Effects of Mutation Testing on Safety Critical Software

Rebecca Johnsson
Nathalie Svensson

Supervisor: Christoffer Nylén
Saab Aeronautics, Linköping
Adrian Horga
IDA, Linköping University

Examinator: Ahmed Rezine
IDA, Linköping University
Abstract

For avionic systems, the safety requirements are stricter than for non-safety critical systems due to the severe consequences a failure could cause. Depending on the consequences of a failure, the software needs to fulfill different testing criteria. More critical software needs more extensive testing. The question is whether the extra testing activities performed for software of higher criticality level results in discovery of more faults. Mutation testing has been used in this thesis as a method to evaluate the quality of test suites of avionic applications from different safety critical levels. The results showed that the extra activities performed at the higher levels do not necessarily result in finding more faults.
Acknowledgement

First of all we would like to thank our supervisor at Saab Aeronautics, Christoffer Nylén, for his support and expertise. We also give a thanks to Anders Isaksson at Saab Aeronautics for giving us the opportunity to perform this thesis in the first place.

We would also like to thank our supervisor and examiner at Linköping University, Adrian Horga and Ahmed Rezine, for giving directions and constructive criticism on our work.

Finally, we would like to thank each other for excellent teamwork.

Linköping, June 2017
Rebecca Johnsson and Nathalie Svensson
# Table of Contents

1. **Introduction** .......................................................... 7
   1.1 Background ............................................................ 7
   1.2 Motivation .............................................................. 7
   1.3 Thesis purpose ......................................................... 8
   1.4 Research question .................................................... 8

2. **Theory** .................................................................. 9
   2.1 Software testing ....................................................... 9
   2.2 Code coverage .......................................................... 9
      2.2.1 Statement Coverage ............................................. 10
      2.2.2 Decision Coverage ............................................. 10
      2.2.3 Condition Coverage ............................................ 11
      2.2.4 Modified Condition Decision Coverage .................... 12
   2.3 Mutation testing ...................................................... 12
      2.3.1 Dead or live ....................................................... 13
      2.3.2 Mutation Score .................................................. 14
      2.3.3 Creating mutants ............................................... 14
      2.3.4 Mutant operator ................................................ 14
      2.3.4 Reducing the Computational Expense ....................... 15
   2.4 Static code analysis .................................................. 16
   2.5 Safety critical systems .............................................. 17

3. **Method** ................................................................ 18
   3.1 Mutation Tool .......................................................... 18
      3.1.1 Mutant Operator Class Implementation ....................... 19
   3.2 Data Collection ........................................................ 20
   3.3 Evaluation .............................................................. 20

4. **Results** .............................................................. 21
   4.1 Overview ............................................................... 21
   4.2 DAL A ................................................................. 21
   4.3 DAL C ................................................................. 22
   4.4 DAL E ................................................................. 22

5. **Discussion** .......................................................... 23
   5.1 Applications ............................................................ 23
   5.2 Mutant operators ..................................................... 23
   5.3 Results ................................................................. 23

6. **Conclusion** ........................................................ 25

7. **Future Work** ...................................................... 26
   7.1 Extending the Mutation Tool ..................................... 26
   7.2 More Data ............................................................ 26
   7.3 Equivalent mutants ................................................ 26

References ................................................................. 27
1. Introduction

Software testing is and has always been a big part of software development. However, it is a costly and time consuming part of the development process. While testing by itself does not add anything to the software being developed, finding bugs and faults is important feedback for the developer to act upon in order to improve the functionality and usability of the software under test. Faults should be found early in the process, since the later they are found the more costly they become.

For safety critical software, extra effort has to be made in order to ensure that the program works as expected. For this it is important to have adequate test suites. A commonly used metric to evaluate the performance of a test suite is code coverage [1]. Code coverage measures to what extent the software under test has been exercised by the test suite. However, this does not necessarily result in high quality test suites.

One way of assuring the quality of the test suites is to use mutation testing. Mutation testing is a testing technique that injects small faults into the software source code to find out whether the test suite are able to detect them. In that way, the quality of the test suite can be evaluated.

1.1 Background

This thesis has been developed at Saab in Linköping. Saab is a global security and defense company that offers products, services and solutions for military and civil safety. The thesis is made at the business area Aeronautics, which focuses on development of military and civil aviation technology.

For avionic systems, the safety requirements are stricter than for non-safety critical systems due to the severe consequences a failure could cause. Because of this, there is a standard to use as a guideline for development called DO-178C [2]. This standard is published by the Radio Technical Commission for Aeronautics (RTCA), which is a private, non-profit association that provides a foundation for technical advance in aviation.

The standard defines different Software Levels, known as Design Assurance Levels (DAL), that determine the effects of a failure. There are five levels that range from level A in which a failure is catastrophic to level E in which a failure has no effect on the safety. Each level defines different testing activities that must be performed to make sure the software works as expected. Saab uses DO-178C for development of safety critical software in avionic systems.

