Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Mutation testing as quality assurance in base station software

by

Niclas Norman

LIU-IDA/LITH-EX-G--14/030--SE

20014-06-17

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final thesis

Mutation Testing as quality assurance in base station software

by

Niclas Norman

LIU-IDA/LITH-EX-G--14/030--SE

2014-06-17

Supervisor: Ola Leifler
Examiner: Ola Leifler

Abstract

Telecom base stations are a critical part of society's information infrastructure. To ensure high quality base station software, automated testing is an important part of development. Ericsson measures the quality of automated tests with statement coverage, counting the number of statements executed by a test suite. Alone, however, statement coverage does not guarantee test quality. Mutation testing is a technique to improve test quality by injecting faults and verifying that test suites detect them. This thesis investigates whether mutation testing is a viable way to increase the reliability of test suites for base station software at Ericsson. Using the open-source mutation testing tool MiLu, we describe a practical method of using mutation testing that is viable for daily development. We also describe how mutation testing reveals a numbers of potential errors in the production code that current test suites miss even though they have very good statement coverage.

Acknowledgment

First I would like to thank Anders Nilsson for giving me the opportunity to do this thesis at Ericsson. It has been very interesting and pleasant weeks at BBI-tools in Linköping. I would also like to thank all the coworkers at BBI-tools for their help and expertise.

In addition I would like to thank my supervisor and examiner Ola Leifler for giving me directions and pointing out problems. All your constructive criticism and ideas have been greatly appreciated.

Last but not least I would like to thank my supervisor at Ericsson, Jonas Svanberg who has given me excellent support and expertise, without your help this would not have been doable.

Niclas Norman
Linköping, Sweden
June 2014

Table of Contents

1 Introduction

When developing software one of the biggest and most important parts of the process is to find errors and bugs. The problem is that it does not generate profit but will increase the cost if neglected. This makes software testing problematic since it must be handled with as little money and time spent as possible. The later an error is detected the more time and money it will cost to correct it. The worst-case scenario is when an error is detected after delivery. You want to find an error or bug as early as possible to minimize the cost.

Software development for embedded systems adds some complexity to the development process. Embedded systems have more limitations in general and hardware additions that a developer must take into consideration. In comparison with an ordinary computer running a multipurpose operating system like Linux, OS X or Microsoft Windows an embedded system is built and programmed for a specific purpose or task.

The telecom industry uses embedded system software for the base stations. A base station is responsible for handling cell phone communication. The software in base stations is very complex and the uptime requirements are very high.

Base station customers are telecom operators. They have high demands on uptime from their subscribers and from government regulations. Since that base stations are used for infrastructure they are critical in emergency situations where cell phone communications is essential. A critical part of the development is to meet these demands. This will put high requirements on tests so that they really check for errors and bugs. The problem is that it can be hard to know if the test suite really covers the errors programmers are prone to make.

1.1 Justification

There are different ways to measure the quality of a test suite including *statement coverage* (Myers, 2004). The statement coverage score tells us if the tests executed the statements in the source code. If we get a value of 100%, we know that the tests executed all the statements in the code. One big problem here is that you could get a 100% coverage score by doing bad tests. Because of this it could be bad to only rely on statement coverage without having additional quality metrics on the test suite. Myers (2004) claims that statement coverage alone is bad. Let us look at the example in figure 1 below:

```
int foo(int a, int b){
  int to_return;

  if(a == b){
    to_return = a;
  }
  return to_return;
}
```

Figure 1. Example of error in program code.

If we run a test for the function in figure 1 with two values like three and three the function will return three.

```
assert( foo(3,3) == 3);
```

This is correct but the function still contains a possible error. If we would send, three and two we do not know the result.

```
assert( foo(3,2) == ?);
```

This is because we missed the initialization of the variable *to_return*. This problem is quite interesting, if we test it with three and three as parameters we execute every statement and we would get statement coverage of 100% but we still have possible errors in the code. This states that we need something more than just statement coverage to tell us the quality of the tests. This is a trivial example and there are tools that would find these types of errors.

Mutation testing is a technique to assess the quality of a test suite (Yue & Harman ,2011). By injecting atomic faults in the source code, mutation testing could be seen as a simulation of errors programmers are prone to make. By testing these mutants, we could get an indication of the quality of the test suite.

1.2 The problem

The work flow for a developer usually starts with a feature or function that needs to be added to the production code. When the development is done, tests are written and compiled with the production code. The testing of the software is conducted in different levels before it is labeled approved. The first level is the tests written by the individual developer. When this is approved the next phase of testing is conducted by adding more of the production software and other tests. For every step, more production software and tests are added which makes the procedure increase in both time and complexity. If we see this process as a testing chain the first part is unit testing where the developer tests his own production code with his own test. The next part of the chain is an integration test, which could introduce for example the production code of the team and their test and so forth.

