Evaluating BabelRTS using Mutation Testing for Java

Md Rakibul Hasan Talukder
Department of Computer Science, Colorado State University, Fort Collins, Colorado, USA
Corresponding author: rakibul, MRHT, talukder; rakibul.talukder@colostate.edu

Abstract

Regression test selection (RTS) tool performs a crucial role on the efficient testing of software while the codebase undergoes different versions within its life-cycle. Instead of running all the test cases blindly on the changed versions of the code, an RTS tool selects a group of test cases which ensures to validate the previously and newly introduced features. But the effectiveness of an RTS tool largely depends on finding fewer test cases and not missing any test cases for a changed version. In this work, we have tried to evaluate an RTS tool, babelRTS via mutation operation and then testing its effectiveness finding how many it misses out of all the killable mutants. An RTS tool can work on multiple language. Here in this project we are focused on java environment. In this work, we have taken 9 java projects having varieties of file sizes and performed mutation and evaluation. Our developed tool has evaluated that babelRTS can kill 96% of killable mutants. As RTS tools are becoming used for multi-lingual support, we have also developed our evaluation tool having multi-lingual support.

Keywords: regression test; test selection; mutation; test-suite effectiveness; killable mutants

1 Introduction

Nowadays every large software under goes several changes frequently. The evolution of product is happening so rapidly and the size and complexity of codebase is increasing in every sprint of development. But after each iteration or sprint of development it is very challenging to determine which portion or feature needs to be tested to ensure the previous functionality as well as new ones. One way of ensuring the overall functionality of the software product is adding new test cases with the previously generated ones, and then run all the test cases. As long as the codebase is not that large and complexity of the product is not so high, it can be managebale. But software products having millions of lines of code can make this impossible and quite inefficent in terms of running time of all the test case. Regression Test Selection (RTS) tries to choose and execute just the tests that are impacted by code changes since tests that are not impacted by code changes ought to produce results that are consistent with previous runs. RTS has been utilized in practice on a large scale and may considerably reduce the effort required for regression testing in this way.

Regression test selection is one of the well practiced approach for selecting efficient test suites while the codebase undergoes multiple versions throughout its life-cycle. It enables QA tester to avoid run the whole test suite. But it is quite challenging for a RTS tool to exclude the test cases which does not have any effect after making new changes to the program. We can say that, the effectiveness of an RTS tool largely depends on the sensitivity of that tool to any mutation of the program. It is important to find an appropriate RTS tool which has a high effectiveness on selecting test suites.

In this work, we want to evaluate the effectiveness of a RTS tool, BabelRTS. We want to build a evaluation tool which will perform two tasks on a high level:

1. Perform a mutation operation on codebase
2. Evaluate how good the BabelRTS is to find test suites for the preformed mutatation operation

The effectiveness of an RTS tool can be evaluated in many ways. Usually how less the selected test cases are and how less time it takes to find all those test cases. But the safety and correctness of an RTS tool is also an important factor. The safety of an RTS tool determines whether the tool has not missed any test cases that may result in fail for any test input. Correctness means the safety of the tool. As RTS tool is becoming very popular to find the selected test automatically, evaluating the safety of this tool is also very important.
Several RTS technologies, such as Ekstazi and STARTS, are built on a dependency tree of modifications made to the software. However, this technique is time expensive since it may necessitate the creation of control flow

graphs, abstract syntax trees, and so on. However, most projects these days are multilingual. In certain projects, java, C, and javascript are all utilized in the same codebase. As a result, efficient RTS tools that handle different languages are required. Consequently, an evaluation tool for an RTS tool is also required which can be tested for multiple language.

In this work, we plan to develop a evaluation script for babelRTS (evalBabelRTS.py) which can perform a statement deletion from a randomly selected file as smartly as possible. The the script will be able to find the values of defined experiment variables after running on multiple java projects. Eventually we will be able to perform calculation on those experiment variables to get our desired evaluation metric. Thus we will be able to measure the correctness of the RTS tool, called babelRTS.

2 Background

There are some ideas or topics that are important for understanding the evaluation approach we used:

Mutation operation Program Mutation generates alternate programs (mutants), each of which differs from the original by a minor syntactic change, and then requests that the tester construct inputs to eliminate the mutants by making each mutant provide a different result from the original version. Mutation operators, which are rules that specify changes to syntactic components in a program, define these modifications. The mutation operators have always been directly responsible for the ability of mutation testing to assist testers in designing good quality tests. Testing with well-planned mutations can be quite successful, whereas testing with badly designed mutations can be useless.