1.2 Motivation

Mutation testing has been around since the early seventies, but has rarely been extensively used since it is very computationally expensive to perform. Since the seventies, research has been made on how to reduce the expenses of mutation testing [3] [4] [5] [6]. This, in combination with the development of faster and more powerful computers has now made it possible to use mutation testing as a way of asserting quality of test suites. Moreover, when developing safety critical software it is important to use test sets of high quality. This makes mutation testing an interesting topic to incorporate in the development process of safety critical software.
1.3 Thesis purpose
The purpose of this thesis is to perform and evaluate mutation testing on applications that have been classified into different Design Assurance Levels as defined by DO-178C.

1.4 Research question
Can mutation testing say something about the quality of test suites corresponding to applications from different safety critical levels?
2. Theory

This section describes the theory behind software testing in general and mutation testing in particular. Code coverage as a metric used to evaluate test suites is described, as well as how static code analysis can be used together with mutation testing. Finally, a description of what a safety critical system is and how it should be developed according to DO-178C are provided.

2.1 Software testing

The idea of software testing has been around since the start of software development. It is believed that approximately 50% of the development cost consists of testing [7]. Today, more and more software driven devices are used in society and our everyday life. Banking systems, medical equipment, avionic systems and so on need to work as expected. Bugs and errors in the software can lead to different sets of consequences. They may only lead to frustration of the user, but in worst case scenarios they might cause great financial setbacks or even endangering users’ health and lives. Hence, the subject of software testing is of great importance.

Software testing is the process of finding and eliminating faults and bugs in a program [8]. A fault is a static defect in the software that usually results in errors. An error is an incorrect internal state of the software which in turn leads to software failure. A failure occurs when the software behaves incorrectly with respect to the expected behavior.

Ideally when testing software, every possible combination of input to the program should be covered. This is however, in most cases, impractical and often impossible. Most software programs, even small ones, allow many such input combinations, and testing them all would become too time-consuming [7].

2.2 Code coverage

Since it is too time-consuming to test all different input combinations to a program, other techniques are used to estimate whether the software is working as expected or not. One metric that is used is called code coverage [1]. Code coverage is defined as the percentage of code that is being executed during tests. The idea behind code coverage is that there is no way to know if some part of the software is correct or not if it has not been executed. A high code coverage percentage is therefore desired.

There are several different types of code coverage. Some are

- **statement coverage**
- **functional coverage**
- **decision coverage**
- **condition coverage**
- **decision condition coverage**
- **modified decision condition coverage**

A condition is an expression that evaluates to a boolean value. It can contain boolean constants, boolean values, function calls with boolean return values or comparison operators (<, >, <=, >= etc.). An example of a condition is the statement \((a > b)\). A decision is made up
of conditions connected with logical connectors (&& or ||), for example \((condition1 \&\& condition2)\).

### 2.2.1 Statement Coverage
Statement coverage is a commonly used code coverage metric. To fulfil statement coverage each executable statement in the source code must be executed at least once.

### 2.2.2 Decision Coverage
Decision coverage, also known as *branch coverage*, is a code coverage metric used to ensure that all reachable statements in the code are executed at least once. To reach decision coverage, every decision in the source code must be evaluated to both true and false at least once, see Table 2-1.

**Table 2-1.** Test cases 1 and 6 are the only ones needed for the decision to evaluate to both true and false.

|   | condition1 | condition2 | condition3 | condition1 && (condition2 || condition3) |
|---|------------|------------|------------|------------------------------------------|
| 1 | false      | false      | false      | **false**                                |
| 2 | false      | false      | true       | false                                    |
| 3 | false      | true       | false      | false                                    |
| 4 | false      | true       | true       | false                                    |
| 5 | true       | false      | false      | false                                    |
| 6 | true       | false      | true       | **true**                                 |
| 7 | true       | true       | false      | true                                     |
| 8 | true       | true       | true       | true                                     |
2.2.3 Condition Coverage

To reach condition coverage every condition in each decision must be evaluated to both true and false at least once, see Table 2-2.

**Table 2-2.** Test cases 1 and 8 are the only ones needed to evaluate each individual condition to both true and false.

|      | condition1 | condition2 | condition3 | condition1 && (condition2 || condition3) |
|------|------------|------------|------------|------------------------------------------|
| 1    | **false**  | **false**  | **false**  | false                                    |
| 2    | false      | false      | true       | false                                    |
| 3    | false      | true       | false      | false                                    |
| 4    | false      | true       | true       | false                                    |
| 5    | true       | false      | false      | false                                    |
| 6    | true       | false      | true       | true                                     |
| 7    | true       | true       | false      | true                                     |
| 8    | **true**   | **true**   | **true**   | true                                     |
2.2.4 Modified Condition Decision Coverage

The modified condition decision coverage criterion (MCDC) is a common test requirement to fulfil when testing safety critical software. To fulfil MCDC, test cases must be created so that each condition in every decision in the source code evaluates to both true and false, and that each decision also evaluates to both true and false [9]. Furthermore, each condition in every decision must independently and correctly affect the outcome of the decision. This requires more test cases than for decision coverage or condition coverage, see Table 2-3. A minimum of n+1 test cases are needed, where n is the number of conditions.

