Regression Test Selection for the R Programming Language

Submitted To

Dr. Emanuel Tutuc
Kevin Olalde
Dell EMC, Server Validation Engineering

Prepared By

Canyon Evenson
Michael Flanders
Jennings Inge
Saarila Kenkare
John Koelling
Janvi Pandya

EE464 Senior Design Project
Electrical and Computer Engineering Department
University of Texas at Austin

Spring 2020

CONTENTS

TABLES

FIGURES

EXECUTIVE SUMMARY

Testing software is a central concern for all software development but remains a time consuming process. While there are many testing tools available for popular languages such as C, C++, Python, and Java, there are not as many available for the R programming language which Dell uses to test their server configurations. Therefore, we have partnered with Dell and created a regression test selection (RTS) tool that reduces the testing time of large code bases written in the R language.

From a pre-existing set of test cases, our RTS tool will select a subset of tests that target the most recent changes to the code. Cutting out unnecessary tests—i.e., tests that do not test the new code changes—reduces the amount of time developers and companies spend on testing. Although cutting out unnecessary tests seems obvious, the process of selecting the right tests needs to be done carefully. Overselection reduces performance, and under selection causes bugs to be missed and buggy software to be shipped to customers.

Our solution to this test-selection problem is a tool that is made of three components: a test-to-source-mapping component, a differences-in-code component, and a source-to-test-mapping component. The test-to-source-mapping component gathers code coverage data and uses it to create mappings between a test and the code that test exercises. After a developer revises a code base, our tool uses the differences-in-code component to figure out which files and functions have changed. Finally, the tool uses the previously created mappings and the source-to-test-mapping component to relate these changed files to their tests which are then selected to be run.

Implementation of our design took a long and winding course as our initially assigned Dell project was canceled. We switched from analyzing one of Dell's internal Java codebases to analyzing an internal R codebase to analyzing an open-source Twitter codebase called AnomalyDetection. During the implementation of our tool, we explored two approaches for mapping tests to source code. One used a custom script and the other used a pre-existing R code coverage tool. We initially went with the custom script because it was quicker but after more extensive testing, we switched back to the code coverage tool approach because it was more accurate.

We tested and evaluated our tool focusing on correctness and speed. We have 41 tests for our tool that helped us find simple bugs and increase our tool's robustness. We tested the performance of our tool using the AnomalyDetection codebase, which has 28 tests, and found that our tool took 1.6 seconds longer than running all tests. Since using our tool should be faster than running all tests, we did not meet performance goals. However, because AnomalyDetection is such a small code base, 38% of our tool's running time was spent on initialization, which is not expected to grow with the size of a code base. Ignoring the initialization time, our tool provided a 25% reduction in testing time. Thus, when used in a more appropriate situation, such as with a code base with hundreds of tests, our tool should provide a significant reduction in testing times.

The issues we had getting started on our project delayed any technical work until late in the first semester of this year. Development of the tool started at the beginning of this semester and although the test-to-source component took longer than expected we were able to finish our project on schedule with zero monetary expenses.

The main safety concerns for our project were to avoid modifying the user's code base and to recommend all necessary tests after each code revision. Modifying the user's code carries the risk of introducing bugs and failing to recommend necessary tests can lead to failures in real-world deployment.

Our recommendations to future developers are to replace the MySQL database with SQLite, add better integration with Git, support additional languages, and add options for outputting code coverage data.

Overall, our project is a success and meets the requirements set by our industry mentor at Dell. Our tool correctly recommends tests to be run after each code revision and is expected to reduce testing times when used with large code bases.

1.0 INTRODUCTION

Software testing is a critical part of development. Dell created this project to provide an improved testing setup for R code bases. Our goal was to make a more efficient and intelligent way of testing R software than was previously available to Dell. Our solution divided the problem into logical subsections, and the report provides detail on their functionality and our process of implementation.

The design problem addressed by our project concerns testing of R codebases and reducing unnecessary time expense. For a given change of code, some of the testing typically done is unnecessary and our project is a way of eliminating that. Our solution to that problem is in one deliverable with three components: a regression test selection tool that encompasses components for test-to-source mapping, differences in code, and source-to-test mapping.

Implementation of our project was altered by our initial attempts, as we modified several aspects (especially coverage measurement in the test-to-source mapping component) as we went along. Additionally, complications with the Dell sponsoring project led us to change our plans. Once it was finished, we tested the project piecewise and in full, with primary emphasis on the accuracy of its performance and secondary emphasis on its speed.

The time spent on this project was a constant consideration from a project management perspective. Especially given the disruptions early on in the first semester, as the original codebase had to be abandoned, we took care to ensure that we would finish the project in full on schedule.

After reviewing safety and ethical considerations of the project, we offer recommendations for anyone who would continue this project, as well as what we would have recommended to ourselves in hindsight. Finally, we close the paper with concluding remarks.

