MUTATION TESTING:
A comparison of mutation selection methods

Master Degree Project in Informatics
One year Advanced Level 30 ECTS
Spring term 2012

Hans Hagman

Supervisor: Birgitta Lindström
Examiner: Gunnar Mathiason
Mutation Testing:
A comparison of mutation selection methods

Submitted by Hans Hagman to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the School of Humanities and Informatics. The project has been supervised by Birgitta Lindström.

24 October 2012

I hereby certify that all material in this dissertation which is not my own work has been identified and that no work is included for which a degree has already been conferred on me.

Signature: ____________________________________________________________
Mutation Testing:
A comparison of mutation selection methods

Hans Hagman

Abstract

Software is all around us in our lives in the industrialized world, and we as a society and individuals need it to function correctly. Software testing fills the role of performing behavior audits, to guide the correction of the software to its intended behavior. The consequences of faulty software can range to the late arrival of trains, to nuclear meltdowns.

This places quality requirements on the software of various levels. Program based mutation testing provides a high level of faultfinding capability. It does this by injecting many synthetic faults into the code under test, as described by mutation operators. These faults are used to search for testcases that would identify such faults, and consequently find real faults that the synthetic faults mimic.

However, mutation testing is costly on three accounts; each mutant of the original code is compiled, each mutant should ideally have an associated testcase to reveal that fault the mutant contains, finally the testcases are analyzed thoroughly by looking the output of the original and mutants to reveal the error in behavior.

In order to reduce cost while maintaining a high level of faultfinding, selective mutation testing is investigated, it uses a subset of all the available mutation operators. The investigation found that using Absolute value-, and Relational operator-, mutation reduces cost of mutation testing by 80%, while uncovering 83% of the injected faults.

Keywords: Software Testing, Mutation, Testing Effectiveness, Selective Mutation
Acknowledgements

Wow, it's done. There are so many people I would like to thank, that someway have helped me with the thesis.

My amazing wife, your patience and devotion, to me, our daughter, our family, is astounding, I love you. Your backrubs, thoughtful words, nudging and pulling to get me going when I get stuck.

My wonderful daughter, you are so beautiful and curious of the world around you. I miss your sister. Forgive me for the aspirations I place on you. Perhaps not helped me directly, but I cherish every hug and kiss you give me, it gives me so much energy.

My parents, for taking care of my daughter, so that I could write on weekends. Yes dad, you can call me ‘magister’ now.

My wife’s parents, for providing a restful place in their home.

Erik, my brother, thanks for your help with the figures, and going over my text.

Nils, my brother, thanks for all the encouraging words.

Classmates who have challenged me throughout the years, and made me grow.

My examiner, Gunnar, with great integrity but still in a sense of camaraderie, you have thoroughly judged my work with utmost care.

Foremost my supervisor Birgitta, the thesis might have been my project, but I was your project, you say that you see my potential. Through our years as teacher and student, I have never learnt so much or grown so much. Thank you, my mentor, my friend.

And every day, everywhere, our children spread their dreams beneath our feet.
And we should tread softly.
- Sir Ken Robinson
# Table of contents

Abstract ............................................................................................................................. II
Acknowledgements............................................................................................................ III

1 Introduction ....................................................................................................................... 8
  1.1 Thesis outline .................................................................................................................. 8

2 Background ...................................................................................................................... 9
  2.1 Software quality assurance ........................................................................................... 9
      2.1.1 Software fault and detection .................................................................................. 9

2.2 Software testing rationale ............................................................................................. 10

2.3 Mutation testing .......................................................................................................... 11
      2.3.1 Mutation testing cost ......................................................................................... 14
      2.3.2 Selective mutation .............................................................................................. 14

3 Problem Description ....................................................................................................... 16
  3.1 Cost effective imperative .............................................................................................. 16
  3.2 High quality testing ..................................................................................................... 16
  3.3 Cost of Mutation based testing .................................................................................... 16
  3.4 Aims and Objectives .................................................................................................... 16
  3.5 Expected outcome ........................................................................................................ 17

4 Method ............................................................................................................................ 18
  4.1 Investigate and identify available mutation selection methods .................................... 18
  4.2 Investigate faultfinding capabilities and cost ............................................................... 19
  4.3 Compare and contrast the selected methods ................................................................. 19

5 Survey of mutation methods and mutants ..................................................................... 20
  5.1 All mutants definition ................................................................................................. 20
  5.2 Methods for selective mutation ................................................................................... 20
      5.2.1 N-Selective ......................................................................................................... 21
      5.2.2 Constrained ABS/ROR mutation ....................................................................... 21
      5.2.3 Expression-selective ............................................................................................ 22

6 Experiment .................................................................................................................... 23
  6.1 Mutation ...................................................................................................................... 23
  6.2 Framework .................................................................................................................. 24
  6.3 Equivalent mutants ...................................................................................................... 26
  6.4 Representative programs ............................................................................................ 26
      6.4.1 Program descriptions ............................................................................................ 27
      6.4.2 Program adaptation .............................................................................................. 28
  6.5 Testcase generation ..................................................................................................... 28
  6.6 Experiment result gathering ......................................................................................... 30
  6.7 Method comparison ..................................................................................................... 30

7 Results ............................................................................................................................ 32
  7.1 Result sanitation ........................................................................................................... 32
  7.2 Mutation ...................................................................................................................... 32
      7.2.1 QuickLZ .............................................................................................................. 33
      7.2.2 MD5 .................................................................................................................... 34
      7.2.3 TriTyp ................................................................................................................ 35
Table of Figures

<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Simple C Function</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>Mutated C function</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>Mutation adequacy score equation</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>Experiment overview</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Selective mutation</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Mutation log excerpt</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>Framework layout</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td>Testcase generation</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>Method comparison</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>QuickLZ Mutation</td>
<td>33</td>
</tr>
<tr>
<td>11</td>
<td>MD5 mutation</td>
<td>34</td>
</tr>
<tr>
<td>12</td>
<td>TriTyp mutation</td>
<td>35</td>
</tr>
<tr>
<td>13</td>
<td>Mutant reduction QuickLZ</td>
<td>36</td>
</tr>
<tr>
<td>14</td>
<td>Mutant reduction MD5</td>
<td>37</td>
</tr>
<tr>
<td>15</td>
<td>TriTyp Mutant reduction</td>
<td>38</td>
</tr>
<tr>
<td>16</td>
<td>Method internal score</td>
<td>39</td>
</tr>
<tr>
<td>17</td>
<td>QuickLZ result</td>
<td>40</td>
</tr>
<tr>
<td>18</td>
<td>MD5 result</td>
<td>41</td>
</tr>
<tr>
<td>19</td>
<td>TriTyp result</td>
<td>42</td>
</tr>
<tr>
<td>20</td>
<td>Constrained ABS/ROR result</td>
<td>43</td>
</tr>
<tr>
<td>21</td>
<td>Expression result</td>
<td>44</td>
</tr>
<tr>
<td>22</td>
<td>N-Selective result</td>
<td>45</td>
</tr>
<tr>
<td>23</td>
<td>Methods result</td>
<td>46</td>
</tr>
<tr>
<td>24</td>
<td>Mutant fault detection classification</td>
<td>51</td>
</tr>
<tr>
<td>25</td>
<td>Common error example</td>
<td>52</td>
</tr>
<tr>
<td>26</td>
<td>MuJava Mutation</td>
<td>58</td>
</tr>
<tr>
<td>27</td>
<td>Fortran Mutants</td>
<td>60</td>
</tr>
</tbody>
</table>
Figure 28 C Mutants........................................................................................................62
Figure 29 Mutation operator conversion Fortran..........................................................72
Figure 30 Mutation operator conversion C......................................................................73
Figure 31 Testcase generation for Trityp........................................................................77
Figure 32 Testcase generation for MD5...........................................................................77
Figure 32 Testcase generation for QuickLZ.....................................................................78
Figure 33 Changed main method in MD5.......................................................................79
Figure 34 Original main method in MD5.........................................................................79
Figure 35 TriTyp changed main Method .........................................................................80
Figure 36 TriTyp original main method.........................................................................80
Figure 37 TriTyp removed method.................................................................................81
Figure 38 QuickLZ introduced main method.................................................................82
1 Introduction

The purpose of this report is to investigate ways to improve the effectiveness and efficiency of mutation testing, using various methods for selective mutation.

This is investigated using a framework to mutate programs and test the faultfinding capabilities of the mutation methods with respect to cost and score.

Mutation testing is very computationally expensive, and selective mutation testing is a means to reduce the cost. Reducing cost while maintaining a high faultfinding capability save precious resources, which can be spent on making the software better.

The experiment showed that for the programs experimented upon; using absolute values and the relational operators testing provides a good compromise of effectiveness and efficiency.

1.1 Thesis outline

Chapter 2 provides an introduction into the domain of software quality and mutation testing; such as what software quality is, how to find faults in software, the reasons why software needs to be thoroughly tested, finally it explains software mutation, its use, benefits, costs and tradeoffs.

The third chapter describes the problems that this thesis is concerned with. Software quality is costly and often overlooked; nevertheless, high quality testing is of central concern. The chapter looks at these issues and sets out goals for the study to address these issues.

Chapter 4 describes in detail how to achieve the goals set out in the previous chapter. Chapter 5 presents the result of the literature survey.

Chapter 6 describes the procedures carried out in the experiment. The framework and tools used are presented, the programs used for mutation is presented, and how the experiment was carried out.

The seventh chapter presents the results of the experiment. Specifically, how much do the different methods safe in terms of resources and their level of faultfinding.

Chapter 8 presents a comparison of the different methods chosen for the experiment. Here the different methods are summarized across the different programs and compared to each other.

Chapter 9 presents the conclusion from the experiment.

The tenth chapter describes related works and reports.

Finally, chapter 11 presents future work.
2 Background

Section 2.1 gives an introduction of the field of software quality assurance. Section 2.2 provides an overview for the rationale of software testing. Then 2.3 describe the concepts of mutation testing.

2.1 Software quality assurance

The purpose of software is to have a computer to perform tasks. Achieving these tasks is done by a set of instructions performed by some hardware. These instructions may be faulty, because of mistakes made when designing the tasks or misunderstanding the purpose of the tasks, introduced within the continuous design and implementation of the software. Eventually, given enough time, anyone will make a mistake, and these mistakes give consequences in the form of bugs in code. These bugs may cause an error, which might lead to a failure, i.e. an incorrect behavior or computation. Detecting and correcting these faults is critical to ensure the reliability of software.

Software quality assurance has two distinct purposes (DeMillo et al., 1979; Branstad et al., 1980; Adrion et al., 1982; IEEE, 1990, 2005; Carnegie-Mellon University, 2002); verification and validation:

- **Verification** (IEEE, 1990, 2005; Carnegie-Mellon University, 2002); is that the software under study to ensure “that you built it right”. This requires that those developing the software have correctly understood the requirements from previous development phase and implemented them to form a correct behavior.

- **Validation** (IEEE, 1990, 2005; Carnegie-Mellon University, 2002); is checking that “you built the right thing.” This relates to a correctly understood and developed purpose of a system, if the customer wants a program for booking rooms, this is what should be delivered to the customer, and not a system for calculating how many people could be in a room to ensure fire safety.

2.1.1 Software fault and detection

Software faults are commonly modeled by the “Fault-Error-Failure chain” or the RIP Model (Morell, 1983, 1988; Offutt, 1988; DeMillo and Offutt, 1991). The RIP model consists of three components - that together explains why software produces incorrect results or fails - **Reachability**, **Infection** and **Propagation**. Understanding how these interact helps testers and programmers to find and correct erroneous code.

- **Reachability** is the notion whether a fault present in code the code can be reached and executed, such a fault may be avoided on certain occasion by branches in the code, and faults residing in unreachable code can never be executed. It is only when the fault is reached and executed, that it may cause problems.

- **Infection** is whether execution of the fault leads to an incorrect internal state of the program. A line containing a fault may be such that executing it will put the program in an incorrect state, an error, thus we have infection. Incorrect states could for instance be the arithmetic overflow, caused by adding two numbers that exceed memory space. Adding two small numbers would not produce the incorrect state, given that the result is within memory space bounds.

- **Propagation** is what is commonly observed as the software performing erroneous, when the error propagates to the interface causing an observable deviation from the intended behavior i.e. a failure. Tough a failure may be observable it is not necessarily observed.
Closely to these concepts is the Observability and Controllability (Schütz, 1994) of software. Observability (Schütz, 1994) is the degree to which the output, internal state and the effects on and by the environment can be observed during testing. Schütz describes this as “...When a system is tested, it is necessary to evaluate or judge the "correctness" or "appropriateness" of the system behavior. In order to do this, one must observe or monitor for each test execution what the system does, how it does it, and perhaps when it does it. The system must facilitate such observation, i.e., it must be observable ...”. Programming languages that support reflection, such as Java (Oracle Corporation, 2012) or Python (Python Software Foundation, 2012), allows for greater observability, as internal states can be monitored during execution.

Controllability is a measure of how well one can control the execution of a program, ranging from strictly deterministic programs, to programs that include elements of randomness that are hard to control (Schütz, 1994). The output of the program may depend on hidden internal variables that are impossible or difficult to control, adding to the problem of accurately verifying proper behavior. Deterministic programs will have a clear and constant path for testing, according to faults as described in terms of the RIP-model, whereas non-deterministic programs will not have a recurring path to reproduce a failure.

Programs are tested using testcases (Balcer et al., 1989; Stocks and Carrington, 1996), these contains: testcase values, expected values, prefix values, postfix values; verification values and exit commands. Prefix values are those that are supplied to the program under test, to set the program up for testing, placing it in a state where it can accept and run supplied testcase values. Postfix values have two subtypes; verification values and exit commands. Verification values are those needed to attain the desired output from the system. Exit values places a program in a state to be tested again or exits the program. Using the output of the program, it is evaluated against the expected values, and if they do not match, the program is deemed to have failed in that particular testcase, indicating the presence of a fault.

2.2 Software testing rationale

Today software exists in almost everything we use in our daily lives, alarm clocks, car brakes, TV’s, stereos, mobile phones, and space shuttles. By their presence in our daily lives, it is important that the software work correctly. Our continued and increased dependency on software require that it be tested properly, as faulty software could cause severe harm to humans, grave economic loss, or place them in undesired or awkward situations. Modern society because of its integration of software would seriously impede its function without working software components.

The core aim of software construction is to produce software that does what it is designed to perform, and that it is performed correct. One of the missions for testers is to assess software quality, by creating tests to determine whether the program lives up to these goals. The software industry today (Myers, 1979; Myers and Sandler, 2004), same as 25 years ago, places as much as 50% of development resources on software testing, a process that is expensive and laborious. Software mogul Bill Gates (Foley and Murphy, 2002) stated: “...we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We’re more of a testing, a quality software organization than we’re a software organization. ...”.

Mature software testing can be defined according to Bezier (1990), as “... a mental discipline that helps all IT professionals develop higher quality software”. The purpose of the testing process is not to designate blame for incorrect code but rather, as a team, work together to jointly produce high quality software. Quality requirements on software continually increase, so does the cost of software testing. Delicate balancing of cost and quality becomes an important question to the industry as a whole. Customers demand high quality software, and
Background

this is a growing demand, software failures can thereof lead to significant economic and public relations damage to software companies.

2.3 Mutation testing

Mutation in software is based on the idea of mutation; small changes are made to the code or input, to produce differing sets of behaviors or attributes. The initial idea was presented by Richard Lipton “Fault Diagnosis of Computer Programs” (Lipton, 1971) in a 1971 student paper, and Timothy Budd (Budd, 1980; Budd et al., 1980) produced the first mutation tool in the field.

There are different uses for mutation in software testing (Woodward, 1990; Offutt and Untch, 2001; Ammann and Offutt, 2008), such as:

- Program-based mutation testing (DeMillo et al., 1978; Budd, 1980; Budd et al., 1980).
- Integration mutation (Kim et al., 2000; Vincenzi et al., 2001).
- Specification-based mutation (Budd and Gopal, 1985).
- Input space mutation (Liu and Tan Hee Beng, 2009).

Program based mutation-testing focuses on introducing mutations into the code to mimic real faults in computer code. This thesis has its focus on program-based mutation; as such, further references to mutation testing refer to program-based mutation testing.

The usage of mutation testing rests on the hypothesis of the competent programmer and the coupling effect (Budd et al., 1978, 1980; DeMillo et al., 1978). The competent programmer hypothesis assumes that a competent programmer will create a program that is close to being correct. Injecting faults that change the behavior could point out errors in the code, by analyzing the output of the original and mutants.

The coupling effect states that finding small faults will be sufficient to remove the complex faults present in the program under test. DeMillo et al. (1978) states about the coupling effect: “Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors.”

A program that has some change applied to it, a mutation, is said to be a mutant. Mutants can be said to be in one of several states (DeMillo et al., 1978; Budd et al., 1980; DeMillo and Offutt, 1993). The states being living, dead on arrival, dead, or equivalent. Running a test case against the mutants, depending on the results, a mutant remains in a state or transitions to another.