Table 2-3. Test cases 3, 5, 6 and 7 are needed to reach MCDC.

| condition1 | condition2 | condition3 | condition1 && (condition2 || condition3) |
|------------|------------|------------|------------------------------------------|
| 1          | false      | false      | false                                    |
| 2          | false      | true       | false                                    |
| 3          | false      | true       | false                                    |
| 4          | false      | true       | false                                    |
| 5          | true       | false      | false                                    |
| 6          | true       | false      | true                                     |
| 7          | true       | true       | true                                     |
| 8          | true       | true       | true                                     |

The modified condition decision coverage criterion was developed to help testers cover and test complex boolean expressions in safety-critical software applications [9].

2.3 Mutation testing

One problem with code coverage is that it does not tell the tester anything about the quality of the tests. Tests could execute every statement in a program, and get 100% code coverage, without detecting faults, see example in Figure 2-1.

```java
int c;
if(a == b){
    c = a;
    return c;
}
```

Figure 2-1. Example code getting 100% statement coverage without detecting the fault.

Testing the “equal” function with input a = 1 and b = 1 gives a correct return value of 1 and all statements in the code would be covered resulting in 100% statement coverage. This, without detecting that variable c is uninitialized resulting in unknown behavior for input values that are not equal.
Another way to evaluate and improve existing test suites is to use mutation testing. Mutation testing is a software testing technique that can be traced back to the early seventies. The idea with mutation testing is to create mutants of the source code by making small syntactic changes in it. These changes are supposed to mimic common errors a programmer could make. Test suites that are designed to detect faults in the source code can then be run on the mutated code in order to see if the faults are found [10]. One example of such a change is to change an AND operator to an OR operator [11] as depicted in Figure 2-2.

| Original code | if (a && b) { //do something... } |
| Mutated code  | if (a || b) { //do something... } |

**Figure 2-2.** Example of a mutant.

Usually, programmers only make small errors in the code they write, according to the **competent programmer assumption** [12]. By making one single syntactic change in the source code like in the previous example, a first order mutant (FOM) is created. Further research on mutation testing has proven contradicting results to the competent programmer assumption though. Purushothaman and Perry [13] conclude that about 90 % of all realistic faults are more complex than those that can be created by first order mutants.

*The coupling effect* [14] is a principle that states that if a set of test data can distinguish all programs with simple errors, it will also be able to distinguish all programs with complex errors. This is however an empirical principle and can only be proven empirically.

Higher order mutation testing is one approach to simulate more complex and realistic faults. A higher order mutant (HOM) is a mutant that is generated by applying several changes to the same code. The concept of higher order mutation testing along with the concept of subsuming HOMs was created by Jia and Harman [15]. A subsuming HOM is constructed by combining a set of FOMs and is therefore harder to kill than to individually kill the FOMs from which the HOM was created. The theory is that if a set of test cases manages to kill the HOM, the HOM can effectively replace all the FOMs.

2.3.1 Dead or live

After executing the available test suite on a mutant, the mutant will be considered one of two things: *dead or live* [14]. If the test suite detects the fault in the mutated code, i.e. fails, then the mutant is *killed* and considered dead. If the mutant is not killed however, i.e. the test case passed, it is considered live and further investigation is required to discover why the mutant survived. If adding more test cases results in killing the mutant then the problem existed within the original test suite and the test suite has been improved.

If it is showed that the mutant could not be killed by adding more test cases, the mutant will be considered either an *equivalent* mutant or a *stubborn* mutant. An equivalent mutant is a mutant that produces the same output as the original code, see Figure 2-3. Because of this it can not be killed [10]. If the mutant is live but not equivalent it is considered stubborn [16].This does not mean that the mutant is unkillable, merely that no test case has yet been discovered with the ability to kill the mutant [17].
2.3.2 Mutation Score

The quality of a test suite is measured by its mutation score. The more non-equivalent mutants that the test suite kills, the higher mutation score it gets and the better the test suite is considered to be. Mutation score is calculated according to the following equation: [10]

\[
\text{mutation score} = \frac{\text{number of dead mutants}}{\text{total number of non-equivalent mutants}}
\]

2.3.3 Creating mutants

To mutate the original source code the programmer can inject faults in the code by hand. Another option is to use an automated tool or an existing framework to generate and apply code changes at appropriate mutation sites, i.e. locations in the source code that could be mutated [18].