2.0 DESIGN PROBLEM

Software is constantly changing to include new features or satisfy new requirements. To ensure that new changes do not break previous functionality, developers must rerun the tests for all previous code that might be affected by the new changes. This process is called regression testing since it ensures that we do not regress and cause old bugs to resurface [1]. Since running all of the tests may take too long, developers can choose to run a manually picked subset of the total tests. However, poor test selection can lead to insufficient test coverage resulting in bugs going unnoticed until after the product ships [1]. Since bug reporting, diagnosis, and patching becomes much more expensive after the product is shipped, these bugs lead to vast increases in the cost of developing and maintaining software [2]. To avoid these problems caused by manual test selection, we have developed an automated regression test selection tool (herein referred to as the RTS tool) that ensures that all necessary tests are run each time a software application is changed.

2.1 Specifications

Our industry contact at Dell has specified one deliverable, a RTS tool for the R programming language. This tool is run by the user after they make a code revision and it automatically selects and runs tests specific to the changes in the most recent code revision. This tool is made up of three components that can each be run individually if the user wishes. The first component (test-to-source) maps test cases to source code, the second (diff-to-test) maps changes in code to test cases, and the third (source-to-test) maps source code to test cases. The tool as a whole and the three subcomponents each have unique input/output specifications. The tool will be compatible with Windows and Linux operating systems and performance will be measured by the accuracy of test recommendations and the speed with which it performs these recommendations. These specifications were developed through research of prior art and discussions with our industry mentor. Additional information is available in Appendix A.

2.1.1 *Input and Output Specifications*

Our RTS tool is made up of three components, each of which has unique input and output specifications that we have provided in Table 1. The main deliverable, the test selection tool, will act as a wrapper for the three components of the program. The user will interact directly with the

RTS tool. The tool will use the three components to check for code changes, regenerate source-to-test and test-to-source mappings, select tests that should be run, and run those tests. All output of each individual component is in the form of JSON. Our first component, the test-to-source tool, will take test cases, or a test suite, as input and output the functions in source code that are exercised by the input tests. The diff-to-test tool, Deliverable 2, will take one of two execution modes as input: 1) redraw mappings or 2) test selection. If selected to redraw mappings, the output will be a list of source and test files changed, if any. If selected to perform test selection, the tool will output a set of tests that should be run to exercise changed code, if any. The input to the third deliverable, the source-to-test tool, will be names of source code files, and the output will be the tests that exercise those files.

Table 1. Input and Output Specifications for Deliverable and Components

	RTS Tool	**Test-to-Source Component**	**Diff-to-Test Component**	**Source-to-Test Component**
Inputs	Developer's intended action	Test case name(s)	Execution mode (Redraw mappings or test selection)	Source code file name(s)
Outputs	Execution of regression tests	JSON report of the source code methods executed by each test case	Redraw mappings: List of source and test files changed. Test selection: Tests that should be run as regression tests.	JSON report of the test cases corresponding to each source code method

2.1.2 *Performance Specifications*

The performance of the final regression test selection product will be based on accuracy and speed. Since the main objective is to decrease testing time by only running relevant tests, the final product will meet performance expectations if the time required to select and run these regression tests is less than the time required to run all the tests [1]. Additionally, the output of the other components used to map test cases to source code files (and vice versa) will be inspected to see how accurately it identifies relevant sections of code.

2.1.3 *Operating Environment Specifications*

The tools are meant to be used during the software development life cycle by Dell's testing team and developers. Dell develops their products in Windows and Linux, therefore our tools will be compatible with both types of operating systems.

3.0 DESIGN PROBLEM SOLUTION

We are using Python as our implementation language, a MySQL database for storing mappings between runs, and R code analysis tools that will assist in creating mappings of source code and tests. After experimenting with different methods of creating our tool, we have now completed proof-of-concepts for the three components that form the RTS tool and tested them on the AnomalyDetection codebase. The AnomalyDetection codebase is described in more detail in section 5.2.1, Performance Testing Methodology.

3.1 Design Concept

Our final design, shown in Figure 1, of a regression test selection (RTS) tool for the R programming language is an overall tool, composed of three distinct components: (1) a test-to-source mapping tool, (2) a diff-to-test mapping tool, and (3) a source-to-test mapping tool. While prior art research helped us make informed design decisions, we drafted our own modular design based on Dell's requirements and our perceptions of developers' needs for a testing tool. We chose a modular design because it clearly encapsulates related functionality, is easier to maintain and test, and allows us to build separate tools that each fulfill one of Dell's requirements. Furthermore, the design allows our end users to use any of the first three components directly or through a RTS tool that provides a simpler interface.