The hardware in the base stations are specialized hardware platforms, which include digital signal processors (will henceforth be known as DSP). For software development on DSP-platforms there is an extension to ANSI C called DSP-C, which adds types and functions exposing specialized operations on DSP. Ericsson uses a proprietary dialect of DSP-C. This adds some complexity to the writing of tests and the development of support tools.

There are many different tools to help developers at Ericsson conduct tests on their software but the developers write the tests themselves. Among the tools used for testing there are different simulators that simulate the real DSP hardware (real DSP hardware can be used). Because of build times and simulator running speeds it takes considerable time to run tests. Execution and build times can be somewhere between one minute up to 30 minutes or even longer. If we look at figure 2 we can see the working flow of running tests with a simulator. We start with our developed production code and a test suite for this code. To be able to get this into the simulator it needs to be built to a binary, this requires some external code like the operating system and

other platform dependencies. The test we have written could also make calls to external functions (called Host test suite in figure 2) or vice versa. When the tests are completed we get information about the outcome of the tests and a coverage score.

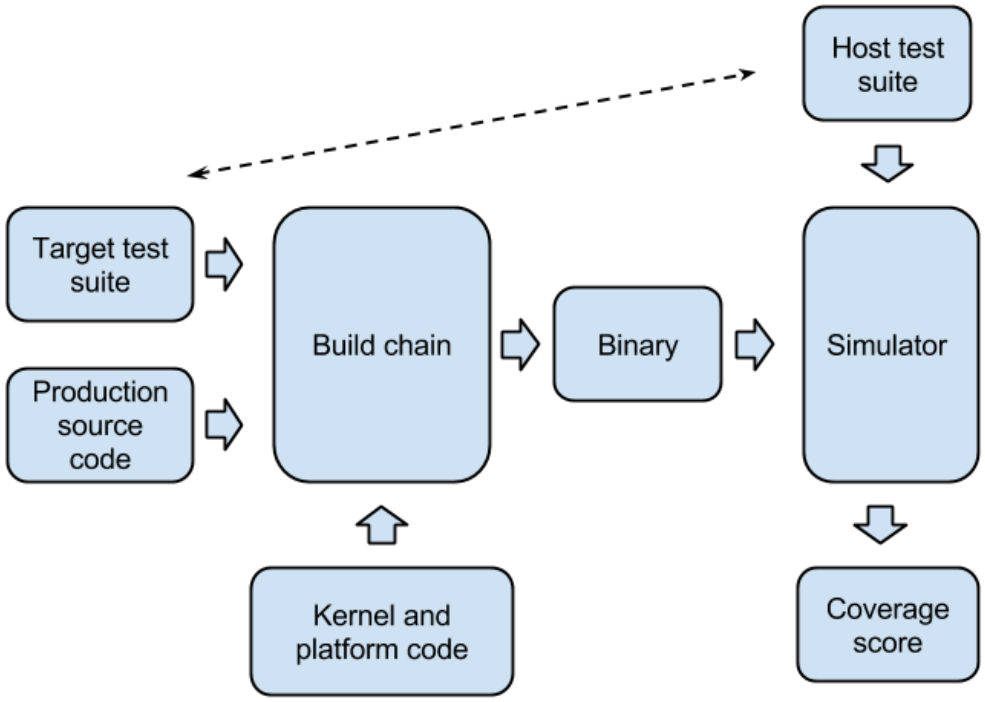

Figure 2. Simplified flowchart over the simulator testing procedure.

In addition to the simulator there is a unit-testing tool called Swift (the simulator is more of an integration-testing tool). Swift is a Linux software, which makes it possible to compile and run directly on a local machine or a server. Instead of using all the external dependencies, Swift creates mocks of these so that the software can be compiled and executed. Mocks are simulations of external dependencies like functions or objects (Hamill, 2009). The mocks behave like the original functions or objects to the extent needed for testing. One important part of mocks is that they can validate how they are used. Swift is based on *Google mock* and *Google test*. Google mock[1] is a framework for generating mock objects. Google test[2] is a framework for creating tests using C++. Figure 3 illustrates the Swift testing procedure. Swift will speed up the running of the test suite considerably but will not be able to handle some of the DSP-C extensions. The file generated by Swift is an executable file that can be run on any normal developer Linux machine.