- Living mutants has as so far, not shown any difference in the behavior of the original and mutated program.
- Dead-on-arrival mutants will not compile. This could occur when the mutation violated the language syntax.
- A dead mutant has been killed by a testcase causing the output of the mutant to differ from the output of the original program. Such a mutant has satisfied its requirement of finding a useful testcase, since it will be killed by that testcase. The testcase is saved and used to expose a certain type of fault that the mutant represents.
- Equivalent mutants are those that produce the same results as the original code, given any possible input. It can prove hard to discern whether a mutant is equivalent or just hard to kill, but researchers have had some success in revealing these equivalent mutants (Baldwin and Sayward, 1979; Tanaka, 1981; Hierons et al., 1999; Grün et al., 2009).

Mutations are introduced into the programs, creating a clone of the original program but modified in a single instance, this is called first order mutation. Each change produces a
modified version of the program, a mutant. Each mutant contains a single fault. Second order mutation (Polo et al., 2009) introduces two mutations in the code, and so on with n-order mutation.

The set of mutants is used to create a set of testcases. The goal is to have a set of testcases that has the power to kill all the mutants. Testcases that can produce a different output on the original and at least one mutant is said to be adequate, failing this they are called inadequate.

Mutation testing has as its focus to test the test set itself, for each mutant the premise is to check whether the different behavior will be detected by a testcase. The basic function is captured well by Geist (1992) as:

“...in practice, if software contains a fault, there will usually be a set of mutants that can be killed only by a test case that also detects that fault. ...”

Mutation testing techniques are said to have a very strong fault detection rate as compared to other techniques (Offutt et al., 1993; Offutt, Lee, et al., 1996; Offutt and Voas, 1996). However, it is also known to be expensive. Frankl et. al. (1997) and Offutt, Pan, et al. (1996) found that when comparing all-uses coverage criteria with mutation testing, mutation performed better; Frankl et al. writes “…overall, mutation testing did better than all-uses. ...”, but did so at a higher cost in terms of mutant generation and testing for equivalent mutants.

Observe the example below, in Figure 1 Simple C Function is a simple C function that sums the values inside in a provided array. For each number in the array, its value is added to the variable s. Running the testcase \([0,0,3,2]\) through the function would give the result of “5” from the output. This number represents the output of the original unmutated code, the expected value.

```c
//Effects: If x null throw NullPointerException
// else return the sum of the values in x
int sum (int[] x){
    int s = 0;
    for (int i=0; i < x.length; i++){
        s = s + x[i]
    }
    return s;
}
```

**Figure 1 Simple C Function**

After Ammann and Offutt (2008)

Figure 2 Mutated C function gives an example of a mutation operator applied to the previous C function, the mutated line is marked by a comment, denoting the type of mutation. Here the arithmetic operator addition is swapped by subtraction. Because of this, the testcase \([0,0,3,2]\), would yield the result of “-5” from the output, while the expected correct result is still “5”.

```c
//Effects: If x null throw NullPointerException
// else return the sum of the values in x
int sum (int[] x){
    int s = 0;
    for (int i=0; i < x.length; i++){
        s = s - x[i]
    }
    return s;
}
```
After Ammann and Offutt (2008)

Testcases containing null or incorrect types, will never reach the mutated instruction and any testcase where the sum of the array is zero, will not kill the mutant. However, this trivial mutant is easily killed by most testcases. Certainly, there are mutants that are far more difficult find than in the example. Differencing results of the two functions indicates that a mutant has been found and killed, since it does not produce the same result in some situations.

However, it should be noted that, mutants might in fact have the intended correct behavior. This is a consequence of incorrectly programmed behavior in the original. It is in the analysis of the testcase using the original and mutant, that critical thinking must be applied to identify the source of failures.

*Mutation score* or *mutation adequacy score* (Hamlet, 1977; DeMillo et al., 1978) is a measure used during and after the generation of mutant killing testcases, see Figure 3 Mutation adequacy score equation, and a score of 100% means that for every mutant, a revealing testcase is noted in the test suite, as such the test suite is said to be *mutation adequate*. However, in practice a threshold is usually set on a lower level e.g. 90%, because of resource restraints and equivalent mutants.

\[ \text{MutationScore} = \frac{\# \text{DeadMutants}}{\# \text{TotalMutants} - \# \text{EquivalentMutants}} \]

*Figure 3 Mutation adequacy score equation*

*Mutation operators* are abstract descriptions on how to change code to create mutants (Agrawal et al., 1989; DeMillo and Offutt, 1991; King and Offutt, 1991; Kim et al., 2000, 2001; Ma and Offutt, 2005a, 2005b), containing mutation *primitives* that describes how to perform a single mutation of code. Members of a set are replaced by other members from the same set. Mutation operators, in this case from the set of Mothra mutants, may contain the sets of:

- Relational operator primitives: [<, >, ==, <=, >=, !=, trueOp, falseOp]
- Arithmetic operator primitives: [+, -, *, **, /, %, leftOp, rightOp]
- Conditional operator primitives: [||, &&, &, |, ^, falseOp, trueOp, leftOp, rightOp].

Among several other sets available to apply on program code. These primitives are introduced into code to alter it, according to a matching set of patterns applicable for mutation. This transformation of code could result in a differing set of output as compared to
the original code; in order to detect this, testcases are executed on the original and mutated code.

The benefits of mutation testing were initially to test other methods of software testing, by using exhaustively reviewed code (Offutt et al., 2004; Ma, Harrold, et al., 2006), to generate mutants (Andrews et al., 2005) to mimic real faults in code. Researchers would generate mutants to use as faults, and different test coverage criteria’s could be compared to gauge their performance.

### 2.3.1 Mutation testing cost

Mutation testing is costly (Weiss and Fleishgakker, 1993; Mresa and Bottaci, 1999) on three accounts; firstly, each possible mutant needs to be generated and compiled. Secondly, a set of testcases need to be deduced that can kill all mutants. This can largely be automated (Offutt, 1988; Ayari et al., 2007; Blanco et al., 2009). Finally, equivalent mutants must be discovered (Wong and Mathur, 1995), which on practical applications is hard, if not impossible to achieve within real world cost and time constraints of software testing.

There are tools for creating mutants such as MuJava (Ma, Offutt, et al., 2006), PIT (Coles, 2012) or MuClipse (Smith and Williams, 2009), using syntax transformation rules which greatly reduces the cost of mutant generation. The tools analyze program code, and when an applicable operator is identified, replace that operator with every possible operator from the same set, as demonstrated in the example above.

Parsing a program will produce a large set of mutants (Acree et al., 1979; Offutt, Lee, et al., 1996). The number varies with the properties of the program, the mutants used and programming language. Operators vary in the number of primitives they contain.

The arithmetic operator in MuJava contains five primitives and the assignment operator in MuJava contains eleven primitives, influencing the number of mutants created based on program properties. Using subsets from the total number of mutants naturally produces a smaller set.

Programming languages also determine the total number of mutants, as different languages have different properties. Mutants have been created to suit the varying program languages. The MuJava mutants have 16 operators, whereas the Mothra mutants have 22 operators. The number of mutants in a program is typically in the order of \(O(\text{Data objects} \times \text{References})\) (Wong and Mathur, 1995; Offutt, Lee, et al., 1996).

Mutants are produced by systematically applying mutation operators on the software. Test data is then generated in such a way that, if the test data kills one mutant, the test data is saved as part of a successful testcase. Testcase deduction must continue as long as there are mutants still alive, usually a threshold is set to save resources.

Tough or equivalent mutants may however, cause problems. Equivalent mutants require special consideration to detect them, so that resources are not wasted on killing a mutant that cannot be killed. Testcase generation must be thorough to ensure a mutant score that is sufficiently high. The process comes with computational and operational cost, as generation should continue until all mutants have some associated testcase. Often the threshold for mutation adequacy is set below 100% to allow for the presence of equivalent mutants. The rigorous process leads to quality, but at the cost of time and resources.

### 2.3.2 Selective mutation

One way to reduce cost while maintaining a high level of mutation adequacy score is to use selection methods to reduce the number of mutants (Krauser et al., 1991; Offutt et al., 1993). Selection methods focus on selecting mutations in such a way that they also uncover tests applicable for other mutants. One such selection method could be to use only one primitive
from each set of operators. E.g. selecting only the mutation “m*n → m-n” from the arithmetic operators.
Problem Description

3 Problem Description

The following chapter will introduce and describe the problem; how to perform efficient and effective mutation testing.

3.1 Cost effective imperative

Software companies, such as IBM Corporation, Apple Inc., Oracle Corporation, SAP AG and Microsoft Corporation amongst others in the software industry, spend several billions on developing software systems, and research states that the industry as a whole spends half (Myers, 1979; Myers and Sandler, 2004) of their development budgets on testing.

Companies need to efficiently and swiftly produce software that fulfill customer needs, while trying to spend as few resources as possible, in order to make a profit from the development of the software itself. Reducing the cost for testing by applying cost effective tests, could give a competitive edge and save resources, but the test procedure cannot tradeoff quality of the tests, as failing software would result in customers taking their business elsewhere, to companies that produce software having less faults.

3.2 High quality testing

The demand on software quality is high and the demand continues to grow. Mutation-based testing provides a high code coverage that performs on par or better than other coverage criteria in testing such as statement and branch coverage (Walsh, 1985; Offutt et al., 2004; Koster and Kao, 2007), and all-uses method (Wong and Mathur, 1995; Offutt, Pan, et al., 1996; Frankl et al., 1997; Kakarla et al., 2011). Offutt et al. (2004) even calls it the golden standard of testing. The exhaustive and structured approach gives a high degree of confidence, that the code under review will be thoroughly tested, since so many conceivable faults are mimicked (Offutt, 1992).

3.3 Cost of Mutation based testing

The software industry and its customers are making increasing demands for quality software, because of the economic and the customer and public relation benefits that it provides; demand is not likely to decrease. The problem of mutation testing is its higher cost (Weiss and Fleshghakker, 1993; Mresa and Bottaci, 1999); however, given that the mutation approach gives a very high diagnostic performance of software systems, the return investment of time and resources is high.

The cost of mutation testing can be measured by using testcases as a base. Testcases can only indicate the presence of a fault, not the fault itself. Where the testcase provides a differing result from the original, the original and mutated software must be compared to investigate if there is a fault.

In order to compare the efficiency of the different methods, the test cases are counted in the generated test suites. There is a cost associated by each test case since it must be executed and its results analyzed every time it is used. It must also be maintained as the software evolves. There are other costs associated with mutation testing and testing in general, however, these issues lies outside the scope of this thesis.

3.4 Aims and Objectives

Mutation testing is as described, costly concerning several factors, one of these is the number of mutants produced, influencing the other factors. Each of these mutants must be compiled and preferably, a suitable testcase is deduced to reveal that mutant.
Literature suggests reducing costs by reducing the total number of mutants, by selective mutation. The focus of this thesis is to examine the different selection methods with respect to their associated cost and level of fault revealing.

Where *effectiveness* is understood as the faultfinding capacity of the test method, and *efficiency*; the cost associated with a method, is understood as the number of generated testcases.

Testcases are a good approximation of cost, since each test case must be executed, and the results of the original and mutants analyzed every time it is used. It must also be maintained as the software evolves. There are naturally other costs related to testing, such as time and computing power, understanding that these may vary over different development projects, and their variation introduces an unknown variable that this thesis cannot take into account.

Selective mutation methods maintain a level of effectiveness while increasing efficiency. If these methods are successful in this, we can use the results to make informed decision on how to use mutation selection methods.

The aim of this thesis is:

*Compare* cost-effectiveness and efficiency of mutant *selection methods* in mutation-based software testing, knowledge that can be used as a basis on how to apply mutation testing selection methods whilst maintaining *effectiveness* and increasing *efficiency*.

Completion of this aim is based on the following objectives:

- Investigate and identify available mutation selection methods.
- Investigate faultfinding capabilities and cost.
- Compare and contrast the selected methods.

The purpose of this study is to investigate the current state of mutant-based software testing with focus on mutant operator selection. There are many benefits from using mutants in the field of software testing, as described previously, amongst others the high degree of confidence from the test process and the ability to mimic real world faults.

### 3.5 Expected outcome

It is the purpose of this thesis to provide a detailed evaluation of selection methods and their respective operators. Describing each selected method including its potential benefits when used in testing, and what limitations that affect the use of such a method.

The resulting comparison of methods can be used to make an informed decision, on which mutation selection method to use, given demands for satisfying high quality software testing, *effectiveness*, while reducing the use of resources, *efficiency*. 
Method

4 Method

Here, the methods and approaches for the different objectives are described.

4.1 Investigate and identify available mutation selection methods.

The first objective is to investigate currently available methods for selective mutation-based testing. A literature review of the methods available is chosen as the best possible way to achieve this. Literature reviews are a proven method to investigate a scientific field; the review provides a comprehensive list of the different applications available in mutation-based testing.

An alternative to the literature study is to interview active researchers, then again, this is dependent on the time these persons have at their disposal. Researchers have written books in the field, and bibliographic references could be used, in the same way as interviews, to point out relevant articles of interest. However, books rarely contain state of the art in fields of research, and references quickly become outdated as the books age.

Interviews of current testing practices in companies can also give insight into the methods used; but only a few tools are available for mutation testing, such as Jester (Moore, 2005) or Certitude (Springsoft Inc., 2012). The probability that mutation-based tools are used in software companies is low, because of the few mutation tools available, and traditional low adaptation of modern testing techniques and tools (Tassey, 2002) in software companies.

The approach for the literature survey is the following. Articles are investigated using scientific databases that provide full text articles, available to students at the University of Skövde. This set of relevant articles is further expanded via citations, until no new results emerge, achieving a transitive closure. From this set of methods described in the articles, a subset of the most promising 2-3 selection methods are selected; based on factors regarding ease of adaptation and available tools.

Methods that need special software adaptations, such as compiler reconstruction is removed as these need special tools, which may not be publicly available. Compiler adaptation is based on the premise that the compiler is changed, to produce a special version of the program under test. Compiler based method requires that the compiler be formally verified, a lengthy and error prone procedure, to ensure correctness this avenue of research is removed.

Methods based on code change premise however, require a much simpler set of tools. Once an operator is discovered, it can be replaced by the applicable set of mutation operators, available from that set. Implementing this requires only a tool to search and replace patterns with a mutated version of the detected string. This ensures validity as pattern recognition and change have readily available working tools.

Methods that are applicable to the Java™ programming language are selected as research programs exists for mutant testing in that language, such as MuJava (Ma, Offutt, et al., 2006), MuClipse (Smith and Williams, 2009) or Javalance (Schuler and Zeller, 2009).

Each of the selected methods will base their selection of mutants on a common pool of mutants, here called “all-mutants”, as a representation of unrestricted mutation testing. The literature review must therefore also find a representation for the all-mutants set, according to the previously described literature search method. All-mutants can be used as a basis for comparison, based on that the methods select their mutation operators from a common pool of mutants.
4.2 Investigate faultfinding capabilities and cost

The different methods are analyzed and their test effectiveness measured. The methods need to be compared using criteria’s of cost, and a measure of faultfinding. Each method for generating mutation-based tests requires that a set of mutants and testcases be somehow generated; the number of testcases deduced for each selection method is used as a measure of cost.

In order to generate the mutants, a mutation tool is used, such as MuJava or MuClipse. Using the all-mutants to generate mutants and testcases, as a base for creating a maximum of mutants available in a program, the accompanying set of testcases created by a method can be used as a base for comparison. Finding input data to kill mutants takes time, and should be generated without introducing bias, preferably with a random generator.

The effectiveness of a method can be measured according to how well the mutants selected by a method, covers the mutants present in the all-mutants set. Effectively, how well does a method measure on the mutation adequacy score, compared to the all-mutants.

4.3 Compare and contrast the selected methods

Using the measurements for evaluation derived from the previous objective, the different methods for software testing can be compared to each other. The results of this comparison can be used as a means for evaluation; what methods are the most appropriate, for reducing cost and of mutation-based testing while keeping as good test effectiveness as possible.

Equivalent mutants are ignored, by setting the mutation adequacy score of the experiment at a threshold and a timer, where remaining mutants are simply considered as equivalent. These represent another possible source of bias, which can be mitigated by full disclosure for peer review, of all generated mutants. The all-mutants set is the base, any other method that requires a lesser set of mutants can be said to be proportionally smaller, and proportionally cheaper to perform, with regard to the lesser number of mutants required for using that method.
5 Survey of mutation methods and mutants

This objective focuses on three distinct sub objectives; research methods for selective mutation, find or synthesize a description of all-mutants, and finally finding a representative program to perform tests.

5.1 All mutants definition

The set of all-mutants is based on the program under test. MuClipse (Smith and Williams, 2009), uses the mutants as defined in MuJava (Ma, Offutt, et al., 2006). The Mutation operators in MuJava are originally based on C/C++/Fortran versions of mutation (Offutt et al., 1993, 2004) and researchers adopted these to the Java programming language. The set of mutant operators for MuJava and their behavior are available in appendix.