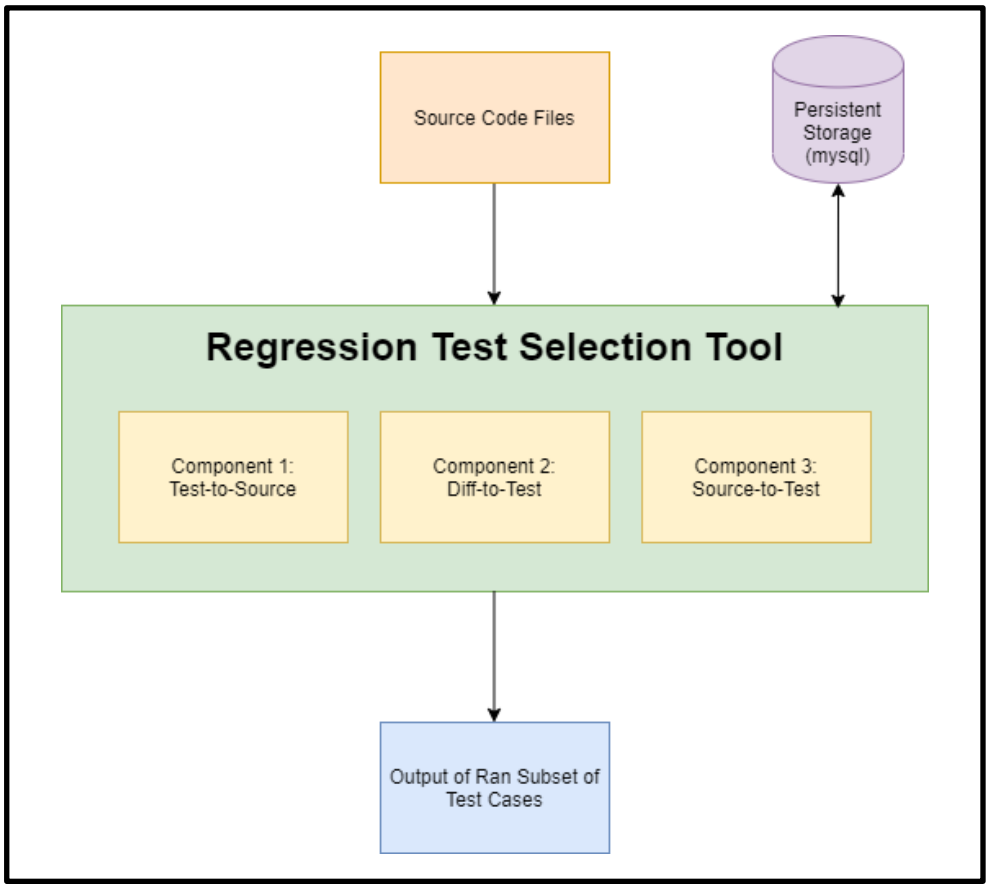

Figure 1. High level Overview of Regression Test Selection Tool

3.1.1 Regression Test Selection (RTS) Tool

The RTS tool, as detailed in Fig. 2, combines three components: a test-to-source component, a diff-to-test component, a source-to-test component, to create a simplified command line interface for Dell's engineers to interact with. The RTS tool should be run after a user makes changes in code, then commits them. At this point, this tool will: 1) invoke the diff-to-test component to compare the current revision to the previous one, 2) invoke the test-to-source component to regenerate mappings for any of the changed files, and 3) invoke the source-to-test tool to find which test cases map to the changed code, and then 4) run the selected subset of tests.