[1] https://code.google.com/p/googlemock/
[2] https://code.google.com/p/googletest/

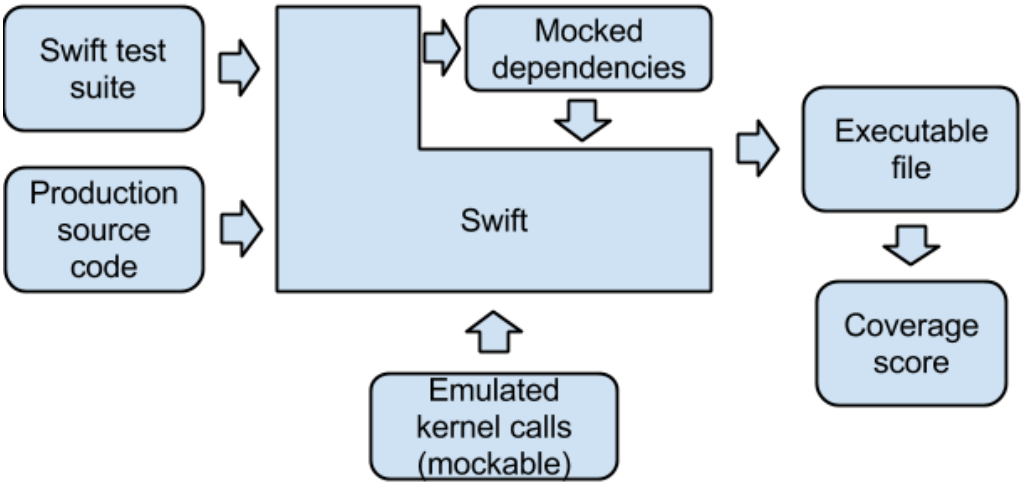

Figure 3. Simplified flowchart of the Swift testing procedure.

After the production software and the tests have been developed, the developer can run it on a test platform (simulator or Swift). After the tests have completed, a coverage score is presented. If the coverage is above a certain threshold, the tests are said to be good enough. This is done in iterations as test cases and production code is modified. When the coverage score has reached above the threshold, this score becomes the new threshold. For instance, if we get a coverage score of 85% and the threshold was 80%, in the next iteration we need to get at least 85% coverage to get an approval. The problem here is that coverage does not say much about the quality of the test, only that the test executed lines in the production code.

To be able to verify the quality of the test cases with a method like mutation testing, which relies on source code fault injection, we need a smooth way to run the mutated versions of the production code against the test suites. The injected fault must also be compiled with the test suite to build an executable binary.

1.3 Thesis purpose

The purpose of this thesis is to answer the question whether mutation testing is a viable way to increase the reliability of test suites for base station software at Ericsson. This question is big and needs to be split into smaller parts and that gives us three questions that are more specific.

1.4 Research questions

- Is mutation testing viable in the Ericsson context of baseband development?
- Is common code coverage a good measure of code quality?
- Could mutation testing increase the quality of the test suites?

1.5 Scope

The more mutants we generate, the more we should be able to discuss about mutations testing. Due to this we have chosen to limit us to one part of the test chain. Since the first part of the test chain is the one with the lowest complexity and running times this will be the focus of this thesis. This gives this thesis a focus on the unit testing part of the production code. As a natural consequence, the focus will be on the tool Swift that is developed just for unit testing. Because of this, we will only be able to discuss errors detected at the unit test level.

Since the base station production code is large, we need to select a subset of it. The focus will be on production code with tests written for Swift.

2 Theory and related work

To get an introduction to the field of software testing we have made a summary of the most important concepts related to this thesis. In addition to this, we have added an introduction to the field of mutation testing.

2.1 Software Testing

IEEE has defined testing as "*An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.*" IEEE Std 829-2008, p11.

The literature often divides testing as two different categories called black box and white box testing (Copeland ,2004). These two are directly related to the definition specified by IEEE. Black-box testing is a method for requirements and specification testing. The concept is to look at the test object as a black box and send input based on the specifications, the return value should be as stated in the requirements. The drawback with this is that we do not know anything about the inside of the tested object. We cannot be sure that there are no errors or bugs present since we do not know that all statements or expressions are evaluated. To be sure of this the test suite must contain tests that test every possible valid and invalid input, which is not feasible. This forces the tester to come up with a test suite that is effective and efficient at finding bugs and errors since we cannot test with every possible input. This is the advantage of the black box testing technique as it gives us efficient and effective test cases (Copeland ,2004). Another advantage is that manual black box testing does not require any programming skills since the tester uses the specifications for the test.

