Driving Blindfolded – Solving The Triage Challenge!

ABSTRACT

When regression tests fail, an engineer must decide which failing tests should be further analyzed and by whom - a process called triage. The challenge is the dearth of information about the bug and its creator at this early stage. This session outlines a novel method that addresses the issue by driving the testing to gather enough information to do fully automated triaging. This paper explains how this innovative approach solves the triage challenge, thus accelerating the bug fixing process by 30%.

1. INTRODUCTION

Regression testing is performed regularly through-out a project in order to ensure that the quality never dips, regresses. Regression testing is normally automated in terms of kicking off the tests and putting together the test results. But the analysis of the test results is not automated. The first thing that needs to be done is to look at the test failures, decide which are real errors and which are just intermittent computer or network issues and then assign the bug to the appropriate engineer. This process is called triaging. As it is impossible to know the exact nature of a bug, before it has been thoroughly analyzed, triaging has to be done while having very little information available at this early stage. The methods used are ad hoc, based on experience, association or static assignment, where an engineer is assigned for each test to first look at the failure. The triaging process takes time as the test results needs to be translated into bug reports and if the results are sent to the wrong person(s) then it will take longer before the issues are fixed.

What is needed is a tool that provides automatic triage which immediately sends the bug reports to the person who committed the faulty revision in order to accelerate the bug fixing process. The earlier the correct engineer gets the bug report the sooner the bug will be fixed. In order for this to work, automatic triage needs to be very robust so that it always sends the bug reports the right way and must be fast to run and use limited amount of resources so that it is efficient to use in large systems. This paper looks at the technology behind a new tool called PinDown, which has a novel approach to automatic triage that addresses these issues.

2. AUTOMATIC TRIAGE

2.1 Mapping Test Results To Revision Information

The Version Control System (VCS) contains the code of the device under test, the test bench, the tests and all the changes made to all of its files. The VCS contains information about each revision of each file or module such as: the committer of the revision, the time of the commit and the code lines that were updated. This is the information needed for triaging: who (committer), what (revision) and when (time of commit). What is needed is a tool that can map test failures to a revision in the VCS in order to extract all the information needed for triaging, including both creating a bug report and assigning it to the right person.

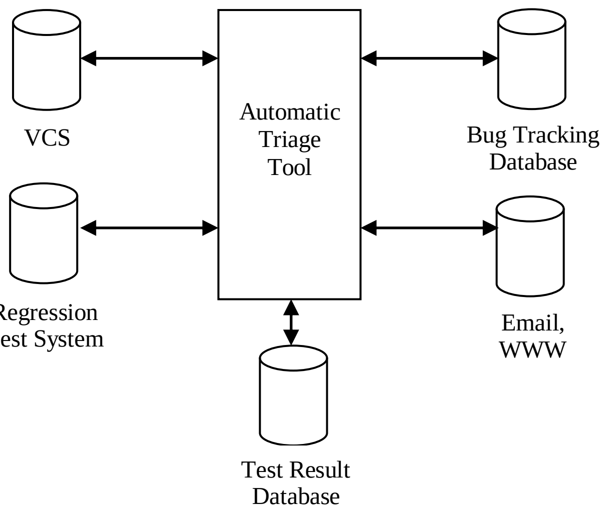

Figure 1. Automatic Triage Tool in System

In order to take advantage of the information available in the VCS an automatic triage tool consequently needs to interface against both the VCS and the regression test system in order to map test results to revision information (see **Figure 1**). The automatic triage tool also needs to be able to communicate the result of its diagnosis automatically to the bug tracking database, email and files on the web and on the normal file system. The automatic triage tool also needs to save the test results and its diagnosis in a test result database to allow for speedy analysis and to be able to amass historical test results information that can be used in its analysis.

Finding the revision at which point a bug was introduced that caused certain tests to fail is done by rerunning the failing tests on older revisions, until the first revision is found for where these tests fail. If tests passed on the previous revision and then started to fail on this revision then we know that the changes made in this revision contained a bug which caused the tests to fail. The revision that contains the bug is called the faulty revision.

Regression testing is a type of verification where the purpose is to ensure that the device under test does not regress in quality. The regression tests should normally pass and if they don't, it is possible to trace back through the revisions until the earliest revision is found for which the test(s) do not pass. This is the faulty revision. However, if a test has never passed, e.g. a new test that was just introduced, then the result may be always-failed. In order to avoid going back too far a limit must be set at which point a test is concluded to be always-failed.

