DATE: February 28, 2020

TO: Dr. Emanuel Tutuc, Mr. Oliver Davidson

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

SUBJECT: Triaging, Diagnosing, and Predicting Software Errors Updated System Design

OVERVIEW OF DESIGN OBJECTIVES

Our project aims to create a regression test selection tool for the R programing language. This tool will recommend specific tests for a developer to run when they make a change to a software application, so that they can verify the correctness of their changes without running the entire test suite of the application. Our main deliverable for this project will be a testing harness made up of three distinct modules, each relating to a mapping of source code to tests. The success of the project will be measured by the accuracy of the tool's test recommendations, the speed at which it performs its tasks, and compatibility with Windows and Linux operating systems.

Design Problem

Software is constantly changed to include new features or satisfy new requirements. To ensure that these changes did not break previous functionality, software developers will run tests to check their correctness. This process is called regression testing [1]. Because regression testing is a time consuming process, developers will often choose to run a subset of the total tests for a software application. However, poor test selection can lead to certain areas of the application not being properly verified, and can cause certain bugs to go unnoticed until after the product ships [1]. This then leads to vast increases in testing and maintenance costs, as bug detection, reporting, and patching become much more expensive the further down the line bugs are identified [2]. Automated regression test selection can mitigate these problems by ensuring that all relevant tests are run each time a software application is changed.

Design Updates

Since our project relies primarily on the mapping of source code to tests, it is crucial that any codebase we use has both good code and good test cases. Upon inspection of the Q source code provided by Dell, we found that it did not contain any tests, and would not be sufficient for the development of our project. Upon starting work again this semester, the situation was exactly the same, and we realized that we needed to adjust our trajectory. One possible plan was to work on writing test cases for Q, but that course of action is outside of the scope of this senior design project. We instead decided to explore other options for codebases. We spoke with our industry mentor about this decision, and he agreed that it was the best path going forward. Any good software tool should work for more than one codebase, and with the method we decided on, we will be testing our tool with a variety of codebases from the start.

We found several open-source, R language codebases, and after examining each, decided on two of them. One of the codebases, by Twitter, is called AnomalyDetection, which detects anomalies in different types of data. This project is small, with only 5 code files and 4 test files. This codebase is what we are initially testing our tool on: the benefit is that since it's small, it serves as a proof of concept, and will be simple to manually check results on. The second repository is called Shiny, by RStudio, and it creates web applications. This codebase is very large, but very well tested. This will be useful for seeing how our tool scales. As tests are added to the Q repository, we will make sure that our tool successfully integrates with that project too.

Deliverables

The team's industry contact at Dell has specified four deliverables for our project:

1. The first deliverable is a module that maps a software application's tests to its methods. To create this mapping, the module will run the test suite of an application and track which functions are called by each test. An example application and a model of its interactions are shown in Fig. 1.

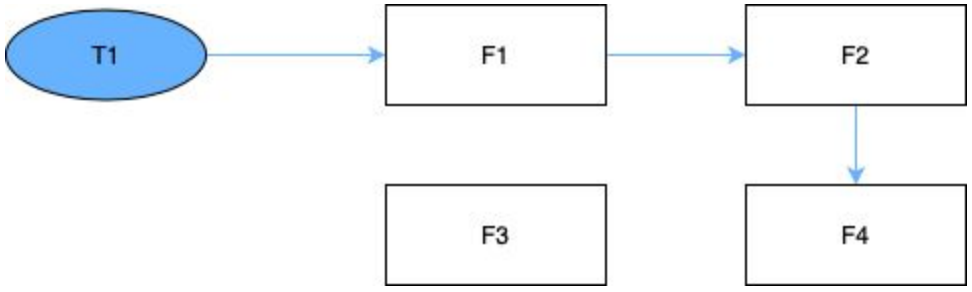

Figure 1. Test to Source Mapping

This application contains one test T1 and four functions F1, F2, F3, and F4. T1 calls functions F1, F2, and F4 but does not call F3, therefore, this module would create a mapping of T1 to F1, F2, and F4.

2. The second deliverable is a module that specifies which tests should be selected for regression testing after an application is modified. This module uses the mappings created by the previous module to connect modified functions with the tests that execute them. Then, the module outputs all tests found in the previous step. An extension of the previous application, with a new test T2 that calls functions F3 and F4 is shown in Fig. 2.