White box testing is not the opposite of black box but a complement (Copeland ,2004). Instead of looking at the specifications, we need to look at the internal structure of the program. Using this method, we have knowledge about the internal structure of the test object and the ability to adjust the input parameters after this knowledge. By doing this the tester can check the different internal states of the test object and find bugs that otherwise would be hard to find. The big disadvantage with white box testing is the requirements of programming skills. In addition, the problem with testing all the paths in the internal structure could be as infeasible as the case with black box testing. One positive aspect of white box testing is that the internal paths of the test object can be checked in a structured way.

If we look at the internal states of a program we can see that this could be translated into a graph. Ammann and Offutt (2008) discuss this view and states that this is the most used metric among testers today. By using a graph with nodes and edges we can reproduce the complete program or a part of it and get a picture of the flow. If we look at figure 4 we can get an idea of what such a graph may look like.

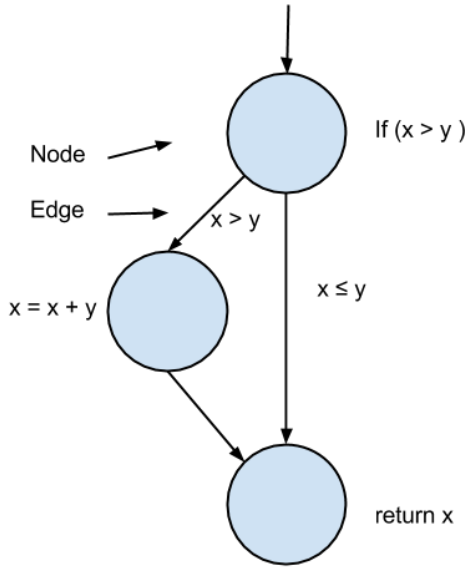

Figure 4 figure over a simple program graph.

The code for the example in figure 4 would look something like this:

```
If (x > y) {
      x = x + y
}
Return x
```

By using this approach we can get different coverage types based on how we visit the nodes. One way is statement coverage that we discussed in the introduction chapter. To get a high statement coverage we need to visit all the nodes in the graph. In the example from figure 4 we need to make sure that we evaluate the *if-statement* to true and check the value returned. If we would do this we would get 100% statement coverage. Statement coverage is the lowest level of coverage (Copeland, 2004). Another coverage type is *branch coverage* where we need to evaluate each decision to true and false. If we look at the example in figure 4 again we would need to do one test that evaluates the *if-statement* to true and one that evaluates false and check the return value in both cases. If we would do that we would get 100% branch coverage. There are more coverage types but these two gives a basic idea of the coverage metric.

If we look at how coverage works and what it measures we can see that we only check how much of the source code we cover with the test. In statement coverage we just check that all the statements have been visited in the graph. Boris Beizer (1990) claims that getting a lower score than 100% statement coverage is bad due to that it leaves parts of a program untested.

If we have bad written tests we could get 100% coverage but still have errors in the code. We gave an example of this in the introduction chapter (see section 1.1). The same is with branch coverage. Inozemtseva and Holmes (2014) argue that high coverage is not an indication of an effective test suite. They also state that expensive coverage methods do not always indicate that

the test suite has high quality. If this is the case we need something to indicate if we have an adequate test suite that is not based on coverage metrics alone.

2.2 Mutation Testing

Mutation Testing is a test method for test suites. By injecting faults in the original program and then run the test suite we get a measurement of the quality of the test suite. The concept of mutation testing is from the 70´s and it has been an exponential increase of published papers from then until 2009 (Jia & Harman, 2011). The increase in interest could be from the increase in computational power from today's computers. Mutation testing is quite computationally intensive since each mutant is a copy of the original program that needs to be tested against the test suite.

Mutation testing is based on two hypotheses; the *Competent Programmer Hypothesis* and *The Coupling Hypothesis* (Yue & Harman ,2011). The Competent Programmer Hypothesis states that programmers are competent and produce programs that are close to the correct version (Assylbekov & DeMillo, 1978). Of course, there are faults in programs but they tend to be small and simple. The Coupling Hypothesis states that tests that find small errors will also find a high proportion of more complex errors. This is due to the sensitivity of tests that find small errors. These two hypotheses are important since mutation testing depends on the ability to find small injected faults in the source code (Jia & Harman, 2011). From that you should be able to tell if your tests are able to find errors not just small but even more complex.

For mutation testing to be useful we need some metrics. The quality of a test suite as defined by mutation testing is measured by its ability to detect mutations in the source code (Jia & Harman, 2011). If a test detects a mutant it is said to be *killed* (a program crash would also count as a killed mutant) and if not killed, we have a *surviving* or *alive* mutant. From this we can get a *mutation score*. The mutation score is the proportion of killed mutants relative to the total number of mutants. One problem that we could get is infinite loops this needs to be handled by the tests by some timeout, if this is not done we would not be able to get a mutation score.

$$mutation\ score\ = \frac{killed\ mutants}{total\ number\ of\ mutants}$$

In the best case we would get a mutation score of one, this would tell us that the test suite is able to kill all the mutants.