One of the first fully functional mutation testing environments created was Mothra [19]. Its development was initiated in 1986 and it was written using the C programming language. The main idea of Mothra is to translate, modify and execute programs. It has provided the basis of a several research works done on mutation testing [5] [20] and it provides the ability for developers to extend the tool with additional functionalities. That has allowed the tool to grow over the years as new research have appeared.

Since Mothra, many tools for automatically generating mutants have been developed, several of them written for Java. Some examples of such tools are Jumble [21], Jester [22], muJava [23] and Judy [24]. For the C/C++ programming language however, there are currently only a few mutation tools available. Most of these are in the stage of development and they are rarely well documented. An example of this is Mull [25], a mutation testing system implemented using LLVM. It is compatible with any LLVM based language and uses small mutant selection to reduce the number of mutants. An example of a relatively more mature mutation tool for C/C++ programs is CCMutator [26]. CCMutator provides the ability to generate higher order mutants.

2.3.4 Mutant operator

The set of code changes required to create a mutant are known as mutant operators [5]. These come in a wide variety that differ depending on the programming language in use. Mothra for example, uses a set of 22 mutant operator classes [27], see Table 2-4.

<table>
<thead>
<tr>
<th>Original code</th>
<th>Mutated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 0;</td>
<td>int i = 0;</td>
</tr>
<tr>
<td>while(...){</td>
<td>while(...){</td>
</tr>
<tr>
<td>i++;</td>
<td>i++;</td>
</tr>
<tr>
<td>if(i &gt;= 5)</td>
<td>if(i == 5)</td>
</tr>
<tr>
<td>break;</td>
<td>break;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Figure 2-3. Example of an equivalent mutant.
Table 2-4. Mothra mutant operators

<table>
<thead>
<tr>
<th>Mutant Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAR</td>
<td>Array Reference for Array Reference Replacement</td>
</tr>
<tr>
<td>ABS</td>
<td>Absolute Value Insertion</td>
</tr>
<tr>
<td>ACR</td>
<td>Array Reference for Constant Replacement</td>
</tr>
<tr>
<td>AOR</td>
<td>Arithmetic Operator Replacement</td>
</tr>
<tr>
<td>ASR</td>
<td>Array Reference for Scalar Variable Replacement</td>
</tr>
<tr>
<td>CAR</td>
<td>Constant for Array Reference Replacement</td>
</tr>
<tr>
<td>CNR</td>
<td>Comparable Array Name Replacement</td>
</tr>
<tr>
<td>CRP</td>
<td>Constant Replacement</td>
</tr>
<tr>
<td>CSR</td>
<td>Constant for Scalar Variable Replacement</td>
</tr>
<tr>
<td>DER</td>
<td>DO Statement End Replacement</td>
</tr>
<tr>
<td>DSA</td>
<td>DATA Statement Alterations</td>
</tr>
<tr>
<td>GLR</td>
<td>GOTO Label Replacement</td>
</tr>
<tr>
<td>LCR</td>
<td>Logical Connector Replacement</td>
</tr>
<tr>
<td>ROR</td>
<td>Relational Operator Replacement</td>
</tr>
<tr>
<td>RSR</td>
<td>RETURN Statement Replacement</td>
</tr>
<tr>
<td>SAN</td>
<td>Statement Analysis</td>
</tr>
<tr>
<td>SAR</td>
<td>Scalar Variable for Array Reference Replacement</td>
</tr>
<tr>
<td>SCR</td>
<td>Scalar for Constant Replacement</td>
</tr>
<tr>
<td>SDL</td>
<td>Statement Deletion</td>
</tr>
<tr>
<td>SRC</td>
<td>Source Constant Replacement</td>
</tr>
<tr>
<td>SVR</td>
<td>Scalar Variable Replacement</td>
</tr>
<tr>
<td>UOI</td>
<td>Unary Operator Insertion</td>
</tr>
</tbody>
</table>

2.3.4 Reducing the Computational Expense

Mutation testing is known to be extremely computationally expensive because of the great amount of mutants that can be created. Many attempts have been made in order to reduce the computational expense of mutation testing. When the notion of weak mutation was
introduced, the original form of mutation testing was renamed *strong mutation* [3]. In strong mutation testing, the mutant must be executed to completion before comparing the output between the original source code and the mutated version. This is computationally expensive to do for a lot of mutants [4].

In [3] the concept of *weak mutation* was introduced. Weak mutation testing considers *components* in the source code. A component could be a variable reference, a variable assignment, an arithmetic expression, a relational expression or a boolean expression [3]. A mutant is created by mutating a component. Then, instead of executing the entire source code that the component is part of, weak mutation compares the output of the mutated component with the output of the original component immediately after the mutated component has been executed. Because mutant components are considered individually, several of them could be executed during the same test run, which saves a considerable amount of execution time [4].