5.2 Methods for selective mutation

The literature search discovered six articles; describing 18 methods for selective mutation testing, (Offutt et al., 1993; Wong and Mathur, 1995; Offutt, Lee, et al., 1996; Mresa and Bottaci, 1999; Barbosa et al., 2001; Vincenzi et al., 2001). These articles contained sets of reduced mutation operands:

- Offutt et al. (1993):
  - N-selective
- Wong and Mathur (1995)
  - Randomly selected X% mutation
  - Constrained ABS/ROR mutation
- Offutt et. al. (1996)
  - Expression/Statement-selective
  - Replacement/Statement-selective
  - Replacement/Expression-selective
  - Expression-selective
- Mresa and Bottaci (1999)
  - EFF
  - EFA
- Barbosa et al., (2001)
  - SS-27
  - CSS-27
  - S-Offutt-27
  - S-Offutt-5
  - S-Selective-5
- Vincenzi et al., (2001)
  - SUS Sufficient Incremental Unit Testing Strategy
  - SIS Sufficient Incremental Interface Testing Strategy
  - U-IS Unit-Interface Incremental Testing Strategy
  - SU-IS Sufficient Unit-Interface Incremental Testing Strategy

These methods for selective mutation are created on different foundations; some are created on a hypothesis or after the fact. The first seven are created from the researchers’ hypothesis of effective mutation testing, that those mutations are those that are the most effective to use and researchers then experiment to investigate this. The remaining methods are created after the fact, by mutating a set of programs to use as a training set, computing the mutation score of the operands, and then selecting the mutants that achieve a high score, to find an efficient set of mutants.
The methods are investigated to ensure their compatibility with the MuJava framework, in order to be adaptable the methods are required to reach 90% adaptability. That is to say if a method contains 10 mutation operators and if more than one mutant operator is non-transferable the method is excluded.

The methods remaining after scrutiny are presented below. The selection methods must be applicable to the mutants available in MuClipse. Interested readers may refer to the appendix for more information on the other mutation methods.

5.2.1 N-Selective
Offutt et al. (1993) introduced “N-Selective” mutation where the operators are removed according to the number of mutants created; the operators that produce the most mutations are removed. Offutt et. al. investigated 2-, 4- and 6-Selective mutation, which is removing the two, four, and six most mutant producing operators. They found that the reduced set is almost as effective as unrestricted mutation for small programs, achieving a saving of up to 60% while retaining a mutation score of 99% or above compared to all-mutants. They used the Mothra mutation system for the Fortran-77 programming language. An in-depth description about Mothra is available in King and Offutt (1991).

N-selective mutation is easily adapted to the MuClipse framework. Each mutant operator is used a number of times, and of those the $n$th-operators that produce the most mutants are removed from the set. Mutation cost reduction can be calculated comparing the reduced set to the total set of mutants.

5.2.2 Constrained ABS/ROR mutation
Constrained ABS/ROR mutation henceforth referred to as Constrained A/R, (Wong and Mathur, 1995), removed all mutants but the ROR: relational operator replacement and ABS: absolute value replacement and force zero at execution.

The results from the experiments concluded that with even small sets of mutants, as low as 10% from the set of all mutants, results in a high mutation score with regard to ‘all mutants’. Achieving scores at 97% and above, while having an 80 percentile savings in the number of generated mutants, thus being very efficient. They also point out that this is hard to generalize since the size of the programs is very small, and that faults that are more complex may be present in larger programs, but remain optimistic to choosing a small set of mutants to save testing resources.

The list below contains the Mothra operators, with the related MuJava operators as sub items:

- ABS: test zero at execution and terminate if so, absolute value replacement.
  - AODU: Arithmetic Operator Deletion Unary
  - AOIU: Arithmetic Operator Insertion Unary
- ROR: relational operator replacement.
  - ROR: Relational Operator Replacement

Mutation using absolute values is not fully supported by the MuJava framework. The Mothra mutation operators (King and Offutt, 1991) for ABS is described as; absolute value at expression and variable level, absolute negative value at expression and variable level, and test zero at execution. Expression level mutation is no longer supported in MuJava, nor is test zero at execution available. Expression-level absolute value mutation was removed in version 2. Variable-level mutation is still supported with the mutation operators AODU: Arithmetic Operator Deletion Unary and AOIU: Arithmetic Operator Insertion Unary (Offutt, 2010; Ma et al., 2011).
Relational expression mutation (King and Offutt, 1991) is described as an expression containing the relational operators, the mutant contains the primitives \([<, >, ==, <=, >=, !, \text{TRUEOP}, \text{FALSEOP}]\). All primitives except the last two are supported in the MuJava framework, with the ROR operator.

### 5.2.3 Expression-selective

Expression-selective or E-selective mutation by (Offutt, Lee, et al., 1996), uses only the expression mutations, to generate mutants. The article describes this method as a means to save significant resources, whilst testing features regarding arithmetic and logical constructs in programs.

The list below contains the Mothra operators, with the related MuJava operators as subitems:

- **ABS**: test zero at execution and terminate if so, absolute value replacement.
  - o **AODU**: Arithmetic Operator Deletion Unary
  - o **AOIU**: Arithmetic Operator Insertion Unary
- **AOR**: arithmetic operator replacement.
  - o **AORB**: Arithmetic Operator Replacement Binary
  - o **AORS**: Arithmetic Operator Replacement Shortcut
- **LCR**: logical connector replacement
  - o **COR**: Conditional Operator Replacement
- **ROR**: relational operator replacement.
  - o **ROR**: Relational Operator Replacement
- **UOI**: unary operator insertion.
  - o **AOIU**: Arithmetic Operator Insertion Unary
  - o **AOIS**: Arithmetic Operator Insertion Shortcut
  - o **LOI**: Logical Operator Insertion

These mutants mostly carry over to the MuJava system (Offutt, Lee, et al., 1996; Offutt, 2005, 2010; Ma et al., 2011) (Ma, Offutt and Kwon, 2011; Offutt, Lee, Rothermel, Untch and Zapf, 1996; Offutt, 2005; Offutt, 2010).

The ABS operator in MuJava, as stated before, only has support for the variable level mutation in MuJava with the operators AODU and AOIU.

AOR has the primitives \([+, -, *, **, /, \%, \text{LEFTOP}, \text{RIGHTOP}]\) in Mothra (King and Offutt, 1991). This operand is subdivided in MuJava (Offutt, 2005) into the operators; AORB and AORS. All primitives except, ** (power of in FORTRAN), LEFTOP and RIGHTOP have matches in the MuJava framework.

LCR uses the FORTRAN language operators \([.\text{AND.}, .\text{OR.}, .\text{EQV.}, .\text{NEQV}]\), as well as the special mutation primitives \([\text{FALSEOP}, \text{TRUEOP}, \text{LEFTOP}, \text{RIGHTOP}]\) each occurrence is replace by the others in the set. The first four primitives are matched by the COR operand, however, the special mutation operands have no match in MuJava.

ROR: relational operator replacement is evenly matched in MuJava with the exception of the operators LeftOp and RightOp.

UOI: Unary operator insertion, in which ”...Each arithmetic expression is negated, incremented by 1 and decremented by 1. Each logical expression is complemented...” (King and Offutt, 1991). AOIU performs negations of arithmetic variables; however, this operator is already included above. AOIS inserts the four increment and decrement operands \([\text{value}++, \text{value}--, --\text{value}, ++\text{value}]\). Finally, logical complements are performed by the LOI in MuJava.
6 Experiment

The experiment rests on six parts; mutation of programs using a research tool, a framework to perform tests, programs to mutate, testcase generation, experiment result gathering, and method comparison. Below is a summarized description. The detail of each stage is described in the following chapters.

The experiment as laid out coarsely in Figure 4 begins by taking a program to mutate. The programs are mutated one time each for every selection method. The mutants are saved to file, and the framework is set to work. It generates input data to run against the mutants. Should the original and any mutant differ in their results, the mutant is marked as dead and removed from further testing, and a successful testcase is saved for later use. The framework is executed long enough to meet the criteria of time or mutation adequacy score, concerning the mutants from the selection method.

The successful testcases are then executed against the set of all mutants to gauge their performance on the whole set. This final stage provides the result needed to compare the mutant selection methods and their effectiveness.

![Figure 4 Experiment overview](image)

6.1 Mutation

Researchers Smith and Williams (2009), adapted the MuJava system by Offutt et al. (2004), used to perform mutation on source code to Eclipse calling it MuClipse. Using MuClipse the code selected is mutated using the MuJava method level operators designed by Ma and Offutt
Experiment

(2005b). MuClipse and MuJava have no support for mutation of statements or variable substitution.

For each selection method, mutants are selected from the set of all mutants as shown in Figure 5 Selective mutation, to form a subset as dictated by the method.

Figure 5 Selective mutation

MuClipse is instrumented to perform the mutation on the software under test using the mutation operators on Java method level, and the mutants are saved to the file system, and used in the testing framework. All the mutations are saved to a log file (see Figure 6), containing mutant name, altered line, method signature, class name, and finally original and mutated code.

Figure 6 Mutation log excerpt

In this figure, we can see that the instruction “triOut + 3” is changed to “triOut / 3” “triOut % 3” and “triOut – 3, creating three different mutants.

6.2 Framework

A framework as depicted in Figure 7 Framework layout is constructed to perform the testing and comparison of the original and mutated versions of the software under test, the code of which is presented in appendix.
The framework performs a set of actions on the mutants and original. First, the file system is traversed to find and list the original and all mutants. The name of the class, the directory, location of files, marked if it is the original, is parsed and saved.

Test data for input is generated, and tested for uniqueness against other test data. The details of the generation are presented in chapter 6.5. This data is supplied to the execution of the original to retrieve and store the expected outcome. The test data does not cover some common testing wisdom, such as null-arguments, out of range input, or incorrect data-types. These sanity tests fall outside the scope of this study, since their purpose is to test other properties of programs, such as exception handling.

The experiment was executed on a HP Pavilion P6110SC Desktop PC (Hewlett-Packard Development Company, 2012), its specifications: AMD 2.4 GHz triple core CPU, 4 GB internal memory, running a clean install of Ubuntu 12.4.

Profiling was performed on the target machine using NetBeans Profiler (Oracle Corporation, 2012), the programs on this system places their upper bounds of execution times at; 72 ms for QuickLZ, 5 ms for MD5, and TriTyp at 7 ms. The profiling was done using the upper bounds values that the test generation tool could produce.

The collection of mutants is executed with a 5-second time limit, assuming that the programs are stuck in a loop after five seconds. This upper bound allows more than ample for the programs to fully execute to completion, execution exceeding this limit it can be safely assumed that programs still running are somehow faulty. Furthermore, the 5-second rule is used in other research applications (Ma, Offutt, et al., 2006) Generally speaking it is undecidable whether a program will ever finish it’s execution, as the Halting problem states (Turing, 1936).

The result of the execution from standard output is returned and saved. The output from the process is compared against the original, if the mutant has the same result as the original, nothing happens for this mutant and iteration.

Should any mutant provide a different answer than the original, several things happen. Affected mutants are marked as ‘killed’, and following iterations will not test the killed mutants. A testcase is created using the test data as input data and the result from the original as the expected result. Testcases retain the input values, and expected results.
Experiment

The above step is repeated as long as, a timer limit is reached or the mutation score has reached a threshold, the details of which are described in 6.3. After the loop comes to its end, the mutants and the number of testcases that killed it is written to a text file, so that the results can be analyzed.

The timer is constructed in order to mitigate the risk that the experiment will never finish. This could occur if there are equivalent mutants remaining among the still alive mutants. This timer is reset each time that testdata successfully kills a mutant. Should the timer reach zero before the target score is reached, the score for following methods is set to the score that was achieved. This strategy allows for a fair comparison of the different methods by using scores on each method that match.

6.3 Equivalent mutants

Equivalent mutants are ignored for the purposes of this thesis; they are of course a source of problem in an industrial setting. However, it is not in the scope of this thesis to identify such mutants or provide a means for detection, but rather focus on the different selection methods to ensure fault detection rates and cost. They are simply regarded as mutants that are difficult to discover, and as such could point to a potential problem in the code under review, in a real world setting.

Mutants that are alive after the time limit runs out, or the required mutation score is reached, are considered, in this work, to be equivalent and thus ignored.

Achieving mutation adequacy score takes into account the ratio of equivalent mutants present in the programs under study. An initial version of the experiment showed that on average, 90% of mutants are detected. Previous studies (Schuler et al., 2009) have shown that the number of equivalent mutants, among the still living mutants after an exhaustive search, to be in the range of 40 - 45%, effectively ~10% of the whole set of mutants. Offutt, Pan, et al. (1996) showed that 9% of the total set were equivalents.

Therefore, in order to reach a score of 90%, in actuality the score to reach is 81%, since the number of suspected equivalent is 10%. The math is 1*0.9*0.9 = 0.81 or 81%.

6.4 Representative programs

Ensuring that the results are valid, programs are selected that either come from the open source community or academia. Constructing a program to include all language constructs and all possible combinations of language functionality would not only be infeasible due to the limitation of available resources, but also a risk since it would introduce a potential bias to the study. However, it is impossible generalize the results to every possible program. The results can however be used as an indication of the efficiency and effectiveness of the methods.

Three programs used in the book by Ammann and Offutt (2008) are chosen, because the programs represent conceivable small programs that are available for peer-review, and some like the “TriTyp” program, are commonly used as examples in the field of software testing. Two larger programs are selected from the open source community because they are judged applicable to the testing framework. These are called MD5 and QuickLZ.

The selection of program is intended to make a cross selection of programs with different properties, and consequently different language elements. Chapter 6.4.1 describes the different programs in detail.
6.4.1 Program descriptions

Five programs are mutated using the MuJava system: OddOrPos, TriTyp, NumZero, MD5, and QuickLZ. The three first programs were taken from the book by Ammann and Offutt (2008).

The descriptions are provided as a means to demonstrate that the programs chosen are diverse.

- NumZero checks an array of integers for zero and if a zero is discovered, a counter is incremented, and finally the number of zeros is reported to standard output.

- OddOrPos checks an array of integers to see if any number in it is either positive or odd, and if so increments a counter. Upon completion of the check, the counted numbers of odd or positive numbers are printed to standard output.

- TriTyp takes three integers as input and checks if they make up a valid triangle.
  - The values are first checked to see if any side is less than zero, if so marks the triangle as invalid.
  - Otherwise the values are compared to see if any sides are of equal length and if so increments a counter.
  - Should the counter be zero, the triangle is checked to see if its sides would make it scalene.
  - If the counter is set to four or more, the program returns that the triangle is equilateral.
  - If not the sides are compared to see if they would match the dimensions of an isosceles triangle.
  - Ending the program is a printout of the type of triangle that matches or if the triangle so happens to be invalid, to standard output.

- The MD5 (Rivest, 1992) program by Howell and Harrison (1999) is a Java translation from C of the MD5 hash function from ssh-1.2.22 source. Howell states that on his personal webpage the program is available for use. It performs a non-reversible cryptographic transformation on any character string of arbitrary length, known as hashing. It is commonly applied in the Free and open-source software community to ensure that the correct file was downloaded, by hashing file contents and providing the hash for comparison, but it also serves other uses.

  For example, the string “hello” is transformed to the MD5 hash “f2eb68435a38be7c3e0b106066b67e”. Changing any character in that string creates a different result, e.g. “hello!” results in the string “e7844642b722521a078e8b36db315b8”. This is called the avalanche effect in the MD5 documentation. The algorithm and a pseudo code version is available in full text in RFC 1321 (Rivest, 1992).

- QuickLZ (Reinhold, 2012) is a commercial open source compression and decompression algorithm. The program is available under GNU General Public License, or commercial license, the author was contacted with details of how it would be used and he gave permission for its intended use. The program compresses or decompresses strings by reading it and substituting recurring sequences of characters with a special byte. This type of compression is referred to as dictionary compression.

  Consider compression of the string “fifteen, sixteen, seventeen, fifteen, sixteen, seventeen”. Compression of the substring “teen,” (note the spacing) can be replaced
Experiment

by a byte $b$ that now represents it. Decompression of the compressed string “ifsixbsevenbfifbsixbseventeen” would mean replacing the byte $b$ with “teen,” restoring the original string.

6.4.2 Program adaptation
The programs are adopted to receive commands from their main methods, as this method is never mutated by the mutation tool. Ensuring that the programs are exercised thoroughly, the call hierarchy of the programs is investigated. The programs TriTyp, NumZero, and OddOrPos programs use all the methods associated with a mutant. The changes introduced into the code are available in appendix.

TriTyp has a function for entering the sides of the triangle, this was removed, and the sides were instead supplied via arguments to the programs.

Integration of MD5 into the framework, never calls the methods MD5.update(byte[], int) and MD5.hashFile(String), to reflect this, the respective mutants are removed from the result. QuickLZ also have a method that is never called by the main method, QuickLZ.sizeCompressed(byte[]), these mutants are also removed.

6.5 Testcase generation
The framework generates test data for input to the mutants and original. This data is used to form a testcase, if and only if it is successful in killing at least one mutant. The use of the generated test data is shown in Figure 8 Testcase generation.
The data must be generated in such a way that it does not introduce bias. This can be avoided by carefully designing the ways the data is generated. The generation must take into account the execution paths in the different programs. TriTyp has an elaborate test data generation to account for the branches available in its structure. The other programs have little in the way of branches and as such require less or no instrumentation to exercise the entire program. The full exercise of the programs is desired, since all the mutants should be tested.

