Generating Coverage for Regression Testing

Jianwen Dong and Yizhuo Du
Cockrell School of Engineering
The University of Texas at Austin

Abstract

Evaluation of Code Coverage in Regression Testing is the problem of identifying the coverage information through the modified versions of the software. The traditional approach requires running all the test suites to gain a complete coverage report. In this project, we design a fully automated tool that can generate an updated coverage report by only running the impacted test suites. The tool automatically detects the affected test suites based on the modifications of the software and runs the selected test suites. The updated coverage report is generated based on the old coverage data and the new coverage data of the affected test suites. Our experiments illustrate that our tool reduces the runtime overhead significantly under certain conditions compared with the traditional approach.

Keywords: *Regression testing, code coverage, automated tool*

1. Introduction

Software development and maintenance are two major parts of the software systems evolution. After the software is updated, regression testing is applied to the modified version of the software to ensure that it behaves as intended. Code coverage analysis is the process of identifying code executed by a particular set of test suites, which is an important component of software verification.

One traditional approach of generating complete coverage information for regression testing is that running all test suites for every modified version of the software. However, that approach is costly because, as software grows in size and complexity, a test suite accumulates more tests and therefore takes longer time to run. Therefore, the runtime overhead of running all test suites can increase dramatically as the software is developed over time.

Based on the drawback of the traditional approach, an efficient approach [1] would be generating correct and complete coverage report by only running the impacted test suites. This alternative approach requires the old coverage data of the previous version of the software and detection of the impacted test suites. By combining the old coverage data and the updated coverage information of the affected test suites, the complete updated coverage report of the software can be generated. As a consequence, the new approach has a better performance in terms of time consumption compared with the traditional approach.

2. Algorithm

The coverage recomputing algorithm provides the same coverage based on the old coverage information and the new incomplete coverage information after running impacted test suites [2].

The first step is to collect the new project coverage. The inputs for the procedure include the old program *P_old*, the new program *P_new*, an RTS system *RTS*, and all the test suites *T*.

Declare types:
FileCoverage: {e} which contain uncovered lines for a source file as a set.
SuiteCoverage: {(filename, file_coverage)}which contains FileCoverage for all the source files for a single test suite as a map.
ProjectCoverage: {(suite_name, suite_coverage)}contain SuiteCoverage for all test suites in a project as a map.

```
Procedure getNewProjectCoverage(P_old, P_new, RTS, T):
    old_coverage = ProjectCoverage for old version // Read the coverage from
    the file
    new_coverage = create an empty ProjectCoverage
    impacted_suites = RTS(Program_old, Program_new) // Invoke the RTS system

    foreach s_i  ∈ impacted_suites:
        coverage_si = run_coverage(s_i) // Invoke the test suite and get
    SuiteCoverage
        new_coverage[s_i]=coverage_si
    endfor

    foreach s_j ∈ (T - impacted_suites):
        new_coverage[s_j]=old_coverage[s_j] // Add the missing coverage
    information
    endfor

    return new_coverage
end getNewProjectCoverage
```

Fig 1. Get New Project Coverage Algorithm

After this procedure, we now have new project coverage information. However, in order to generate a coverage report, we need a map from filename to FileCoverage. The procedure to generate such a map has an input type ProjectCoverage, and the algorithm is as follows:

```
Procedure getCoverageForFiles(project_coverage):
    files_coverage = init_map<Filename, FileCoverage> // Initially all lines
    in all files are marked as covered.

    foreach (suitename, suite_cov) ∈ project_coverage
        foreach (filename, file_cov) ∈ suite_cov
            foreach uncovered_line Li ∈ file_cov
                if Li is also uncovered in all other suites
                    files_coverage[filename][Li]=uncovered
                endif
            endfor
        endfor
    endfor

    return files_coverage
end getCoverageForFiles
```

Fig 2. Generating File Coverage Map Algorithm

Based on the output of the above procedure, we now can generate a coverage report for the whole project.

3. Implementation

3.1. Techniques

In our implementation, we need an RTS system to get the impacted test suites, a coverage tool to generate coverage for a specific suite.

After several trials, we found that an RTS system called Starts can easily run in our experiment, and we choose to use Jacoco as our coverage tool. Besides, Maven Invoker is used to run JUnit tests, invoke Jacoco coverage, and clear outdated coverage data.

3.2. Design

To store the old coverage data, we create a file "statemap.data" under the project directory. This file will contain the ProjectCoverage(coverage information for all test suites). Each time when we run our tool, we will calculate the latest ProjectCoverage and override this file with this latest result. When a user wants to forget the coverage history, this user can simply delete this file and clean "Starts". In this way, all the test suites will be run in the next execution of our tool.

To make the coverage report clear, we will read the source file line by line and check whether this line is covered or not in our latest coverage information. If the line is not covered, we will print "// <-- MISSED LINE" at the end of the line. Then users will know where to improve the test suites or debug the code.

4. Results

4.1. Correctness

In order to test the correctness and reliability of our tool, we have conducted different tests on it. There are mainly 4 dimensions of the test cases for our tool:

# of impacted test suites	Single	Multiple
# of modified source files	Single	Multiple
# of modified test suites	Single	Multiple
whether modified file names	True	False

Table 1. Dimensions of Test Cases for the Tool

After testing on our test subject, it shows that the generated coverage report by our tool is the same as the report generated with Jacoco by running all the test suites.

