Generate Test Selection Statistics with Automated Mutation Testing

Course Code: DV2572 Master Thesis in Computer Science

Madhukar Enugurthi
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Madhukar Enugurthi
E-mail: maen16@student.bth.se

University advisor:
Omid Gholami
E-mail: Omid.gholami@bth.se
Department of Computer Science and Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet: www.bth.se
Phone: +46 455 38 50 00
Fax: +46 455 38 50 57
ABSTRACT

Context: The goal of this research is to form a correlation between code packages and test cases which is done by using automated weak mutation. The correlations formed is used as the statistical test data for selecting relevant tests from the test suite which decreases the size of the test suite and speeds up the process.

Objectives: In this study, we have done an investigation of existing methods for reducing the computational cost of automatic mutation testing. After the investigation, we build an open source automatic mutation tool that mutates the source code to run on the test cases of the mutated code that maps the failed test to the part of the code that was changed. The failed test cases give the correlation between the test and the source code which is collected as data for future use of the test selection.

Methods: Literature review and Experimentation is chosen for this research. It was a controlled experiment done at the Swedish ICT company to mutate the camera codes and test them using the regression test suite. The camera codes provided are from the continuous integration of historical data. We have chosen experimentation as our research because as this method of research is more focused on analyzing the data and implementing a tool using historical data. A literature review is done to know what kind of mutation testing reduces the computational cost of the testing process. The implementation of this process is done by using experimentation.

Results: The comparative results obtained after mutating the source code with regular mutants and weak mutants we have found that regular mutants and weak mutants are compared with their correlation accuracy and we found that on regular mutation operators we got 62.1% correlation accuracy and coming to weak mutation operators we got 85% of the correlation accuracy.

Conclusions: This research on experimentation to form the correlations in generating test selection statistics using automated mutation testing in the continuous integration environment for improving test cases selection in regression testing

Keywords: Automation testing, test selection, weak mutation, computational cost.
**ACKNOWLEDGEMENTS**

First and foremost, we would like to thank our supervisors. My master thesis has been an amazing experience, sincere thanks to my supervisor **Mr Omid Gholami** who guided me whenever necessary with his valuable expert guidance and reviews. Thank you so much for the support and concern Omid Gholami.

I would like to acknowledge all the participants who helped with the experiment and thank you for their valuable suggestions which helped me to obtain good results for this research.

I would also like to thank my friends who supported me during my thesis work and encourage me to achieve my thesis goals Special thanks to my parents Enugurthi Srinivasa Chary and Vijaya Nirmala without their support and help I would not have accomplished my thesis.
LIST OF FIGURES

Figure 2-1 Process of Mutation Testing 1 ............................................................................................................. 10

Figure 2-2 Architecture of Milu tool 1 ................................................................................................................. 11

Figure 4-1 Experimentation Process 1 .................................................................................................................. 18

Figure 6-1 Comparison of results between regular and weak mutation 1 ...................................................... 23

Figure 6-2 Sample C program 1 .......................................................................................................................... 25
**LIST OF TABLES**

*Table 2-1 Different mutation tools* ................................................................. 12

*Table 4-1 Different mutation operators* ............................................................. 20

*Table 4-2 Historical Data from the Swedish ICT company* ........................ 20

*Table 5-1 Results obtained while using regular mutation* ............................. 21

*Table 5-2 Results obtained while using weak mutation* ............................... 22

*Table 6-1 Sample C program output when using CAR mutation operator* ........ 25

*Table 6-2 Standard deviation 1 for failed test cases while using regular mutation* 26

*Table 6-3 Standard deviation 2 for failed test cases while using weak mutation* 27

*Table 6-4 Effective size of regular and weak mutation* .................................. 27

*Table 6-5 Reduction of computational cost* ................................................... 27
LIST OF ABBREVIATIONS

MTS - Mutation System
AST - Abstract syntax tree
SCA - Source code analyzer
MOCS - Mutation operator constraint script
TES - Test execution
SUT - Software under test
TC - Test case
MT - Mutation testing
ST - Statement
BB - Basic Block
CAR – Constant array replacement
6.1.2 External threat .................................................................................................................. 26
6.1.3 Reliability ........................................................................................................................ 26
6.2 Standard Deviation 1 for Regular Mutation Using MILU Tool .................................................. 26
6.3 Standard Deviation 2 for Weak Mutation Using MILU Tool ..................................................... 27
6.4 Overhead of Weak Mutation and Regular Mutation ............................................................... 27
6.5 Reduction of Computational Cost .......................................................................................... 28

7 CONCLUSION AND FUTURE WORK .................................................................................. 29
7.1 Conclusion ............................................................................................................................ 29
7.2 Future Work .......................................................................................................................... 29

REFERENCES .......................................................................................................................... 30
1 INTRODUCTION

Software testing is a way of finding errors and bugs while developing a software one of the most common thing is to check the software is having an error or not. There is a different testing process to test a software based on the requirements. Regression testing is a process of testing the changed program to check whether the program works properly even after introducing the new functionalities without creating any bugs [1]. The main purpose is to verify that the bugs are fixed and the changes made to the program doesn’t create any new bugs to the previous working of the program. In recent years, the companies try to use distinctive methodologies to speed up the software development process. With the increase in the code base and test suite, running regression testing is a time-consuming process especially while trying to integrate the changes continuously. With the usage of traditional methods, it is not possible to perform the continuous integration [2]. There are some Regression testing techniques which are Test suite reduction, Test case prioritization and Test selection.

1.1 Research Focus

The focus of my thesis research is to produce a proof of concept implementation of an automated method for systematically introducing errors into the software platform. The process of introducing errors is done using mutation testing. As a part of this research, experimentation is used as a research method to collect the results. In the experimentation process, we are using weak mutants to build our automation mutation tool. The research was conducted at the Swedish ICT company. It has a very large regression testing suite that runs for several hours in a day. This regression test suite makes sure that any changes in source code do not break anything.