Appendix C - Pseudo code for Testdata generation, with figures Figure 31 Testcase generation for TriTyp, Figure 32 Testcase generation for MD5 and Figure 33 Testcase generation for QuickLZ, which provides a walkthrough of the code which generates testdata. The test data is generated using a random generator for the classes, with special considerations on how the tests are generated per class. The testcases are generated isolated from each other, on a per method basis. The testcases are tested for uniqueness among each other, also on a per method basis.

The program OddOrPos uses the random generator in its basic form, a random number of integers 1 through 10 are requested, and selected within a span of all possible $2^{32}$ integers in Java.

NumZero has more control in its test data generation, first for a chance of 1:10 a zero is added to the data, if not a random number is added to the testcase as above.

TriTyp has a more elaborate of the test data generation, to ensure that as much of the possible paths are exercised in the program. First, a random test generator method is selected from the eight available: Isosceles, Equilateral, Scalene, invalidSide, zeroOnASide, lessThanZeroOnASide, randomPlus, and randomWhatever:

- Isosceles path generates three random sides, valid or not within range of 0 to 10000, and finally a random side is set to be the hypotenuse of the two other sides.
- Equilateral simply generates a random positive value, range of 0 to 10000, and set all sides to this value.
- Scalene generates three random sides range of 0 to 10000, selects a random side and with the aid of Pythagoras' theorem, calculates that random side to conform to the theorem.
- InvalidSide generates three sides, and using Pythagoras' theorem, makes sure that the triangle cannot form a valid triangle.
- ZeroOnASide generates threes sides and then sets a random side to have the value of zero.
- LessThanZeroOnASide performs the same as the previous but instead sets a random side to less than zero.
- RandomPlus simply generates three sides within range of 0 to 10000.
- RandomWhatever has no controls at all; it simply generates three random numbers within the integer range of Java.

Testcases for QuickLZ are generated by first selecting the compression level of the algorithm, currently only level 1 or 3 are available, and then randomly creates a set of 1 to 100 words, each between 1 to 20 characters in length. These words are created by a method that for up to the requested word length generates a random character from a to z.

MD5 is tested in the same fashion, a testcase containing a set of random words up to 100, each of these words containing up to 20 characters.
6.6 Experiment result gathering

The testing framework continuously prints the results of the mutant testing to file, presenting the current state. This allows the result to be collected from these files. The files contain:

- Total time of execution
- Mutation score on the current set of mutants
- Total testcases generated from the test data
- Total number of trials to find usable test data
- Results of successful testcases
  - Original output
  - Output of killed mutants
  - Test data
- Killer for a mutant

6.7 Method comparison

The second function of the framework is to compare the results. In this mode, the framework loads all mutants and the original. However, instead of generating test data for a subset of mutants, a test suite is loaded from file, containing the test cases created from the experiment. The test suites are executed on the 'all-mutants' set to compare the number of mutants killed.

After the experiment has performed the task of generating successful testsuites for the subset of mutants given by the selection methods, they are evaluated. The test suites are used on the full set of mutants, 'all-mutants'. The framework (see Figure 9 Method comparison) uses these testcases against all-mutants to compute the mutation adequacy score with regard to this set.

![Figure 9 Method comparison](image)

Each individual mutant in the set all mutants represents a potential fault, which can only be discovered by a testcase that also kills the mutant. Hence, the mutation adequacy score can be used to measure the quality of the set of testcases. The higher the score is, the more effective the generated test suite is with respect of its faultfinding capability.
Experiment

In order to compare the efficiency of the different methods, the test cases are counted in the generated test suites. There is a cost associated by each test case since it must be executed and analyzed every time it is used. It must also be maintained as the software evolves.

The comparison is executed on a basis of individual mutation methods, for each method and program, its testcases are executed, and the mutation score is computed. The efficiency and effectiveness can then be compared using the mutation score and the number of successful testcases.
7 Results

The result from the experiment is presented below.

7.1 Result sanitation

The results from the testing, is collected and a scrutinized for any oddities. The results from the programs NumZero and OddOrPos and some N-selective methods were not satisfactory.

NumZero only yielded one or two testcases after many trials, regardless of selection method, indicating that those programs contained a large set of trivial mutants that could be killed by a vast majority of testcases. These testcases all killed the same number of mutants in the all-mutants set for NumZero; 92%. Because of the limited usable results, NumZero is removed from this study.

OddOrPos gave similar results, most mutants were killed by almost any testcase, but had a few lingering mutants left. Given the small size of the program and the many testcases executed against it (64978 unique testcases), and presenting little in the way of usable results, it too is removed from this study.

A higher level of N-Selection gives a small number of mutants and therefore results that become hard to quantify, since the individual importance of each single mutant is high. Finding a testcase for 90% of the mutants becomes a moot task, since possible equivalent mutants cannot be accounted for in the confines of a small set of mutants. These methods still find some successful testcase to expose trivial mutants. However, trivial mutants are also found using the low-level N-Selection methods, further reducing the impact of the high-level N-Selective methods.

7.2 Mutation

Mutation is performed using the mutation tool MuClipse created by Smith and Williams (2009), the results of the mutation of the programs is presented below. Mutation has the added property of profiling programs, since mutations are applied only if a certain feature in the language is used.

N-selective, Constrained A/R, and Expression mutation is performed for each program. Creating the N-selection mutants takes no consideration for the dead on arrival mutants or equivalent mutants, since no tests have been executed prior to the mutation process.

N-selective mutation was performed up to the 11th level for QuickLZ, and MD5, at this level, two mutants remain for both programs. TriTyp is N-selected to the sixth level, as the seventh level had no mutants. NumZero is N-selected to the fifth level, containing one mutant. OddOrPos is N-selected up to the seventh level, it also containing one mutant.

Performing N-selective mutation to high levels comes with a caveat. When generating testdata until 90% of the mutants are killed, the importance of individual mutants goes up. Having five mutants in a selection to kill, all of the mutants must be killed in order to reach a score of at least 90%. Whereas having a set of ten mutants allows for all but one mutant is killed in order to reach a score of at least 90%. This can be counteracted by reducing the mutation adequacy score to 85%. This would only require that the set of mutants are 7 or more to allow two mutants to remain alive. However, doing so increases the risk that faults will go undetected.
Results

7.2.1 QuickLZ
This program was the largest of all the programs, producing the most mutants, 2896 in total, and the details are shown in Figure 10 QuickLZ Mutation. Six were removed as they are outside the call hierarchy of the main method. 204 mutants were dead on arrival. 483 mutants are never killed by any testcase, and deemed equivalent. Removing these, the sum total of mutants is 2203. The operators are listed in their N-selection order, sorted by the column ‘Created’.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Created</th>
<th>Distribution</th>
<th>Outside</th>
<th>DOA</th>
<th>Sum</th>
<th>Equivalents</th>
<th>Equivalents removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOIS</td>
<td>1138</td>
<td>39,30%</td>
<td>0</td>
<td>162</td>
<td>976</td>
<td>176</td>
<td>800</td>
</tr>
<tr>
<td>AORB</td>
<td>412</td>
<td>14,23%</td>
<td>0</td>
<td>0</td>
<td>412</td>
<td>93</td>
<td>319</td>
</tr>
<tr>
<td>LOI</td>
<td>362</td>
<td>12,50%</td>
<td>0</td>
<td>23</td>
<td>339</td>
<td>34</td>
<td>305</td>
</tr>
<tr>
<td>ROR</td>
<td>335</td>
<td>11,57%</td>
<td>5</td>
<td>0</td>
<td>330</td>
<td>71</td>
<td>259</td>
</tr>
<tr>
<td>LOR</td>
<td>132</td>
<td>4,56%</td>
<td>0</td>
<td>0</td>
<td>132</td>
<td>22</td>
<td>110</td>
</tr>
<tr>
<td>SOR</td>
<td>118</td>
<td>4,07%</td>
<td>0</td>
<td>0</td>
<td>118</td>
<td>45</td>
<td>73</td>
</tr>
<tr>
<td>COI</td>
<td>107</td>
<td>3,69%</td>
<td>1</td>
<td>0</td>
<td>106</td>
<td>15</td>
<td>91</td>
</tr>
<tr>
<td>ASRS</td>
<td>106</td>
<td>3,66%</td>
<td>0</td>
<td>0</td>
<td>106</td>
<td>19</td>
<td>87</td>
</tr>
<tr>
<td>AOIU</td>
<td>89</td>
<td>3,07%</td>
<td>0</td>
<td>1</td>
<td>88</td>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>COR</td>
<td>52</td>
<td>1,80%</td>
<td>0</td>
<td>0</td>
<td>52</td>
<td>4</td>
<td>48</td>
</tr>
<tr>
<td>AORS</td>
<td>25</td>
<td>0,86%</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>4</td>
<td>21</td>
</tr>
<tr>
<td>AODU</td>
<td>1</td>
<td>0,03%</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>AODS</td>
<td>19</td>
<td>0,66%</td>
<td>0</td>
<td>18</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>COD</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOD</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>2896</td>
<td>6</td>
<td>204</td>
<td>2686</td>
<td>483</td>
<td>2203</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10 QuickLZ Mutation
Results

### 7.2.2 MD5

MD5 is the second largest program, producing 2519 mutants. The result of the mutation is presented in Figure 11 MD5 mutation. Of these 15 was out of the call hierarchy from the main method, 16 were dead on arrival, and 82 mutants were never killed and marked as equivalent. Removing these, 2406 mutant remains. The operators are listed in their N-selection order, sorted by the column ‘Created’.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Created</th>
<th>Distribution</th>
<th>Unexercised</th>
<th>DOA</th>
<th>Sum</th>
<th>Equivalents</th>
<th>Equivalents removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOIS</td>
<td>1348</td>
<td>53.51%</td>
<td>4</td>
<td>10</td>
<td>1334</td>
<td>57</td>
<td>1277</td>
</tr>
<tr>
<td>LOI</td>
<td>365</td>
<td>14.49%</td>
<td>2</td>
<td>5</td>
<td>358</td>
<td>1</td>
<td>357</td>
</tr>
<tr>
<td>AORB</td>
<td>332</td>
<td>13.18%</td>
<td>0</td>
<td>0</td>
<td>332</td>
<td>1</td>
<td>331</td>
</tr>
<tr>
<td>AOIU</td>
<td>304</td>
<td>12.07%</td>
<td>1</td>
<td>1</td>
<td>302</td>
<td>1</td>
<td>301</td>
</tr>
<tr>
<td>LOR</td>
<td>48</td>
<td>1.91%</td>
<td>0</td>
<td>0</td>
<td>48</td>
<td>1</td>
<td>47</td>
</tr>
<tr>
<td>ASRS</td>
<td>44</td>
<td>1.75%</td>
<td>0</td>
<td>0</td>
<td>44</td>
<td>2</td>
<td>42</td>
</tr>
<tr>
<td>ROR</td>
<td>31</td>
<td>1.23%</td>
<td>6</td>
<td>0</td>
<td>25</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>SOR</td>
<td>24</td>
<td>0.95%</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>COI</td>
<td>15</td>
<td>0.60%</td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>AORS</td>
<td>4</td>
<td>0.16%</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>COR</td>
<td>2</td>
<td>0.08%</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>AODS</td>
<td>1</td>
<td>0.04%</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>LOD</td>
<td>1</td>
<td>0.04%</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>AODU</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COD</td>
<td>0</td>
<td>0.00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td>2519</td>
<td></td>
<td>15</td>
<td>16</td>
<td>2488</td>
<td>82</td>
<td>2406</td>
</tr>
</tbody>
</table>

*Figure 11 MD5 mutation*
7.2.3 TriTyp

TriTyp is the smallest program, having only 337 mutants. No mutant was dead on arrival, none was out of the call hierarchy, and 23 were suspected to be equivalent. 314 mutants remain, as shown in Figure 12 TriTyp mutation. The operators are listed in their N-selection order, sorted by the column ‘Created’. Distribution is based on the sum of mutants.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Created</th>
<th>Distribution</th>
<th>DOA</th>
<th>Sum</th>
<th>Equivalents</th>
<th>Equivalents removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOIS</td>
<td>134</td>
<td>39,76%</td>
<td>0</td>
<td>134</td>
<td>21</td>
<td>113</td>
</tr>
<tr>
<td>ROR</td>
<td>85</td>
<td>25,22%</td>
<td>0</td>
<td>85</td>
<td>1</td>
<td>84</td>
</tr>
<tr>
<td>LOI</td>
<td>38</td>
<td>11,28%</td>
<td>0</td>
<td>38</td>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>AORB</td>
<td>36</td>
<td>10,68%</td>
<td>0</td>
<td>36</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>COI</td>
<td>24</td>
<td>7,12%</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>COR</td>
<td>14</td>
<td>4,15%</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>AOIU</td>
<td>6</td>
<td>1,78%</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>LOR</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ASRS</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SOR</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AORS</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AODS</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LOD</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AODU</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>COD</td>
<td>0</td>
<td>0,00%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sum</td>
<td>337</td>
<td>0</td>
<td>337</td>
<td>23</td>
<td>314</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12 TriTyp mutation

7.3 Mutant reduction

The various selection methods all reduce the number of mutants to a certain degree ranging from a minimum mutant reduction to below 1% spanning to 64%. The reduction of mutants is of the total set of mutants including the suspected equivalent mutants, DOA mutants, and those outside call hierarchy. This reduction of mutants also has some correlation on the number of testcases created. As the numbers of mutants decreases, the number of testcases also decreases.

The reduction of mutants implies that fewer testcases need to be generated, thus giving an increase in efficiency. Fewer testcases also means that efficiency is higher as there are less tests to investigate and results to evaluate. Furthermore, the time required to generate the test cases needed also decreases, assuming a uniform distribution of fault revealing testcases.
7.3.1 QuickLZ

QuickLZ is the largest of the programs with 2686 mutants, as shown in Figure 13 Mutant reduction QuickLZ. The word selective is removed for layout reasons. It has a reduction in the total set of mutants ranging from a reduction to less than 1%. N-selective shows a clear trend in mutant reduction as the value of n goes up.

**Mutant reduction QuickLZ**

<table>
<thead>
<tr>
<th>Constrai ned</th>
<th>Expression</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutant reduction to</td>
<td>15,599%</td>
<td>33,805%</td>
<td>63,663%</td>
<td>48,325%</td>
<td>35,704%</td>
<td>23,418%</td>
<td>18,503%</td>
<td>14,110%</td>
<td>10,164%</td>
<td>6,217%</td>
<td>2,941%</td>
<td>1,005%</td>
</tr>
<tr>
<td>Selective Mutants</td>
<td>419</td>
<td>908</td>
<td>1710</td>
<td>1298</td>
<td>959</td>
<td>629</td>
<td>497</td>
<td>379</td>
<td>273</td>
<td>167</td>
<td>79</td>
<td>27</td>
</tr>
</tbody>
</table>

*Figure 13 Mutant reduction QuickLZ*
Results

7.3.2 MD5
MD5 has 2488 mutants. It has a reduction in the number of mutants ranging from a reduction by 0.08% with 11-Selective to 54% for 1-Selective; the details of which are presented in Figure 14 Mutant reduction MD5. The word selective is removed for layout reasons. The program has comparatively few relational operations, as is mirrored by the few mutants of the Constrained A/R method.

![Mutant reduction MD5](image)

**Figure 14 Mutant reduction MD5**
7.3.3 TriTyp
TriTyp with 337 mutants, had at best a reduction in the number of mutants to 6% with 5-Selective, the least reduction was with 1-Selective, 60%, as shown in Figure 15 TriTyp Mutant reduction.

![Mutant reduction TriTyp](image)

**Figure 15 TriTyp Mutant reduction**

7.4 Selection method results
The testcases from the experiment was used to gauge its effectiveness on the all-mutants. This mutation adequacy score provides the basis of comparison together with the number of testcases. Since the equivalent mutants have been exposed, they are removed. They are not within the scope of this thesis.

The different methods were executed until reaching the intended mutation adequacy score internally for that method, or that the timer expired. This resulted in that the methods did not necessarily reach the intended score. In the interest of full disclosure the internal scores for each method is presented in Figure 16 Method internal score.
The table reflects the mutation adequacy score for that individual method and the mutants it contains. In this chapter mutation adequacy score is calculated as referred to in Figure 3 Mutation adequacy score equation, however besides equivalent mutants, DOA-mutants and those mutants outside the call hierarchy are also removed from the sum of total mutants.

The goal was to achieve a similar score across all methods, the reason that comparison is possible if the same score is reached on each method. Practically speaking however, this is infeasible. As the testcases are passed up to the next level of N-Selective, the next method may already reach a score higher than the previous on those testcases. To use a similar selection process across all methods, the highest score for all the methods is selected, as it is unknown whether those methods contains more equivalent mutants than other methods.
7.4.1 QuickLZ
Regardless of mutation method the mutation score never falls below 63% for mutation adequacy score, as Figure 17 QuickLZ result shows. QuickLZ have ten mutants that are considered trivial, that is, the mutant is killed by any testcase generated by the experiment. These trivial mutants represent less than a half percent, as such, they do not influence the results in any significant way.

