A Method to Select Tests Based on Code Coverage

Jack Marvin and Trevor Hammock

Jack_Marvin@mentor.com Trevor_Hammock@mentor.com

Abstract

We all know that to discover software defects, tests need to be run. But how many tests are "good enough"? Many companies do full regression suite runs to validate new software versions. Depending on your situation, this may be fine. If you are part of a start-up, where there is a high rate of code change and potentially fewer tests, it may be preferred. But if you are QA for a mature product with many tests and/or limited resources, you might wonder about the value of running all the tests when little or no changes have occurred, and the turnaround is costly. If you decide to run fewer tests, there are pros and cons. You can save on compute resources, but you may not detect a defect as early, especially if your tests cover another teams' code and they lack coverage. And if another team is your internal customer and their functionality becomes broken, they will not be pleased.

There are some available tools that can help with determining which tests cover code changes. Jest[1] for JavaScript, pytest[2] for Python. They have their strengths and weaknesses.

There is often the fear 'what if we miss something'? What about side-effects? Who owns tests upstream and downstream from me in a flow?

The application described in this paper addresses these concerns. From the changes that are detected what should be tested is determined with risk reduced to tolerable levels. The result is a system that you can be confident will provide proper coverage with the fewest number of tests.

Biography

Trevor Hammock is a Software QA/Developer at Siemens, with 5 years of testing experience ranging from automation to C++ and python for test suite development.

Jack Marvin is a Software QA Engineer at Siemens, with 20 years of experience in automation, exploratory testing, and performance testing.

1 Introduction

This paper is intended for those that run regression test suites regularly and want to run less tests while maintaining coverage, which can benefit any testing interval such as hourly, nightly, or weekly. Given that there is a finite number of resources shared across all teams, and some teams or builds may get higher priority, a reduction in tests run may allow the same level of quality with less resource contention.

The system we will describe to accomplish this task is written in a publicly available coding language (Python) and uses a publicly available application (Gcov[3]) to provide data. The system is named DTS for Dynamic Test Selector. DTS creates a list of tests by determining what code changed and which tests cover those changes.

The act of reducing tests, while providing large run time benefits, imposes some risk of not detecting a defect as early as in a non-DTS system; possibly creating a reduction in confidence by others on the team. These risks may cause enough fear to reduce the adoption rate, especially for teams with a high rate of code change. Fears can be reduced by continually providing a backstop of running as before at a desired interval, possibly increased as confidence is increased.

One example of running tests with low value is when the developer is on vacation. Another is the code change is non-detectable (a change to a comment or a style change). These runs often occur when automation is created to run every build. DTS can easily be inserted into that automation to exclude running those tests.

2 DTS Process

DTS uses the idea of mapping a code change to what tests cover that change, which tells us what tests need to be run and which can be deferred to run later. Additional metrics can also be used to calculate which tests should be prioritized over others, greatly increasing efficiency.

The mapping is determined by a code-coverage tool. We use Gcov[3]. There are other tools available, some require a license. Gcov is a component of GCC, the Gnu Compiler Collection, which is the default compiler toolset shipped with the Linux operating system.

2.1 Steps:

1. A Gcov[3]-instrumented build is done where the instrumented source code is the code that a team wants to track.

2. The regression test suite is run with that Gcov build.

3. A database is constructed that maps tests to functions.

4. A tool finds the changed functions in the build to be tested and determines the tests that cover those functions. Some optional filtering can be performed to further reduce the test set.

5. The regression suite is run as usual, except for selecting only the tests affected.

2.2 Benefits of Running Less Tests

There are many benefits of running fewer tests. The most pronounced being faster approval of code changes, especially if you are competing for resources. In our fast-paced environment with constant pressure to deliver results, achieving testing completion sooner is a plus

Some other benefits include a possible reduction of hardware (computer, network, and disk space), fewer fails from the same defect and less false fails from the testing system or hardware incurring costly analysis time.

2.3 Other Benefits of DTS

While a reduced test load is the main priority, DTS provides some other benefits having to do with code and test analysis. The test analysis piece is mostly for how much tests overlap or test case effectiveness.