1.2 Contribution

Large code base requires large complex regression test suite. The regression testing performed in the Swedish ICT company is taking a lot of time. For every small change, the whole test suite needs to run. Running the whole test suite takes seven hours of time. This implies that the developer has to stay/wait before knowing the regressions testing is done after the change in code packages/source code. This results in delaying the continuous integration process and also increase in the development cost. The correlation we form using the error code which is mapped to test cases is used as a reference to run only particular tests in the regression test suite which results in saving a lot of time than running the complete test suite for seven hours. The correlations provided from our thesis work is used as a reference data for future use in the Swedish ICT company while performing regression testing.

1.3 Thesis Outline

This section describes how this research document is structured. The chapter that follows this section explains briefly about Regression testing, Test Automation and Mutation testing which is been part of this thesis research. The chapter following this is related work and this chapter describes the research done so far with relevance to this research. What are the additional contributions that can be made and in which area does this research fit the best. Furthermore, Research gap and research questions are formulated in this section. Following it with the research methodology which explains how the literature review and experimentation has been carried out and what resources does the literature review and experimentation demands so far. A literature review was done to answer our research question 1 (RQ1) and we have chosen the experimentation methodology to answer our research...
question 2 (RQ2). The next section consists of Results and analysis followed by Discussion, Conclusion and Future scope of this research has been explained in an elaborative was toward the end of the documentation.

1.4 Problem Formulation

It is observed that at the Swedish ICT company, it has very large regression test suite which is run every day for any small change in the code base to make sure that no bugs are being introduced. Their large code base has a very large regression test suite which requires several hours to run. The programmer has to wait for the results before committing the changes, thus it leads to a lot of waste in time and expenses.

The research was done on regression testing technique, in practice, these are hard to implement on an industrial level. While regression testing we don’t need to wait for all the test results. If we could execute the test related to the changes then we could save a lot of time. Using test selection, we can select test cases that are related to particular code packages from the previously failing tests. But the Swedish ICT company framework was quite stable with no proper historical failure test data. This raised the problem with how the failed test data can be formed from the code packages and by doing mutation can we map the failed test cases to code packages.

The Swedish ICT company have lines of code for their default security cameras which are in several packages. So, correlating these code packages to test cases can reduce the time required by selecting only the related test cases to modified code package. Correlating code to test cases requires historical data of failed tests, which we didn’t have in our situation. The traditional regression test method involves white box analysis and these methods should be performed for every change in the code base and cannot be used for the regression test suite. There is no research on this context. So, we need fault data to correlate the packages. This could be done by introducing mutants with an automation mutation tool using weak mutation and then testing against the regression test suite to correlate code packages to test cases with the error data. The correlations can be saved and used for statistical test selection without the need to run an analysis like in traditional methods [1].
2 BACKGROUND WORK

2.1 Test suite reduction

It is used to reduce the saving and reusing tests during the software maintenance by eliminating redundant test from the test suite. Eliminating the tests from the test suite speed up the process of the test suite. Test suite minimization is being mostly based on coverage criteria. Leitner et al [3] proposed that minimization of the suite is to produce a shorter version of the test case, the requirement for testing is that even shorter version of the test cases should still be able to reproduce the failures. This is done to produce the reduced size of the failing test cases which is based on delta debugging technique.

2.2 Test case prioritization

Test case prioritization is the order of making scheduling of the test cases in order to execute the test cases which have the high priority [4]. Rothermel and Harrold[5] define that the prioritization technique can improve the rate of fault detection of tests suite. Test case prioritization approach for software evolution which prioritizes the test cases based on their killing mutants on different statements was proposed by L Zhang [6]. The main purpose of this strategy is to lower the fault detection time [6].

2.3 Test selection

Test selection strategy deals with the problem of selecting the test cases that will be used to test the changed part of the software [7]. In regression test selection, test cases are selected because their execution is relevant to the changes between the previous and current version of SUT (system under test). Based on Rothermel formulation of a problem it can be said that the test case selection technique for regression testing focuses on identifying the modification traversing test cases in the give test suite [5]. Yar and Lishmot presented a test case selection based on symbolic execution approach of the SUT [8]. This technique statically analyses the code and specifications to determine the input partitions, then it produces the test cases so that each input partitions can be executed at least once [8]. Vokolos and Frankl also proposed a textual difference approach for test selection. This is based on the difference between source code of 2 versions of SUT. This approach is modifying the part of SUT by applying the diff Unix version [9].

2.4 Automation of testing

Automation testing is a type of testing that is capable of executing the tests and compares the output with the original test run [10]. From the research done, we came to know automation mutation testing is done only at the unit level in industries. The build automation mutation tool can work under functional level testing.

2.5 Mutation Testing
This is a type of testing where we mutate certain statements in the source code and checks if the test cases can find the errors [11]. The process of mutation testing is explained in Figure 2-1. Though mutation testing is good in checking the quality of test suite, it is very hard to put into practice in industrial level as it produces a large set of mutants which leads to an increase in the execution cost and also mutation testing requires a lot of human effort in analyzing the original programs output to that of mutated programs.

In the mutation testing, Mutations are classified into two types one is first order mutants and other is higher order mutants.

Traditional mutation testing consists of first order mutants, the first order mutants is created by introducing a single fault to the source code. In the first order mutation, the mutants are easily killed because it is easy to detect only one particular type of mutants. Coming to higher order mutants this is created by inserting two or more mutants/fault. Our focus is on introducing higher order mutants. We need to introduce a number of mutants that can be killed, and also to cover the whole functionality of whole code higher order mutants are introduced [12].
2.6 Milu Tool

Milu is an automated mutation testing tool used for generating the first and higher order mutants to the source code. This tool is specially designed to mutate the C program. In traditional type of mutation testing, they used the only first order of mutants on the test cases. But in the Milu tool, there is an option of generating both first and second order mutants. The details regarding the higher order mutation are provided found elsewhere [13]. As a C program mutation tool, Milu adopts 77 mutation operators of Agarwal et al [14]. In this tool there is also an option of customizing the mutation operators, this feature helps us to produce weak mutants to the source code provided by the Swedish ICT company using Milu tool. As the Swedish ICT company source codes are mostly based on the C. The data provided to us by the Swedish ICT company is also based on C. So, we used the Milu tool in our experiment to generate the mutants.