**QuickLZ Result**

<table>
<thead>
<tr>
<th>Constrained Expression</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testcases</td>
<td>26</td>
<td>26</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>19</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Mutation score</td>
<td>81,03%</td>
<td>81,03%</td>
<td>79,44%</td>
<td>79,44%</td>
<td>79,44%</td>
<td>76,99%</td>
<td>72,17%</td>
<td>72,17%</td>
<td>72,17%</td>
<td>72,17%</td>
</tr>
</tbody>
</table>

**Figure 17 QuickLZ result**
7.4.2 MD5

The results are shown in Figure 18 MD5 result. MD5 has a consistently high level of mutation score regardless of method. This would indicate that MD5 have many trivial mutants as opposed to QuickLZ.

**MD5 Result**

<table>
<thead>
<tr>
<th>Constrained Expression</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testcases</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Mutation score</td>
<td>87.70%</td>
<td>87.70%</td>
<td>87.32%</td>
<td>87.32%</td>
<td>87.32%</td>
<td>87.32%</td>
<td>87.32%</td>
<td>87.32%</td>
</tr>
</tbody>
</table>

**Figure 18 MD5 result**

MD5 shows a consistently high mutation score regardless of which method is used. The results suggest that this program has a high degree of testability, as described by Voas and Miller (1995), as “...the likelihood that the code can fail if something in the code is incorrect. ...”. Mutants have the property of mimicking faults; consequently, in this program, they easily fail; or rather die easily. This is an indication of code that has high quality.
7.4.3 TriTyp
TriTyp is the oldest and smallest of the programs under test, the results from the experiment is presented in Figure 19 TriTyp result. The mutation score is around 60 – 77%.

**TriTyp Result**

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Number of Testcases</th>
<th>Mutation Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constrained</td>
<td>16</td>
<td>67.66%</td>
</tr>
<tr>
<td>Expression</td>
<td>23</td>
<td>77.45%</td>
</tr>
<tr>
<td>1-Selective</td>
<td>14</td>
<td>67.66%</td>
</tr>
<tr>
<td>2-Selective</td>
<td>14</td>
<td>67.66%</td>
</tr>
<tr>
<td>3-Selective</td>
<td>13</td>
<td>60.83%</td>
</tr>
<tr>
<td>4-Selective</td>
<td>13</td>
<td>60.83%</td>
</tr>
<tr>
<td>5-Selective</td>
<td>13</td>
<td>60.83%</td>
</tr>
</tbody>
</table>

Figure 19 TriTyp result
8 Method comparison

The different methods all select a set of mutants from the ‘all-mutants’ set. Since they share a base, the methods are sub-sets of this, allowing them to be compared. The methods are compared with regard to efficiency (number of testcases) and effectiveness (mutants revealed).

Effectiveness is the aptitude to find faults. Effectiveness is compared using the mutation score of the different methods, those with higher scores finds more mutants than those with lower score and thus are more efficient.

Efficiency is the number of tests required to find faults. If more testcases are required, the method is regarded as less efficient, than a method that uses less testcase. However, the mutant reduction is a good indicator of what the final cost will be, as cost may vary across iterations of the experiment, but reduction will not.

A method is considered cost-efficient if it reduces the number of testcases while maintaining the ability to find faults i.e. its effectiveness.

8.1 Constrained A/R Efficiency and Effectiveness

This method uses Relational Operator Replacement (ROR) [==, !=, >, >=, <, <=] and the ABSolute Value (ABS) mutations; AODU: Arithmetic Operator Deletion Unary and AOIU: Arithmetic Operator Insertion Unary. An operand from this set is replaced by any other operand from the same set. This results in a varied number of generated mutants over the programs depending on their properties. Constrained ABS/ROR is similar to branch testing, because these operands are present in many if/else statements. Linear programs will have few mutants, and those with many branches have many mutants. Figure 20 Constrained ABS/ROR result shows the data for the experiment on Constrained A/R.

<table>
<thead>
<tr>
<th>Constrained A/R</th>
<th>QuickLZ</th>
<th>MD5</th>
<th>TriTyp</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of testcases</strong></td>
<td>26</td>
<td>2</td>
<td>16</td>
<td>44</td>
</tr>
<tr>
<td><strong>Mutation score</strong></td>
<td>81,026%</td>
<td>87,697%</td>
<td>67,656%</td>
<td>83,360%</td>
</tr>
<tr>
<td><strong>Total mutants</strong></td>
<td>2203</td>
<td>2406</td>
<td>337</td>
<td>4946</td>
</tr>
<tr>
<td><strong>Mutants dead</strong></td>
<td>1785</td>
<td>2110</td>
<td>228</td>
<td>4123</td>
</tr>
<tr>
<td><strong>Selective Mutants</strong></td>
<td>419</td>
<td>327</td>
<td>91</td>
<td>837</td>
</tr>
<tr>
<td><strong>Mutant reduction to</strong></td>
<td>16%</td>
<td>13%</td>
<td>27%</td>
<td>20,301%</td>
</tr>
</tbody>
</table>

Figure 20 Constrained ABS/ROR result

MD5 has the least number of relational operators and has the highest reduction in mutants, because of this; the mutation indicates that MD5 is linear to a higher degree then the other programs. While MD5 achieves a high mutation adequacy score in the Constrained A/R method, this is in all likelihood due to many trivial mutants in the program. The cost of achieving the 94.76% score in MD5 was at a mere two testcases, indicating high efficiency and effectiveness.
QuickLZ and TriTyp have a higher level of relational mutants, and thus have a lower reduction in the number of mutants. They also score lower than MD5. The cost is much higher for these programs and two to three times the cost of MD5.

Constrained A/R has the lowest average of testcases, making it the most efficient of the methods. It also has the second highest average score with 83,306% making it the second most effective.

8.2 Expression Efficiency and Effectiveness

The method uses the operators:

- **ABS**: test zero at execution and terminate if so and absolute value replacement.
- **AOR**: arithmetic operator replacement.
- **LCR**: logical connector replacement
- **ROR**: relational operator replacement.
- **UOI**: unary operator insertion.

These operators are used at various levels in each program and as such, the savings depend on the properties of the program. The details of the Expression mutation method is presented in Figure 21 Expression result.

<table>
<thead>
<tr>
<th>Expression</th>
<th>QuickLZ</th>
<th>MD5</th>
<th>TriTyp</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of testcases</td>
<td>26</td>
<td>2</td>
<td>23</td>
<td>51</td>
</tr>
<tr>
<td>Mutation score</td>
<td>81,026%</td>
<td>87,697%</td>
<td>77,448%</td>
<td>84,027%</td>
</tr>
<tr>
<td>Total mutants</td>
<td>2203</td>
<td>2406</td>
<td>337</td>
<td>4946</td>
</tr>
<tr>
<td>Mutants dead</td>
<td>1785</td>
<td>2110</td>
<td>261</td>
<td>4156</td>
</tr>
<tr>
<td>Selective Mutants</td>
<td>908</td>
<td>665</td>
<td>141</td>
<td>1714</td>
</tr>
<tr>
<td>Mutant reduction to</td>
<td>33.80%</td>
<td>26.73%</td>
<td>41.84%</td>
<td>34.654%</td>
</tr>
</tbody>
</table>

Figure 21 Expression result

MD5 had a reduction to 32.23%. MD5 has the highest score of all the programs, and does so with only a few testcases, indicating high efficiency and effectiveness.

QuickLZ and TriTyp, however, have a lower score at a higher cost, requiring 2 to 3 times as many testcases without reaching the high score that MD5 has.

Expression has the second highest average of testcases, making it the second most efficient after Constrained ABS/ROR, it also has a slightly higher mutation score.

8.3 N-Selective Efficiency and Effectiveness

This method differs radically from the other methods; its selection is based on an algorithm. The number of mutants used vary depends on the n-level used. The saving in percent of mutants is a function of the distribution of mutants across the operators.

Assuming an equal distribution of mutant generation across the operators, one more level of N-Selective mutation in MuJava, will reduce the cost by 1/15 or 6.666...%. However, in the
Method comparison

five programs implicitly profiled by mutant generation, the largest set is AOIS with a 54% distribution of mutants in MD5. The selection of programs in this experiment shows that the mutants are not equally distributed across the different mutation operators. Generally, since programs work under different conditions and the use of the programming language varies influencing the distribution of mutants. Figure 22 shows the results from the experiment with N-Selection, since the highest level of n-selective on TriTyp is 5, the results of N-Selective is trimmed to level 5 of N-selective.

<table>
<thead>
<tr>
<th>N-Selective</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testcases</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuickLZ</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>19</td>
<td>15</td>
<td>20,6</td>
</tr>
<tr>
<td>MD5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3,0</td>
</tr>
<tr>
<td>TriTyp</td>
<td>14</td>
<td>14</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13,4</td>
</tr>
<tr>
<td>Total</td>
<td>40</td>
<td>40</td>
<td>39</td>
<td>35</td>
<td>31</td>
<td>37,0</td>
</tr>
<tr>
<td>Score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuickLZ</td>
<td>79,437%</td>
<td>79,437%</td>
<td>79,437%</td>
<td>76,986%</td>
<td>72,174%</td>
<td>77,494%</td>
</tr>
<tr>
<td>MD5</td>
<td>87,323%</td>
<td>87,323%</td>
<td>87,323%</td>
<td>87,323%</td>
<td>87,323%</td>
<td>87,323%</td>
</tr>
<tr>
<td>TriTyp</td>
<td>67,656%</td>
<td>67,656%</td>
<td>60,831%</td>
<td>60,831%</td>
<td>60,831%</td>
<td>63,561%</td>
</tr>
<tr>
<td>Average</td>
<td>78,139%</td>
<td>78,139%</td>
<td>75,864%</td>
<td>75,047%</td>
<td>73,443%</td>
<td>76,126%</td>
</tr>
<tr>
<td>Reduction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuickLZ</td>
<td>63,663%</td>
<td>48,325%</td>
<td>35,704%</td>
<td>23,418%</td>
<td>18,503%</td>
<td>37,923%</td>
</tr>
<tr>
<td>MD5</td>
<td>46,383%</td>
<td>31,994%</td>
<td>18,650%</td>
<td>6,511%</td>
<td>4,582%</td>
<td>21,624%</td>
</tr>
<tr>
<td>TriTyp</td>
<td>60,237%</td>
<td>35,015%</td>
<td>23,739%</td>
<td>13,056%</td>
<td>5,935%</td>
<td>27,596%</td>
</tr>
<tr>
<td>Average</td>
<td>56,761%</td>
<td>38,444%</td>
<td>26,931%</td>
<td>14,328%</td>
<td>9,673%</td>
<td>29,048%</td>
</tr>
</tbody>
</table>

Figure 22 N-Selective result

MD5 has the lowest number of testcases while achieving a mutation score of 76,126%. TriTyp and QuickLZ had 4.333... to 6.666... times the number of testcases as compared to MD5, and have a lower score.

The effectiveness of N-selective tends to go down as the level of n increases, which selects fewer mutants, however the efficiency of N-selective also goes down as the number of testcases decreases. However, higher values of n will uncover some mutants, but not equal to lower values of n. This effect is commonly seen in other testing efforts, with diminished return per time unit in the ongoing search for bugs.

8.4 Analysis

The different method varies somewhat in score, and number of testcases, as presented in Figure 23 Methods result. The N-Selective methods never reach 80% as opposed to the other methods. Since N-Selective has the same results for levels 1, 2, and 3, furthermore levels 4 and 5 score comparatively that they can be removed from the analysis.
Method comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Constrained A/R</th>
<th>Expression</th>
<th>1-Selective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number testcases</td>
<td>44</td>
<td>51</td>
<td>37</td>
</tr>
<tr>
<td>Mutation score</td>
<td>83,360%</td>
<td>84,027%</td>
<td>76,126%</td>
</tr>
<tr>
<td>Mutant reduction</td>
<td>20,301%</td>
<td>34,654%</td>
<td>56,761%</td>
</tr>
</tbody>
</table>

Figure 23 Methods result

Constrained ABS/ROR achieves a mutation score of 83,360%, and does so with 44 testcases. The operators account for only 20,301% of the mutants but still achieve a high score. However, it has the drawback of not adapting to programs that use little of its operators.

Expression also achieves the highest score with 84,027%, while using on average 51 testcases, more than Constrained ABS/ROR. It also has the drawback of not adapting to programs.

N-Selective shows a great promise through its adaptability, it will conform, to a degree, to the mutants in that program, while achieving a score around 76,126%, with 37 testcases, thus performing the poorest among the methods.

A striking feature of N-Selective is that it will adapt to any program profile, as it reduces the cost of mutants by removing the n largest subsets of mutants. This may cause issues as programs with a low distribution in the operators, may skew the testcases. A program that has a high percent of mutants in the highest levels of n would reduce the number so much that only a very small portion would remain. This may cause problems as no tests are specifically generated for that type of errors in the code.

Consider this scenario; there are three modules in a software suite to test, Alfa, Beta, and Gamma. Alfa is security critical, Beta is important and Gamma is the GUI, which can simply be restarted, should it fail. Alfa would require a high level of N-selection to achieve high fault detection. Beta would require a medium level, and Gamma could use a low level. This takes excellent advantage to the adaptability of N-selective mutation.
9 Conclusion

The purpose of this thesis was to investigate the effectiveness and efficiency of different mutation selection methods, on program-based mutation. In order to achieve this, a literature survey was conducted to investigate for different mutation selection methods, as described in section 5.

Five Java programs were selected for the experiment. Using a mutation tool, MuJava, these programs were mutated in accordance with the different selection methods. These mutants were then used to generate testcases, as a basis for comparison of the different mutation selection methods, the process of which is described in chapter 6.

Chapter 7 presents the results of the mutation and methods.

The test suites generated for each of the selection methods is then executed against all-mutants to measure their effectiveness in finding mutants, while efficiency is measured by the size of the test suites. The resulting data is then used to compare the different methods in chapter 8.

This achieves the goals set out in chapter 3.4.

9.1 Research contribution

This thesis investigates the use of mutation testing using mutation selection methods with special attention to effectiveness and efficiency.

The experiment shows that for achieving the maximum level of efficiency and effectiveness, Constrained ABS/ROR is the best choice. It strikes a good balance of mutation adequacy score for all-mutants whilst, having the lowest cost in number of testcases and the highest reduction in number of mutants.

However, should the tester decide to achieve a higher mutation score Expression is the better choice if more resources are available, as it has a slightly higher score but with significantly higher number of test cases and a lower reduction in number of mutants.

N-Selective also finds some mutants, but at a much lower score than the two other methods. It only recommended to use, when the software tested is non-critical in nature.

9.2 Discussion

9.2.1 Threats to validity
The experiment is executed only one time, this allows for the randomness of the experiment to have a high impact. The results may be skewed in a particular fashion. This would be mitigated by performing the experiment additional times.

The set of programs are few, lowering the confidence of the results. Adding more programs would add to both the impact and trustworthiness of the results.

9.2.2 Ethical implications
This thesis suffered some setback during its course; an initial version used a faulty migration of mutants from C++ to Java, which invalidated the experiment. This was immediately brought to the supervisor’s attention, which prompted a rerun of the experiment.

The programs are used with the implicit consent, and in the instance of QuickLZ also explicit consent, from the authors. The author of the MD5 program was contacted using the included e-mail, but the address no longer works. The programs OddOrPos, NumZero, and TriTyp,
have clear references to them (Ammann and Offutt, 2008) and is used under scientific fair use principles.

This thesis publishes no personal data, except to point out the supervisor, examiner, and student of this paper.

9.2.3 Societal implications
The aim of this thesis is to investigate the efficiency and effectiveness of mutation selection methods. Because of the ubiquitous use of software programs, and the fact that they should be tested to uphold some level of quality, this could have some impact in society. Practitioners can use this thesis as grounds for determining which method to use to preserve precious resources.
10 Related work and reports

10.1 N-selective experiment

Offutt et al. (1993) presented the first experiment with n-selective mutation, using Mothra and a collection of FORTRAN programs. This thesis experiment investigated N-selective mutation, but instead increments the value of \( n \) by one, whereas Offutt et al. only used 2-4- and 6-selective mutation.

This experiment provided a high efficiency with a savings in the number of mutants by up to 60%, and an effectiveness of above 99% for all the tested levels for the programs.

This paper tries to compare the attributes of N-Selective as compared to other forms of mutation selection.

10.2 Breeding High Impact Mutations

Breeding High Impact Mutations by Schwarz et al. (2011), tries to maximize the impact of mutants; coverage impact and impact on return value (Schuler and Zeller, 2010) and invariant impact (Schuler and Zeller, 2009) that mutants have.

These genetic algorithms create mutants with high impact, Schwarz et al. (2011): “... [mutants] not detected by the test suite, and at the same time are well spread all over the code. ...”. The idea is to generate fewer mutants that are hard to detect, thus improving both efficiency and effectiveness.

In their experiment, the algorithm selects 10 – 20% of the total of mutants, achieving a high efficiency. However since they focus on finding mutants that are hard to discover, they don’t investigate effectiveness.

10.3 Mutation efficiency survey

There are in general three solutions (Offutt and Untch, 2001) to reduce the cost of mutation testing; fewer, smarter or faster.