Statement Deletion Mutation One of the significant mutation can be removing a line form the codebase randomly. But a program file notly contaisn program statement but also includes various information of the implementation of the program. Usually developers write comments which are not programming statement. Again usually, in a nwere version of any software, it does not remove any statement that can result in build failure of the project. So, statement deletion from a program file is quite challenging.

Killable Mutants: In this context of work, we are calling those mutants killable which can pose an effect to the output of the program. We have assumed that it is reasonable to believe that mutating a program will increase the chance of introducing bugs which will result in test case failure. So, in our context of work, if any test cases fail or provide erroneous output on a mutant, that mutant is killable.

Killed Mutants: In our context of work, we target to evaluate a RTS tool to see if it can generate a group of test cases to have an effect on the found killable mutants. Again, we are considering test cases faliure to determine if the selected test suites by provided any RTS tool have an effect on those killabled mutant. If they have an effect, those mutants are called killed mutants.

BabelRTS: This tool provides a list of test files after each execution. For the first run, it provides all the test files, as delta from previous change to current change is the whole project. But later for each execution it provides the test files that are only triggedred by the subsequent changes on the code bases. For each execution of the BabelRTS tool, its creates or changes a .babelrts file. This file preserves the information of latest changes on the codebases.

3 Evaluation goal

We want to evaluate an Regression Test Selection (RTS) tool named **BabelRTS**. Our goal is to measure how much effective this tool is to find the test suites when a change occurs in the codebase. We want to measure the correctness of BabelRTS. We want evaluate how much killable mutants BabelRTS can kill. As RTS tool can have multi-lingual support, one of our evaluation goal is to develop the evalBabelRTS.py in a way so that it can have support for different langes too. In our work, our goal was to execute the experiment on java projects. So, here babelRTS is being evaluated in terms of a specific language. But assuring felxibility of supporting for many languages is one of the other goals.

4 Evaluation approach

Our approach for evaluating babelRTS is to introduce a mutation first into the codebase. We have only considered **A Random Statement Deletion** as mutation operation. Then we will find if the performed deletion triggers any test cases to fail. We define this mutant as **killable mutants**. Then we will run ***babelRTS*** to select the test cases for this mutant. Now if this selected tests by babelRTs produce at least one failure or error, this mutant is killed by babelRTS. We will look for the killable and killable mutants for each mutation attempt. Eventually we

will found total number of killable mutants and killed mutants. Thus we can evaulate the correctness and safety of the babelRTS taking ratio of killed and killable mutants.

4.1 Pseudo code

```
Script pseudocode:\\
-killable_mutants = 0\\
-killed_mutants = 0\\
For revision In revisions:\
        -compile and test revision
        -runBabelRTS
        For i=0 To MAX_TIMES:
                -choose random source file
                -choose random line in source file
                -delete line
                -compile and test revision
                If mutant Is killed:
                        -killable_mutants = killable_mutants + 1
                        -runBabelRTS
                        -run selected tests
                        If mutant Is killed:
                                -killed_mutants = killed_mutants + 1
Return killed_mutants/killable_mutants
```

4.2 Variables, Metrics

We considered couple of variables to get the insight on how well our tool is working, how good the test suites are and how well the babelRTS is doing.

Independent Variables:

1. project_name
2. language
3. max_revision_number
4. max_deletion_number

Experiment Variables per Mutation: These variables are considered for each mutation attempt. Using the values for these variables, we can get interesting insights about the mutation process and effectiveness of the tool or approach.

1. number_of_files_in_selected_project : non-negative integers
2. is_revision_compileable : 0 or 1
3. is_revision_run_by_all_tests : 0 or 1
4. randomly_selected_file_path : string
5. randomly_removed_line: string
6. is_Mutant_Compiled: 0 or 1
7. is_Mutant_killable : 0 or 1
8. number_of_errors_mutant_killable_for : non-negative integers
9. number_of_failures_mutant_killable_for : non-negative integers
10. is_Mutant_killed_by_babelRTS : 0 or 1
11. number_of_errors_mutant_killed_for : non-negative integers
12. number_of_failures_mutant_killable_for : non-negative integers