The mutation is quite simple; by modifying the source code we create a mutant. These modifications follow rules, for example with the logical connector replacement rule we create mutants by switching logical operators like AND for OR (Jia & Harman, 2011). These modifications rules are called *mutation operators*. The example on the next page is a typical logical connector replacement.

```
if (a == 1 && b == 1)
       //do something
return something
```

is substituted with

```
if (a == 1 || b == 1)
       //do something
return something
```

This is done for every AND operator, which will create one mutant for every different substitution made this way (Jia & Harman, 2011). For each new mutation we get a new version of the source file. In each version one single mutation is made. To distinguish between the different mutants, only one atomic mutation is made for each new version. There are a number of different mutation operators that can be applied to create mutants.

We have already mentioned one problem with mutation testing. It has a high computation cost due to all the mutants that are generated (Jia & Harman, 2011). Another problem is the *human oracle problem*. To be able to test something we need to have some idea of what to expect, if we have an input we need to know what output the program will return from the given input. This is not a problem directly linked to mutation testing, instead it is more of a general problem when it comes to testing. In addition to the human oracle problem mutants not detected by the test suite need manual work to check if they are possible bugs.

One of the biggest issues with mutation testing is the so called *equivalent mutant problem* (Jia & Harman, 2011). An equivalent mutant is a mutant that would produce the same result as the original program. For a better explanation we can look at the example below.

```
for(int i = 0; i  < size; ++i )
for(int i = 0; i != size; ++i )
```

These two lines will get the exact same result from the program. This is due to that the evaluation of the variable i will be false until i is not equal to $size$. Since we start at zero and increment by one each turn in the loop we will continue for as long i is not equal to $size$.

The test suite will not kill these mutants and that is correct, since they behave exactly as the original program (Jia & Harman, 2011). The problem is that they will be added to the list of not killed mutants so the test score will be misleading. The formula needs to be corrected to take these in consideration. If we look at the formula again it would look something like this.

$$mutation\ score\ = \frac{killed\ mutants}{total\ number\ of\ not\ equivalent\ mutants}$$

Budd and Angluin (1982) found this problem undecidable, it is impossible to distinguish an equivalent mutant from the original program for a test. There are a lot of studies and papers discussing different techniques for minimizing this problem (Jia & Harman, 2011). The main problem is that there is no easy way of handling the equivalent mutant problem. The consequences of not finding the equivalent mutants are that the mutant score can generally not become one. To identify the equivalent mutants we need to manually check the surviving mutants, which require additional time for the developers.

One method of decreasing the amount of mutants that needs to be manually checked is to select a couple of mutation operators to generate mutants from (Jia & Harman, 2011). Offutt et al (1996) has in their study found that there are five different mutation operators that are more efficient than others. These operators generate fewer mutants and are almost as efficient as using more operators. They show tendencies to generate fewer equivalent mutants than other operators do. This is a big gain since it will decrease the manual work required in checking the surviving mutants. The operators they are proposing to use are:

- Absolute value insertion, which forces an expression take a positive, negative and zero value.
- Arithmetic operator replacement, which substitutes each arithmetic operator for another legal arithmetic operators (plus for a minus).
- Logical connector replacement, which substitutes each logical connector for another logical connector (AND for OR).
- Relational operator replacement, which substitutes each relational operator for another relational operator (equals for a lesser than)
- Unary operator insertion, which inserts a unary operator in front of an expression (forces an expression to take a negative value).

Even if mutation testing has some problems Assylbekov et. al (2013) states that mutation testing can increase the code coverage score. This is because it will have an improving effect on the test suite. If the mutation finds deficiencies in the test suite, it will force an improvement of the tests. They also found a correlation of 0.5 between mutation score and coverage score. The coverage score was based on branch, statement and method coverage. Smith and Williams (2009) also argue that by using mutation analysis testers can improve coverage, in their case statement coverage. Another thing that the authors point out is that mutation testing is forcing testers to examine source code. In their conclusion they state that as long as mutants represent faults that programmers do, mutant analysis can be an effective method in improving a test suite.

3 Method

We started with a short planning phase where we decided to split the work into three different phases. In the beginning, we did not know much about the problem area. We needed some kind of survey to get a good understanding. We needed to check for earlier studies and results that had been done in the field. In addition we checked for the techniques used in those studies to get an idea how to attack the problem. To be able to answer our first research question:

Is mutation testing viable in the Ericsson context of baseband development?