![Figure 2-2 Architecture of Milu tool](image)

The above Figure 2-2 is the architecture of the Milu tool. The figure explains the process of testing a program in the Milu tool. SCA is the source code analyzer which analyzes the input provided to the tool. AST is the abstract syntax tree of the system. MTS is the mutation system, which is the main component in the Milu tool. It generates the mutant template by taking AST and SCA. The Test harness is the component present in the Milu tool which helps in reducing the runtime cost of the mutation testing process. MOCS is a mutation operator constraint script which is used to introduce new mutation operators by the user. This feature helps our research work. It tries to help us introduce weak mutants to the code. TES is the test execution system.

2.6.1 The motivation for using Milu tool

After doing the literature review on the tools present for mutation testing we came to know that there are different types of mutation testing tool in which the tools present for Java program is more
compared to C program mutation. Our concern is to mutate the C codes for which we have found three mutation testing tools which are mentioned in the table.

<table>
<thead>
<tr>
<th>Mutation tool name</th>
<th>Useful for experiment</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic mutation testing tool for C</td>
<td>No</td>
<td>Generated higher order mutants but cannot reduce the computational cost.</td>
</tr>
<tr>
<td>C mutate tool</td>
<td>No</td>
<td>Generated higher order mutants but cannot introduce a particular type of mutants to the code</td>
</tr>
<tr>
<td>Milu mutation tool</td>
<td>Yes</td>
<td>Generated both first and higher order mutants and can reduce the run-time cost</td>
</tr>
</tbody>
</table>

Table 2-1 Different mutation tools

We have found Semantic mutation testing tool [15], C mutate[16] and Milu mutation tool [13], for Semantic mutation tool we can change the source code of the tool but using semantic mutation tool it only generated only one type of mutants. Using semantic mutation tool leads to an increase in the computational cost. C mutate is also not used in our experiment because this tool does not have the feature to introduce our required mutants to the source code so this tool is also being excluded. The motivation for using Milu tool is there is an option called MOCS (mutant operator constraint script) which allows the user to introduce particular mutants to the source code which in our case are a weak mutation. And also, in the Milu tool has the capability of reducing the runtime cost. By doing weak mutation we will reduce the compilation cost which reduces the whole computational cost of the mutation testing process. This is the motivation for considering Milu as our tool.

2.7 Advantages to the Swedish ICT company

Advantages of doing this research work at the Swedish ICT company by forming the correlations to the code packages and test cases the test suite size gets reduced which helps in reducing the time taken to run the regression testing. As it takes seven hours to run the complete test suite every time the code is changed. Providing the correlations for the Swedish ICT company for code packages and tests may result in running only particular tests for particular code change which saves a lot of time and also reduces the cost of the testing process.
3 RELATED WORK

This chapter contains the details of the research done to date and related to testing selection which is based on automated weak mutation testing.

A lot of research has been carried out in the past regarding the automation testing but mostly at the unit level of testing. As I mentioned most of the research is done at the unit level for the test selection Rothermel and Harrold [5] gave a good analysis of regression test selection on the functional level of testing.

Some researchers have tried to improve the problems regarding the test selection such as testing is a cost-effective process by using historical data, taking this into consideration the fact that using mutation testing is a costly process. by following the weak mutation process, we can lower the computational cost of the system. By using this we are not affecting any code quality. The functionally based approach is mainly used for the industries. In the present research, we have investigated only particular mutants which are used for weak mutation testing that helps in getting us both the expected results and also which is cost optimal.

Present companies are now adopting the agile process, Continuous integration is the most important aspect of continuous development as more tests are run and then the code is integrated. The less time spends on running tests can help speed up the development process significantly. The test case selection method is about picking out one of the test cases from the test suit. As test case selection problem T should have all the test that are exposed faults in p. The author Fisher et. al proposed that test case selection using integer programming to tests FORTRAN programs [17]. The test cases the selection process of his algorithm will work on two matrices which explain that the program segments to test cases and two program segments [2]. One of the drawbacks of this approach is we cannot control the flow changes in the program. The test case dependency matrix does not work with the control flow structure of software under test. if we change the flow of the matrix is updated then execution cost may increase, and it becomes an expensive method.

Agarwal et al [14] proposed a model which is based on the reduction of execution technique. Reducing the execution of the program is the execution trace is the set of statements executed by given tests [2]. Dynamic slice of the program is the execution of a subset of a program, reducing the effort for testers. For example, adding /deleting a statement which has simple output and no define or uses variables, this statement can be modified revealing. A new statement has no variable, its addition will not be affecting the existing slices, the result is an empty solution [2].

Howden [18] proposed a modification known as weak mutation testing. In this weak mutation testing is computationally more efficient because it provides a less stringent test. Weak mutation testing is defined as in a program P and component present in program P is C. If the program is mutated, then P’ is the mutated program and C’ is the mutated component of program P. weak mutation defines if C’ causes to compute a different value from the value computed by C on at least one execution of C’. Even though C’ has produced different values for C, P’ will produce the same value as P.

Reducing the number of mutants using weak mutation can also lead to fewer mutants this is selective mutation testing. Offut [19] et al., has done research based on the Mothra’s [20] weak mutation, 22 weak mutation operators they found few operators that can produce 90% of the mutants that make the tests fail.

Offutt and Lee[19] in their Leonard weak mutation system, Component as the location of the program in which the states of original and the mutated program are compared.
L. Zhang [21] in his PhD has combined regression and mutation testing to an efficient way to prioritize tests. Changes did by programmers during development and introduced changes (mutants) were used to evaluate the test suites. He introduced a ReMT algorithm to work with mutation testing on regression test suites. ReMT has three parts. Preprocessing component – which maps the mutants. Core ReMT has other two components, Mutant coverage checking and dangerous-edge reachability checking. The mutant coverage uses the coverage information for test selection based on history. If tests did not have any history the ReMT executes the tests to gather results. The dangerous-edge reachability coverage is used to check if the mutation results can be used again.

A) Can only be reused if no dangerous edge is executed from the beginning to mutation, done by dynamic coverage.
B) No dangerous edge can be executed from mutation to end stare, done by dangerous-edge reachability analysis

When evaluated on different programs, results show that ReMT can reduce the cost.