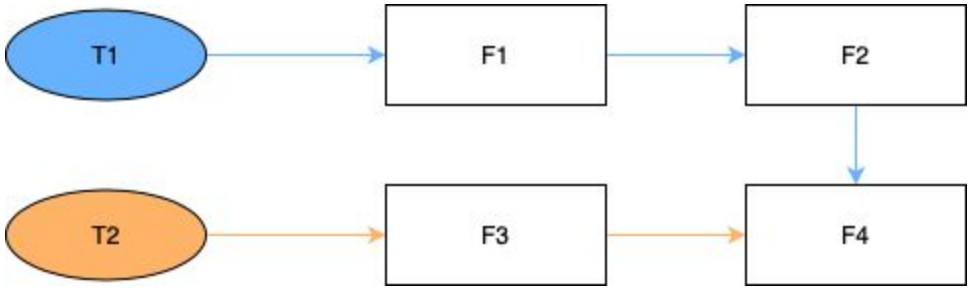

Figure 2. Test to Source Mapping (Multiple Tests)

In this example, if function F4 is modified, then tests T1 and T2 are selected for regression testing. Alternatively, if function F2 is modified, then only test T1 is selected.

3. The third deliverable is a module that takes a source code file as input and returns the tests that exercise the functions in the file. This module also uses the mappings created by the first module. For every function in a source file, this module looks up the tests associated with it and outputs each one. An associated test is one that calls any of the functions in the given file. Fig. 3 extends the example application again so that it has two sources files S1 and S2. S1 contains functions F1, F2, and F4 while S2 contains function F3.

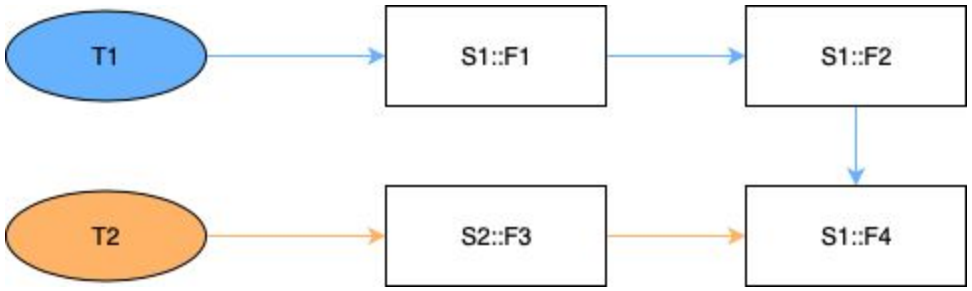

Figure 3. Test to Source Mapping (Multiple Source Files)

Given source code file S1, this module would return tests T1 and T2. If S2 was the input, then only T2 would be returned.

4. The final deliverable is a testing harness that combines all of the previous modules. The harness is run by the developer after making a code revision and automatically selects and runs tests specific to the changes in the most recent code revision.

Specifications

As described above, our project is an regression test selection tool that consists of four modules with each having 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.

Input and Output Specifications

There are four input-output specification pairs provided, one for each deliverable, as outlined in Table 1. The test-to-source module, deliverable 1, will take test cases (or a test suite) as input and output the functions in source code that are tested. The diff-to-test module, deliverable 2, will take its execution mode as input. If selected to redraw mappings, the output will be a list of source and test files changed, to be later used by module 1 in redrawing mappings. If selected to perform test selection, the revision history as provided by git will be used to output a set of tests that should be run.The source-to-test module, deliverable 3, will have the input of source functions and output appropriate tests. Each of the modules will present their outputs as a JSON report.

The fourth deliverable, the testing harness, will act as a wrapper for the remainder of the program. The developer will interact directly with the testing harness. The harness will check for revisions of code, regenerate source-to-test and test-to-source mappings if necessary, select tests that should be run, and run those tests.

Table 1. Input and Output Specifications for Items From "Deliverables" Section

	Deliverable 1 (Test-to-source-code mapping)	**Deliverable 2** (Diff-to-Test mapping)	**Deliverable 3** (Source-code-to-test mapping)	**Deliverable 4** (Harness)
Inputs	Test case name(s)	Execution mode (Redraw mappings or test selection)	Source code file name(s)	Developer's intended action
Outputs	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	Execution of regression tests

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.