Experiment Variables per Execution

1. total_number_of_compilable_mutants = Σ (is_Mutant_Compiled=1)
2. total_number_of_non-compilable_mutants = Σ (is_Mutant_Compiled=0)
3. total_number_of non-killable_mutants = total_number_of_compilable_mutants - Σ (is_Mutant_killable=1)

4. total_Mutant_killable = Σ (is_Mutant_killable=1)
5. total_Mutants_killed_by_babelRTS = Σ (is_Mutant_killed_by_babelRTS=1)

Ratio of non-killable mutants to compilable mutants: This metric provide us the insight on how effective the test suites are to trigger an error or a failure. Non killable mutants are those which can be compiled but test suites cannot raise an error or failure for those mutated versions. As our approach of evaluating RTS is as good as the provided test suites are, we think this metric is important.

$$\frac{total\_number\_of\textbf{non-killable\_mutants}}{Total\_number\_of\_\textbf{compilable\_mutants}}$$

Ratio of compilable mutants to total mutation attempts:

$$\frac{total\_number\_of\textbf{compilable\_mutants}}{Total\_number\_of\_\textbf{mutation\_attempts}}$$

Ratio of killed to killable mutants: This ratio is the main evaluation metric to measure the correctness and safety of the BabelRTS tool. this metric will provide what percnatge of killable mutants are killed by the test suites slected by the babelRTS.

$$\frac{Total\_number\_of\_\textbf{killed\_mutants}\_by\_babelRTS}{Total\_number\_of\_\textbf{killable\_mutants}}$$

4.3 Steps used to perform each evaluation:

To measure the the defined metrics, we first had to get the values of defined experiment variables (section 4.2). Here are the steps on how we measure the metrics:

1. A list of java projects from git repositories is cloned. All of the cloned projects are put in the same directory.
2. BabelRTS package is also kept on the same directory with all the java projects.
3. Values for independent variables are provided: Name of all the projects, their language, maximum revision number and maximum statement deletion number.
4. evalBabelRTS.py has been run after that.
5. After the execution an excel sheet having all the experiment variables per mutation has been exported.
6. Summation and other necessary computation (according to definition) in the excel sheet are performed to calculate the values for ***experiment variables per execution***.
7. Eventually, three ratios: non-killable mutants to compilable mutants, compilable mutants to total mutation attempts and killed to killable mutants - these are computed to get the final evaluation.

4.4 Tools

We have developed a program in ***python*** that takes a list of projects and perform all the steps in the pseudo code (4.1) and give output of all the experiment variables per mutation described in 4.2 into an excel sheet. As our work is focused on testing evaluation for java projects we had to deal with java compilation requirements, test logs and project structure. Consequently, we had to install appropriate version of tools mostly related to java.

The tools we have used:

1. JAVA Development Kit (JDK 11)
2. Maven 3.6.3
3. Git
4. Python 3.9

5 Implementation of evaluation tool

We wanted to build an RTS evaluation tool to support multiple language. We developed the tool in a way so that projects of multiple language can be evaluated in a single run. We have built the tool having support for java projects currently. The git repository of implementation is https://github.com/rafi075/evaluation-babelRTS.git

1. Take a projects_names, source_code_path, test_suite_path and language as input.
2. Fix some parameters for each run:
   (a) **maximum_revision_number:** number of latest git commits to consider for each project
   (b) **maximum_deletion_number:** number for how many times mutation operation will be performed for each git commit revision

3. Go to each project, git checkout to each commit, and compile each revision first and run all the test cases
4. Get out of the *selected_project* directory.
5. Run the BabelRTS tool to create the .babelrts file into the *selected_project* for the first time so that later mutated version can compare with this original version.
6. Get into the *selected_project* directory again
7. Then perform statement deletion for *maximum_deletion_Number* of times.
8. For each mutated version, compile the code using maven install command skipping all the tests
9. If compilation is not successful, save the information we have got till now to a dataframe for exporting in an excel sheet and exit.
10. if compilation is successful, run all the test cases that have come along the project directory.
11. Extract test cases error or failure information from the log via pattern matching for java environment.
12. if the found-error-failure-count == zero , then it means, the mutant is not killable. Save all the information till now into the dataframe for export and exit.
13. Now, if mutant is killable, increase the killable mutant count and tell babelRTS which file has been changed explicitly. And then call the the babelRTS method to find all the seletected cases for the current mutant.
14. BabelRTS gives all the file paths for seltected test cases. Convert all file paths into class names.
15. Run the selected test classes using maven test classNames command.
16. Again extract information from test logs to see if there is any failed or erroneous test cases.
17. if count of errors or failures is zero, save the information for experiment variables to the dataframe and exit.
18. Otherwise the mutant is killed, increase the count for killed mutants, save all the values for experiment variables into the dataframe and exit the program.
19. Finally, export the data frame into an excel sheet.