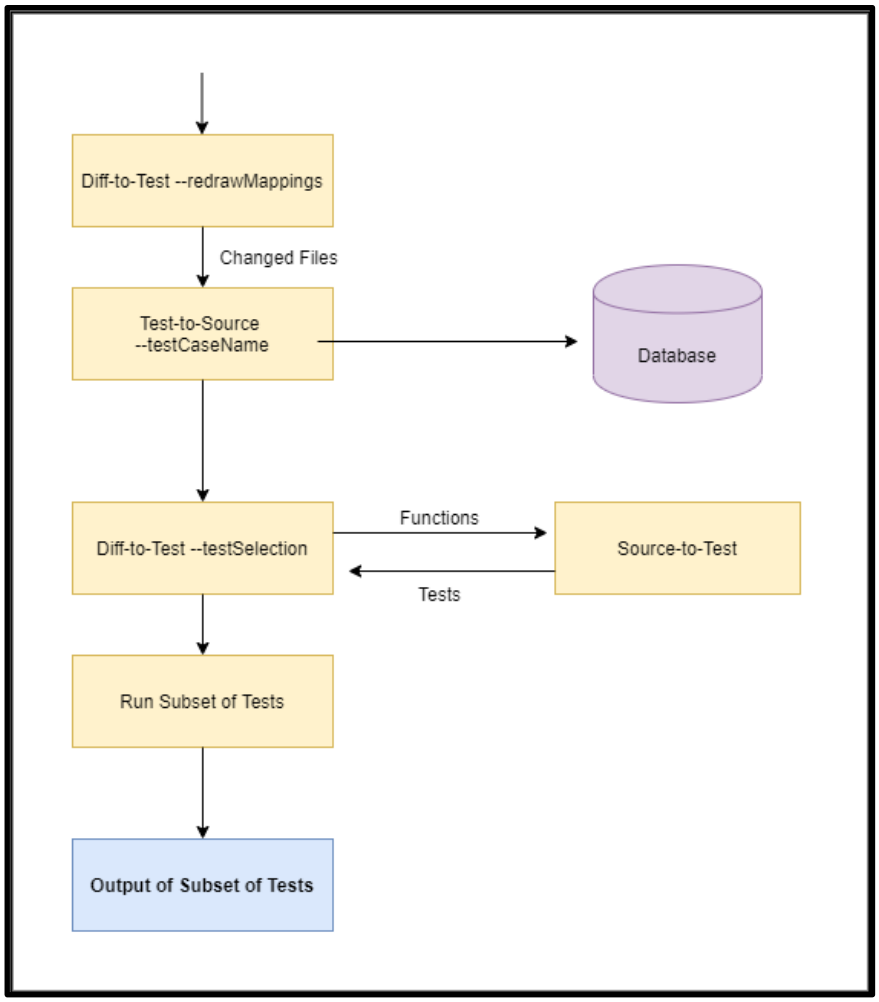

Figure 2. Regression Test Selection Tool

The first step of the RTS tool is to run the diff-to-test component with the mode set to redraw the mappings, which will result in a list of files that have been changed. These files will be given as input to the test-to-source component , which will redraw mappings from any functions in those files to test cases and store the new mappings in the database. Then, the diff-to-test component is run again with the test selection mode, which will communicate with the source-to-test tool to get the updated mappings, now stored in the database. It will then run the resulting tests.

3.1.2 Test-to-Source Component

In order to reduce time costs, this component of the RTS tool maps specific tests to the areas of the source code that the test exercises, as detailed in Fig. 3. The input is a specific test case, or test suite, and the output is a list of which functions from the source code were covered by the

tests. Using a parsing method implemented in Python, the test-to-source tool uses a MySQL database to keep track of what tests have been run. By interfacing with the diff-to-test tool, this tool will be able to find out if the code or tests have been changed, and if so, it will regenerate mappings. Otherwise, it will use the already stored mappings in the database to output the exercised functions.

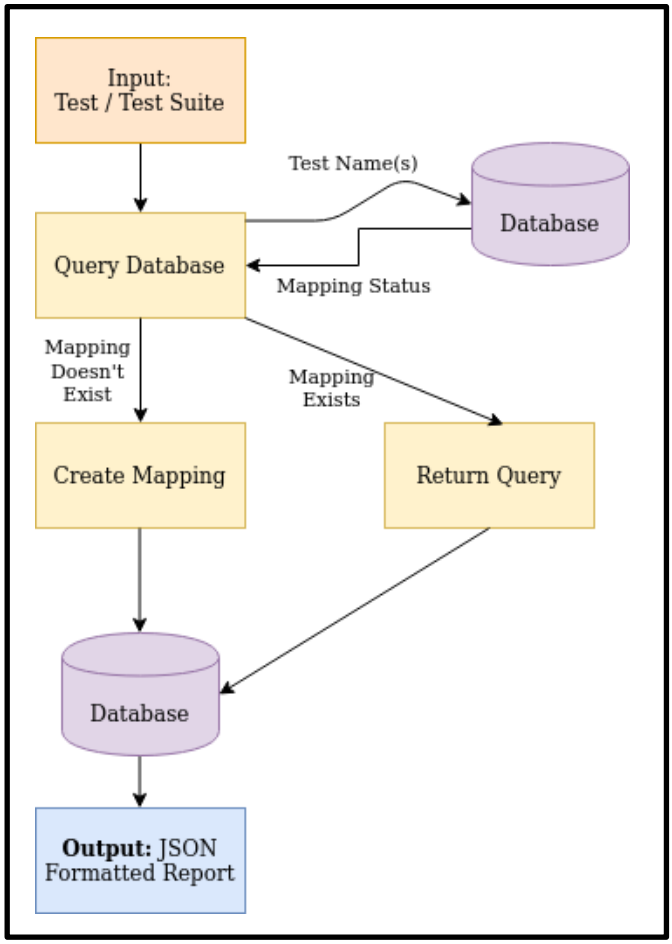

Figure 3. Test to Source Mapping

Although Python is our implementation language, R is the source code language, so we rely on both R scripts and Python processing for implementation. We tried out two different methods of accomplishing this goal, and selected the one that was best for accuracy, as that is our most important metric.

The implementation method we are now using relies on the code coverage tool covr. We utilized covr last semester to produce code coverage reports with regards to several R code repositories,

and worked to adapt it to the Anomaly Detection source we've specified. Before running any R commands, we first split up the tests. Our algorithm goes through all of the test cases in the repository's test directory and splits up the file into separate tests. For example, a file named test1.R could contain 4 tests. We would enumerate these as test1-0.R, test1-1.R, test1-2.R, and test1-3.R. When we map from test to source, we then map at this lower level, as opposed to the entire file. Otherwise, we could be running unnecessary tests, which is the problem that our tool is attempting to solve.