- **Fewer** reduces the number of mutants, and with it the computational cost
  - Selective mutation
  - Mutant sampling
- **Faster** is concerned with running more tests on mutants at the same timeframe.
  - Schema-based Mutation Analysis
- **Smarter**
  - Weak mutation
  - Separate compilation
  - Compiler mutant integration
  - Architecture Distribution
  - Improved algorithms

The report mentions different ways to improve the field of mutation testing, by either faster, smarter and fewer. This thesis has focused on Selective mutation, which falls into the fewer section. Interested readers who wish to expand their knowledge in this field should read the article to get a broad introduction of the various ways to perform mutation testing.
10.4 Constraint solvers

Constraint solvers (DeMillo and Offutt, 1993) can analyze the code to find paths that lead up to the place where the mutants is located. A testcase is then generated (via backtracking) for that mutant and can then be executed to illustrate conditions where the mutant is killed.

This focuses on examining the paths of the program, which may uncover dead code if the constraint solver is unable to reach sections of code. This will reduce the cost of mutation testing, since the task of deducing testcases for branch or mutation coverage can be automated to some measure.

10.5 Reduced test set using Higher Order Mutation Testing

Higher Order Mutation Testing (Jia and Harman, 2009) is a technique of combing a number of First order mutants (FOM), to form a Higher Order Mutant (HOM). A HOM is a mutant that is constructed “to seek out combinations of simple faults that partially mask one another, so that the combination of faults is harder to detect than any of the individual constituent faults.”

Their idea is to replace FOMs, with HOM that subsumes them. Assuming that the HOM is constructed from and replace FOMs, at a ratio of at least 1:2, the efficiency is at least 50%. This would also reduce test effort since the number of mutants are reduced. This reduces the number of mutants while retaining test effectiveness according to the researchers.

Intuitively there is a great benefit in a process that identifies complex subtle faults, while also uncovering simpler faults at the same time. However, this avenue lacks complementary research effort in the testing community.

10.6 Reduced Test effort using Weak Mutation

Offutt and Lee (1994) have shown that weak mutants can be used as a computation efficient mutation testing, however, this method cannot detect complicated defects, introduced by certain mutants, since some masks mutants placed in sequence from each other may cancel another out.

Weak mutation checks the states of the internal values before and after the location of the mutant, and doesn’t necessarily follow the path all the way to its termination. This path may mask certain errors (implicit information loss (Voas and Miller, 1995)), for instance when modulo operations are used.

10.7 Increased speed with Virtualization

The “Mutation 2012” conference presented new ideas for mutation testing, especially mutation using virtualization. Durelli et. al. (2012) using virtualization for mutation testing, the researchers state that “…Experimental results show that the VM-based implementation achieves speedups of as much as 89% in some cases.” (Durelli et al., 2012). This avenue of research may be what propels mutation testing into the commonplace of software testing.

Because of the computational savings, there is no need to reduce the mutants, and would greatly increase both efficiency and effectiveness. However, the results of the testcases should still be toughly analyzed, in order to investigate the correct function of the code under test.
11 Future Work

This chapter will discuss possible future work as observed from the contents of this thesis.

11.1 Industrial environment test

Research is seldom done for the sake of research itself, it has to be applied for use outside of the academical world. Many researchers have stated that the mutation testing science is beginning to reach maturity for industrial use.

As such, it would be interesting to see mutation testing put to practical use. To address this, the results from this thesis could be used to test the efficiency and effectiveness of the methods herein as compared to the test efforts in an industrial project.

11.2 Experiment optimization

The experiment is only executed one time; as such, the results are limited in the sense that the random effects are large. This could be reduced by running several iterations of the experiment to increase the confidence of the results.

Furthermore, the set of programs is relatively small. Adding several programs to the set would increase confidence of the results.

11.3 Target elusive mutants

Investigate the methods effectiveness with focus on their respective disjunctive mutation operators. Each of the methods makes a selection of the available mutation operators. Each operator will provide a differing level of fault finding ranging from; trivial mutants that will be killed by many testcases, to the intermediary mutants that requires more effort, to those that are killed by only a few testcases, e.g. they are hard to find and kill, and the same would be applicable to a real mimicked fault. An example of this is given in Figure 24 Mutant fault detection classification.

![Figure 24 Mutant fault detection classification](image)

However, consider the following example: certain types of faults in code, may under the right conditions, prove very hard to find and in some cases easy. This feature is in part dependent
Future Work

on the structure of the program, as mutants in nested iterations or selections may be unreachable. Voas & Miller (1995) approaches this notion as the concept of 'testability' in a program.

Constraint solvers as described by (DeMillo and Offutt, 1991) approaches this notion of code reachability by parsing the program and tries to device input values to reach a certain section of code, effectively performing coverage of that part of the program. However, this method is not able to find all faults of code.

Weak mutation could be combined with this technique to achieve a high coverage in the code, specifically in parts of the program that is hard to reach with the constraint solver.

11.4 Investigate set of common faults

Programmers tend to make the same errors when they program, this is the basic assumption in mutation testing, as described in the competent programmer hypothesis. For instance new programmers in C would encounter the issue of undeclared functions, see Figure 25, though they are, in their mind, present in their code. This error is due to the order in which the compilers work in C, any and all symbols must be declared before their use. In this instance all that is needed to solve the issue declare the function at the top at the file.

```
int main()
{
    func();
}
void func()
{
    ...
}
```

```
int main()
{
    menu();
}
```

```
void menu();
int main()
{
    menu();
}
void menu()
{
    ...
}
```

**Figure 25 Common error example**

The research describes a set of mutation operators used in the field of mutation testing. How well do the set of mutants available today, map to the actual faults done in the day-to-day work of programmers? Does the set of mutant operators cover the faults that these programmers make? It would be interesting to collect real world faults and compare these to faults that mutation testing would uncover, to see how well the overlap or if there is critical operators missing.
Bibliography


Fifth International Conference on Software Testing, Verification and Validation. IEEE, Montreal, Canada, pp. 682 – 690.


Ma, Y.-S., Offutt, J., 2005a. Description of Class Mutation Mutation Operators for Java.

Ma, Y.-S., Offutt, J., 2005b. Description of Method-level Mutation Operators for Java.


Tassey, G., 2002. The economic impacts of inadequate infrastructure for software testing.


## Appendix A - Mutant Operators

In this appendix, the mutation operators of MuJava are described and how they relate to other constructed mutation operators for other program languages.

### MuJava Mutants

The mutants of MuJava are designed for the Java programming language. MuJava has two distinct types of mutation operators; method level and class level mutants.

### Method level mutants

Method level mutation (Ma and Offutt, 2005b) or traditional mutation changes the syntax on the method level of the program. MuJava uses 16 sets of mutation operators, with 55 primitives. MuJava has no names for the primitives, and the names for them are constructed from the operator name and the change introduced after applying that operator.

Earlier versions of the operators also included an absolute value operator, but was removed in the second version (Offutt, 2005, Offutt, 2010-05-18), however absolute value modification is provided via AOD-unary and AOI-unary on a variable level.

<table>
<thead>
<tr>
<th>Mutation operator</th>
<th>Mutation Primitive</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AORB</strong>: Arithmetic Operator Replacement – Binary</td>
<td>arorb+</td>
<td>a / b =&gt; a + b</td>
</tr>
<tr>
<td></td>
<td>arorb-</td>
<td>a / b =&gt; a - b</td>
</tr>
<tr>
<td></td>
<td>arorb*</td>
<td>a / b =&gt; a * b</td>
</tr>
<tr>
<td></td>
<td>arorb/</td>
<td>a - b =&gt; a / b</td>
</tr>
<tr>
<td></td>
<td>arorb%</td>
<td>a / b =&gt; a % b</td>
</tr>
<tr>
<td><strong>AORU</strong>: Arithmetic Operator Replacement - Unary</td>
<td>aoru+</td>
<td>a = -9; =&gt; a = +9;</td>
</tr>
<tr>
<td></td>
<td>aoru-</td>
<td>a = +9; =&gt; a = -9;</td>
</tr>
<tr>
<td><strong>AORS</strong>: Arithmetic Operator Replacement - Shortcut</td>
<td>aors++</td>
<td>++p =&gt; ++p</td>
</tr>
<tr>
<td></td>
<td>++aors</td>
<td>--p =&gt; ++p</td>
</tr>
<tr>
<td></td>
<td>aors--</td>
<td>p++ =&gt; p--</td>
</tr>
<tr>
<td></td>
<td>--aors</td>
<td>++p =&gt; --p</td>
</tr>
<tr>
<td><strong>AOIU</strong>: Arithmetic Operator Insertion - Unary</td>
<td>aoiu+</td>
<td>count =&gt; +count</td>
</tr>
<tr>
<td></td>
<td>aoiu-</td>
<td>count =&gt; -count</td>
</tr>
<tr>
<td><strong>AOIS</strong>: Arithmetic Operator Insertion - Shortcut</td>
<td>aois++</td>
<td>i =&gt; i++</td>
</tr>
<tr>
<td></td>
<td>++aois</td>
<td>i =&gt; ++i</td>
</tr>
<tr>
<td></td>
<td>aois--</td>
<td>i =&gt; i--</td>
</tr>
<tr>
<td></td>
<td>--aois</td>
<td>i =&gt; --i</td>
</tr>
<tr>
<td><strong>AODU</strong>: Arithmetic Operator Deletion - Unary</td>
<td>aodu+</td>
<td>+p =&gt; p</td>
</tr>
<tr>
<td></td>
<td>aodu-</td>
<td>-p =&gt; p</td>
</tr>
<tr>
<td><strong>AODS</strong>: Arithmetic Operator Deletion - Shortcut</td>
<td>aods++</td>
<td>j++ =&gt; j</td>
</tr>
<tr>
<td></td>
<td>++aods</td>
<td>++j =&gt; j</td>
</tr>
<tr>
<td></td>
<td>aods--</td>
<td>j-- =&gt; j</td>
</tr>
<tr>
<td></td>
<td>--aods</td>
<td>--i =&gt; i</td>
</tr>
<tr>
<td><strong>ROR</strong>: Relational Operator Replacement</td>
<td>ror&gt;</td>
<td>a % 2 == 1 =&gt; a % 2 &gt; 1</td>
</tr>
<tr>
<td></td>
<td>ror==</td>
<td>a % 2 == 1 =&gt; a % 2 == 1</td>
</tr>
<tr>
<td></td>
<td>ror&lt;</td>
<td>a % 2 == 1 =&gt; a % 2 &lt; 1</td>
</tr>
</tbody>
</table>
## Appendix A - Mutant Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ror&lt;=</td>
<td>a % 2 == 1 =&gt; a % 2 &lt;= 1</td>
</tr>
<tr>
<td>ror==</td>
<td>a &gt; 0 =&gt; a == 0</td>
</tr>
<tr>
<td>ror!=</td>
<td>Side2 &lt;= 0 =&gt; Side2 != 0</td>
</tr>
</tbody>
</table>

### COR: Conditional Operator Replacement

- cor&&: 1 | a > 0 => 1 && a > 0
- cor||: triOut == 2 & Side1 + Side3 > Side2 => triOut == 2 || Side1 + Side3 > Side2
- cor&: a & b => a | b
- cor|: a & b => a | b
- cor^: Side1 <= 0 || Side2 <= 0 || Side3 <= 0 => (Side1 <= 0 || Side2 <= 0 || Side2 <= 0) ^ Side3 <= 0

### COI: Conditional Operator Insertion

- coin!: a == 0 => !(a == 0)

### COD: Conditional Operator Deletion

- cod!: !(a == 0) => a == 0

### SOR: Shift Operator Replacement

- sor>>: binary >>>> 7 => binary >> 7
- sor<<: binary >>>> 8 => binary << 8
- sor>>>: binary >>>> 1

### LOR: Logical Operator Replacement

- lor&: a << 1 | b => a < 1 & b
- lor|: a & 0x7f => a | 0x7f
- lor^: a & b => a ^ b

### LOI: Logical Operator Insertion

- loi~: i => ~i
- loi~: length => -length

### LOD: Logical Operator Deletion

- lod~: ~i => i

### ASRS: Assignment Operator Replacement Shortcut

- asrs+ = a = b => a += b
- asrs= = i += 3 => i -= 3
- asrs*= = i += 3 => i *= 3
- asrs/= = i += 3 => i /= 3
- asrs%= = i += 3 => i %= 3
- asrs&= = a | b => a & b
- asrs|= = a & b => a | b
- asrs^= = a | b => a ^= b
- asrs<< = a >>= 1 => a <<= 1
- asrs>>> = a <<= 1 => a >>= 1
- asrs>>>> = a >>= 1 => a >>>= 1

### Figure 26 MuJava Mutation

**Class level mutation**

The definition of the Java Class level mutants can be found in (Offutt et al., 2006, Ma and Offutt, 2005a, Ma et al., 2006). They are not presented here since mutant at this level is not the focus of this thesis. The area of class level mutation represents a completely new path for mutation testing.
Appendix A - Mutant Operators

**Mothra Mutation operators**

These are the Mutation operators for the first mutation program Mothra, written by (King and Offutt, 1991), here a description of the operators are presented with a conversion to the MuJava equivalents. Many mutants are based on data substitution; these groups of operators are not available in MuJava.

<table>
<thead>
<tr>
<th>Mutant name</th>
<th>Description</th>
<th>MuJava Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AAR</strong>: array reference for array reference replacement</td>
<td>Replace an array for another array.</td>
<td></td>
</tr>
<tr>
<td><strong>ABS</strong>: absolute value insertion</td>
<td>Absolute value replacement; positive or negative, test if value equals zero and if so kill the mutant.</td>
<td></td>
</tr>
<tr>
<td><strong>ACR</strong>: array reference for constant replacement</td>
<td>A constant is replaced by one of all the arrays available in the program.</td>
<td></td>
</tr>
<tr>
<td><strong>AOR</strong>: arithmetic operator replacement</td>
<td>Replace an arithmetic operand with *, **, /, %, +, -, RIGHTOP, LEFTOP.</td>
<td>Partial match: AOR-B, and AOR-S not **, RIGHTOP or LEFTOP.</td>
</tr>
<tr>
<td><strong>ASR</strong>: array reference for scalar variable replacement</td>
<td>Replaces this scalar by any applicable array reference.</td>
<td></td>
</tr>
<tr>
<td><strong>CAR</strong>: constant for array reference replacement</td>
<td>An array is replaced by any constant available.</td>
<td></td>
</tr>
<tr>
<td><strong>CNR</strong>: comparable array name replacement</td>
<td>Replaces an array with another array of the same size.</td>
<td></td>
</tr>
<tr>
<td><strong>CRP</strong>: constant replacement</td>
<td>A constant is changed: const++, const--, const-(const/10), const+(const/10).</td>
<td>Partly AOIS: aois++, aois--, ++aois, --aois</td>
</tr>
<tr>
<td><strong>CSR</strong>: constant for scalar variable replacement</td>
<td>A constant is replaced by any scalar available in the program.</td>
<td></td>
</tr>
<tr>
<td><strong>DER</strong>: do statement end replacement</td>
<td>Replaces labels after a DO statement with any other labels available. DO statements are also replaced by ONETRIP that forces an iteration.</td>
<td></td>
</tr>
<tr>
<td><strong>DSA</strong>: data statement alterations</td>
<td>A DATA expression is changed: DATA++, DATA--, DATA-, (DATA/10), DATA+(DATA/10)</td>
<td>Partly by AOIS: aois++, aois--, ++aois, --aois</td>
</tr>
<tr>
<td><strong>GLR</strong>: goto label replacement</td>
<td>Replaces a goto label operand by another label statement in program</td>
<td></td>
</tr>
<tr>
<td><strong>LCR</strong>: logical connector replacement</td>
<td>Replaces a logical operand by, AND, OR, EQV, NEQV, FALSEOP, TRUEOP, LEFTOP,RIGHTOP</td>
<td>First two operands via LOR: lor&amp;, lor</td>
</tr>
<tr>
<td><strong>ROR</strong>: relational operator replacement</td>
<td>Replace operand with, (LT., LE, GT, GE, EQ, NE, FALSEOP, TRUEOP, LEFTOP,RIGHTOP</td>
<td>First six operands via ROR, remainders have no match.</td>
</tr>
<tr>
<td><strong>RSR</strong>: return statement replacement</td>
<td>Statements in functions or subroutines are replaced with return.</td>
<td></td>
</tr>
<tr>
<td><strong>SAN</strong>: statement analysis</td>
<td>Statements at the start of a block are replaced with TRAP(), TRAP() kills the execution.</td>
<td></td>
</tr>
<tr>
<td><strong>SAR</strong>: scalar variable for array reference replacement</td>
<td>Replaces an array by a compatible scalar reference.</td>
<td></td>
</tr>
<tr>
<td><strong>SCR</strong>: scalar for constant replacement</td>
<td>Replaces a constant by a scalar.</td>
<td></td>
</tr>
<tr>
<td><strong>SDL</strong>: statement deletion</td>
<td>Replace any statement with continue.</td>
<td></td>
</tr>
</tbody>
</table>
### Mutant Operators

<table>
<thead>
<tr>
<th>Mutant</th>
<th>Range</th>
<th>MuJava Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGCR: Constant replacement using global constants</td>
<td>Constants</td>
<td></td>
</tr>
<tr>
<td>CLSR: Constant for scalar replacement using local constants</td>
<td>Constants</td>
<td></td>
</tr>
<tr>
<td>CGSR: Constant for scalar replacement using global constants</td>
<td>Constants</td>
<td></td>
</tr>
<tr>
<td>CRCR: Required constant replacement</td>
<td>Constants</td>
<td></td>
</tr>
<tr>
<td>CLCR: Constant replacement using local constants</td>
<td>Constants</td>
<td></td>
</tr>
<tr>
<td>OAAA: arithmetic assignment mutation</td>
<td>arithmetic assignment</td>
<td>ASRS: arithmetic</td>
</tr>
<tr>
<td>OAAN: arithmetic operator mutation</td>
<td>arithmetic</td>
<td>AORB</td>
</tr>
<tr>
<td>OABA: arithmetic assignment by bitwise assignment</td>
<td>Bitwise assignment</td>
<td></td>
</tr>
<tr>
<td>OABN: arithmetic operator by bitwise operator</td>
<td>Bitwise</td>
<td></td>
</tr>
<tr>
<td>OAEA: arithmetic assignment by plain assignment</td>
<td>Plain assignment</td>
<td></td>
</tr>
<tr>
<td>OALN: arithmetic operator by logical operator</td>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td>OARN: arithmetic operator by relational operator</td>
<td>Relational</td>
<td></td>
</tr>
<tr>
<td>OASA: arithmetic assignment by shift assignment</td>
<td>arithmetic assignment</td>
<td></td>
</tr>
<tr>
<td>OASN: Arithmetic operator by shift operator</td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td>OBAA: Bitwise assignment by arithmetic assignment</td>
<td>arithmetic assignment</td>
<td></td>
</tr>
<tr>
<td>OBAN: Bitwise operator by arithmetic assignment</td>
<td>arithmetic</td>
<td></td>
</tr>
</tbody>
</table>

### Figure 27 Fortran Mutants

### C Mutants

The C mutants in Figure 28 C Mutants, used by the selective mutation methods, are designed by Agrawal et al. (1989), they are more extensive than the mutants designed for MuJava. Many C mutation operators replace language operators with others from another group, such as relational operators for arithmetic operators; MuJava does not have such transformations, resulting in a low match of mutant operators.