4.2. Performance

The time complexity of the recomputing algorithm is O(N*F*L) where N is the total number of test suites, F is the total number of files in the project, and L is the average number of lines for the files in the project.

The space complexity of the recomputing algorithm is also O(N*F*L). So the complexity is linear to the size of the project.

To show the overall running time of our tool, we need to define some symbols:

N: number of test suites
T_α: average running time per test suite
T_{RTS}: running time for the RTS(selection part)
T_{COV}: running time for generating a coverage report
M: number of impacted test suites
T_{our}: running time of the recomputing algorithm

Running time with our tool = $T_{RTS}+M*(T_\alpha+T_{COV}) +T_{our}$
Running time without our tool: $N*T_\alpha+T_{COV}$

Let's assume that each commit to the project will not change the size of the project dramatically. In this case, T_{RTS}, T_{COV}, T_{our}, can be treated as fixed values.

In order to optimize the performance of the regression testing, if the running time of test suites is fixed, the following condition should be satisfied:

$$M < (N*T_\alpha + T_{COV} - T_{RTS} - T_{our}) / (T_\alpha+T_{COV})$$

which is intuitive. Because if M is small, it means that only a small part of the test suites are impacted, and our algorithm can avoid many unnecessary runs. However, if M is large, or even the same as N, the running time of RTS and our algorithm may slow down the regression test. The condition here may not hold if the running time is a random variable, instead of a fixed value. In this case, the condition here is just an estimation for the threshold.

We then simulate our model on a project with N=50, Trts=5s, Tcov=3s, Tour=1.5s, and the running time of test suites is uniformly randomly distributed in (15s, 25s) with an average T_α of 20s.

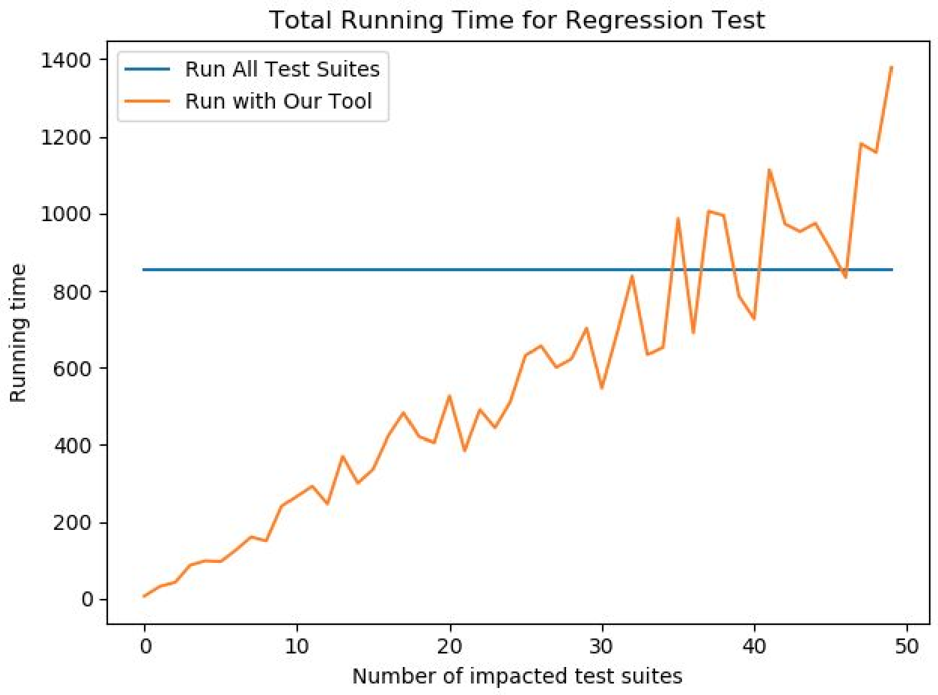

Fig 3. Total Running Time for Regression Test

After the calculation, the estimated threshold is 43.33, and we can see from Fig 3 that around the threshold, both the running times are similar, and it fluctuates because the running time is not a fixed value in this simulation.

5. Conclusion

In this project, we designed and implemented an automated tool to efficiently generate code coverage reports for regression testing. The tool generates the updated complete coverage report by combining the stored old coverage data and updated coverage data of the impacted test suites. Both coverage data are applied to the coverage recomputing algorithm to generate the updated complete coverage report. Because the tool is fully automated, we integrated an RTS system to detect the affected test suites. By conducting several experiments on our tool, we found that our tool reduces the time overhead of generating coverage reports when the amount of modified test suites is relatively small for each modification. Therefore, such an approach could save time for getting code coverage on regression testing.

Reference

[1] P.K. Chittimalli and M.J. Harrold, "Re-Computing Coverage Information to Assist Regression Testing", *Proc. Int'l Conf. Software Maintenance*, pp. 164-173, 2007.

[2] M.M. Tikir and J.K. Hollingsworth, "Efficient instrumentation for code coverage testing", *international symposium on Software testing and analysis*, pp. 86-96, 2002.