We needed to develop a prototype for generating mutants. With the prototype we could generate empirical data to be able to answer the other research questions:

Is common code coverage a good measure of code quality?
Could mutation testing increase the quality of the test suites?

3.1 Exploratory

After the planning we continued by conducting an overall study of software testing. To get an initial understanding about the topic and its terminology we started by reading about software testing on Wikipedia[3]. This gave us an overview and basic knowledge of the terms used. We also checked the references on Wikipedia to check that they included published papers and well known secondary sources. We then used some of the books referenced there to get a deeper knowledge about software testing. We also used Linköping's University Library to find more literature about software testing. This was done by using the tools Unisearch and the library Catalogue. The search terms used was *software testing*, *software testing techniques* and *introduction to software testing*.

After we got an understanding about the topic software testing we moved over to the field for this thesis, which is mutation testing. This is a sub field to software testing. From the literature about software testing, we realized quite early that the books about software testing are quite limited in addressing the field of mutation testing. To get a basic understanding of the field we started by reading on Wikipedia[4] about mutation testing. To continue to get a better knowledge of mutation testing we started to check for scientific papers. We used searching tools provided by Linköping's University Library called Scopus and Unisearch. The search terms used were *Mutation testing, Software development, Software testing, Mutation score and Testing Coverage*.

When we got a good knowledge about the field of mutation testing we started looking for existing tools. We only focused on the ones created for the ANSI C language. We searched for these tools by using the papers found and Google search. We searched on Google with the phrase "mutation testing tools ANSI C".

[3] http://en.wikipedia.org/wiki/Software_testing
[4] http://en.wikipedia.org/wiki/Mutation_testing

We also needed to find some good test objects, in this case files that were suitable for mutation and with a Swift test suite. Only parts of the production code has Swift tests written for them. We selected a set of files by discussing with developers of the Swift tool since they had a good knowledge of which files in the production code that had Swift tests written for them. We also ran Swift on the different examples we found to get a basic idea of how Swift worked. The criteria we had for choosing files were that they needed Swift tests and a good coverage score. From the proposed files, we checked for the ones with the highest coverage score.

3.2 Implementation

For generating mutants, we used the mutation tool MiLu. We started by doing some simple test programs to learn how MiLu worked. This was time consuming due to incomplete documentation. To learn how to use it we checked the source code to be able to reverse engineer how to configure it. In the beginning, we tested MiLu with the default mutation operators and continued by trying different operators. When we got MiLu working with our examples we started to test it on production code. The first thing we noticed was that MiLu requires the source file to be run with for example "gcc -E". This is stated in the information about MiLu. This means that the source file needs to be run through the preprocessor before MiLu is able to mutate the file.

By doing some research we found that the Swift tool generates intermediate files where the preprocessor has run. With these intermediate files, we could start generating mutants with MiLu. This requires Swift to run once before the mutant generation can start.

The problem was that when running MiLu on our simple tests it worked well but when running it on the production code it crashed. We started investigating the problem and found the error. This was done in a few iterations until we got MiLu working for all the different test objects.

After we got MiLu working, we developed a tool to help us run Swift on all the mutants. This tool is based on a bash script (the complete script is included in appendix A) that runs MiLu on a user-specified file and mutates it based on selected mutation operators. When the mutation is done we copy and replace each mutant with the production code and run Swift. We keep track on tests that was successful (successful test means a surviving mutant) and add them to a list. This list contains mutants that the test suite did not catch. When all the mutants have been tested we get a score that tells us the ratio between killed mutants and the total number of mutants after we have subtracted the skipped mutants from the total. When the script is done testing all the mutants it generates an output file with statistics from the tests. In figure 5 we can see an illustration for how the mutation script works. Starting with a file of production code and ending with mutation statistics.

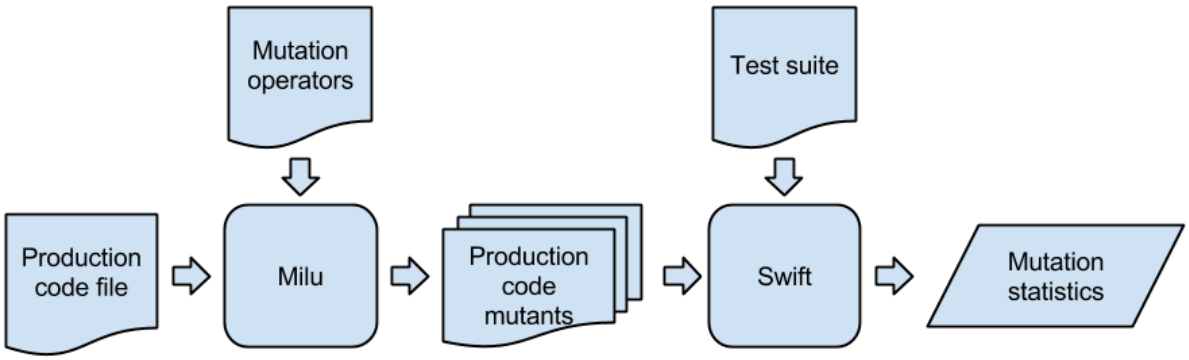

Figure 5. Flowchart over the mutation testing script.