The table below is taken from the article by Agrawal et al. with some spelling corrections and a mapping to the MuJava mutants. This table serves as the basis for conversion between the different mutation systems for C and Java.
<table>
<thead>
<tr>
<th>Mutant Operators</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OBBA: Bitwise assignment mutation</strong></td>
<td>Bitwise assignment</td>
<td>asrs&amp;=, asrs^=, asrs</td>
</tr>
<tr>
<td><strong>OBBN: Bitwise operator mutation</strong></td>
<td>Bitwise</td>
<td>LOR</td>
</tr>
<tr>
<td><strong>OBEA: Bitwise assignment by plain assignment</strong></td>
<td>Plain assignment</td>
<td></td>
</tr>
<tr>
<td><strong>OBLN: Bitwise operator by logical operator</strong></td>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td><strong>OBNG: Bitwise negation</strong></td>
<td>Bitwise</td>
<td>loi~, lod~</td>
</tr>
<tr>
<td><strong>OBRN: Bitwise operator by relational operator</strong></td>
<td>Relational</td>
<td></td>
</tr>
<tr>
<td><strong>OBSA: Bitwise assignment by shift assignment</strong></td>
<td>Shift assignment</td>
<td></td>
</tr>
<tr>
<td><strong>OBSN: Bitwise operator by shift operator</strong></td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td><strong>OCOR: Cast operator by cast operator</strong></td>
<td>Casts</td>
<td></td>
</tr>
<tr>
<td><strong>OEAA: Plain assignment by arithmetic assignment</strong></td>
<td>Arithmetic assignment</td>
<td></td>
</tr>
<tr>
<td><strong>OEBA: Plain assignment by bitwise assignment</strong></td>
<td>Bitwise assignment</td>
<td></td>
</tr>
<tr>
<td><strong>OESA: Plain assignment by shift assignment</strong></td>
<td>Shift assignment</td>
<td></td>
</tr>
<tr>
<td><strong>OIPM: Indirection operator precedence mutation</strong></td>
<td>Expressions</td>
<td></td>
</tr>
<tr>
<td><strong>OLAN: Logical operator by arithmetic operator</strong></td>
<td>Arithmetic</td>
<td></td>
</tr>
<tr>
<td><strong>OLBN: Logical operator by bitwise operator</strong></td>
<td>Bitwise</td>
<td></td>
</tr>
<tr>
<td><strong>OLLN: Logical operator mutation</strong></td>
<td>Logical</td>
<td>cor&amp;&amp;, cor||</td>
</tr>
<tr>
<td><strong>OLNG: Logical negation</strong></td>
<td>Logical</td>
<td>COI, loi~</td>
</tr>
<tr>
<td><strong>OLRN: Logical operator by relational operator</strong></td>
<td>Relational</td>
<td></td>
</tr>
<tr>
<td><strong>OLSN: Logical operator by shift operator</strong></td>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td><strong>ORAN: Relational operator by arithmetic operator</strong></td>
<td>Arithmetic</td>
<td></td>
</tr>
<tr>
<td><strong>ORBN: Relational operator by bitwise operator</strong></td>
<td>Bitwise</td>
<td></td>
</tr>
<tr>
<td><strong>ORLN: Relational operator by Logical operator</strong></td>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td><strong>ORRN: Relational operator mutation</strong></td>
<td>Relational</td>
<td>ROR</td>
</tr>
<tr>
<td><strong>ORSN: Relational operator by shift operator</strong></td>
<td>Shift</td>
<td></td>
</tr>
<tr>
<td><strong>OSAA: Shift assignment by arithmetic assignment</strong></td>
<td>Arithmetic assignment</td>
<td></td>
</tr>
<tr>
<td><strong>OSAN: Shift operator by arithmetic operator</strong></td>
<td>Arithmetic</td>
<td></td>
</tr>
<tr>
<td><strong>OSBA: Shift assignment by bitwise assignment</strong></td>
<td>Bitwise Assignment</td>
<td></td>
</tr>
<tr>
<td><strong>OSBN: Shift operator by bitwise operator</strong></td>
<td>Bitwise</td>
<td></td>
</tr>
<tr>
<td><strong>OSEA: Shift assignment by plain assignment</strong></td>
<td>Plain assignment</td>
<td></td>
</tr>
<tr>
<td><strong>OSLN: Shift operator by logical operator</strong></td>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td><strong>OSRN: Shift operator by relational operator</strong></td>
<td>Relational</td>
<td></td>
</tr>
<tr>
<td><strong>OSSA: Shift assignment mutation</strong></td>
<td>Shift assignment</td>
<td>asrs&lt;=, asrs&gt;=, asrs&gt;&gt;=</td>
</tr>
<tr>
<td>OSSN: Shift operator mutation</td>
<td>Shift</td>
<td>SOR</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>SBRC: break replacement by continue</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>SBRn: Break out to nth level</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>SCRB: continue replacement by break</td>
<td>Continue</td>
<td></td>
</tr>
<tr>
<td>SDWD: do-while replacement by while</td>
<td>do-while</td>
<td></td>
</tr>
<tr>
<td>SGLR: goto label replacement</td>
<td>Goto</td>
<td></td>
</tr>
<tr>
<td>SMVB: Move brace up and down</td>
<td>Statement</td>
<td></td>
</tr>
<tr>
<td>SRSR: return replacement</td>
<td>Return</td>
<td></td>
</tr>
<tr>
<td>SSDL: Statement deletion</td>
<td>Statement</td>
<td></td>
</tr>
<tr>
<td>SSOM: Sequence Operator Mutation</td>
<td>Statement</td>
<td></td>
</tr>
<tr>
<td>STRI: Trap on if condition</td>
<td>if statement</td>
<td></td>
</tr>
<tr>
<td>STRP: Trap on statement execution</td>
<td>Statement</td>
<td></td>
</tr>
<tr>
<td>SMTC: n-trip continue</td>
<td>Iterative statements</td>
<td></td>
</tr>
<tr>
<td>SSWM: switch statement mutation</td>
<td>switch statement</td>
<td></td>
</tr>
<tr>
<td>SMTT: n-&lt;trip trap</td>
<td>Iterative statement</td>
<td></td>
</tr>
<tr>
<td>SWDD: while replacement by do-while</td>
<td>While</td>
<td></td>
</tr>
<tr>
<td>VASM: Array reference subscript mutation</td>
<td>Array subscript</td>
<td></td>
</tr>
<tr>
<td>VDTR: Absolute value mutation</td>
<td>Scalar reference</td>
<td></td>
</tr>
<tr>
<td>VGAR: Mutate array references using global array references</td>
<td>Array reference</td>
<td></td>
</tr>
<tr>
<td>VGLA: Mutate array references using both global and local array references</td>
<td>Array reference</td>
<td></td>
</tr>
<tr>
<td>VGPR: Mutate pointer references using global pointer references</td>
<td>Pointer reference</td>
<td></td>
</tr>
<tr>
<td>VGSR: Mutate scalar references using global scalar references</td>
<td>Scalar reference</td>
<td></td>
</tr>
<tr>
<td>VGTR: Mutate structure references using global structure references</td>
<td>Structure reference</td>
<td></td>
</tr>
<tr>
<td>VLAR: Mutate array references using local array references</td>
<td>Array reference</td>
<td></td>
</tr>
<tr>
<td>VLPR: Mutate pointer references using local pointer references</td>
<td>Pointer reference</td>
<td></td>
</tr>
<tr>
<td>VLSR: Mutate scalar references using local scalar references</td>
<td>Scalar reference</td>
<td></td>
</tr>
<tr>
<td>VLTR: Mutate structure references using only local structure references</td>
<td>Structure reference</td>
<td></td>
</tr>
<tr>
<td>VSCR: Structure component replacement</td>
<td>Structure component</td>
<td></td>
</tr>
<tr>
<td>VTWD: Twiddle mutations</td>
<td>Scalar expression</td>
<td></td>
</tr>
</tbody>
</table>

Figure 28 C Mutants
Appendix A - Mutant Operators

Bibliography for appendix


OFFUTT, J. 2010-05-18 2010-05-18. RE: Questions regarding the articles about MuJava. Type to HAGMAN, H.

Appendix B - Mutation Methods

Mutation Methods

The different mutation methods were researched from the full text databases available at the University of Skövde. The literature search discovered seven articles for performing selective mutation testing. Please note that for ease of presentation; underlined mutants are transferable to the MuJava framework.

Literature results

The literature search unveiled 247 articles about mutation testing, where six articles describes 18 methods for selective mutation testing (Offutt et al., 1993, 1996; Wong and Mathur, 1995; Mresa and Bottaci, 1999; Barbosa et al., 2001; Vincenzi et al., 2001).

The search procedure involved searching databases available to students at the University of Skövde. The keywords used were "mutation", "selective", "cost", "reduction" and "constrained". The article search was extended by investigating the references used, also citation investigation was also used.

These articles contained sets of reduced mutation operands:

- Offutt, (1993):
  - N-selective
  - Randomly selected X% mutation
  - Constrained ABS/ROR mutation
- Offutt et. al., (1996):
  - Expression/Statement-selective
  - Replacement/Statement-selective
  - Replacement/Expression-selective
  - Expression-selective
- Mresa and Bottaci, (1999):
  - EFF
  - EFA
- Barbosa et al., (2001):
  - SS-27
  - CSS-27
  - S-Offutt-27
  - S-Offutt-5
  - S-Selective-5
- Vincenzi et al., (2001):
  - SUS Sufficient Incremental Unit Testing Strategy
  - SIS Sufficient Incremental Interface Testing Strategy
  - U-IS Unit–Interface Incremental Testing Strategy
  - SU-IS Sufficient Unit–Interface Incremental Testing Strategy

Offutt 1993

N-Selective

Offutt et. al. (Offutt et al., 1993) introduce “N-Selective” mutation where the operators are removed according to the number of mutants created; the operators that produce the most mutations are removed. Offutt et. al. investigated 2-, 4- and 6-Selective mutation, which is
removing the two, four and six most mutant producing operators. The results were that the reduced set was almost as effective as unrestricted mutation for small programs, achieving a saving of up to 60% while retaining a mutation score of 99% or above compared to all-mutants. They used the Mothra mutation system for the Fortran-77 programming language, for in-depth description about Mothra see (King and Offutt, 1991).

N-selective mutation is easily adapted to the MuClipse framework. Each mutant operator is used a number of time, and those two, four, or six operators that are used the most can be removed from the set. Mutation cost reduction can be calculated comparing the reduced set to the total set of mutants. Updated mutation adequacy score is calculated by comparing the hits and misses to the hits and misses of the total set.

**Wong and Mathur 1995**

**Randomly selected X% Mutation**

“Randomly selected x% mutation” by Wong and Mathur (1995), selected randomly from the full set of mutants available to them, where x represents a selection of a certain percentage of all mutants, investigated from ten to 40%, with increments of five. They also used the Mothra mutation program for Fortran-77.

The method is adapted to MuJava by randomly selecting a certain percentage of mutations to from a reduced set of mutants. Cost and mutation adequacy score is calculated as before.

**Constrained ABS/ROR mutation**

“Constrained ABS/ROR mutation”, by Wong and Mathur (1995), simply removed all mutants but he “ROR” (relational operator replacement) and “ABS” (absolute value replacement and force zero at execution). The results from the experiments concluded that with even small sets of mutants, as low as 10% from the set of all mutants, results in a high mutation score with regard to all mutants’, 97% and above, while achieving an 80% saving in the number of generated mutants, thus being very efficient. They also point out that this is hard to generalize since the size of the programs is very small, and that faults that are more complex may be present in larger programs, but remain optimistic to choosing a small set of mutants to save testing resources.

Mutation using absolute values is not supported by the MuJava framework on expression level; it was originally used in but was later removed (Offutt, 2005, 2010), variable level is still supported. Relational expression mutation is available in the framework.

**Offutt et. al. 1996**

**Expression-selective**

Expression-selective or E-selective mutation by Offutt et al. (1996) uses only the expression mutations, to generate mutants. The expression mutants:

- **ABS**: test zero at execution and terminate if so, absolute value replacement.
- **AOR**: arithmetic operator replacement.
- **LCR**: logical connector replacement
- **ROR**: relational operator replacement.
- **UOI**: unary operator insertion.

These mutants carry over well to the MuJava system, with the exception of the ABS operator, this operator is only available at variable level in Java. The article describes this method as a
means to save significant resources, whilst testing features regarding arithmetic and logical constructs in programs.

**Expression and Statement-Selective**

The Expression and Statement selective method by Offutt et al. (1996), combines expression selective mutation with Statement Selective mutation.

The expression mutants:

- **ABS**: test zero at execution and if so terminate, and absolute value replacement.
- **AOR**: arithmetic operator replacement.
- **LCR**: logical connector replacement
- **ROR**: relational operator replacement.
- **UOI**: unary operator insertion.

The statement mutants:

- **DER**: DO statement end replacement
- **DSA**: DATA statement alterations
- **GLR**: GOTO label replacement
- **RSR**: RETURN statement replacement
- **SAN**: statement analysis
- **SDL**: statement deletion

Statement mutations do not exist in the MuJava framework; as such, these mutants cannot be tested in this thesis.

**Replacement-of-operand and statement-selective**

Replacement and statement-selective mutation by Offutt et al. (1996), uses a combination of replacement mutation operators combined with the statement operators. Replacement replaces a variable for another variable. Neither of these mutations is supported in the MuJava system.

Replacement mutation:

- **AAR**: array reference for array reference replacement
- **ACR**: array reference for constant replacement
- **ASR**: array reference for scalar variable replacement
- **CAR**: constant for array reference replacement
- **CNR**: comparable array name replacement
- **CRP**: constant replacement const++, const--, const-, const-(const/10), const+(const/10).
- **CSR**: constant for scalar variable replacement
- **SAR**: scalar variable for array reference replacement
- **SCR**: scalar for constant replacement
- **SRC**: scalar for constant replacement
- **SVR**: source constant replacement

The statement mutants:

- **DER**: DO statement end replacement
- **DSA**: DATA statement alterations
Appendix B - Mutation Methods

- GLR: GOTO label replacement
- RSR: RETURN statement replacement
- SAN: statement analysis
- SDL: statement deletion

The replacement mutants and statement mutants, except for CRP have no equivalent operators in the MuJava framework; therefore, no comparison can be performed with these. CRP can be matched partly by AOIS: aois++, aois--, ++aos, --aos

Replacement-of-operand/expression-selective

Replacement-of-operand/expression-selective, Offutt et al. (1996), is the third combination in this article and combines replacement and expression statements to perform mutation. The expression mutants will work in MuJava but the replacement-of-operand mutations will not since there are no equivalent functions in MuJava.

Replacement-of-operand mutation:
- AAR: array reference for array reference replacement
- ACR: array reference for constant replacement
- ASR: array reference for scalar variable replacement
- CAR: constant for array reference replacement
- CNR: comparable array name replacement
- CRP: constant replacement const++, const--, const-, const-(const/10), const+(const/10).
- CSR: constant for scalar variable replacement
- SAR: scalar variable for array reference replacement
- SCR: scalar for constant replacement
- SRC: scalar for constant replacement
- SVR: source constant replacement

The expression mutants:
- ABS: test zero at execution and terminate if so, and absolute value replacement.
- AOR: arithmetic operator replacement.
- LCR: logical connector replacement
- ROR: relational operator replacement.
- UOI: unary operator insertion.