5.1 Mutation implementation

Our mutation operation follows the following pseudo code:

```
def sdMutation():
    - selected_file_path
    - selected_deleted_line
    - get all elegible files from the selected project
    while:
        while: (find a non empty file)
            - choose a random file from the file list
            - get line count
            - find a file wih line count > 0
            - break

     - try finding elgible statement for MAX_TIMES

     if check_if_any_elgible_statement_found == true :
        - perform_delete_operation
        - break
     else:
        - continbue to pick another random file
 return    selected_file_path , selected_deleted_line
```

5.2 Finding eligible statement

One of the challenging part of statement deletion mutation is to find an eligible statement which can trigger a test case fail. A file can contain comments or api documentation etc. Deleting those will have no effect on test cases. Moreover, picking a line that results in compilation failure will also waste an attempt. So we have tried to smart-pick a line that will increase the chance of test case failure. Otherwise most of the randomly picked line will be non-killable.

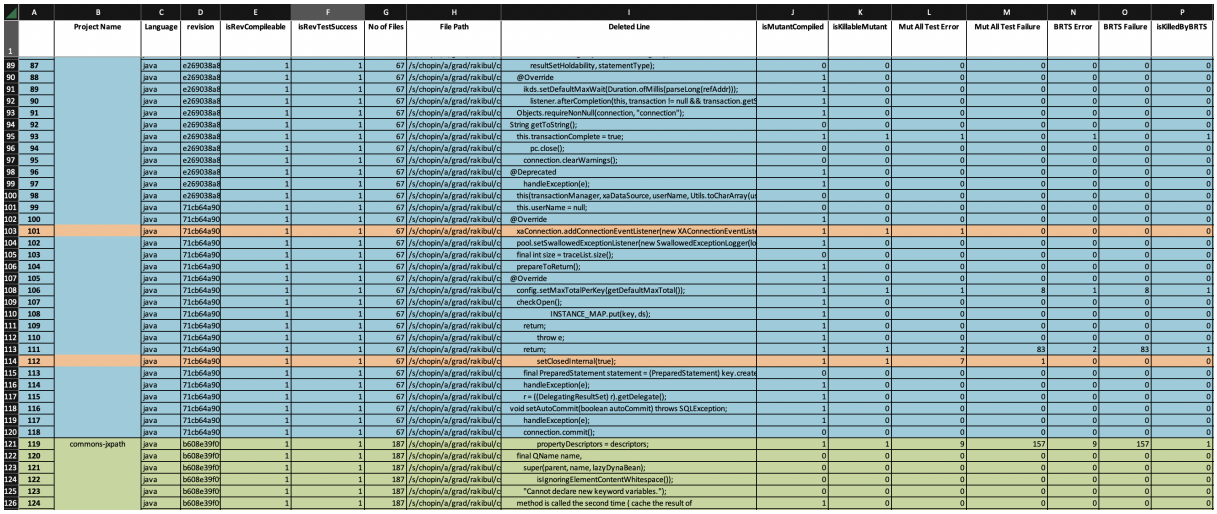
We have tried to exclude the following expressions for java environment:

- Empty line

- Single line comment with \\
- Multiple line comment start * or end *\
- Line starting with *
- Line contains **{** ***or*** **}**
- Line contains ***if, for, import, package, return, public, private***

6 Results

Here is a snieppet for collected raw data for experiment variables:
(Git repository to find more logs and results: https://github.com/rafi075/evaluation-babelRTS.git)