Performance Specifications

The performance of the final product will be based on accuracy and speed. Upon discussion with our industry mentor, accuracy is the more important of the two metrics. Performance will be contrasted against the alternative of running the code's entire test suite.

Accuracy will be measured in terms of relevance of tests selected. After each code revision, the tool should specify a minimal number of tests that fully cover the affected portions of code. This means that the code will be validated without additional unnecessary tests being run.

Speed will be measured in terms of time relative to executing the entire test suite. This will be more pertinent for large code bases with a large number of tests. Eliminating the execution of extraneous tests would allow our tool to take less total time than executing the entire test suite [1]. We will measure our total run time using the three variables outlined in [1] and as shown in Equation 1 below. For small codebases, the additional time to deploy our tool would partially offset the efficiency of selecting targeted tests, and thus speed is not as important a metric for our project when applied to small code bases.

$$I = Harness\ initialization\ and\ test\ selection\ time$$

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

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

$$(Eq.\ 1)\ Total\ run\ time = I + T + M$$

DESIGN SOLUTION

We are using Python as the implementation language, a MySQL database for storing the outputs between runs, and R code analysis tools which will assist with creating mappings of source code and tests. After experimenting with different methods of creating our tool, we have now completed proofs-of-concept for the three modules that form the testing harness, and tested them on the AnomalyDetection codebase.

Design Concept

Our final design—shown in Fig. 4—of a regression test selection (RTS) tool for the R programming language is composed of four distinct modules: (1) a test-to-source mapping tool, (2) a diff-to-test mapping tool, (3) a source-to-test mapping tool, and (4) a test harness that combines the previous three tools. We chose a modular design to allow our end users to use any of the first three modules directly or through a testing harness that provides a simpler interface.