Mresa and Bottaci, 1999

EFF

The EFF (an abbreviation of efficient) set used in the article by Mresa and Bottaci (1999) was deduced as being the most efficient after performing an initial experiment to compute an efficiency score for each mutation operator. The set is formed from the operators that fulfilled the moreEfficient predicate, and constructed the EFF set for Mothra:

- SAN: Statement analysis
- AOR: arithmetic operator replacement
- SDL: Statement deletion
- ROR: relational operator replacement
Appendix B - Mutation Methods

- **UOI**: unary operator insertion

The mutation operators UOI, ROR and AOR, all match equivalent operators in the MuJava framework, but not operators SAN and SDL. Hence, this method is not applicable.

**EFA**

The EFA (an abbreviation of efficient but also ABSolute value mutation) was constructed from the EFF set since the researchers observed that EFF missed many ABS mutants; "... test sets that were adequate for eff mutants failed to kill a disproportionately high number of abs mutants..."

- **SAN**: Statement analysis
- **AOR**: arithmetic operator replacement
- **SDL**: Statement deletion
- **ROR**: relational operator replacement
- **UOI**: unary operator insertion
- **ABS**: test zero at execution and terminate if so, and absolute value replacement.

The mutation operators UOI, ROR and AOR, all match equivalent operators in the MuJava framework, but not operators SAN and SDL.

**Vinchenzi et. al. (2001)**

Almost all of the operators from this article (Vincenzi et al., 2001) have no equivalents in the MuJava framework, as such they are merely presented here with short summaries as towards their possible transition to the MuJava framework.

These sets were constructed by investigating the performance of mutants on a set of test programs. The mutants that showed the best coverage while reducing test effort were selected to form sets:

**SS-27**

- SWDD while Replacement by do-while
- SMTC Multiple Trip Continue
- SSDL Statement Deletion
- **OLBN** Logical operator replacement by bitwise operator
- OASN Arithmetic operator by shift operator
- ORRN Relational operator mutation
- VTWD Twiddle Mutations; increment or decrement a value by 1
- VDTR Absolute value mutation
- Cccr Constant for constant replacement
- Ccsr Constant for scalar replacement.

The only operands that carry over to MuJava are some of VDTR’s primitives, OLBN operators and ORRN mutation operators.

**CSS-27**

- SSDL Statement Deletion
- **ORRN** Relational operator mutation
Appendix B - Mutation Methods

- VTWD Twiddle Mutations; Increment or decrement a value by 1
- Ccsr Constant for scalar replacement

Only ORRN have equivalent mutants in the MuJava framework with the ROR operator.

**S-Offutt-27**
- Cccr Constant for constant replacement
- Ccsr Constant for scalar replacement
- CRCR Required constant replacement

None of these operands is matched in the MuJava Framework.

**S-Offutt-5**
- Vprr Pointer Reference Replacement
- Varr Array reference replacement
- VTWD Twiddle Mutations; Increment or decrement a value by 1
- VDTR Absolute value mutation
- Vsrr Scalar Variable reference replacement

VDTR is the only operator that can be partially transferred to the MuJava framework.

**S-Wong**
- STRP Trap on statement execution
- OLLN Logical operator mutation
- OLNG Logical Negation
- ORRN Relational Operator Replacement
- VTWD Twiddle Mutations; Increment or decrement a value by 1
- VDTR Absolute value mutation

The OLNG, ORRN, and VDTR mutants can be transferred to a similar mutant in the MuJava framework.

**6-Selective-27**
- Vsrr Scalar Variable reference replacement
- CRCR Required constant replacement
- Ccsr Constant for scalar replacement
- VDTR Absolute value mutation
- Cccr Constant for constant replacement
- SRSR Return statement replacement

VDTR is the only mutant in this method that is transferrable.

**6-Selective-5**
- Cccr Constant for scalar replacement
- Vsrr Scalar Variable reference replacement
- Ccsr Constant for scalar replacement
- CRCR Required constant replacement
- VDTR Absolute value mutation
- ORRN Relational Operator Replacement
Appendix B - Mutation Methods

ORRN and VDTR can be transferred to the MuJava framework.

**SUS**

- SMTC Multiple trip continue
- SSDL Statement Deletion
- OEBA Plain assignment by bitwise assignment
- **ORRN** Relational Operator Replacement
- VTWD Twiddle Mutations; Increment or decrement a value by 1
- **VDTR** Absolute value mutation
- OBSN Bitwise comparison by shift operator
- OASN Arithmetic operator by shift operator
- ORRN Logical operator by relational operator
- SWDD while replacement by do while
- Varr Array reference replacement
- **Oido** Increment / Decrement (++c, c++, --c, c--)
- SSWM Switch statement mutation
- OEAA Plain assignment by arithmetic assignment
- ORBN Relational operator by bitwise operator
- SRsr Return statement replacement
- STRI Trap on if condition
- Varr Scalar Variable reference replacement
- OABN Arithmetic operator by bitwise replacement
- Cccr Const constant for constant replacement

ORRN, VDTR and Oido have matching operators in the MuJava framework.
Appendix B - Mutation Methods

**SCUS:**
- Cccr Constant for constant replacement
- OABN Arithmetic operator by bitwise replacement
- OAEA Arithmetic operator by plain assignment
- OASN Arithmetic operator by shift operator
- OBSN Bitwise comparison by shift operator
- OESN Plain assignment by arithmetic assignment
- OESA Plain assignment by bitwise assignment
- OESA plain assignment by shift assignment
- Oido Increment / Decrement (++c, c++, --c, c--)
- OLLN Logical operator by logical operator
- OLRN Logical operator by relational operator
- OLSN Logical operator by shift operator
- ORAN Relational operator by arithmetic operator
- ORBN Relational operator by bitwise operator
- ORLN Relational operator by logical operator
- ORRN Relation Operator replacement
- SBRC break replacement by continue
- SBRn Break out to nth enclosing level
- SDWD do while replacement by while
- SMTCA Multiple trip continue
- SMVB Move brace up or down
- SRSR Return statement replacement
- SSDL Statement deletion
- SSWM Switch statement mutation
- STRI Trap on if condition
- SWDD while replacement by do while
- Varr Array reference replacement
- VDTR Absolute value mutation
- VSCR Structure component replacement
- Vsrr Scalar Variable reference replacement
- VTWD Twiddle Mutations; Increment or decrement a value by 1

Oido, ORRN, and VDTR operator have matching operators in the MuJava framework.
Mutation operator conversions

Below is two matrices showing how the different mutant methods relate to the MuJava framework. An x notes that the operator has a match by a related operator in the framework.

<table>
<thead>
<tr>
<th>Operator / Method</th>
<th>Constrained ABS/ROR</th>
<th>Expression-selective</th>
<th>Replacement-selective</th>
<th>Statement-selective</th>
<th>EFF</th>
<th>EFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>aods</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aodu</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>aois</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aoiu</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>aorb</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aoru</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>asrs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cod</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cor</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lod</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loi</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ror</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matched Operators</td>
<td>3</td>
<td>5</td>
<td>none</td>
<td>none</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Total Operators</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 29 Mutation operator conversion Fortran
## Appendix B - Mutation Methods

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>aods</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>aodu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aolis</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aolu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>aorb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>aoru</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>asrs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cod</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>col</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lod</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loi</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ror</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sor</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Matched Operators**

- 1
- none
- 1
- 3
- none
- 1
- 4
- 4

**Total Operators**

- 9
- 3
- 3
- 5
- 6
- 6
- 6
- 20
- 31

*Figure 30 Mutation operator conversion C*
Bibliography for appendix


Appendix C - Pseudo code for Testdata generation

This appendix presents the pseudo code, for how the testdata is generated. The algorithms only take into account valid input data, this goes against traditional software testing practices. The test data does not cover some common testing wisdom, such as null-arguments, out of range input, or incorrect data-types. These sanity tests fall outside the scope of this study, since their purpose is to test other properties of programs, such as exception handling.

```
procedure generateForTriTyp
    static int Isosceles = 0;
    static int Equilateral = 1;
    static int Scalene = 2;
    static int zeroOnASide = 3;
    static int lessThanZeroOnAside = 4;
    static int randomPlus = 5;
    static int randomWhatever = 6;
    static int invalidSide = 7;

    path = random.int.range(0 to 7);
    randomSide = random.int.range(0 to 2);
    switch (path)
        case Isosceles:
            side1 = random.int.range(1 to 10000);
            side2 = side1;
            side3 = side1;
            array.add(side1);
            array.add(side2);
            array.add(side3);
            // Set one side to the hypotenuse of the two other sides
            if (randomSide == 0)
                array.set(0, hypotenuse(side2, side3));
            if (randomSide == 1)
                array.set(1, hypotenuse(side1, side3));
            if (randomSide == 2)
                array.set(2, hypotenuse(side1, side2));
            break;
        case Equilateral:
            side1 = random.int.range(1 to 10000);
            array.add(side1);
            array.add(side1);
            array.add(side1);
            break;
        case Scalene:
            side1 = random.int.range(1 to 10000);
            side2 = random.int.range(1 to 10000);
            side3 = random.int.range(1 to 10000);
            array.add(side1);
            array.add(side2);
            array.add(side3);
            if (randomSide == 0)
                array.set(0, squareRoot(squared(side2) + squared(side3)));
            if (randomSide == 1)
                array.set(1, squareRoot(squared(side1) + squared(side3)));
            if (randomSide == 2)
                array.set(2, squareRoot(squared(side2) + squared(side1)));
            break;
        case invalidSide:
```
Appendix C - Pseudo code for Testdata generation

```java
side1 = random.int.range(1 to 10000);
side2 = random.int.range(1 to 10000);
side3 = random.int.range(1 to 10000);
array.add(side1);
array.add(side2);
while (side3 == squareRoot(squared(side1) + squared(side1, 2))) {
    side3 = random.int.range(1 to 10000);
} array.add(side3);

case zeroOnASide:
    side1 = random.int.range(1 to 10000);
side2 = random.int.range(1 to 10000);
side3 = random.int.range(1 to 10000);
if (randomSide == 0) {
    array.add(0);
    array.add(side2);
    array.add(side3);
}
if (randomSide == 1) {
    array.add(side1);
    array.add(0);
    array.add(side3);
}
if (randomSide == 2) {
    array.add(side1);
    array.add(side2);
    array.add(0);
} break;

case lessThanZeroOnAside:
    side1 = random.int.range(1 to 10000);
side2 = random.int.range(1 to 10000);
side3 = random.int.range(1 to 10000);
if (randomSide == 0) {
    array.add(0 - side1);
    array.add(side2);
    array.add(side3);
}
if (randomSide == 1) {
    array.add(side1);
    array.add(0 - side2);
    array.add(side3);
}
if (randomSide == 2) {
    array.add(side1);
    array.add(side2);
    array.add(0 - side3);
} break;

case randomPlus:
    side1 = random.int.range(1 to 10000);
side2 = random.int.range(1 to 10000);
side3 = random.int.range(1 to 10000);
array.add(side1);
array.add(side2);
array.add(side3);
break;
```
Appendix C - Pseudo code for Testdata generation

```java
    case randomWhatever:
        side1 = random.int.range(-2,147,483,648 to 2,147,483,647);
        side2 = random.int.range(-2,147,483,648 to 2,147,483,647);
        side3 = random.int.range(-2,147,483,648 to 2,147,483,647);
        array.add(side1);
        array.add(side2);
        array.add(side3);
        break;
    end switch
end Procedure;
```

**Figure 31 Testcase generation for Trityp**

```java
    Procedure generateForMD5
        words = random.int.range(1 to 100);
        String s = "";
        for (int i = 0; i < words; i++) {
            s = generateRandomWord(random.int.range(1 to 20));
            array.add(s);
        }
    end Procedure;

    Procedure generateRandomWord(int wordLength) {
        String word;
        for(int i = 0; i < wordLength; i++) {
            char temp = random.char.range('a' to 'z');
            word.append(temp);
        }
        return word;
    }
end Procedure;
```

**Figure 32 Testcase generation for MD5**
Appendix C - Pseudo code for Testdata generation

```java
Procedure generateForQuickLZ
    // Selection of compression level
    if (random.int()%2==0)
        array.add("1");
    else{
        array.add("3");
    }

    words = random.int.range(1 to 100);
    String s = "";

    for (int i = 0; i < words; i++) {
        s = generateRandomWord(random.int.range(1 to 20));
        array.add(s);
    }
end Procedure;

procedure generateRandomWord(int wordLength) {
    String word;
    }
    for(int i = 0; i < wordLength; i++) {
        char temp = random.char.range('a' to 'z');
        word.append(temp);
    }
    return word;
}
end Procedure;
```

Figure 33 Testcase generation for QuickLZ
Appendix D - Program adaptations

The programs were changed in their main methods to allow for the framework to interact with them using the standard input and output of the system. Below are the adaptations of the three programs used.

```
public static void main(String args[]) {
    // This main() method was created to easily test
    // this class. It hashes whatever's on System.in.

    MD5 md = new MD5();
    String hashThis = args[0];
    for (int i = 1; i < args.length; i++) {
        hashThis += " " + args[i];
    }
    System.out.println(md.hashPassword(hashThis));
}
```

**Figure 34 Changed main method in MD5**

```
public static void main(String args[]) {
    // This main() method was created to easily test
    // this class. It hashes whatever's on System.in.

    byte buf[] = new byte[397];
    // arbitrary buffer length designed to irritate update()
    int rc;
    MD5 md = new MD5();
    byte out[] = new byte[16];
    int i;
    int len = 0;

    try {
        while ((rc = System.in.read(buf, 0, 397)) > 0) {
            md.update(buf, rc);
            len += rc;
        }
    } catch (IOException ex) {
        ex.printStackTrace();
        return;
    }
    md.md5final(out);
    System.out.println("file length: "+len);
    System.out.println("hash: "+dumpBytes(out));
}
```

**Figure 35 Original main method in MD5**
Appendix D - Program adaptations

```java
public static void main(String[] argv) { // Driver program for trityp
    int A, B, C;
    int T;
    /* System.out.println (instructions); System.out.println
     * ("Enter side 1: "); A = getN(); System.out.println
     * ("Enter side 2: "); B = getN(); System.out.println
     * ("Enter side 3: "); C = getN();
    */

    A = Integer.parseInt(argv[0]);
    B = Integer.parseInt(argv[1]);
    C = Integer.parseInt(argv[2]);

    T = Triang(A, B, C);

    System.out.println("Result is: " + triTypes[T]);
}
```

**Figure 36 TriTyp changed main Method**

```java
public static void main (String[] argv) { // Driver program for trityp
    int A, B, C;
    int T;

    System.out.println (instructions);
    System.out.println ("Enter side 1: ");
    A = getN();
    System.out.println ("Enter side 2: ");
    B = getN();
    System.out.println ("Enter side 3: ");
    C = getN();
    T = Triang (A, B, C);

    System.out.println ("Result is: " + triTypes[T]);
}
```

**Figure 37 TriTyp original main method**
Appendix D - Program adaptations

```java
// Read (or choose) an integer
private static int getN ()
{
    int inputInt = 1;
    BufferedReader in = new BufferedReader (new InputStreamReader (System.in));
    String inStr;
    try
    {
        inStr    = in.readLine ();
        inputInt = Integer.parseInt(inStr);
    }
    catch (IOException e)
    {
        System.out.println ("Could not read input, choosing 1.");
    }
    catch (NumberFormatException e)
    {
        System.out.println ("Entry must be a number, choosing 1.");
    }
    return (inputInt);
}  // end getN
```

Figure 38 TriTyp removed method
public static void main(String[] args) {
    int level = Integer.parseInt(args[0]);
    String inString = "";
    int i = 1;
    while (i < args.length) {
        inString += " " + args[i];
        i++;
    }
    //System.out.println(" inString>"+inString+" ");
    byte[] inBytes = null;
    byte[] compressed = null;
    try {
        inBytes = inString.getBytes("UTF8");
    } catch (UnsupportedEncodingException e1) {
        System.err.println("UTF8 not supported on input");
        e1.printStackTrace();
    }
    compressed = compress(inBytes, level);
    //I need a printout of this 'middle' value to correct for cyclic faults, in compression/decompression
    //Can't place in separate method since it too would be mutated
    // http://stackoverflow.com/questions/332079/in-java-how-do-i-
    //convert-a-byte-array-to-a-string-of-hex-digits-while-keeping-l
    char[] hexArray = {
        '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
    byte[] bytes = compressed;
    char[] hexChars = new char[bytes.length * 2];
    int v;
    for (int j = 0; j < bytes.length; j++) {
        v = bytes[j] & 0xFF;
        hexChars[j*2] = hexArray[v/16];
        hexChars[j*2 + 1] = hexArray[v%16];
    }
    System.out.println(" compressed hex> "+new String(hexChars));
    byte[] outBytes = decompress(compressed);
    String outString = "";
    try {
        outString = new String(outBytes, "UTF8");
    } catch (UnsupportedEncodingException e) {
        System.err.println("UTF8 not supported on output");
        e.printStackTrace();
    }
    System.out.println(" outString>"+outString);
    }
}