For instance, DTS can determine how much code a test covers, which may be useful for determining which tests can be removed and not affect quality. There is also an opportunity for test case creation training. An example is whitebox testing where we know X feature covers Y function, so we create a test that covers Y.

DTS can also identify which functions have a large impact. i.e., a function that impacts a large portion of the test suite. This can help determine which functions are more important, providing more data for DTS refinements. If you know a function is covered by many tests, you can choose tests that overlap functions and leave out ones that do not.

You can use DTS as an automated way to map/categorize tests to a given feature, providing another view of which tests are important when a feature changes.

Impact of changes can accurately be quantified. DTS can verify a statement that nothing changed, or the change should only affect one product, being more accurate than human guesses and assumptions.

The tool can aid detection of tests that won't fail or have a high likelihood of not failing. If over time, tests are not selected and did not miss a defect, it may be likely that the test cannot fail. Investigation can determine if the test is poorly constructed or useless.

Early feature evaluation builds can be proven adequate quickly by any team member besides development and QA. For example, a person on the marketing staff can prove runnability prior to build delivery.

Code churn (how often and how much code changed) can be measured to indicate release stability. DTS determines source code differences. It could report those to describe code churn.

2.4 Limitations and Risks

The Gcov[3] database is only as good as the build the tests were run with. Future changes and new tests will not be known until the next scheduled run. There is a trade-off since Gcov runs can be 20X the normal run time, meaning you cannot do a Gcov run simultaneously with the code changes. But a short interval between a Gcov build with new code changes and test runs reduces the opportunity for missing a defect.

And proving future changes not affecting current features is valuable. That is, no regressions. Depending on your role and where you are in the release cycle, it may not be a concern. If you are a developer, you may not have many tests created by QA. If you are QA, it might be early in the release cycle where you have not yet had the time to create tests. Another option is if you do have newer tests for new features, you can choose to add a selection from them to your DTS-created test list.

Which tests comprise an optimal set? What is optimal? There are trade-offs of test coverage with run time. Is the runtime of one long running test, with a higher chance of finding a bug, better than running a larger number of quick tests, that are less likely to find a bug? DTS only has one narrow algorithm of optimal test selection and clients could benefit from multiple types of selection algorithms, where the needs of the situation dictate what algorithm is preferable.

Another option if your clients expect a delivery without these failures being missed is to choose to select all the tests that cover a change instead of removing duplicates.

For whom is DTS meant for? If you have a small number of tests that complete quickly enough to meet your expectations, DTS may not be worth the effort. If tests do not complete as soon as desired due to hardware resources being finite, resulting in contention, DTS is probably worthwhile. If certain teams or build types get increased priority, and others compete for resources, DTS is probably useful for all teams.

The usage of DTS may vary depending on if it is being used by a development engineer or a quality assurance engineer. The developer wants a fast turnaround time to not delay development. QA staff may prefer more thorough testing that may take more time to complete. Where you are in the release cycle can also have an effect, you may not have many tests for new features at certain points and may have more coverage at others. Public releases may expect running more tests to minimize risk than internal builds. And as the build stabilizes, you may decide to run fewer tests.

There is a potential for other groups' tests to be the only ones to fail. If the Gcov build is only instrumented for the source code of your concern, a defect in another team's code may be missed if your tests are the only ones to cover it. Similarly, downstream features should also be considered. This concern grows more likely with an increased quantity of shared code. These risks can be mitigated by running non-DTS at your desired interval (weekly) and proper team collaboration.

Code changes to common functions may cause a high percentage of tests to be selected. Additional filtering may be necessary to reduce the number of tests, providing an opportunity to miss a defect. Running as before with an increased interval will catch the missed defect.

Running fewer tests may mean rare (1/10000) race-condition or uninitialized variable kinds of defects may not be detected as early.

DTS is designed for use with Gcov, which generates coverage for gcc compiled code (C-based languages). Products can be comprised of other languages, which Gcov will not cover. Other tools may need to be developed or discovered. Running these tools would be costly because it would increase the turnaround time of getting code coverage but would reduce the likelihood of uncovered tests.