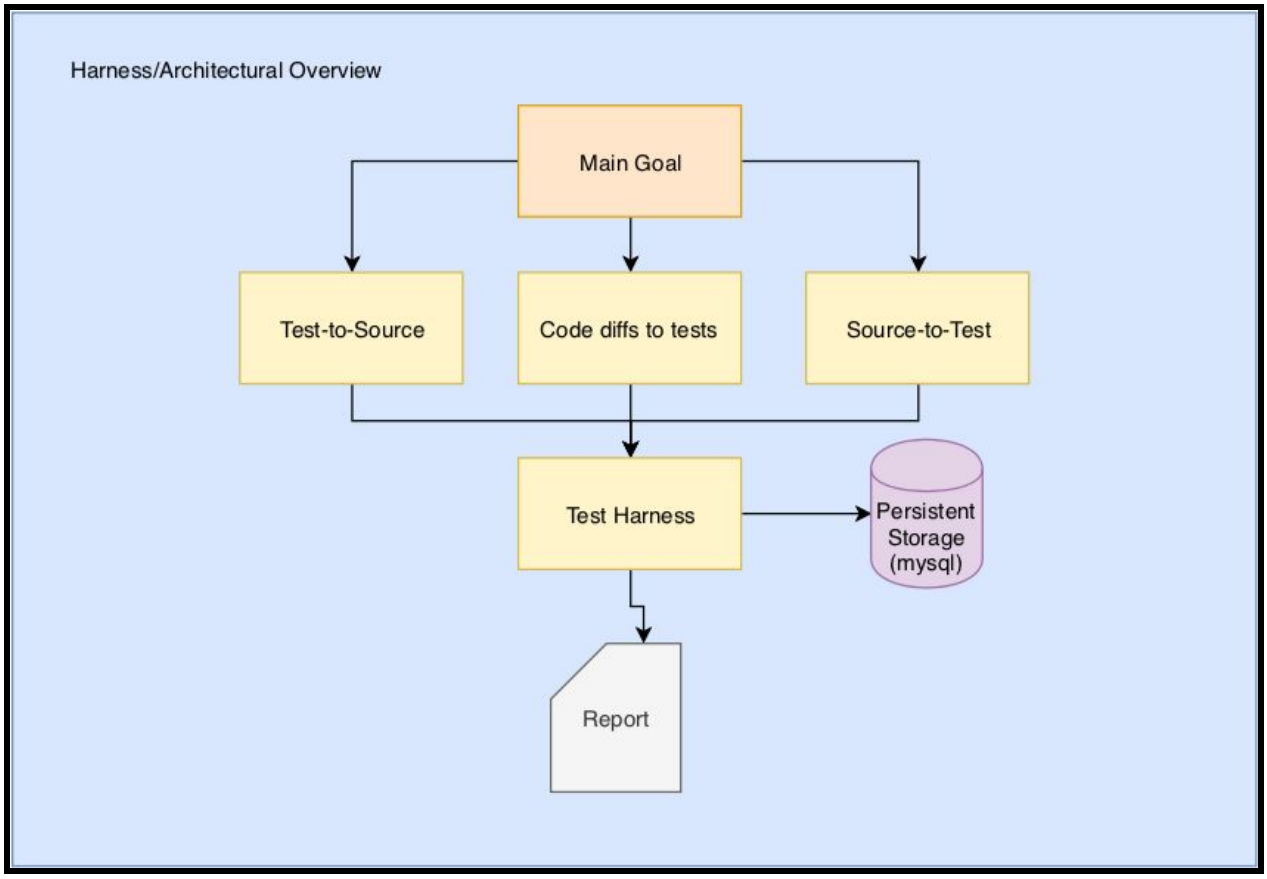

Figure 4. High level overview of our design for a regression test selection tool

Module 1: Test-to-Source Mapping

In order to reduce time costs, this module maps specific tests to the areas of the source code that the test exercises, as detailed in Fig. 5. The input to the module is a specific test case, or test suite, and the output of the module is a list of which functions from the source code were covered by the tests. Using a parsing method implemented in Python, the tool uses a MySQL database to keep track of what tests have been run. By interfacing with module 2, this module 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.

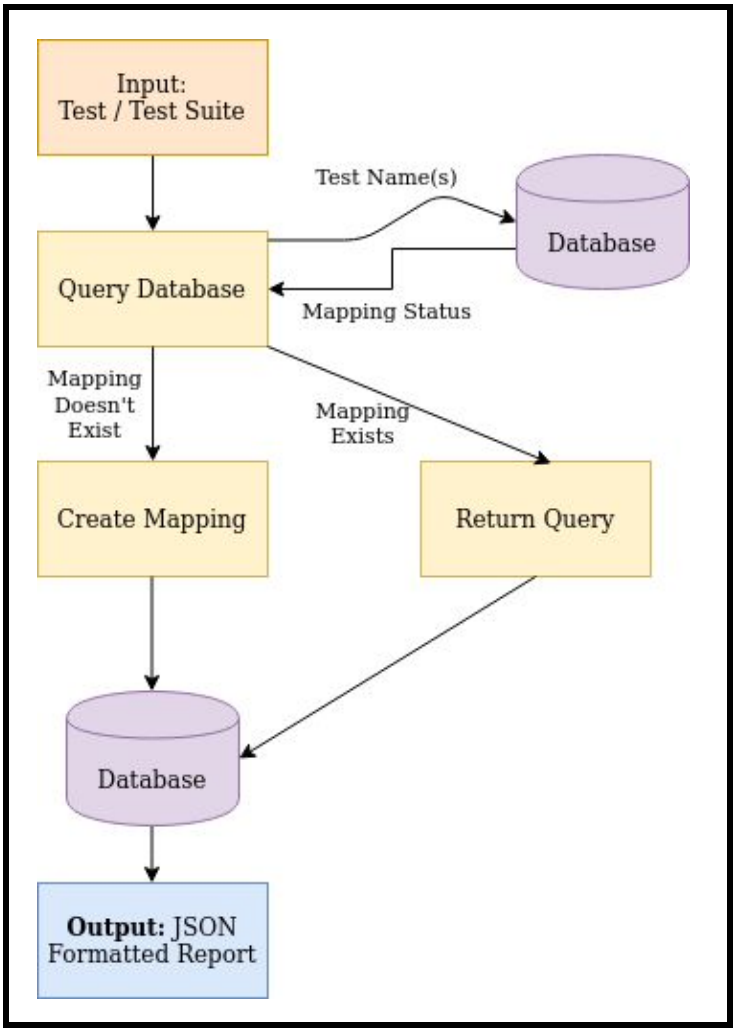

Figure 5. Test to Source Mapping

An implementation of this module has now been completed. 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 first implementation method was using the static analysis tool covr. We utilized this tool 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. In order to reach the

desired output of code coverage percentages for each of the test files with regards to source code functions, we specified a method level granularity within the tool by file comparison. Running the covr python wrapper script on the AnomalyDetection codebase had a significant overhead and produced an incorrect output. We have decided to move forward with the second implementation, as it produced accurate mappings in less time.