Hussain in his thesis proposed a mutant clustering, which chooses mutants by clustering algorithms. It generated first order mutants then using the clustering algorithms few mutants are selected from every cluster to use in the mutation testing [22]. Rest of the mutants are removed from the test suite. Hussain studies show that clustering can select fewer mutants while maintaining the mutation score.

Edward [1] in his thesis has proposed a tool to analyze the historical test failure data and use this to select test cases when a particular package is changed. The difference engine developed collects the previous and current regression test data and uses this data to form the correlations to failed test data. But the problem is using it in most of the industrial test suite is very stable and the test fails very rarely to make the correlation between the data.

The most popular regression testing techniques are mostly white-box testing methods. None of these can be used for black-box testing, where code coverage cannot be used. This thesis is a continuation of Edward approach to get correlations by making the tests fail by introducing the errors into software under test with mutations. This way we can get the correlations without waiting for the test to fail which can only happen scarcely in industrial usage. These correlations can be used for test selections even in the black box testing

Aim and Objectives

3.1 Aim:

The goal of this research is to form a correlation between code packages and tests which is done by using automated mutation testing. The correlations formed is used as the statistical test data for selecting relevant tests from the test suite which decreases the size of the test suite and speed up the process

3.2 Objectives:

- Investigation of existing methods for reducing the computational cost of automatic mutation testing.
- Build the automatic mutation tool that mutates the source code.
- Run the tests on the mutated code and map the failed test to the part of the code that was changed.
- The failed test cases give are the correlation between the tests and the source code which are collected as data for future use of the test selection.

### 3.3 Research Questions

**RQ1)** Does the weak mutation testing process increases or decreases computational cost?

**Motivation:** Mutation testing increases the amount of code needs to be tested which in turn leads to the increase in the execution and computational cost. This in return increases the overall cost of the testing process. To make this process cost efficient, we need to use a particular type of mutants that is used to help in bettering the process. From the literature review, we can identify the particular mutation testing that reduces the computational cost which can answer our RQ1.

**RQ2)** How can we form the correlations between code packages and tests?

**Motivation:** The existing process to get the correlations is to have the failed test data from the developers during the time of developing phase, obtaining the data is not possible in many cases, so we will introduce errors into historical test data present at the Swedish ICT company to mutate so that we get failed test cases which are the data for correlations.
4 RESEARCH METHOD

4.1 Literature Review

A literature review was conducted based on the guidelines provided in Creswell’s book “Research design” [23]. By conducting this literature review the gaps and the related work in the research has been identified. The following steps are the process for literature review which is shown below.

STEP 1: Identifying the keywords

Initially, we need to start with the keywords with the current area of the research. Few keywords are “Automation testing”, “Mutation operators”, “Test selection”, “Weak Mutation”. Later we improved the search string by exploring the results obtained from the previous keywords. Initially, the search string used is “Test automation” AND “Regression testing” for which we have found 433 results for a basic search on IEEE. Later we further modified the search engine as “Mutation Testing” AND “Cost reduction technique”, we got 300 articles which are few useful results and few articles are not related to our research has been excluded. Therefore, the search string has been further modified to (Higher Order OR Weak Mutants) which resulted in 90 articles. Reading the refined articles, we found 45 relevant articles. From these articles, few are not available for the full text which is excluded from the initial search. 10 articles are snowball sampled from the research articles.

STEP 2: Searching and gathering the literature

From the database provided by BTH such as IEEE Xplore, Inspec, Google Scholar etc. are used to search the relevant articles, journals and books with the obtained keywords and search strings. The articles which are related to our research work has been saved to the Zotero as the reference material for the use of the thesis. The articles obtained were thoroughly studied. The keywords used in this article is used for further researching of the articles.

STEP 3: Reading and analyzing the Articles:

The articles collected from the previous step was read and analyzed carefully. The information obtained from the articles abstract and journals are studied carefully and the articles which suit our research are marked as the reference papers.

STEP 4: Writing the review

The information which is gathered from the references is used in the background study and related work of the current research.

STEP 5: References

In the end, we have added a complete bibliographical list of articles, journals, books that are used as references for our current research work.

4.1.1 Inclusion Criteria:

- Articles which are in English are included.
- Articles with Full text.
- Articles that focus on Mutation testing and Automation testing
Articles describing the reduction of computational cost for mutation testing is included.
Comparative research articles focused on the evaluation of mutation testing.

4.1.2 Exclusion Criteria:

- Articles which are not focused on our research topic is excluded.
- Articles which are not peer-reviewed are excluded.
- Articles published prior to 2008 are excluded.
- Articles which are in the languages other than English are excluded.
- Some are articles are not available for full text which is been excluded.

By deep investigation on execution cost reduction for mutation testing helps us gain the knowledge on the reduction of computational cost using weak mutation, will able to answer my RQ1.

4.2 Experimentation

To obtain an answer to our research question in this research the method we have chosen is Experimentation. Experimentation is one of the popular research methods in computer science and it is also defined as the systematic approach to the research. Experimentation is the process which is used to manipulate the independent variables and controls and measures any changes in independent variables [24]. We have chosen experimentation as our research because as this method of research is more focused on analyzing the data and implementing a tool using historical data. Controlled experimentation is done in our thesis for which dependent and independent variables are provided in the section 4.2.1 and 4.2.2. To find the better automation tool, evaluation is a must. The evaluation of algorithm can be only done by the experimentation process.

4.2.1 Independent variables

The independent variables are manipulated to get the quantifiable data or qualitative results from the experiment. Independent variable is the input. In our case, the independent variable is the mutation operators because the mutation operators are being controlled and it manipulates the source code of the cameras to make the test cases fail. So, in our experiment, the independent variables are the mutation operators.

4.2.2 Dependent variables

Dependent variables are the one which is affected by the independent variable. In our case, dependent variables are the failed test cases which answer our research question 2 (RQ2).