One disadvantage of weak mutation testing is the inability to guarantee exposure of all errors [4]. For example, with weak mutation, test data could be created so that there would be a difference in output between the mutated and original component. With strong mutation though, that same test data might not generate an observable difference in the output created after execution of the entire mutant. This means that the mutant would be considered dead with weak mutation but live with strong mutation, creating false positives in the case of weak mutation.

Another method developed in order to reduce the computational cost of mutation testing is called *selective mutation* [5]. As previously mentioned, mutation testing has the potential to create a lot of mutants. With selective mutation only a few adequate classes of mutations to use are chosen to reduce the number of mutants created. This has been shown to perform very well, especially for smaller programs, with a coverage almost as good as with non-selective mutation.

Offutt et al. [6] did some work on how to reduce the number of mutants created. Their study presented a form of selective mutation and their results showed that by using only five of the 22 operator classes used in Mothra, the number of mutants created could be significantly reduced while still covering almost all the potential faults that strong mutation covers. Using only five operator classes led to a reduction of the execution time needed. The five classes chosen in the study of Offutt et al. were ABS (absolute value insertion), AOR (arithmetic operator replacement), LCR (logical connector replacement), ROR (relational operator replacement), and UOI (unary operator insertion).

### 2.4 Static code analysis

One way to automatically create mutants is to do a static analysis on the source code to find appropriate mutation sites. IEEE defines static analysis as “The process of evaluating a system or component based on its form, structure, content, or documentation.” [28].

Static code analysis is used to analyze software without executing it, as compared to dynamic analysis in which the software is analyzed during execution. By statically analyzing the code, an abstract syntax tree (AST) can be created. An AST is an intermediate representation of the syntactic construction of the source code. One way to generate an AST is to use a compiler. The AST can then be traversed in order to find mutation sites to modify so that a mutant is generated. One example of a mutation site could be a binary operator in an equation, see Figure 2-4.
2.5 Safety critical systems

A safety critical system is a system that holds very small, or close to no margins for errors. Examples of such systems are avionic systems and banking systems where errors and bugs can lead to severe economical setbacks or human losses.

When developing safety critical software there are certain guidelines that should be followed. For airborne systems, these guidelines can be found in the document “Software Considerations in Airborne Systems and Equipment Certification”, also known as DO-178C [2].

DO-178C provides guidelines for the planning, development, and verification of airborne software. By following these guidelines, a certain degree of confidence should be achieved in the safety of the system under development. The safety criticality of the system, i.e. the impact a failure has on the system, is defined in the DO-178C as five software levels, also known as Design Assurance Levels (DAL). Each DAL reflects a certain failure condition. The failure condition states the effect a failure causes on the airplane or its occupants [2]. As can be seen in Table 2-5, the more critical levels require more coverage during testing than the less critical ones.

Table 2-5. Required coverage for different Design Assurance Levels.

<table>
<thead>
<tr>
<th>Level</th>
<th>Failure Condition</th>
<th>Effect of Anomaly</th>
<th>Required coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Catastrophic</td>
<td>Prevent continued safe flight and landing.</td>
<td>Statement coverage, Decision coverage, MCDC</td>
</tr>
<tr>
<td>B</td>
<td>Hazardous</td>
<td>Serious or fatal injuries to occupants.</td>
<td>Statement coverage, Decision coverage</td>
</tr>
<tr>
<td>C</td>
<td>Major</td>
<td>Discomfort to occupants, possibly including injuries.</td>
<td>Statement coverage</td>
</tr>
<tr>
<td>D</td>
<td>Minor</td>
<td>Inconvenience to occupants.</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>No Effect</td>
<td>No effect on the aircraft safety.</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 2-4. An example of an AST
3. Method

This section describes how mutation was performed by developing a mutation tool and how the applications to perform mutation testing on was chosen.

3.1 Mutation Tool

Mutants needs to be created in order to apply mutation testing on software. There are two possible approaches for this:

1. Mutate code manually.
2. Mutate by using an automated tool.

Manual mutation is very time consuming compared to automated mutation. Because of this the automated mutation approach was chosen in this study.

The automated tool needed to be compatible with C/C++ code since this is the programming language of the source code and test programs used in this thesis. No well documented mutation tool for C/C++ could be found though, and since only basic functionalities were needed, an automated mutation tool designed for the specific use of this thesis was developed from the beginning.

The tool needed the ability to locate mutation sites based on selected mutant operators and to edit (mutate) these sites accordingly. The tool was built based on Clang/LLVM. Clang is a front end compiler for C like languages using LLVM as its back end compiler. It consists of a library based architecture that has the ability to parse and analyze source code and is therefore very useful when creating source to source transformation tools. The mutation tool was developed by using libTooling [29], which is a library for writing tools based on Clang. The tool parses the source code into an abstract syntax tree (AST) that is traversed in order to find mutation sites based on the selected mutant operator. Once such sites are found the code is manipulated in such a way that new copies of the source code are created.