Then, we use an R script to get a list of every function name in the code repository. We then dynamically generate an R script which measures code coverage percentages between each test case and function, run it from Python and redirect its output to a text file. Back in Python, we process that text file to find the coverage percentage for each pairing and if the coverage percentage is greater than 0, classify it as a mapping and store it in the database.

The user can input a specific test name and all of the functions that test calls will be returned. There is also a command line parameter which allows the user to say if the program should go through the mappings process or not. If so, it redoes all mappings and stores any new ones in the database. If not, it queries the database for the given test.

3.1.3 Diff-to-Test Component

The purpose of this component is to find the changes made in any specified codebase. As shown in Fig. 4, this diff-to-test tool has two execution modes. The first execution path will take a revision of code and select associated tests. The second execution path will inform the RTS tool of files that have been changed. If any changes have been made, the tool will request that a new mapping be generated.

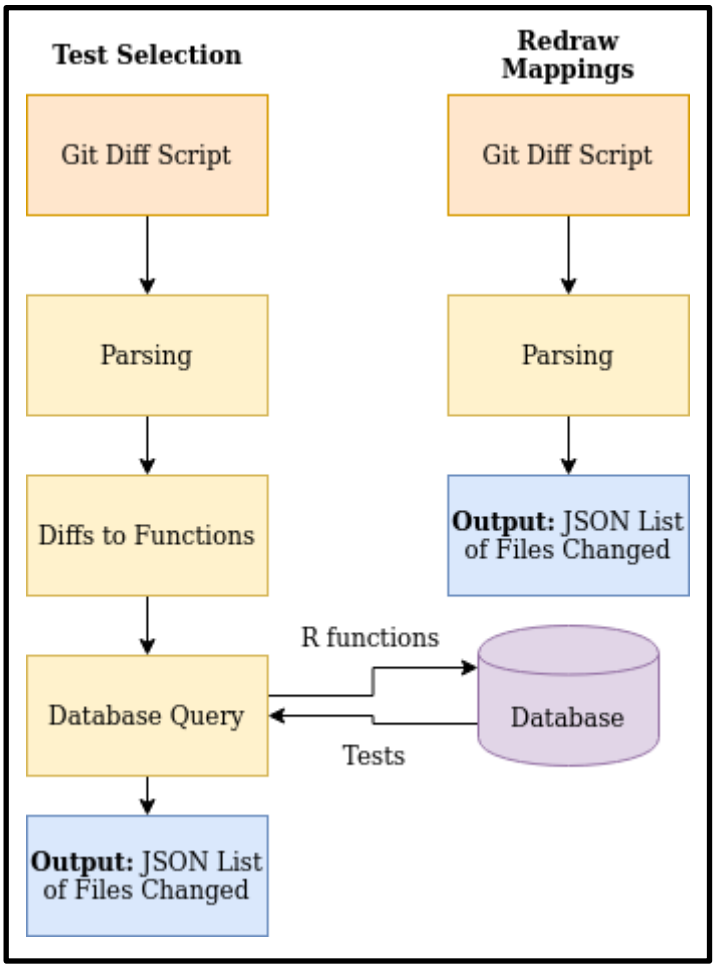

Figure 4. Code Diff to Tests

The first execution mode, test selection, will recommend the tests based on areas of changed code, so that only certain fragments of the program will be exercised. Git will be the source code's version control system, and this tool will integrate with Git's built in revision information tools. This tool will reference the mappings in persistent storage to produce a list of tests that exercise changed areas of code. Firstly, Git's built-in revision information tools are used to review the changes made in source code using the "git diff" command, and a list of code changes are compiled. Secondly, the output is parsed to yield a list of the specific functions that have changed. Thirdly, the database is queried for their source-to-test mappings, and outputs a list of associated tests to run. This tool is implemented in a single Python wrapper script so that it can be called by the overall RTS tool.

The second execution mode, redrawing mappings, is invoked by the RTS tool to update the mappings stored in the database. This will occur after any code change so that the database (and therefore test selection) will be up to date. The process begins by querying Git for code changes. Then, the output is parsed and a list of files changed is returned to the RTS tool. The RTS tool will then use that to conduct a targeted update of the mapping database, using the test-to-source component.

3.1.4 Source-to-Test Component

The source-to-test component returns the tests that are mapped to any specified source code files. Fig. 5 shows the setup of the source-to-test component. The input is the name of an R source code file, and the output is a list of the tests that exercise the source code file. The tool will gather the information about the source to test mappings from the MySQL database. These mappings will already have been generated by the test-to-source component. The output is a JSON report that shows all functions and test cases that map to the source code file.