The second implementation method involves using open-source functions that we researched and combining them with Python processing tools. The final result is a single Python script which runs helper R scripts as needed. The first step of the module is 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. Finally, the third step of the algorithm involves Python parsing. It 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. Once each test is split up, 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.

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.

The second method has been tested on the Anomaly Detection codebase and the results have been verified.

Module 2: Differences in Code

As shown in Fig. 6, this module 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 harness of files that have been changed. If any changes have been made, the module will request that a new mapping be generated.

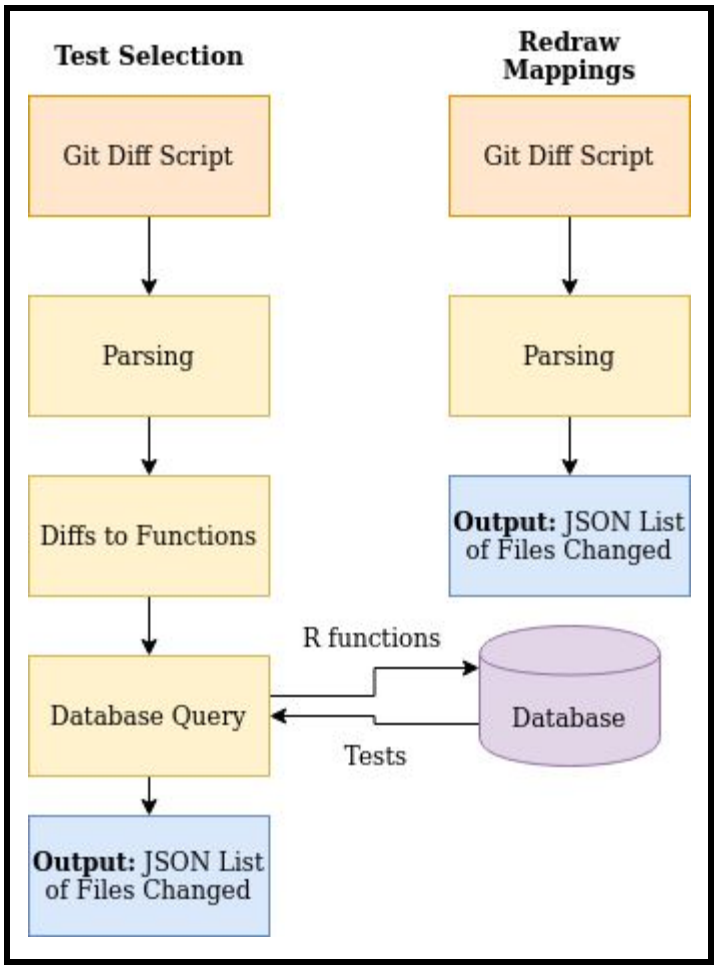

Figure 6. Code Diff to Tests

The first execution mode, test selection, will recommend the tests based on areas of changed code, so to exercise only certain fragments of the program. Git will be the source code's version control system, and this module will integrate with Git's built in revision information tools. This module will reference the mappings in persistent storage created by other modules to produce a list of tests that exercise changed areas of code. This module will be conducted in three steps. Firstly, the module uses Git's built-in revision information tools to review the changes made in source code using the "git diff" command, and compiles a list of code changes. Secondly, the module parses through the output to yield a list of the specific functions that have changed. Third, the module queries the database for their source-to-test mappings, and outputs a list of associated tests to run. This module is implemented in a single python wrapper script so that it can be called by the harness.

The second execution mode, redrawing mappings, is invoked by the testing harness 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. The module will parse the output and return a list of files changed to the testing harness. The testing harness will then use that to conduct a targeted update of the mapping database, using the test-to-source module.