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1		Project Name	Language	revision	isRevCompileable	isRevTestSuccess	No of Files	File Path	Deleted Line	isMutantCompiled	isKillableMutant	Mut All Test Error	Mut All Test Failure	BRTS Error	BRTS Failure	isKilledByBRTS
89	87		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	resultSetHoldability, statementType);	0	0	0	0	0	0	0
90	88		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	@Override	1	0	0	0	0	0	0
91	89		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	ikds.setDefaultMaxWait(Duration.ofMillis(parseLong(refAddr)));	1	0	0	0	0	0	0
92	90		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	listener.afterCompletion(this, transaction != null && transaction.getS	1	0	0	0	0	0	0
93	91		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	Objects.requireNonNull(connection, "connection");	1	0	0	0	0	0	0
94	92		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	String getToString();	0	0	0	0	0	0	0
95	93		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	this.transactionComplete = true;	1	1	1	0	1	0	1
96	94		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	pc.close();	0	0	0	0	0	0	0
97	95		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	connection.clearWarnings();	0	0	0	0	0	0	0
98	96		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	@Deprecated	1	0	0	0	0	0	0
99	97		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	handleException(e);	1	0	0	0	0	0	0
100	98		java	e269038a8	1	1	67	/s/chopin/a/grad/rakibul/c	this(transactionManager, xaDataSource, userName, Utils.toCharArray(us	0	0	0	0	0	0	0
101	99		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	this.userName = null;	0	0	0	0	0	0	0
102	100		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	@Override	1	0	0	0	0	0	0
103	101		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	xaConnection.addConnectionEventListener(new XAConnectionEventListe	1	1	1	0	0	0	0
104	102		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	pool.setSwallowedExceptionListener(new SwallowedExceptionLogger(lo	1	0	0	0	0	0	0
105	103		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	final int size = traceList.size();	0	0	0	0	0	0	0
106	104		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	prepareToReturn();	1	0	0	0	0	0	0
107	105		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	@Override	1	0	0	0	0	0	0
108	106		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	config.setMaxTotalPerKey(getDefaultMaxTotal());	1	1	1	8	1	8	1
109	107		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	checkOpen();	1	0	0	0	0	0	0
110	108		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	INSTANCE_MAP.put(key, ds);	1	0	0	0	0	0	0
111	109		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	return;	1	0	0	0	0	0	0
112	110		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	throw e;	1	0	0	0	0	0	0
113	111		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	return;	1	1	2	83	2	83	1
114	112		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	setClosedInternal(true);	1	1	7	1	0	0	0
115	113		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	final PreparedStatement statement = (PreparedStatement) key.creat	0	0	0	0	0	0	0
116	114		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	handleException(e);	1	0	0	0	0	0	0
117	115		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	r = ((DelegatingResultSet) r).getDelegate();	1	0	0	0	0	0	0
118	116		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	void setAutoCommit(boolean autoCommit) throws SQLException;	1	0	0	0	0	0	0
119	117		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	handleException(e);	1	0	0	0	0	0	0
120	118		java	71cb64a90	1	1	67	/s/chopin/a/grad/rakibul/c	connection.commit();	0	0	0	0	0	0	0
121	119	commons-jxpath	java	b608e39f0	1	1	187	/s/chopin/a/grad/rakibul/c	propertyDescriptors = descriptors;	1	1	9	157	9	157	1
122	120		java	b608e39f0	1	1	187	/s/chopin/a/grad/rakibul/c	final QName name,	0	0	0	0	0	0	0
123	121		java	b608e39f0	1	1	187	/s/chopin/a/grad/rakibul/c	super(parent, name, lazyDynaBean);	0	0	0	0	0	0	0
124	122		java	b608e39f0	1	1	187	/s/chopin/a/grad/rakibul/c	isIgnoringElementContentWhitespace());	0	0	0	0	0	0	0
125	123		java	b608e39f0	1	1	187	/s/chopin/a/grad/rakibul/c	"Cannot declare new keyword variables.");	0	0	0	0	0	0	0
126	124		java	b608e39f0	1	1	187	/s/chopin/a/grad/rakibul/c	method is called the second time (cache the result of	1	0	0	0	0	0	0

Figure 1: Raw data of experiment variables in excel.