Based on the coupling effect [14] it is believed that complex faults will be discovered if all simple ones are, motivating the use of first order mutation. Furthermore, compared to higher order mutation, first order mutation was easier and faster to implement and because of this, the tool only performs first order mutation. This means that each mutant only contains a single syntactic change in contrast to multiple changes.

Based on the promising results of Offutt et. al [6], the five previously mentioned operator classes were selected to be used as mutant operators by the mutation tool under development. However, to implement the ABS mutant operator, functions from the standard library of C/C++ needed to be used, and questions of how to do this were raised, and whether it would be compatible with the code to be mutated. Therefore, the ABS mutant operator was not implemented.

1 https://clang.llvm.org/features.html#libraryarch
3.1.1 Mutant Operator Class Implementation

This section provides a description on how the mutant operator classes were implemented. Table 3-1 illustrates the four chosen mutant operator classes and the respective operators they apply to.

Table 3-1. Illustration of the mutant operators of the four mutant operator classes.

<table>
<thead>
<tr>
<th>Mutant Operator Class</th>
<th>Description</th>
<th>Respective Mutant Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOR</td>
<td>Arithmetic Operator Replacement</td>
<td>+, -, /, %, *</td>
</tr>
<tr>
<td>LCR</td>
<td>Logical Connector Replacement</td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>Relational Operator Replacement</td>
<td>&lt;, &gt;, &lt;=, &gt;=, ==, !=</td>
</tr>
<tr>
<td>UOI</td>
<td>Unary Operator Insertion</td>
<td>++, --</td>
</tr>
</tbody>
</table>

The mutation tool is divided into four different tools, each managing one of the mutant operator classes.

To implement the tools that handle the AOR, LCR and ROR mutant operator classes the mutation tool searches for binary operators in the original source code. Once a binary operator is found, mutants will be created by replacing the binary operator with all the respective mutant operators for that class. For example, if the binary operator < is found by the ROR tool, that operator will be mutated into >, ==, !=, >= and <=, creating a new mutated file for each mutant operator. This means that five new mutated files, each containing one mutation, will be created every time the ROR tool encounters an operator corresponding to the ROR class, see Figure 3-1.

![Figure 3-1. Illustration of ROR mutation.](image)

The implementation of the UOI class differs from the other mutant operator classes implemented. UOI does not depend on finding binary operators. Instead it searches for expressions and predicates that contains variables of type integer. Once such a variable is found, the mutation tool inserts the increment operator (++) and the decrement operator (--) before and after the variable, i.e. postfix and prefix, see Figure 3-2

![Figure 3-2. Illustration of UOI mutation.](image)
The output obtained after running each tool is a set of mutants and a file containing information about all the mutants; what has been mutated, to what and where in the code the mutation can be found.

The mutation tool does not mutate any declarations of variables, neither left hand sides of assignment operators. Furthermore, mutations were applied to all sub-expressions recursively.

### 3.2 Data Collection

Relevant code to mutate had to be chosen in order to make a comparison of mutation testing on different levels of DAL. What differentiates one DAL level from another is essentially the degree of testing done on the application. Therefore, the idea was to choose an application from the highest level, i.e. a DAL A application and then disabling tests in order to represent the lower levels. Then all levels would be represented using the same source code, only differing in the number of test cases. This would place the focus of the study on the test suites and not the source code, and it would make the comparison of results between different levels more accurate since the same mutants could be used for all levels. This method was not feasible to use though, since no information existed about which level of DAL a test case belonged to. Instead, a list of available applications to test from was provided by Saab and a selection was made from these based on their DAL level.

Three applications were chosen, representing level A, C and E. Originally each DAL level was intended to be tested, i.e. five applications. However, there were no applications from level B or level D available to test in the scope of this thesis.

Each application chosen consists of several separate source files together with test suites created with the Google C++ Testing Framework [30]. In order to test an entire application every source file containing mutation sites was mutated individually. After generating mutants for one file, the test suites for the entire application were run on each mutant. Data was then collected on whether the mutant survived or not together with statistics on the amount of mutants created. The process of mutating and running tests was then iterated through every file in the application.

### 3.3 Evaluation

After running the test suites on all mutants of each application the data collected was summarized and mutation score was computed for all applications. However, no identification of equivalent mutants was performed since the process of determining whether a mutant is live because it is equivalent or not is very complex and time consuming. Each surviving mutant would have to be manually examined and a lot of knowledge is needed about the expected behaviour of the software under test to be able to determine equivalence. Due to this, all surviving mutants were treated as non-equivalent.
4. Results

In this section the results from running the test suits on the mutated files are presented.

4.1 Overview

The applications that were tested are named Application A, C and E according to their DAL level. Only source code files (no header files) were mutated and table 4-1 illustrates some general information about each application.