Module 3: Source-to-Test Mapping

Fig. 7 shows the setup of the source-to-test module. 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 module will gather the information about the source to test mappings from the MySQL database. These mappings will already have been generated by Module 1. The output is a JSON report that shows all functions and test cases that map to the source code file.

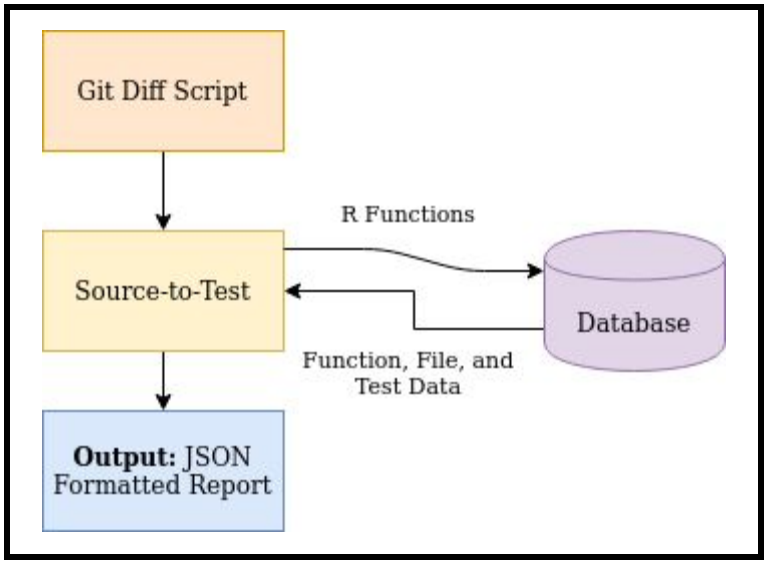

Figure 7. Source to Test Mapping

The implementation of this module has been completed and tested. For usage internal to the tool--i.e. usage by other modules--we have created a python class that can retrieve the source-to-test mappings by simply calling “SourceToTest.getJson(fileName)”.

Module 4: Testing Harness

The testing harness, as detailed in Fig. 8, combines the three previous modules to create a simplified command line interface for Dell’s engineers to interact with. The harness should be run after a user makes changes in code, then commits them. At this point, this module will: 1) invoke module 2 to compare the current revision to the previous one, 2) invoke module 3 to find which test cases map to the changed code, and then 3) run the selected subset of tests.