3.3 Evaluation

When we had a tool that worked to our needs, we started the collection of empirical data. First, we reviewed the amount of mutants different mutation operators generated for all of the selected files. For each file, we selected one mutation operator and ran MiLu, and took notes on the amount of generated mutants. This was done for all the different mutation operators supported by MiLu. To get statistics about the impact mutation testing would have on a developer, we needed to check for how long time it took Swift to run tests on the different mutation operators and files. To get the statistics of the mutation score we ran the script for each test object and mutation operator. This allowed us to study the difference in mutation score of the various mutation operators for the different files.

We also needed to check the mutants that passed through the test suites to be able to evaluate the mutation score in consideration to equivalent mutants. By checking the difference between a surviving mutant and the original code, we should be able to draw conclusions about whether it could be a bug or not. We based our selection of mutants to present on real errors that passed the tests.

When we were done with the empirical data collection, we started to analyze the result. During the analysis we compared the mutation score with the Swift coverage score. We investigated those with a low mutation score, we checked the test code and the source code that we mutated. By checking this, we could get an idea of how good the tests really were. We also measured the amount of mutants different mutation operator generated and the time it took to execute tests on these mutants.

4 Result

Since we did split the work into three phases, we categorize the result into the same parts. This was made to make it easier to understand the connection between the method and the result.

4.1 Exploratory

One of the first papers found was a comprehensive survey of the mutation-testing field made by Yue Jia and Mark Harman (2011). This paper is a survey study of the field of mutation testing from its beginning in the 70's to 2009. This survey has more than 250 references, which made it easier to find papers more focused on our problem area. The most relevant papers are presented in the theory chapter (see chapter 2).

In the survey paper, they also listed the different existing tools created for mutation testing. From this list we looked at the ones created for ANSI C. From the google search, we found the same tools as listed in the survey article. The tool needed to meet some criteria to be able to work in the Ericsson context. These criteria were:

- It needs to work in a Linux environment, since it should be able to run at a Linux server.
- It needs to be Open source, since we should be able to modify it if needed.
- It needs to be able to run from a terminal, since we should be able to run it remotely.
- It must not have a graphical user interface, since we should be able to put it in an existing execution chain.
- It must have the ability to only generate mutants, we should not need to specify a test suite to make the tool generate mutants.

After reading about the different tools and trying to get an understanding of how they work we chose one of the tools. The tool that matched the criteria's best was MiLu, which is created by Yue Jia and Mark Harman (2008) as a part of Yue Jia's PhD thesis. We chose this tool because it was stated to have full ANSI C support and had the greatest support of different mutation operators among the different tools. It was also able to generate mutants without specifying a test suite and it is created for a Linux environment. MiLu is open source and can be downloaded from github[5].

MiLu is a C mutation testing tool made by Yue Jia and Mark Harman (2008). It utilizes a parser to parse the source code into abstract syntax nodes and constructs an abstract syntax tree. This makes MiLu generate mutants fast since it only needs to traverse the tree and mutate the node that matches the mutation operator. MiLu can handle the process from generating mutants to running test on each of the mutants. The user can specify which file to be mutated, the test suite for that file and the mutation operators to be used. The operators MiLu is able to handle are based on a subset of the mutation operators proposed by Agrawal et al. (1989) in the paper Design of mutant operators for the C programming language. These mutation operators are classified by four different characters starting with O, S, V and C. O stands for Operator, S for Statement, V for Variable and C for Constant. The three remaining are abbreviations or

[5] https://github.com/yuejia/Milu

mnemonics of the operator function. The list below shows short examples of the mutation operators supported by MiLu. The mnemonics and the explanations are from Agrawal et al. (1989), the examples are theirs but we have made some minor modifications.