<table>
<thead>
<tr>
<th></th>
<th>Lines of Code</th>
<th>Number of Files</th>
<th>Number of Files Mutated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application A</td>
<td>4493</td>
<td>34</td>
<td>19</td>
</tr>
<tr>
<td>Application C</td>
<td>6849</td>
<td>44</td>
<td>24</td>
</tr>
<tr>
<td>Application E</td>
<td>2448</td>
<td>33</td>
<td>13</td>
</tr>
</tbody>
</table>

Only some of the files corresponding to each application contains mutation sites, i.e. something to mutate, as illustrated in Number of Files Mutated.

4.2 DAL A

The results from running the test suits on application A can be seen in table 4-2. Table 4-2 illustrates the number of mutants generated of each mutation class, how many of these that survived, how many that were killed and the resulting mutation score.

<table>
<thead>
<tr>
<th></th>
<th>Number of Mutants</th>
<th>Killed Mutants</th>
<th>Live Mutants</th>
<th>Mutation Score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR</td>
<td>1060</td>
<td>861</td>
<td>199</td>
<td>81</td>
</tr>
<tr>
<td>LCR</td>
<td>171</td>
<td>163</td>
<td>8</td>
<td>95</td>
</tr>
<tr>
<td>AOR</td>
<td>165</td>
<td>159</td>
<td>6</td>
<td>96</td>
</tr>
<tr>
<td>UOI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>1396</td>
<td>1183</td>
<td>213</td>
<td>85</td>
</tr>
</tbody>
</table>
4.3 DAL C

The results from running the test suits on application C can be seen in table 4-3. Table 4-3 illustrates the number of mutants generated of each mutation class, how many of these that survived, how many that were killed and the resulting mutation score.

**Table 4-3. Data collected from running the test suites on application C.**

<table>
<thead>
<tr>
<th></th>
<th>Number of Mutants</th>
<th>Killed Mutants</th>
<th>Live Mutants</th>
<th>Mutation Score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR</td>
<td>790</td>
<td>625</td>
<td>165</td>
<td>79</td>
</tr>
<tr>
<td>LCR</td>
<td>133</td>
<td>130</td>
<td>3</td>
<td>98</td>
</tr>
<tr>
<td>AOR</td>
<td>483</td>
<td>475</td>
<td>8</td>
<td>98</td>
</tr>
<tr>
<td>UOI</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td>Total</td>
<td>1414</td>
<td>1234</td>
<td>180</td>
<td>87</td>
</tr>
</tbody>
</table>

4.4 DAL E

The results from running the test suits on application E can be seen in table 4-4. Table 4-4 illustrates the number of mutants generated of each mutation class, how many of these that survived, how many that were killed and the resulting mutation score.

**Table 4-4. Data collected from running the test suites on application E.**

<table>
<thead>
<tr>
<th></th>
<th>Number of Mutants</th>
<th>Killed Mutants</th>
<th>Live Mutants</th>
<th>Mutation Score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR</td>
<td>300</td>
<td>225</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>LCR</td>
<td>44</td>
<td>36</td>
<td>8</td>
<td>82</td>
</tr>
<tr>
<td>AOR</td>
<td>50</td>
<td>38</td>
<td>12</td>
<td>76</td>
</tr>
<tr>
<td>UOI</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>394</td>
<td>299</td>
<td>95</td>
<td>76</td>
</tr>
</tbody>
</table>
5. Discussion

This section discusses the results and the method used, together with the limitations of the thesis and the developed mutation tool, followed by source criticism.

5.1 Applications

The applications to mutate were chosen mainly based on their availability. As mentioned in the method, a list of applications available for testing were provided. Applications were then chosen based on their DAL level without considering the size of the applications, how many mutants they would generate or if they were equally complex in their behaviour. This is something that should be considered when evaluating the results.

Furthermore, only three applications were tested, one from each available level. This means that the results are very specific to one certain application. By choosing other applications the results might differ substantially. This could have been avoided by testing more applications from the same DAL level and then calculate an average mutation score for each level.

Another limitation of the method used is that only three out of five DAL levels could be tested. Since the goal of the thesis is to compare mutation score between different DAL levels, testing all five levels would have provided a more accurate result. As mentioned in the method, the idea was to use the same application for all levels, bypassing the problem of finding comparable applications from each level.

5.2 Mutant operators

When running the mutation tool on real production code from Saab it was discovered that very few UOI mutants were created. As can be seen in the results only eight UOI mutants out of a total of 3204 mutants were created. This makes it impossible to draw any proper conclusions about this particular mutation class.

The absence of UOI mutants can be explained by mainly one reason. The UOI tool searches for integer variables to mutate and Saab’s production code rarely contains any but instead contains classes they created themselves. To solve that, the tool could be extended to look for the classes of type integer used by Saab.

Since only four of the five operator classes described by [6] were used in this thesis, the quality of the mutation testing might have been decreased by an unknown factor.