Here are some snippets of the log:

Case 1: Log snippet successfull killing of mutant by BabelRTS:

```
commons-collections src/main/java src/test/java java *.java
HEAD is now at d879329c Remove empty line
Revision Build sucessful. Running babelRTS
Mutation Number: 0 Revision: d879329cd339e2b52e283de14233aff30cd83999
File List Len : 343
Selected File ['/s/chopin/a/grad/rakibul/cs514/commons-collections/src/main/java
/org/apache/commons/collections4', 'OrderedMapIterator.java'] Line Count 48

Selected File ['/s/chopin/a/grad/rakibul/cs514/commons-collections/src/main/java
/org/apache/commons/collections4/map', 'FixedSizeMap.java'] Line Count 174

Eligible line: map = (Map<K, V>) in.readObject(); // (1)

Build successful for mutated program. Now test execution
Found error+failure in Mutant. Current Killable mutant 1
Rel Path: src/main/java/org/apache/commons/collections4/map/FixedSizeMap.java
Lang: java

Testlog from babelRTS changed file
mut_error_failure_by_babelRTS 3 error_by_babelRTS 0 failure_by_babelRTS 3
Returning to UnMutated versiob
```

Case 2: Log snippet of mutation having no effect

```
Mutation Number: 3 Revision: 90d8c7a4f6d99a06f1ecc48add6c8b57d84a67af
File List Len : 103
Selected File ['/s/chopin/a/grad/rakibul/cs514/commons-text/src/main/java/org
/apache/commons/text', 'StringTokenizer.java'] Line Count 1139

Eligible line:TSV_TOKENIZER_PROTOTYPE.setTrimmerMatcher(StringMatcherFactory
        .INSTANCE.trimMatcher());

Build successful for mutated program. Now test execution
Mutation has no effect. No test fail while running all_test
Returning to UnMutated versiob
```

Case 2: Log snippet of mutated version build failure

```
Mutation Number: 1 Revision: 90d8c7a4f6d99a06f1ecc48add6c8b57d84a67af
File List Len : 103
Selected File ['/s/chopin/a/grad/rakibul/cs514/commons-text/src/main/java/org
/apache/commons/text/io', 'StringSubstitutorReader.java'] Line Count 316

Eligible line:          final int actualReadCount = buffer(requestReadCount);

Build failed for mutated program
Returning to UnMutated versiob
```

The largest run we have performed comprises 360 mutation attempts. He is a summary of final calculated values of metrics we discusses.

#Project	#Revision	#Mutation	Total Attempts	#compiled	#killable	#killed	$\frac{killed}{killable}$	$\frac{nonKillable}{comilable}$	$\frac{compilable}{attempts}$
6	3	20	360	188	41	39	0.95	0.78	0.52
3	2	50	300	197	75	69	0.92	0.61	0.67
3	2	5	30	29	8	8	1.00	0.62	0.71

7 Discussion

We have performed the experiment on 9 different java projects:

#Project	Total files
commons-dbcp	67
common-jxpath	187
common-lang	239
common-net	213
common-pool	49
common-collections	343
common-cli	23
common-text	103

We have performed the experiments on projects having wide range of file sizes. We have varied the maximum deletion number from 5 to 50. Using increased number of maximum mutation for a particular revision has increased the number of killable mutants.

According to our design of experiments, the performance of BabelRTS to select the tests with correctness and safety is quite good. In total 690 mutation attempts have been performed. Average ratio of killed to killable mutants are 0.96. That means 96% of the cases, BabelRTS is able to select the test cases that are triggered by the mutation. There were some cases where BabelRTS was not able to find the appropriate test cases. For example, the following two lines have been deleted, they have raised failure test cases when run by all test cases but not by BabelRTS.

- *setClosedInternal(true)*
- *xaConnection.addConnectionEventListener(new XAConnectionEventListener());*

Out of 690 attempts for mutation, succesful compilation count is $(188 + 197 + 29) = 414$, total killable mutant count is 124. The average ratio of non-killaable mutants to compilable mutants is: 0.67. That means 67% of the compiled mutated versions fail to trigger any test cases. We can say that test cases written for those projects

were not as good as to trigger failure case for mutated version.

Another interesting factor of our developed tool is, out of total mutation attempts, on an average only 63% were successfully compiled after mutation. Our mutation algorithm, for 37% of cases picks such statements which results in build failures.