Functional coverage won't have the accuracy of line coverage. Besides reduced granularity, code outside of any function is a blind spot. However, line coverage being more volatile, may prove more difficult

3 What We Want to Test

Our teams run hundreds of thousands of tests every day. We want to be sure we discover defects as soon as possible so they can be resolved easily and before visible to customers, which can be internal or external. Defects that customers find can result in revenue loss due to the software causing them more work and can result in a lack of confidence.

Developers usually run a smaller number of tests than QA and want a quick turnaround so they can be confident and move on to the next change. QA can be more patient and expect the additional tests to take more time.

We also have multiple binaries for each build. One for each supported operating system (OS). We decided that is adequate to usually run on one OS for nightly builds and all OSs for public releases.

We have limited hardware resources. All tests cannot be run for all builds. We end up forcibly reducing our test runs by alternating run modes or other variants. Now we have DTS, which tells us which tests to run without reducing coverage. This has worked well for teams with many developers simultaneously making many changes to teams with a few developers making much less changes.

Table 1 shows team ABC data from incorporating DTS in November 2021 for nightly builds, not for public releases. DTS is not used for Tue and Thu to prove accuracy. Reduction will be increased as confidence becomes sufficient to use DTS for all builds.

Year.Month	No. of Jobs (K)
2021.01	555
2021.02	478
2021.03	556
2021.04	547
2021.05	581
2021.06	580
2021.07	593
2021.08	502
2021.09	479
2021.1	555
2021.11[1]	294
2021.12	268
2022.01	302
2022.02	126
Average Pre-DTS	542.6
Average Post-DTS	247.5
Average % Reduction	**45.61**

1: DTS Usage Started

Table 1 Number of tests by year and month

4 Some Adoption Variants to Consider

Given that you may miss a defect when running fewer tests, which may cause a lack of trust by developers and others on the team, you may wish to consider different strategies to adopt DTS. Some variables to consider include how much testing the developer does, their expertise (how often do they make defects), and how often they make changes.

The risk tolerance of the team ends up driving the adoption, including for which builds to apply to. Some teams are happy to run much fewer tests with the chance of missing a defect. Others are more careful and fearful and want to be sure to run all tests for every build.

These are some ideas that we came up with. Other teams may discover alternatives that they decide fit their development process, and as DTS becomes more common to both Dev and QA, other options may be discovered. We have also seen minor enhancement requests as usage increases by QA and Dev. Dev requests are usually for more analysis of code changes while QA's are more for analysis of tests. These requests being minor indicates the initial success and strength of DTS.

4.1 QA Usage

1. Switch 1-N (where N is all) regression test runs for DTS until accuracy confidence is high. Example: You run a full regression suite five times a week. You could start slow and change one of those to be a list of tests determined by DTS or you could switch to DTS for all five.

2. Run DTS in parallel to show DTS selected tests have equivalent coverage. Or to avoid duplication, compare the generated test list to determine if it selected the failing tests.

3. Use DTS for nightly builds and not for public releases until confidence is high.

4.2 Developer Usage

1. Provide developers with a risk vs accuracy report to convince them of the reward. Strong data would be to show that no defects have been missed in QA testing for those developers.

2. Sell the idea to key developers that others will follow.

5 Conclusion

Although there is a risk of missing a defect when using DTS, the benefits for us far outweigh that risk. Our reduced testing load by ~50% without missing a defect proves we can be confident. The other benefits of increased productivity and determining test value are also a huge plus.

As our teams begin adoption, they have been satisfied and pleased with the large time savings, which gives room for other work, including implementing other changes sooner. And since no defect has been missed, we have been able to move forward with using DTS for more builds and across more teams.

As we make improvements with test selection (which tests provide the most coverage, keeping in mind fail history) and resource consumption (CPU time and memory) we hope to ultimately use it for all builds.

References

1. Jest https://jestjs.io/ (accessed June 29, 2020)
2. Pytest https://docs.pytest.org/en/7.1.x/ (accessed June 29, 2020)
3. Gcov https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (accessed June 24, 2020)