Diagnosing a test failure to the faulty revision (or less common, concluding that the test has always failed) is a very robust method. The result is deterministic ("this test fails since this faulty revision"), which is different from many other diagnosis methods which lists the most likely sources of an error. For triage it is important to deterministically be able to point out the committer in order to assign the bug report to the correct engineer automatically. The committer of the faulty revision may not have introduced an actual error. Instead the committer may have done an incomplete update, where the changes made were appropriate but the committer forgot to update associated tests or configuration files. It may not even be the committers responsibility to update the other files necessary to make the update complete. However, even in this scenario the committer is the best person to assign the bug report to as this person can quickly notice what is missing to make the update complete and assign the bug report to the appropriate person and add the required changes to the bug report. Automatic triaging is not about blame, but about finding who is the person best suited to analyze a failure in order to fix a bug as soon as possible.

2.2 Linear vs. Binary Search

The algorithms that can be used to step back and rerun a failing test on older revisions in order to find the first faulty revision are linear or binary search. The linear approach is costly as it tests all older revisions in reverse chronological order, but on the other hand the first faulty revision will be found with certainty. The binary search method is faster, where only the middle revision is tested between a passing revision and a failing revision, but it only works when you have just one bug as it requires the test results to provide directional information, whether the next revision to test should be younger or older.

Revision	1	2	3	4	5
Test Result	P	F_1	F_1	P	F_2

Figure 2. Test Result Example

If you have had two bugs introduced recently and one of the bugs has been fixed, then the binary search will not work. It will find the faulty revision of one of the bugs, but not necessarily the faulty revision of the bug which is still open. The linear approach will be able to identify the correct faulty revision, but at a higher test cost as more revisions will be tested.

In the example in **Figure 2**, a test fails at the latest revision, which is revision 5. An error was introduced in revision 5 which can be concluded as the test result on the previous revision, revision 4, is a pass. The linear algorithm will capture this as it will rerun the failing test on revision 4. However, the binary search algorithm will fail to find the correct faulty revision even for such a simple example. The first step in the binary search algorithm is to identify a revision in the middle of the revisions for when the test last passed and the latest failure, which in this example is revision 3. The test fails also on revision 3, which is why the binary search method continues to test revision 2, which is in the middle of this revision and the last pass. Finally the binary search algorithm concludes that the faulty revision is revision 2. It is correct that revision 2 is faulty, but what the binary search algorithm misses is that this issue has already been fixed in revision 4. Only the linear approach works in this case.

Also, if there is no known previous passing revision available, for example the first time you perform automatic triage, then binary search does not work, only linear search works.

As a conclusion, the problem with linear search is that it is not fast enough to be of practical use in large systems. The problem with binary search is that it cannot handle multiple bugs, which means it is also not of practical use in large systems.

2.3 Representative Testing

This paper introduces a novel algorithm which is fast and able to handle multiple bugs and which can be used on large systems.

First, all test failures are analyzed and grouped into bugs, where each group of test failures is assumed to be caused by the same bug. The grouping takes into account test results, build characteristics, log files and historical test results provided by a test result database.

The second step is to select the best agent to represent each group of failures. The agent is selected based on the following factors:

- Test and build failure pattern, e.g. do all tests fail for a specific build type or does one test fail across all build types
- Test times
- Build times
- Checkout times from the VCS
- Known bugs
- Historical test data, available in the test result database.

The goal is to find agents that provide fast and robust results.

Revision	1	2	3	4	5
Test 1	P	F			F
Test 2	P	F			F
Test 3	P	F			F
Test 4	P	F			F
Test 5 (agent)	P	F	F	F	F

Figure 3. Representative Testing

The example in Figure 3 shows a system where all tests failed on the latest revision, i.e. revision 5. Test 5 is selected as agent and the failure is diagnosed down to revision 2 which is the faulty revision. Once the faulty revision for test 5 is known, the rest of the tests are tested on revision 1 and 2 to verify that they also have started to fail here due to the faulty revision no 2. Thus the conclusion is that revision 2 is a faulty revision which caused all tests to fail and the fault has not yet been completely fixed as the diagnosis of test 5 shows. There is a chance that the fault in revision 2 was partially fixed benefiting some of tests 1-4, but that these tests then suffered from a fault that was introduced later, e.g. in revision 3 or 4. If this, less likely scenario, is true, than these bugs will be correctly diagnosed once the fault in revision 2 has been fixed. The importance is that the diagnosis of the agent is robust. As with all testing, once you have fixed an issue, more issues may be revealed later.