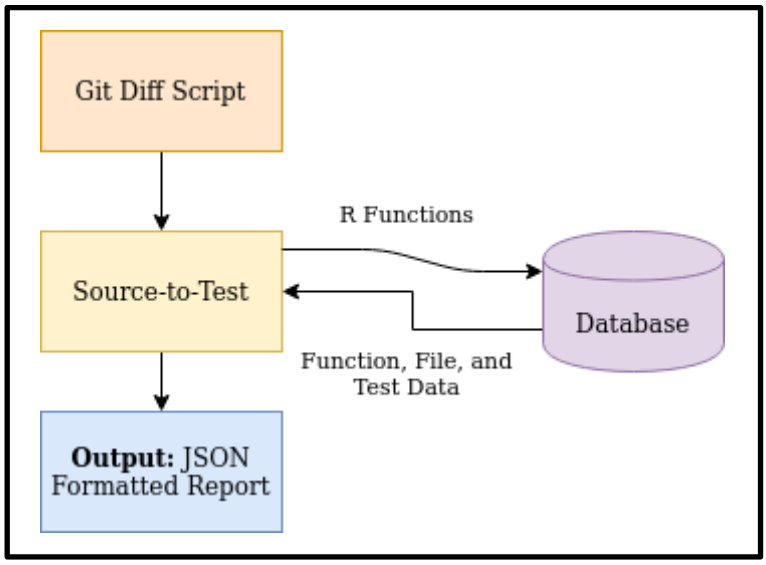

Figure 5. Source to Test Mapping

4.0 DESIGN IMPLEMENTATION

Initially, our project was supposed to operate on a Java-based codebase of Dell's OpenManage Enterprise. As per our preliminary project design, the input to our tools would consist of the provided Java codebase, its test and revision history, and code coverage reports run through

JaCoCo. Shortly into the semester, the project support for the Java-based codebase was rescinded, and our team, along with the guidance of our industry mentor, realigned our project to operate another codebase named Q. This codebase is based in the R programming language as opposed to Java. The transition of our project into an R based project was resolved by our decision to use the covr coverage tool instead of JaCoCo, and maintain our mappings in a MySQL database. The implementation of the actual project components and overall RTS tool would still be performed in Python.

While the Q codebase consisted of robust source code implementation, the shortage of tests within the codebase led us to explore other open source projects on GitHub on which we could model and test our project. In particular, we moved toward a codebase called AnomalyDetection, an open source project by Twitter. Using AnomalyDetection, we implemented the technical aspects of our project, including the three components test-to-source, diff-to-test, and source-to-test. Each of the three components was implemented in a subteam of two members for parallel development, and efficient implementation of modular testing.

For the test-to-source tool, we needed an approach to determining the mappings between test code and source code. As there were two members of this component's subteam, we decided to have each member work on an implementation method in parallel to evaluate which would better fit our metrics of correctness and performance. The method we chose to go with was the covr code coverage tool, detailed in the design solution. The other method we tested relied on code parsing.

This alternative method involves using open-source functions that we found through research and put in a Python wrapper. The first step of the tool is to run an R script which goes through the input code repository and finds all code files and each of the functions defined inside them. It returns these values to the Python script, which then stores any new information in the database. The second step of the algorithm is an R script which maps R functions to each other. This part of the code was found through research; it uses no library, but instead inspects the code to map each function to the ones that it calls. The parser finds any mentions of functions, which creates a

mapping of a test to the functions that it directly calls. We then use the mapping from step 2 to create a mapping from a test to all the functions that it calls both directly and indirectly.

After completing implementation of this method, we realized it contained a couple of major problems: the test to function mapping was done based on parsing. So, if a function call was conditional, or even if a functional call was commented out, our tool would create a mapping regardless, resulting in incorrect results.

For the diff-to-test and source-to-test tools, the implementation was done according to test driven development. For example, for the diff-to-test component, the subteam worked on building units of code that performed minimal functions, such as writing a function in a bash script that calls "git diffs" and returns the line numbers changed in the code. Similarly, for the source-to-test component, the subteam used unit tests to build up the database interface for their component, using object relational mapping. In both cases, these individual units were then tested extensively before integrating together to form the overall component.

Since this project requires no monetary investment, the two major metrics were correctness and speed, with correctness being more important. This alternative method was significantly faster than the method detailed previously, but as it was less accurate, this benefit wasn't worth the price of incorrect mappings. After discussing this trade-off with our industry mentor, we decided to go forward with the implementation detailed in 3.1.

5.0 TEST AND EVALUATION

Our tests focused on correctness and speed. For testing of individual and combined component correctness, we wrote unit and integration tests to check the output of each component. To test the speed of our tool—the tool being the overall RTS tool that combines all components—we manually recorded our tool's runtime to compare it against the default solution, running all tests. A more detailed explanation of how we performed these tests is provided in this section. We break down our methodology and results for testing correctness in sections 5.1.1 and 5.1.2 and for testing performance in sections 5.2.1 and 5.2.2.