4.2.3 Experimental Design

After the initial literature review on the research topic we have started using an automation mutation tool based on C. Initially we gave used a sample piece of code for injecting errors into the code which we succeeded. Later on, the Swedish ICT company have provided a set of code for which we have to mutate and write the test cases to it such that the test cases fail which form the correlations. The code is based on C language. We started out an experiment by injecting the errors into the C code and making them run under their test suite. After the number changes made to the code base and making them run under the test suite we have obtained the failed test cases. The mapping of error code with the failed test cases data is collected.
We have used Milu an Automated mutation tool. To inject the mutants into the source code we have used Milu as our automated mutation tool. Initially, we have started using this with a sample program. In the beginning, using the tool, we have generated the mutants using the default operators present in Milu in which we got succeeded. Later we tried to change it by providing weak mutation operators. When we introduced weak mutants to the sample code it started working. Later we came to know that Milu needs source file which is able to run with GCC -E. The source file should run through the preprocessor before we start the mutation process.

On doing the literature review we identified a Swift tool that generates the intermediate files which make the preprocessor run. Swift is a Linux software which is used to compile and run them directly on the local server. It creates mocks of that compiles and execute software directly. This particular tool needs to be run once before we mutate the code or start generating the mutants. When the mutation started to work we have written a script that introduces only weak mutants into the source code which helps Milu tool to mutate only specific type of mutants which decreases the mutation testing cost. Running the mutated code on the test cases which resulted in failing. The figure 4-1 shown below is the process of mutation script work with the source code and ended with the statistical data which are our correlations.

For the experimentation, we have used nine source codes. The whole experimentation process is done for three months. In each month three sources codes have been mutated and tested and each code is iterated three times. The source codes are tested using regular mutation and weak mutation. The controlled experimentation is done for this experiment. The experiment is repeated with both regular mutation and weak mutation. We have repeated the experiment three times for each source code. After every iteration, we have changed the test cases and ran the experiment again. This process of writing the test cases and doing three iterations to the source codes, we are able to do three source codes for one month for both regular and weak mutation. This process took three months for completing all the nine source codes with three iterations each.

In the regular mutation, we are introducing mutants that cover all type of mutants. The parameters used in this type of regular mutation is statement mutants, operator mutants, constant mutants. Some of the operators used while doing regular mutations are stated in table 4-1 [25]. While introducing of weak mutants, we have introduced the mutants that are weakly killed by the test cases from the regular mutants. The operators in table 4-1 are the operators used in our experiment while doing regular mutations. These operators cover all the parameters of the code i.e. all the parts of the code are covered. While doing the weak mutation we have selected the operators that are weakly filled by these operators.
<table>
<thead>
<tr>
<th>Mutation operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVR</td>
<td>Scalar variable replacement</td>
</tr>
<tr>
<td>UOI</td>
<td>Unary operator insertion</td>
</tr>
<tr>
<td>SAN</td>
<td>Statement Analysis</td>
</tr>
<tr>
<td>DSR</td>
<td>Data statement alteration</td>
</tr>
<tr>
<td>DER</td>
<td>Do statement END replacement</td>
</tr>
<tr>
<td>CRCR</td>
<td>Constant Replacement</td>
</tr>
<tr>
<td>VDRT</td>
<td>Absolute value insertion</td>
</tr>
<tr>
<td>VSRR</td>
<td>Array reference for array reference</td>
</tr>
</tbody>
</table>

**Table 4-1 Different mutation operators**

### 4.3 Rejection Criteria

**Survey:** The research method survey is used in triangulating the market research and polls of some professional opinions will not play a role in building a model. The survey is conducted when there is an existing tool, and the implementation is solely based on the choice of practitioners [26].

**Case study:** The case study is an examination technique used to explore a solitary marvel is a specific time period. Generally reasonable for mechanical assessment of programming models or apparatuses to maintain a strategic distance from the scale-up issues. The contrast between contextual investigation and experimentation is that the trial is directed to test over the factors that are controlled by the professional while then again, a case study chooses the factors speaking to the commonplace circumstance.

Hence, Rejecting the above research methods (i.e. Survey and case study) we have come to a conclusion that Experimentation and Literature review are perfectly suitable for our research. The Figure 4-1 shown below explains the view of the experimentation process.

### 4.4 Challenges encountered in experimentation process

The test results are obtained even there are many constraints and challenges amid experiment

The major problem we faced during the experimentation is a collection of data. The historical data from the Swedish ICT company is been updated for every 28 days. The updating of data affected the correlations accuracy. Though the Swedish ICT company have a large set of data they don’t hold it for the long run.

Coming to the literature review, our experimentation is based on mutation of C packages in the Swedish ICT company. While doing Literature review we find it hard to get the papers and articles related to this topic. This was open to the major challenge we faced initially before starting the experimentation process.

### 4.5 Software Platform

For this research work, on automation weak mutation was performed on different source codes and evaluated in a LINUX environment. We have used automation tool Milu which is built by mutating the C files. The test cases are written in the test scripting language.
4.6 Data Collection

In this research, Experiment is performed with the historical test data from the daily test suite results after the continuous integration of the Swedish ICT company products. Currently, in the Swedish ICT company, the historical test data is stored only for a one-month period of time. The historical test data keeps up-dating for every month. In the Swedish ICT company, they use Jenkins as the automation server for continuous integration process, as Jenkins holds one month of data while the historical data present in the Swedish ICT company is available for one month. The data collected from the Swedish ICT company historical test data for our thesis is done for two months. Two months of data is not sufficient enough to form the correlations for code and tests. But we tried to form with the available data.

<table>
<thead>
<tr>
<th>S.no</th>
<th>Data in the Swedish ICT company</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Changes made by developers to source codes/ camera codes</td>
</tr>
<tr>
<td>2</td>
<td>Data of the Swedish ICT company ID’s</td>
</tr>
<tr>
<td>3</td>
<td>Test cases Results</td>
</tr>
<tr>
<td>4</td>
<td>Information regarding Client-Product</td>
</tr>
<tr>
<td>5</td>
<td>IP protocols address</td>
</tr>
</tbody>
</table>

Table 4-2 Historical Data from the Swedish ICT company

The data shown in the above table is the historical data present in the Swedish ICT company. Changes made by the developer which are provided by information about the number of files changed due to commits made. Data of the Swedish ICT company ID’s indicates the employee’s information stored in the database. Information regarding the client Product indicates that the Swedish ICT company project data which is confidential. Test case results indicate that it is the kind of data where the test cases are failed or passed. Our main focus is on Mutating the source code provided by the Swedish ICT company and running the test cases on the mutated source code.
5 RESULTS

5.1 Experimentation using regular mutation

After the experiment is implemented using Milu results are collected. Below is the no of mutants introduced to the program, no of test cases failed and correlations accuracy.