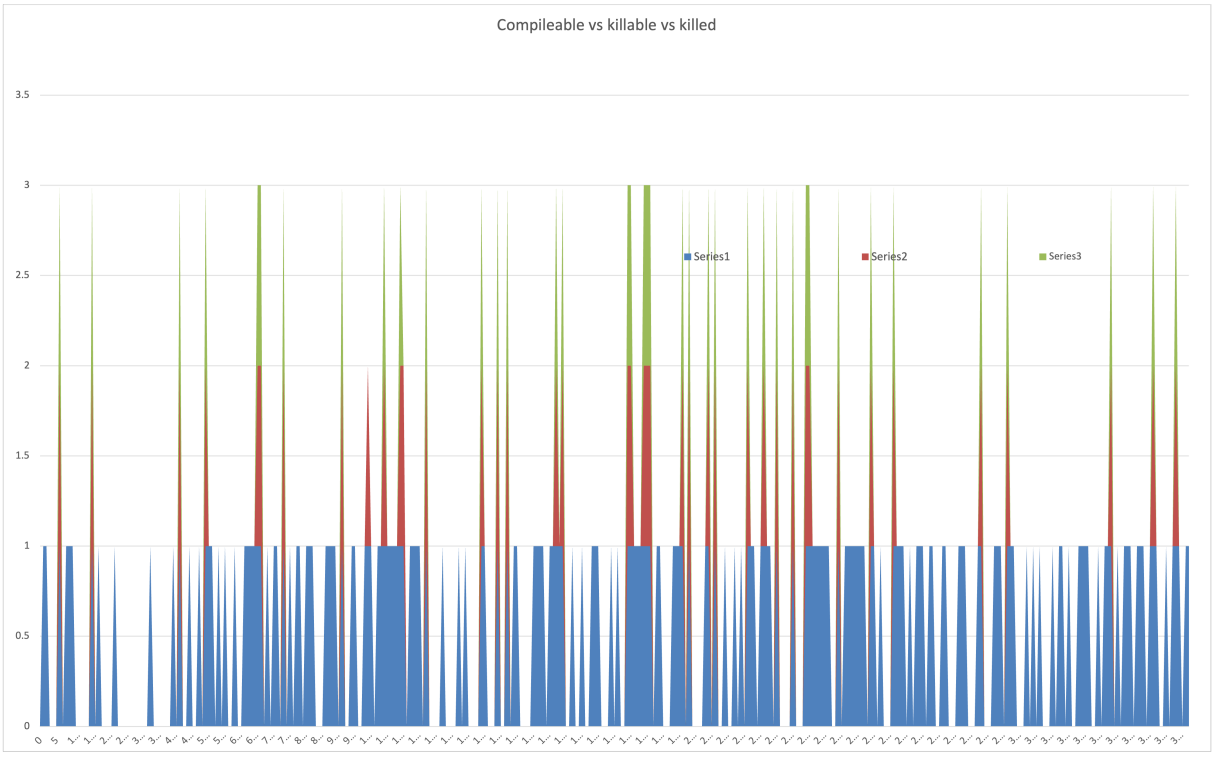


Figure 2: Hit events for compilable vs killabe vs killed

In the figure 2, X-axis represents mutation attempt from 1 to 360. Peak of the blue regions are the hits for compilation success. Red regions are hit events for killable mutants. Killed mutants are represented by green region. We can clearly observe that not all the attempts are compiled. Among them some are recognised as killable mutants. But almost green and red regions overlap with each other pretty well axcept few cases.

8 Challenges,Threats to validity

Identify what problems could exist. Here are the typical threats. List them and comment on them.

1. **Pre-built bad test suites** If there are not good test suites, they might not give any error or failure for the mutated program. So, even after running all the test cases coming with the project, detecting mutation is as good as the already generated test cases.
2. **Project dependency** We mainly tested with java maven projects. As we need to compile and run the project before performing any mutation, we needed to make sure the unmutated revision ran without any errors. And here comes the complexity of installing different versions of packages according to the requirements of the project. We had to install two different versions of maven 3.3.9 and 3.6.3
3. **Finding eligible line for mutation** Finding eligible statement for deletion was challenging. We have tried not to select a statement pure randomly. We have added bias to pick a statement smartly so that it does not result in build failure or trigger any test cases. It seems we fail to pick smartly for 37% of cases
4. **Killable mutatants dependency on maximum_deletion_count**: It seems increasing the value for maximum_deletion_count results in increased percentage of killable mutants. From the result summary table we can see that the lowest ratio for non-killable vs comilable is 61% which has occured when maximum-mutation-count was maximum 50. Increasing project number or revision number which has resulted in maximum total attempts for a single run (360) could not increase this ratio as expected.
5. **Long run time of java projects** As java project needs to be compiled first, it takes a lot of time to run those iteration where mutated versions have been successfully built. The larger the project size, the longer

it takes. We have executed 360 iterations in almost 5 hours where 188 attempts were compiled first and then tested by all test cases.