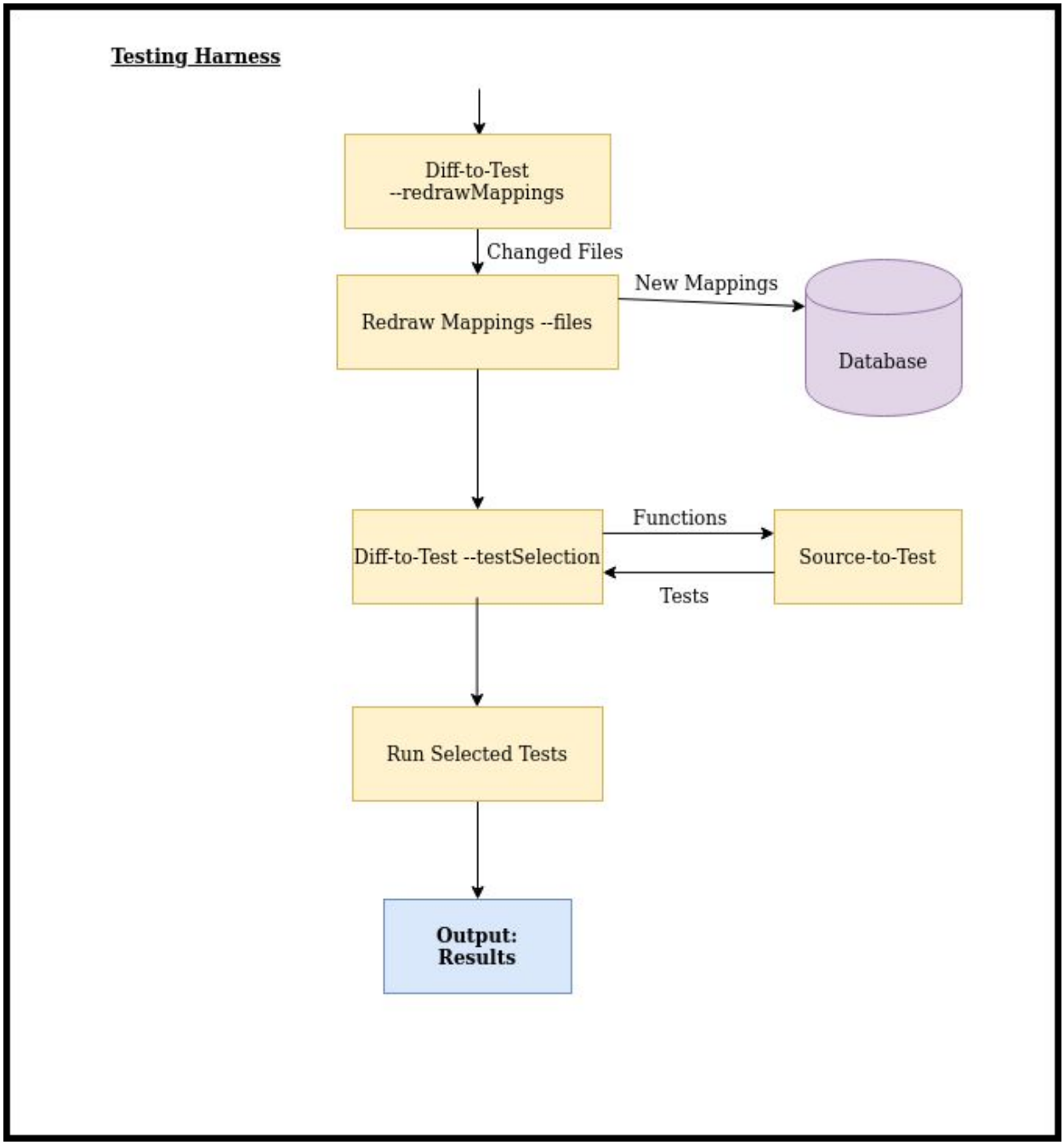

Figure 8. Testing Harness

When Module 2 is run, it will check if the test-to-source mappings have already been prepared. If not, then this means that either: 1) none of the mappings have been created or 2) the source code

file is new--i.e. it was introduced in the latest code revision. In the first case, the mappings will need to be created before using this module--by running Module 1--and the module will warn the user about this. In the second case, we will need to recreate mappings for any new or modified test files since mappings will not already be stored for any newly created or changed files. If the source still does not have a mapping stored in the database, then we will again warn the user. Finally, we will output the source-to-test mappings to users as a JSON report as shown in Fig. 8.

User Interface

Our tool will use Python3's argparse library to allow the user to specify the task to run via the command line. To switch between which module to run, a user would type one of "--source-to-test", "--test-to-source", or "--diff-to-source". If no argument is specified, the whole harness will run. For the "--source-to-test" and "--test-to-source" options, the user would also specify one or more target file names using the "--file-name" option. For example, if a developer wants to see all tests that exercise the "MathFunctions.r" file, then they would type "python3 TestingHarness.py --source-to-test --file-name=MathFunction.r". Users can also instruct the harness to create file mappings using the "--create-mappings" option. If new test files or test cases are added, users can optionally specify which test files to re-create mappings for by adding on the "--file-name" option.

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.htm.

APPENDIX A: RELEVANT STANDARDS

Our industry contact has not specified any specific standards, therefore we have adopted the standards laid out in Dr. Milos Gligoric's Ph. D. dissertation [1]. This paper recommends four standards for regression test selection tools: efficiency, safety, precision, and performance. The first standard, efficiency, is met when selecting tests takes less time than the total running time of unselected tests. The next standard, safety, is met when an RTS tool can guarantee that all unselected tests are unaffected by the changes in the new code revision. The counterpart to this standard, precision, is met when an RTS tool can guarantee that all selected tests are affected by the changes in the new code revision. The final and most crucial standard is performance. To meet this standard, the time it takes an RTS tool to select tests, run those tests, and collect the results must be shorter than the runtime of the entire test suite.