Once a faulty revision is found for each agent, the algorithm validates the other tests in the group whether they also have the same first faulty revision. As the algorithm validates the assumptions made during grouping, a correct result is ensured.

If any assumptions are found to be wrong then we will go back to grouping, following by selection and testing of agents and non-agents. Consequently, the better test failure grouping, the better performance, but the end-result is never affected.

Representative testing does not require a certain underlying search algorithm. Both linear and binary search algorithms may be used together with representative testing. What representative testing does is to speed up the diagnosing process to such a degree that it allows the linear search method to be used, thus providing results that are both fast and accurate.

2.4 Result Noise

A common problem when running large regression test suites is the intermittent instability of the computers and networks used for testing. This may lead to builds or tests failing, or never completing. These results, called result noise, do not reflect the state of the device under test and should be filtered out.

It is important to be able to filter out result noise from real design results in order to reach the correct conclusions. This is done by retesting results which look suspicious or which do not behave as suspected. Another mechanism is to verify that the test is up and running by checking the results of tests that verify the flow and not the device under test.

2.5 Reporting

When it has been concluded in which revision a bug was introduced it is possible to provide information about the bug from three different sources:

1) The VCS contains key data about the revision. This includes the username of the committer, the time of the commit and the lines that were modified. This information is required in order to create a bug report with all the necessary information and to assign the bug report to the correct person, i.e. the committer of the faulty revision.

2) A comparison between test results of the faulty revision with test results from a previous revision for which the same tests pass provides information about the unique characteristics of how each bug manifests itself. This method is applied to the sometimes lengthy build logs and test logs in order to distinctly point to the section in the logs where the error started to manifest itself.

3) A comparison between the test results for different build configurations, all based on the same faulty revision, provides information about the build configuration characteristics where the bug manifests itself.

3. CONTINUOUS INTEGRATION

Continuous Integration (CI) is a continuous process where regression testing is done often, ideally on each commit to the VCS. If testing is performed on each commit then it is easy to pinpoint the failing revision and identify the committer. The downside is the amount of testing that needs to be done, more specifically:

$$\text{total test time} = \text{time for test suite} * \text{number of commits.}$$

Equation 1Test Time – Standard Continuous Integration

The total test time can be addressed in several ways. Instead of testing each commit in series, you can use a larger number of computers and test several commits in parallel (see **Figure 4**). This is a traditional trade-off between time and computer resources that is valid for all verification. Another, very common, way to address this issue is to reduce the time a test suite takes, a dedicated integration test, and then perform a larger regression test at a later stage. In that setup the CI methodology can capture bugs and perform automatic triaging if the integration tests fail, but not at the later stage, where we get a bug slip through which can only be handled manually.

With automatic triage it is possible to handle continuous integration in a more efficient way. The total test time is only:

$$\text{total test time} = \text{time for test suite} + \text{diagnosis time}$$

Equation 2 Test Time - Continuous Integration with Automatic Triage

If there are no failures then there is no extra diagnosis effort. Even if there are test failures, thanks to the method of representative testing, the extra diagnosis time is small, typically 10-20% of the total test time. This requires a multiple less testing than traditional CI.

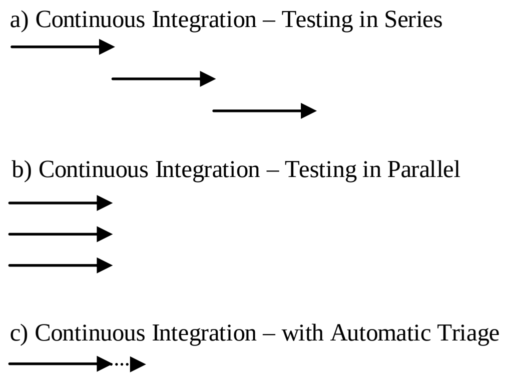
a) Continuous Integration – Testing in Series
b) Continuous Integration – Testing in Parallel
c) Continuous Integration – with Automatic Triage

Figure 4 Continuous Integration

Not only is this a more efficient way of diagnosing the failure, but also it verifies all changes together which means it captures cross-dependencies between the recent commits. Also, there is no slip-through of bugs, when using an automatic triage tool as this can diagnose bugs at any stage in the process.

4. RESULTS

This new algorithm for automatic triage has made it possible to use automatic triage in large real test regression systems. Using automatic triage in this way has been measured to cut the average bug fix lead time by 30%. Faster bug fixing, translates to better product quality during development, which in the end means reduced time to market.