manual debug.

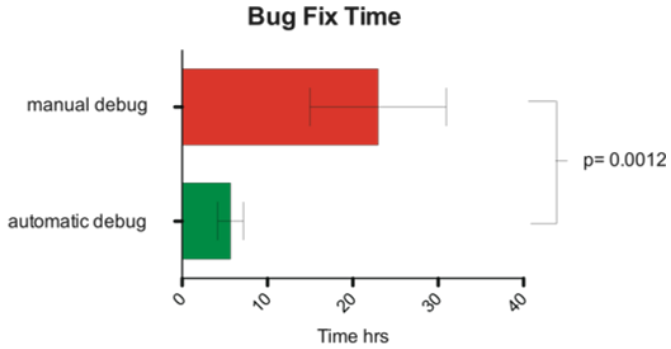



**Figure 9. Comparison of manual and automatic debug time. Time was measured from failure report until issue was fixed in the revision control system. A statistically significant difference (p=0.0012) between manual and automatic debug was found. With automatic debug, bug fixing time is 4 times faster.**

The reason why this occurs could be found in other metrics. When counting how many emails that were discussing a test failure before it was closed, we found that there was 5 times more emails for manually debugged test failures than for automatically debugged failures.


**Figure 10. Comparison of the number of emails concerning test failures after manual and automatic debug. A statistically significant difference (p=0.0002) between manual and automatic**

**debug was found. With manual debug there are 5 times more emails discussing failures compared to automatic debug.**

We have since this paper was published continued to measure the effects of automatic debug of regression failures and we have confirmed that these results are robust over many projects. We have also asked managers to estimate how much the projects have been shortened due to the speed-up in bug fixing which have resulted in estimates up to 11% shorter project time, thanks to automatic debug.

To summarize, it is a major advantage for a project to automate debugging of regression failures. Bugs are fixed faster with less human effort and debate. This saves both human resources and project time.

## V. SUMMARY

A standard approach is to run regression test suites as often as possible and assume that the first time the test fails will point to the commit that broke the test, or at least a range of commits, where the bad commit is within this range. This is a fallacy. We measured this to be wrong in 41% of the cases in real-life ASIC projects and this was analyzing the ideal scenario where every commit had been tested.

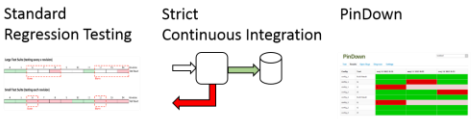
This is an unsuitable approach as engineers stop acting on the reports if the accusations are wrong this often. This will produce debates and not quick bug fixes.

The reason is that complexity is allowed to build up from the first time the test failed until the last time the regression test was run. With many commits from several engineers you get overlaying failures, random instability, build failures preventing tests to be debugged, intermittent errors and real-life revision control systems cannot be simply translated to a simple time line.

The solution is to either prevent the complex bug scenarios to be able to appear or to use a tool that can handle them. Strict continuous integration, where a smoke test acts as a gate keeper for whether a commit is allowed to enter the main branch solves the problem for small non-random test suites. An automatic debug tool for regression tests, such as PinDown, is the second solution as it can handle this complexity. Unlike continuous integration, PinDown can handle large test suites and random tests.

### A. *Comparison between the regression test setup alternatives*

Figure 11 contains a summary of the advantages and disadvantages of the three alternatives:




| | | | |
|---|---|---|---|
| **Bad Commit Known** | No | Yes | Yes |
| **Benefits** | | | |
| Automatic Triage | No | Yes | Yes |
| Bugs Fixed Faster | No | Yes | Yes |
| Less Debate | No | Yes | Yes |
| Shorter Project | No | Yes | Yes |
| **Support** | | | |
| Large Test Suites | Yes | No | Yes |
| Random Tests | Yes | No | Yes |
| **Cost** | | | |
| Debug Testing | Yes (manual) | No | Yes (automatic) |

**Figure 11. Comparison between Standard Regression Testing, Strict Continuous Integration and PinDown**

The only explanation that may be necessary to Figure 11, as it has not been explicitly explained earlier in this paper is the cost. The cost of PinDown is that you need to re-run tests on older revisions, typically max 10 tests in parallel, which consumes simulation licenses and farms on the slot. For Standard Regression Testing you must do some manual debug testing, because you don't know why it failed. That too consumes license and computer resources (typically just 1), but even worse it consumes some working time of an expensive engineer. However, strict continuous integration does not have any cost at all for debugging. The reason is that you must run the tests anyway, and in the case of continuous integration you run them prior to integration rather than after integration. However there is no extra test runs introduced by the continuous integration setup, so there is no such cost.

***Please note that continuous integration only works for small test suites. This means that you still need to debug large test suites either with PinDown or manually.***

### B. *Common Questions and Answers*

Q: We tell people to run smoke tests before they commit. Does this mean we are running strict continuous integration?

A: No, for two reasons. First, because it is not strictly enforced. Most regression bugs are mistakes and people may also make a mistake during the check-in process. They may check in something different than they actually tested. They may forget to check-in one of the files. The second reason is that if you

call it “smoke-test” this probably mean you are running larger test suites after integration. Best case you are running continuous integration for the smoke-test, but the rest are running as Standard Regression Testing.

Q: We are using Jenkins (or other continuous integration tool) to kick off tests regularly on the Linux farm. Is that strict continuous integration?

A: Just using Jenkins (or similar) to kick off jobs regularly on the farm is Standard Regression Testing.

Q: We run really often so we know which commit that caused the failure, right?

A: No, as this paper shows this is not the case. Even in the ideal case, where you run on every commit, you will only know which commit that initially caused a test to fail. However, that is often not the reason why the test still fails some time later (in 41% it is not as shown in this paper). No matter how often you run regression tests you will not be helped in knowing whether a test still fails for the same reason as it did initially.

If you are running random tests with new seeds in each run, then you will not even know which commit that caused a test to fail initially, as each seed produces a new scenario.

### *C. What should you choose?*

If your entire test suite only takes a couple of minutes to run and does not contain any random tests then you should go for the strict continuous integration setup. It is a cheap and robust approach which is very popular in the software industry. The reason is that many software segments have short directed test suites, which makes this a perfect solution.

However, if you have large test suites or random tests then PinDown is the most appropriate solution.

## REFERENCES


[1] PinDown Automatic Debugger from Verifyter, http://verifyter.com

[2] “Continuous Integration”, Martin Fowler, 2006

[3] www.jenkins-ci.org

[4] “Measuring the Gain of Automatic Debug”, Daniel Hansson, Heli Uronen-Hansson, Microprocessor Test and Verification Dec 2013