6. **Unexpected error occurrences** There are cases when our program could not extract error or failure information from the test logs. We have tried to catch those unexpected exceptions. Out 0f 360 attempts, unexpected exception has occurred only 4 times. As we have used pattern matching for extracting information from JAVA test log, any different logs can raise exception.

9 Related work

RTSLinus is the first dynamic RTS method proposed by Celik, Vasic, Milicevic, and Gligoric (2017) to go outside predefined language restrictions. This method dynamically locates all file artifacts a test requires at operating system level. The authors also offer a collection of extension points that facilitate easy interaction with build and testing frameworks. They tested the method on 21 Java projects that leave the JVM by launching new processes or running native code, totaling 2,050,791 lines of code, using the program RTSLinux, which they created as a loadable Linux kernel module. Their findings demonstrate that, in comparison to running all tests, RTSLinux typically skips 74.17% of tests and saves 52.83% of test execution time.

Another lightweight RTS tool, EKSTAZI, developed by Gligoric, Eloussi, and Marinov (2015) can easily interface with build systems and testing frameworks increasing the tool's adoption potential. EKSTAZI does not need integration with version-control systems and records the dynamic dependencies of tests on files. EKSTAZI was put into practice for Java+JUnit and Scala+ScalaTest, and its performance was assessed over 615 revisions of 32 open-source projects (totaling almost 5M LOC). The outcomes demonstrate that EKSTAZI decreased the end-to-end testing time by an average of 32% as compared to doing all tests.

STARTS, developed by Legunsen, Shi, and Marinov (2017), a static static RTS tool, which has been assessed based on the proportion of selected tests over all tests and the related run time to conduct those tests.

A methodology for coverage-based regression test selection (RTS) and a specially created Python tool are presented by Kauhanen, Nurminen, Mikkonen, and Pashkovskiy (2021). In this tool, mutation testing was used to assess, and the outcomes of running all tests were contrasted with those of performing a selection of tests that the tool had preselected.

Elsner et al. (2022) have built system-aware multi-language RTS strategy, which performs selective compilation and execution of impacted code modules and regression tests for a pull request. Their strategy was focussed on code bases which can be a combination of different coding languages involving cross language references.

10 Conclusions

In this work, we have tried to evaluate an RTS tool, BabelRTS to see how much correctly it can select the test cases if any new changes occur in the codebase. To create the environment of changing codebase, we have incorporated mutation to provide different versions for BabelRTS to work on. We developed a tool that can execute a specific kind of mutation : statement deletion. Then we have run the test cases that babelRTS has selected to find out if those are triggered correctly. We have executed 690 mutation attempts in total, among them 63% attempts were suitable for being killable mutants. Only 33% of them were recognized as killable mutants. But eventually we have been able to determine that BabelRTS can kill almost 96% of them. While performing the evaluation experiments, we have come to realize that killability of mutants largely depends on how carefully test cases are written. Moreover, smartness of picking effective statement also determines the percentage of mutated version being unable to build or run. But as RTS tools are being multilingual, we also have developed our evaluation tool in a way so that more language support can be provided easily.

References

Celik, A., Vasic, M., Milicevic, A., & Gligoric, M. (2017). Regression test selection across jvm boundaries. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering* (p. 809–820). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3106237.3106297 DOI: 10.1145/3106237.3106297

Elsner, D., Wuersching, R., Schnappinger, M., Pretschner, A., Graber, M., Dammer, R., & Reimer, S. (2022). Build system aware multi-language regression test selection in continuous integration. In *Proceedings of the 44th international conference on software engineering: Software engineering in practice* (p. 87–96). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3510457.3513078 DOI: 10.1145/3510457.3513078

Gligoric, M., Eloussi, L., & Marinov, D. (2015). Ekstazi: Lightweight test selection. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 2, pp. 713–716).

Kauhanen, E., Nurminen, J. K., Mikkonen, T., & Pashkovskiy, M. A. (2021). Regression test selection tool for python in continuous integration process. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 618-621.

Legunsen, O., Shi, A., & Marinov, D. (2017). Starts: Static regression test selection. In *Proceedings of the 32nd ieee/acm international conference on automated software engineering* (p. 949–954). IEEE Press.