<table>
<thead>
<tr>
<th>Code packages provided by the Swedish ICT company</th>
<th>No of test cases written</th>
<th>No of Test cases failed</th>
<th>The strength of the test suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code 1</td>
<td>15</td>
<td>9</td>
<td>60%</td>
</tr>
<tr>
<td>Source code 2</td>
<td>18</td>
<td>8</td>
<td>44.4%</td>
</tr>
<tr>
<td>Source code 3</td>
<td>12</td>
<td>8</td>
<td>66.6%</td>
</tr>
<tr>
<td>Source code 4</td>
<td>30</td>
<td>16</td>
<td>53.3%</td>
</tr>
<tr>
<td>Source code 5</td>
<td>22</td>
<td>11</td>
<td>50%</td>
</tr>
<tr>
<td>Source code 6</td>
<td>17</td>
<td>12</td>
<td>70.5%</td>
</tr>
<tr>
<td>Source code 7</td>
<td>8</td>
<td>6</td>
<td>75%</td>
</tr>
<tr>
<td>Source code 8</td>
<td>25</td>
<td>19</td>
<td>76%</td>
</tr>
<tr>
<td>Source code 9</td>
<td>14</td>
<td>11</td>
<td>78.5%</td>
</tr>
</tbody>
</table>

*Table 5-1 Results obtained while using regular mutation 1*

This data is extracted from the Milu mutation tool when running on the source codes provided in the Swedish ICT company.

The obtained results are found in every program that is mutated. Now using accuracy evaluation metrics. How much strong is the test suite written is able to find the mutants in the code is tested. This will be of no of test cases failed is divided with no of test cases failed and multiplied by 100 to get into a percentage.

Average of Strength of the test suite = 62.1%

The average strength of the test suite is for the used Milu mutation tool is 62.1%
5.2 Experimentation using weak mutation

The table shown below is the experimental results when weak mutants are introduced into the source code.

<table>
<thead>
<tr>
<th>Code packages provided by the Swedish ICT company</th>
<th>No of test cases written</th>
<th>No of Test cases failed</th>
<th>The strength of the test suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code 1</td>
<td>15</td>
<td>14</td>
<td>93%</td>
</tr>
<tr>
<td>Source code 2</td>
<td>18</td>
<td>13</td>
<td>72.2%</td>
</tr>
<tr>
<td>Source code 3</td>
<td>12</td>
<td>10</td>
<td>83.3%</td>
</tr>
<tr>
<td>Source code 4</td>
<td>30</td>
<td>25</td>
<td>83.3%</td>
</tr>
<tr>
<td>Source code 5</td>
<td>22</td>
<td>18</td>
<td>81.8%</td>
</tr>
<tr>
<td>Source code 6</td>
<td>17</td>
<td>15</td>
<td>88.2%</td>
</tr>
<tr>
<td>Source code 7</td>
<td>8</td>
<td>7</td>
<td>87.5%</td>
</tr>
<tr>
<td>Source code 8</td>
<td>25</td>
<td>23</td>
<td>92%</td>
</tr>
<tr>
<td>Source code 9</td>
<td>14</td>
<td>12</td>
<td>85.7%</td>
</tr>
</tbody>
</table>

*Table 5-2 Results obtained while using weak mutation 1*

In the Milu mutation tool after introducing weak mutants, we are able to generate more accurate data than just using random mutant provided by the tool.

The obtained results are found in every program that is mutated. Now using accuracy evaluation metrics. How much strong is the test suite written is able to find the mutants in the code is tested. This will be of no of test cases failed is divided with no of test cases failed and multiplied by 100 to get into percentage

Average of Strength of the test suite = 85%

The average strength of the test suite is for the used Weak mutation Milu mutation tool is 85%.
ANALYSIS AND DISCUSSION

From the results obtained after the experimentation is done we have obtained some accuracy in the correctness of the relationship when we used mutation testing with the normal mutation operator present in the Milu tool we have 62.1% of the correlations to the mutated code and the failed test cases. When we introduced an only particular type of mutation operators that is weak mutation type with the Milu tool we have 85% of the correlation accuracy.

By observing the correlation accuracy obtained through results in this research we can confirm that the statistical significance of forming the correlations in between normal mutation operators and weak mutation operators we can say performing the weak mutation operators during mutation testing can make more test cases fail. This research states that using weak mutation operators we can produce more test cases fail than other operators.

![Comparison of tool](image)

Figure 6-1 Comparison of results between regular and weak mutation

The chart Figure 6-1 shown above is the comparison data of the results obtained from the Milu mutation tool when using regular mutants and when using weak mutants to form the correlation between the code and test cases. Answers to the research questions are given below:

RQ1) Does the weak mutation testing process increases or decreases computational cost?

**Motivation:** Mutation testing increases the amount of code needs to be tested which in turn leads to the increase in the execution and computational cost. To make this process cost efficient, we need to use a particular type of mutants that is used to help in bettering the process.

The Answer to RQ1: From the Literature review we found that using less number of weak mutants i.e., generating few mutants can result in reducing the execution and computational cost. Though we are generating fewer mutants the process of failing the test suit does not get affected because the generated mutants are worth enough in making the test cases fail and results in getting killed by the test cases. So weak mutants can defiantly reduce the computational cost [18]. In the literature review, it is stated that
the weak mutants are also 91% effective in reducing the cost of mutation testing and 62% for regular mutation.

In the related work, Zhang has combined regression and mutation testing to an efficient way to prioritize the tests. He used ReMT algorithm in his research work which also reduces the cost of mutation testing. When compared our process with Zhang we came to know that Zhang ReMT algorithm can only reduce the runtime cost. But coming to our model we are able to reduce both runtime cost and compilation cost which is more effective than Zhang algorithm.

RQ2) How can we form the correlations between code packages and tests?