5.1 Testing for Correctness

A RTS tool's correctness is more important than its speed. If the tool outputs false positives—i.e., tests that do not need to be run since they are not related to the changed code—then its performance will suffer. On the other hand, failing to recommend a test that is related to the changed code, i.e., a false negative, could result in shipping buggy code to customers. As more code and tests are added, these false positives and negatives become more problematic while performance actually increases. Therefore, we spent most of our time and effort creating a large test suite that covers correctness.

5.1.1 Methodology

To test component correctness, we wrote unit tests for each component. Each unit test is made of three parts: an input for the component code, the output that results from running the component with the input, and a known-to-be-correct output that is checked against the component's output.

As an example, Fig. 8 shows one of our unit tests for the Source-to-Test component. This unit test checks that calling the function "SourceFile.get_by_file_path" on a valid source file returns the names of all test cases that exercise the input source file. The input to this unit test is the source file name, "one.r." The output of the component is the data in "sourceFile.testCaseNames." The correct, expected output is stored in the variable "expected" and checked against the component's output on the last line.

```
def test_gets_the_test_names(with_database):
    sourceFile = SourceFile.get_by_file_path("one.r")
    expected = set(["first_test", "second_test"])

    assert expected == set(sourceFile.testCaseNames)
```

Figure 6. Example unit test.

5.1.2 Results

We have 41 tests that cover all use cases of our components. Besides just checking for correct behavior, these tests also cover functionality such as creating database entries on initial runs, displaying help and usage messages, and handling corner cases and errors. Since most of the

tests were written early during the development process, they helped us find simple bugs and increase our tool's robustness from the very beginning.

5.2 Performance Testing

The performance goal is to save companies time in testing software. This means that our tool should run quicker than running all tests. If our tool results in slower testing times than running all tests, then there is no benefit in using our tool; developers would just run all tests since doing so is safe—i.e., there is no chance of missing a test—and requires no extra effort or set up. Accordingly, we aimed for our tool's runtime to be less than the runtime of all tests. The rest of this section discusses how we recorded and compared these runtimes.

5.2.1 Methodology

We tested the performance of our tool on Twitter's AnomalyDetection code base because it is easy to set up and has decent test coverage. The code base is a statistical R package used for detecting irregular behaviour in time series data, e.g., peaks or dips in the volume of tweets [3]. Twitter uses AnomalyDetection to measure system metrics after releasing new updates and during A/B testing [3], and other companies use it for damage control—a sudden, large amounts of tweets referencing a company is often a bad sign.

We measured the overall runtime of our tool by breaking our tool down into initialization time, mapping update time, and test time as shown in Equation 1. Initialization time is the time our tool takes to start up, call code from R packages, and figure out what code has changed. Mapping time is the time to create and update mappings for any changed code; if new tests are added, then our tool must gather code coverage data and create new mappings for these tests. Finally, we measure the time to run the selected tests.

$$Total\ run\ time\ =\ I\ +\ M\ +\ T$$

$$I\ =\ Initialization\ and\ identification\ of\ changed\ code$$

$$M\ =\ Time\ to\ create\ and\ update\ stored\ mappings$$

$$T\ =\ Time\ to\ run\ all\ selected\ tests$$

Equation 1. Equation for measuring performance.

To come up with realistic, unbiased code changes to test our RTS tool on, we used the Git command line tool [4] and previous code history to simulate developers' workflows. Using Git, we reverted some of the most recent developer changes, ran our tool to create the first mappings (i.e., the "I" from Equation 1), re-applied the reverted developer changes, and then ran the regression test selection process.

5.2.2 Results

Overall, the performance of our RTS tool is better than we expected leading us to believe our tool would save developers time when used with larger code bases. The results in Table 2 , as shown on the next page, show that our tool's runtime is only 1.6 seconds longer than the runtime of all tests. While we did not meet the standard set by Eq. 1 for the AnomalyDetection codebase, this slight time runover is due to the small number of tests (28) for the codebase.

Method	Time taken by one-time setup (sec)	Initialization Time (sec)	Time to create and update mappings (sec)	Time to run all selected tests (sec)	Total test runtime (sec)
RTS Tool	52	3.36	0.1	5.36	8.8
Running all tests	0.0	0.0	0.0	7.2	7.2

Table 2. Time comparisons (averaged and in seconds) for our RTS tool versus running all tests.

We believe that using our tool on a codebase with more tests would result in greater time savings. Since there are only 28 tests in AnomalyDetection, our near-constant initialization time (3.36 seconds) takes up 38% of our tool's total runtime. If we ignore this initialization time, the runtime of our tool is actually 1.84 seconds or 25% faster than running all tests. If we used our tool on a codebase with hundreds or thousands of tests, our initialization time would remain

small and become negligible as more tests are added. This is because the time to identify changed functions and query the database does not increase much relative to the amount of tests added. To better illustrate this, imagine running a RTS tool on a codebase with only one test. The time to run the one test plus the time to run the RTS tool is clearly greater than the time to run just the one test. Now imagine the other extreme: a codebase that has thousands of tests. With this code base, the time used by RTS plus the time of, say, even running 30% of the tests will be an improvement over running all thousands of tests.