5.3 Results

When testing a test suite using mutation testing, a mutation score as high as possible is desired. In the result it can be seen that no mutation score for any of the applications was higher than 87% which is quite a low score. An acceptable score would be somewhere close to 100%. For lower levels of DAL a lower mutation score was expected, since they have less testing criterias to fulfil. The overall low mutation scores among the applications could indicate that the test suites are not able to detect all potential faults, but it might also be because of equivalent mutants. When computing mutation score all equivalent mutants are removed from the equation. In this thesis though, no identification of equivalent mutants are done. If there are any equivalent mutants, i.e. mutant that cannot be killed, they will be part of the calculation, resulting in lower mutation scores.
As can be seen in the results, the DAL C application received the highest mutation score of 87%. Since DAL A applications must fulfill more types of code coverage than lower levels according to DO-178C it would be reasonable to believe that the DAL A application would receive the highest mutation score. Instead it received a slightly lower one of 85%. This could indicate that the test cases for DAL A applications are not as sufficient as they should be and that the added testing activities of DAL A applications compared to DAL C applications do not add to the quality of the test suites. However, a more likely scenario is that this indicates that since only one application was tested on each DAL level, the results are biased to the specific applications tested and more applications would have to be tested in order to get a more accurate result.

The DAL E application received the lowest mutation score at 76%. This was expected since DAL E applications are less tested according to DO-178C. However, the mutation score of the DAL E application is not very far from either the DAL A or DAL C applications indicating once again that either the added testing activities to higher levels does not add to the accuracy of the test suites, or the results are dependent on each specific application making it hard to compare the results.

The applications differ quite a lot in code size and number of mutants created. Since only one application from each level is tested, the results are closely related to the specific application chosen. Choosing another application might have resulted in another mutation score. This might be a reason for why the DAL C application got a higher mutation score than the DAL A application.

Something that is recurrent for all applications is that a large proportion of the mutants created were ROR mutants. In the DAL A application, 76% of the mutants generated were ROR mutants, 56% of the generated mutants in the DAL C application were ROR mutants and in the DAL E application, 76% were ROR mutants. Out of 3204 generated mutants, 2150 were ROR mutants, which equals 67%. This is partly due to the ROR class generating the most mutants in general since five mutants are created for each mutation site corresponding to the ROR class. It is also due to the specific applications being tested since mutants generated by the ROR class directly correlates to the amount of relational operators in the application code.

What is interesting however, is that compared to the LCR and AOR classes that received quite a high mutation score in all three applications, the ROR class received quite a low mutation score. The ROR tool mutate relational operators, for example >= to >. By for example, removing the equal sign in a comparison, one is able to test boundary values. For the most critical DAL levels, MCDC has to be fulfilled, which is an extensive coverage criteria in which, amongst other things, conditions must be evaluated to both true and false. If MCDC is fulfilled, then one would think that mutants created by changing operators in conditions would be discovered by the test suites. The low mutation score of the ROR mutants does not verify this theory though. One reason for the low score could be due to equivalent mutants. If not, it could indicate that the test cases at Saab do not test all boundary values necessary.

What can be seen though is that DAL A has a higher mutation score of the ROR mutants than C and E, which it should have since DAL A has more coverage criterias to fulfil, than level C and E.
6. Conclusion

Based on the results alone, it could be seen that the test suites of different DAL levels reach quite similar mutation scores. This would imply that the quality of the test suites are quite similar for all the applications. It could also be seen that the ROR class generates the most mutants as well as the most surviving mutants, which could indicate that the test suites at Saab do not test all necessary boundary values.

The goal of this thesis was to discover if mutation testing can say something about the quality of the test suites corresponding to different DAL levels. The idea was that higher DAL levels would receive higher mutation scores. By only taking the results into account it could be seen that this was not the case. While DAL E received a lower score than the other levels, DAL C received a higher score than DAL A, and their scores were quite similar. However, by taking all the possible limitations into account it can be seen that further research is required in order for mutation testing to say anything about the test suites and their correspondence to different DAL levels.
7. Future Work

This section describes some possible future works to improve the accuracy of the results in this thesis.

7.1 Extending the Mutation Tool

At this moment only four of the five mutant operators suggested by [6] are implemented in the mutation tool. To improve the results of this thesis we would like to extend the tool to also include the ABS mutant class.

We would also like to improve the existing UOI class by not only looking for integer variables, but also for variables that could be incremented/decremented specific to the code under test.

Another suggestion would be to extend the tool to create more complex faults by creating higher order mutants. This way the test suites could be tested for more complex faults.

7.2 More Data

In this thesis only three applications were tested. We would like to further investigate the effects of mutation testing on different levels of DAL by collecting data from more applications. This way the results would not be as application specific and a more accurate evaluation could be made.

It would also be preferable if data could be collected from all DAL levels to get a more extensive study.

7.3 Equivalent mutants

Lastly, we would like to extend the research by going through all the surviving mutants in order to discover if they are equivalent or not. This would provide a more accurate mutation score.
References