Motivation: The existing process to get the correlations is to have the failed test data from the developers during the time of developing phase, obtaining the data is not possible in many cases, so we will introduce errors into historical test data present at the Swedish ICT company to mutate so that we get failed test cases which are the data for correlations.

The Answer to RQ2: From the experimental results it is clear that to find the correlations for code packages and tests we need to initially mutate the code with automation mutation tool. Once the mutated code is obtained we run this mutated code on the test cases of the original program which results in failing the test cases of the original program. Before it used to be for particular code a test case is mapped. Now for a code package, test cases are being mapped. As we have done this correlation only some amount of data in the Swedish ICT company. As the code base of the Swedish ICT company is very large and complex and it is holding a large amount of data. The evaluation of the correctness of the correlation will be considered by checking the accuracy in the correlations which we have done in the result section. The code with the highest correlation accuracy will be best correlation pair for the code packages and tests.

We have considered programming components to use weak mutation process. The programming components used are Arithmetic expressions, Relational expressions, Boolean expressions. These are the three program components used to do weak mutation testing. That is, we are targeted to introduce mutants only to the arithmetic expressions, relational expression and Boolean expression. The generated mutants while using weak mutation is done only for these four program components. From the research study, we came to know that these program components are more effective while for weak mutation [27].

The variants we used while using weak mutants are stated and discussed below. These are the weak mutation variants used when conducting the experiment.

Expression-EX-WEAK. WEAK/1: This compares the states after the execution of the innermost expression with the mutation operators.
Statement- ST-WEAK/1. WEAK/1: This compares the state after the first execution of the mutated statements.

Basic-Block- BB-WEAK/1. WEAK/1: This compares the state of the first execution of the basic mutation block.

Basic-Block- BB-WEAK/1. WEAK/N: This compares the state of nth execution of the basic mutated block.

This variant of weak mutants is explained with an example in this document. As the source code provided cannot be written in the document due to the code confidentiality. The process of using the variants are explained using a sample code.

Let us take a sample C code on “min” program. This code is replaced using weak mutation testing. The operator used in this CAR mutation operator. CAR is the constant array replacement operator.

```c
1 int min (int* input, int size)
2 {
3     int i, result;
4     result = input [0];
5     for (i =1; i> size; i++) {
6         if (input[i] < result)
8         result = i;
9    }
10    return result;
```

**Figure 6-2 Sample C program 1**

Using the C mutator operator CAR (constant array replacement)

<table>
<thead>
<tr>
<th>Variants used</th>
<th>Original program code</th>
<th>Mutating code result using CAR Operator for the same code</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX-WEAK/1</td>
<td>Result</td>
<td>Input[result]</td>
</tr>
<tr>
<td>ST-WEAK/1</td>
<td>Input[i] &lt; result</td>
<td>Input [i] &lt; input [result]</td>
</tr>
</tbody>
</table>

**Table 6-1 Sample C program output when using CAR mutation operator 1**

The table explains what kind of mutants are introduced when the CAR mutation operator is used while weak mutation testing with the variants EX-WEAK/1 and ST-WEAK/1

For BB-WEAK/1 the program state in line 8 will be compared. For BB-WEAK/N the program state in line 8 is compared every time the line is executed.
6.1 Threats to internal and external validity

6.1.1 Internal threats

Internal threat is the type of threat that the researcher does not acknowledge. This type of threat occurs because of the problem with the tools used for experiment or the design required for experimentation itself. We need to make sure the challenges faced during the previous research should not be encountered again. To avoid this internal threat, we need to repeat the experiment multiple times. In our case, the testing is done to make sure that number of test cases failed. By repeating this experiment multiple times, the internal threat has been mitigated.

6.1.2 External threat

In the external threat, the results should be generic and could be applicable to the generic environment. This is called an external threat. Even though the results are not generic, and it can correspond to the results that would be valid for the other model’s application and environment [28]. Coming to the threat in our research the data we used is the historical data provided by the Swedish ICT company though we formed the correlations to the maximum extent. Explaining correctly the data may be old to test but this testing is helpful in the future use of reducing the test suite. Therefore, this threat has also been mitigated.

6.1.3 Reliability

Research is reliable when the bias can be reduced [29]. The research is done on limited programs and test cases, analyzing the theory on only a few test cases and programs. Reliability was achieved after evaluating the results with statistical tests. This way reliability is achieved from the study.

6.2 Standard Deviation 1 for regular mutation using Milu tool

<table>
<thead>
<tr>
<th>Source codes</th>
<th>Mean value</th>
<th>Standard deviation</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code 1</td>
<td>6.33</td>
<td>2.05</td>
<td>4.22</td>
</tr>
<tr>
<td>Source code 2</td>
<td>6</td>
<td>1.63</td>
<td>2.66</td>
</tr>
<tr>
<td>Source code 3</td>
<td>6.33</td>
<td>1.24</td>
<td>1.55</td>
</tr>
<tr>
<td>Source code 4</td>
<td>12.33</td>
<td>2.88</td>
<td>8.22</td>
</tr>
<tr>
<td>Source code 5</td>
<td>9.66</td>
<td>1.24</td>
<td>1.55</td>
</tr>
<tr>
<td>Source code 6</td>
<td>9.66</td>
<td>2.05</td>
<td>4.22</td>
</tr>
<tr>
<td>Source code 7</td>
<td>4.66</td>
<td>1.24</td>
<td>1.55</td>
</tr>
<tr>
<td>Source code 8</td>
<td>14</td>
<td>3.75</td>
<td>14</td>
</tr>
<tr>
<td>Source code 9</td>
<td>8.33</td>
<td>2.49</td>
<td>6.22</td>
</tr>
</tbody>
</table>

*Table 6-2 Standard deviation 1 for failed test cases while using regular mutation 1*
6.3 Standard Deviation 2 for weak mutation using Milu tool