6.0 TIME AND COST CONSIDERATIONS

We were able to deliver a finished product on schedule with zero monetary expenses despite several setbacks and initial difficulty in getting the project off the ground. Our main roadblocks were getting access to the Dell resources needed to start development, finding a code base to work with, and implementing the test-to-source component of our tool.

The challenges we faced in getting a language and code base to work with delayed the start of any technical work until late in the first semester. At that point we switched to an open-source code base to sidestep many of the issues we faced in getting access to Dell resources. After we switched to using AnomalyDetection to test our project, we were able to complete our risk reductions on time, the results of which saved us a significant amount of development work.

We started development at the start of the spring semester, and from that point onwards, we met all technical deadlines except for completion of the test-to-source component. This component constituted the majority of our development work. Determining how we should track function calls was a challenge, and we considered multiple options. Although we met our initial deadlines for this tool, testing showed that our implementation had major flaws. Despite this we were still able to finalize our project and wrap up development on schedule.

7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN

Our main safety concerns were maintaining the integrity of the user's code and ensuring that our tool's test selection process did not exclude any relevant tests.

We designed our tool so that it would not need to make any modifications to the user's code base. Any time software is modified there is the potential for a bug to be introduced, therefore, it was vital that our tool did not modify the user's code in any way.

We tested our code extensively to ensure that our tool did not fail to recommend necessary tests for a given code revision.If our tool failed to recommend a test case that would have found a bug in the new code revision, the user might end up shipping a faulty product.

8.0 RECOMMENDATIONS

Looking back on our project, there are several changes that we would recommend to future developers, and even to our past selves, to improve the user experience when using our tool.

When looking at the test-to-source and source-to-test components we would recommend a higher degree of control for the user when specifying the creation of mappings. If our tool was to expand and integrate with other languages such as R, Java, C, and C++, these components would need to specify options for both percentage and mapping based tools to allow language-independent usage, since code coverage tools can vary in their implementation.

For the diff-to-test component, we would like to give users a wider selection of previous code versions to choose from rather than just the most recent version. In the current version, this component returns code changes between the two most recent Git repository versions, which aligns with the majority of use cases. By adding support for different branches and commits to compare to the current code revisions, this component would allow testing across a wider range of source code revisions.

Because we are using a code coverage tool to generate the mappings, the project could also be extended to output this information and suggestions about coverage to the user. For example, if our tool found that a function was left completely untested—i.e., the function has 0% code coverage—our tool could tell the user that they have gaps in their testing coverage.

The largest revision we would make would be changing our persistent storage solution. Although

interacting with MySQL was a good learning experience, setting up and maintaining the database can be somewhat cumbersome for our tool's users. If we switch to SQLite for local storage, we could store the mappings in the Git repo and allow users to pull down mappings associated with commits to avoid having to redraw mappings on each machine.

9.0 CONCLUSION

Our project has succeeded in meeting the requirements set forth by our industry mentor at Dell. Given a changed code base, we are able to call our finalized RTS tool to analyze code changes made, and return a subset of tests that must be run to exercise functionality of the changed code areas. Through test and evaluation of our methodologies, we have shown that in a scaled environment where there are many test cases, our tool would allow users to spend less time running tests than through brute force methods. Ultimately, this project will not only increase the quality of software at Dell, but also reduce product-testing workload and better the Dell developer experience.

REFERENCES

[1] M. Gligoric, “Regression Test Selection,” Ph.D. dissertation, Dept. Comp. Sci., University of Illinois at Urbana-Champaign, Champaign, IL, 2015.

[2] The National Institute of Standards and Technology, (2002, June 28). “Software Errors Cost U.S. Economy $59.5 Billion Annually, ” [Online]. Available: http://www.abeacha.com/NIST_press_release_bugs_cost.html

[3] Twitter, AnomalyDetection, (2015), GitHub repository, https://Github.com/twitter/AnomalyDetection.

[4] L. Torvalds, (2005), https://Git-scm.com/.

APPENDIX A – RELEVANT STANDARDS

APPENDIX A – RELEVANT STANDARDS

We have adopted a lot of our standards from the standards listed in Dr. Milos Gligoric's Ph. D. dissertation over regression test selection (RTS) [1]. The dissertation recommends four standards for regression test selection tools: efficiency, safety, precision, and performance. Efficiency is met when selecting tests takes less time than the total running time of unselected tests. Safety is met when an RTS tool can guarantee that all unselected tests are unaffected by the changes in the new code revision. Precision, the counterpart to safety, is met when an RTS tool can guarantee that all selected tests are affected by the changes in the new code revision. To meet the performance standard, the time a RTS tool takes to select and run tests and collect the results must be shorter than running the entire test suite. If running all of the test cases takes less time than it takes to run our RTS tool, then developers would have no need for the RTS tool.