<table>
<thead>
<tr>
<th>Source codes</th>
<th>Mean value</th>
<th>Standard deviation</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code 1</td>
<td>11.33</td>
<td>2.49</td>
<td>6.22</td>
</tr>
<tr>
<td>Source code 2</td>
<td>10.33</td>
<td>2.49</td>
<td>6.22</td>
</tr>
<tr>
<td>Source code 3</td>
<td>8</td>
<td>2.16</td>
<td>4.66</td>
</tr>
<tr>
<td>Source code 4</td>
<td>20</td>
<td>4.08</td>
<td>16.66</td>
</tr>
<tr>
<td>Source code 5</td>
<td>15.66</td>
<td>6.88</td>
<td>6.88</td>
</tr>
<tr>
<td>Source code 6</td>
<td>12.66</td>
<td>2.05</td>
<td>4.22</td>
</tr>
<tr>
<td>Source code 7</td>
<td>5.33</td>
<td>1.69</td>
<td>2.88</td>
</tr>
<tr>
<td>Source code 8</td>
<td>19.33</td>
<td>3.29</td>
<td>10.88</td>
</tr>
<tr>
<td>Source code 9</td>
<td>9.33</td>
<td>2.49</td>
<td>6.22</td>
</tr>
</tbody>
</table>

Table 6-3 Standard deviation 2 for failed test cases while using weak mutation

We have considered nine source codes for our experimentation process. These nine source codes have been tested with both regular mutation and also with the weak mutation. As these nine source codes cover all the default camera codes in the Swedish ICT company we have tested only these nine source codes. For each source code, the experiment is repeated for 3 times. The mean value is calculated based on the number of test cases failed for each source code during 3 iterations. Table 6-2 represents the mean value and standard deviation values of the failed test cases while using regular mutation. Table 6-3 represents the mean value and standard deviation values of the failed test cases while using weak mutation. From the table 5 we can see the standard deviation varies from 1.24 to 3.75 while using regular mutation, and from table 6 we can see the standard deviation varies from 1.69 to 6.88 this variation is because the provided source code lines are different for different code and to writing the test cases to the provided code also varies from one source code to another source code. For some codes, the test cases written are more and the failed test cases are also more compared to the small source code. The main motivation to write more test cases to the large source code is to cover the whole code functionality and make the maximum test cases fail to that particular code.

6.4 Overhead of weak mutation and regular mutation

<table>
<thead>
<tr>
<th>Source Codes</th>
<th>Effective Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Code 1</td>
<td>2.19</td>
</tr>
<tr>
<td>Source Code 2</td>
<td>2.05</td>
</tr>
<tr>
<td>Source Code 3</td>
<td>2.94</td>
</tr>
<tr>
<td>Source Code 4</td>
<td>3.01</td>
</tr>
<tr>
<td>Source Code 5</td>
<td>2.12</td>
</tr>
<tr>
<td>Source Code 6</td>
<td>2.14</td>
</tr>
<tr>
<td>Source Code 7</td>
<td>2.46</td>
</tr>
<tr>
<td>Source Code 8</td>
<td>2.51</td>
</tr>
<tr>
<td>Source Code 9</td>
<td>2.40</td>
</tr>
</tbody>
</table>

Table 6-4 Effective size of regular and weak mutations

To find the standardized difference between regular mutants and weak mutants we have calculated Cohen’s d effect size. As the introduced mutants are different in both regular mutation and weak mutation to find the mean difference between those we have calculated the Cohen’s value which is
shown in table 6-4. From the table 6-4 we can see the Cohen’s d value for different source codes are 2.19, 2.05, 2.94, 3.01, 2.12, 2.14, 2.46, 2.51, 2.40.

6.5 Reduction of Computational Cost

The percentage of reduced computational cost in both regular and weak mutation is shown in the table below.

<table>
<thead>
<tr>
<th>Mutation Process</th>
<th>Percentage of reduced computational cost</th>
<th>Measure in execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Mutation</td>
<td>70%</td>
<td>5.5 hours</td>
</tr>
<tr>
<td>Weak Mutation</td>
<td>93%</td>
<td>1 hours</td>
</tr>
</tbody>
</table>

Table 6-5 Reduction of Computational Cost

As from the Literature review, we came to know that while using the regular mutation the computational cost is being reduced by 61% in which we got a 70% reduction of the computational cost. In the weak mutation from the literature review, it is stated that weak mutation can reduce the computational cost by 91%. In our tool, we have reduced both runtime cost and execution reduction cost and produced 93% of the reduction of the overall computational cost.
7 CONCLUSION AND FUTURE WORK

7.1 Conclusion

This research on experimentation to form the correlations in generating test selection statistics using automated mutation testing in the continuous integration environment for improving test cases selection in regression testing. Mutation testing is a costly testing process to do at the industrial level. Though it is a costly process we have found that using the weak mutation operators can reduce the computational cost and execution cost of the testing process. If the computational cost is reduced, we can save up to 80% of the cost of the testing process. Coming to the experimentation we mutated the code packages to obtain failed test cases. We also compared the Milu tool with regular mutant’s introduction to the code and weak mutant introduction to the code. For which we have obtained the results when using the regular mutants, we have formed 62.1% of the correct correlations. When we mutated the source code with the weak mutants using Milu tool we have obtained the correctness of correlation up to 85%. Extracting this data is major constraint faced during our research. The present Swedish ICT company continuous integration system needs to be improved and the storage capacity of their historical data needs to be increased. From the experimentation results we can also conclude that the computational cost while using weak mutation is reduced by 93%. Finally, we conclude that that correlation to the code packages and tests can be formed and regression test suite size is minimized from these correlations. The implementation was done with weak mutation method. The weak mutation method can be used for black box test selection. Although all languages constructs weren’t tested. The proposed hypothesis can be used for test case selection in industrial application and shown to be effective to reduce the test suite size by 85% and can be a major time-saving method. From the experiment performed it can be concluded that by introducing mutants to the software under test and testing it against the regression tests suite can correlate code packages to test cases and these correlations can be used for test selection. The Execution cost and computational cost can be controlled by using weak mutation testing using higher order mutants.

7.2 Future Work

By performing this experiment and literature review and after scrutinizing the results, this research proposes that by producing the failed test data using mutation testing can lead to form the correlations. We have done this research only based on the C programming language as the Swedish ICT company products are mainly in the C language. Future work on this research can be carried out by forming the relations to different programming languages and we can also build an automation tool that can automate the test selection statistics directly. In this research, the proposed hypothesis is shown to be working but methods more accurate and efficient can also be further improved if there is a better method than weak mutation testing.
REFERENCES