- OAAN: Arithmetic Operator Mutation
  - a +b → a * b
- OBBN: Bitwise Operator Mutation
  - a & b → a| b
- OAAA: Arithmetic Operator Assignment
  - a+= b → a -= b
- OBBA: Bitwise Operator Assignment
  - a &= b → a|= b
- ORRN: Relational Operator Mutation
  - a < b → a <= b
- OLLN: Logical Operator Mutation
  - a && b → a || b
- OLNG: Logical Negation
  - (x<y&&p>q) → !(x<y)&&(p>q)
- OCNG: IF Expression Negation Replacement
  - IF((x<y&&p>q)) → IF(!(x<y)&&(p>q))
- CRCR: Required Constant Replacement
  - k = j + p → k = 1 + p
- OIDO: Operator Increment Decrement Mutation
  - ++x → --x
- SBRC: Break Replacement by Continue
  - break → continue
- SSDL: Statement Deletion
  - printf("Hello") → /* printf("Hello") */

These are only a subset of the total amount of C mutation operators proposed by (Agrawal et al., 1989). We decided only to use the mutant generation part of MiLu since it would not recognize the test files written for Swift. We do not utilize the compiler and testing parts of MiLu.

The result of the search for test objects gave us four files that had good Swift tests written for them and contained production code suitable for mutation. Since these files contain real production code we have renamed them. One of the biggest reasons we chose the files are for their coverage score. If we would have chosen files with lower coverage it would be harder to investigate if mutation testing could increase the quality of the test suite. By using a function built into Swift we could get the coverage score which is based on line coverage. Line coverage works like statements coverage; it checks which lines in the assembler code that have been executed. By running the program you get a list of the files under test, it shows the amount of lines in the program, and how many of these lines were executed by the tests. The lines that did not execute are also shown. We can see an example of this in table 1. The assembler lines not executed are matched to the lines of the production code, this to make it possible for the developer to locate it.

File	Number of lines	Number of Executed lines	Coverage score	Missing lines, line numbers
src/file_under_test_1.c	26	26	100%	
src/file_under_test_2.c	27	27	100%	
src/file_under_test_3.c	28	28	100%	
src/file_under_test_4.c	40	39	97%	87
TOTAL	121	120	99%	

Table 1 Coverage score from Swift.

This tells us that we have 100% line coverage with the exception of *file_under_test_4* where one line is not covered. To be specific line number 87 is not executed by the tests.

Table 2 shows some statistics about the files we chose. This is done to get some basic understanding of the files without revealing the content. When calculating the lines the comments in the beginning of the file were excluded. *Line number for mutation* is the line we tell the mutation script to run tests from (see section 4.2). *Number of lines in test file* denotes the number of lines in the file that contain the test code for a specific file.

File name	Number of lines	Number of lines after preprocessor	Line number for mutation	Number of lines in test file
file_under_test_1	77	5964	5884	153
file_under_test_2	97	5973	5884	159
file_under_test_3	86	5964	5883	298
file_under_test_4	118	6005	5885	298

Table 2 Statistics from chosen files.

The reason that *file_under_test_3* and *file_under_test_4* has the same number of lines for their test files is that they are using the same test file for their testing.

4.2 Implementation

MiLu needs the files to be preprocessed since it utilizes an abstract syntax tree and needs type information to be able to construct this tree (see section 3.2). During preprocessing this information is inserted in the file. Because of the syntax tree MiLu is able to generate mutants very fast. One problem we encountered was that when we started to test MiLu on the production code we noticed that the code we got from MiLu that was not part of the mutation was different from the source code. MiLu made some changes that it was not supposed too. We also noticed that with some source files we got segmentation faults from MiLu and some other yielded error messages.

The first problem was that statements lacked support for some types of internal statements. This made MiLu get to a state where it crashed. Some of these were anticipated controlled crashes; assertions were implemented in the program. We had to extend the support in expressions by adding the ability to handle initializer list expressions, declaration statements and null statements. In statements we had to extend the support for do while and unary operator. By adding these in *ASTPrinter.c* we got MiLu to work without crashing.

The second issue we encountered was that when comparing the source code MiLu generated from the production code the *do-while* statements were removed. So we looked into this and found that MiLu lacked support for do while statements, this is quite important since many macro use this. ANSI C macros are often written like *do-while* statements to be able to handle semicolons in a correct way. This is handled in the *ASTPrinter.c* file in MiLu. We added a fix for this by telling MiLu that if we got an abstract syntax node of the type *do-while* add the string “do {” followed by the statements inside the *do-while* and add “} while (0)” at the end to the mutated source files. This fixes the problem for files we use but will not work if there are real do while statements in the code. The result will look something like this:

```
do {
statement;
} while(0);
```

Another quite important feature that MiLu did not support was string concatenation. In the book The C programming language Kernighan & Ritchie (1988) states that you should be able to write strings like this "Hello" "World". MiLu only handled the first string and skipped the following so we did only get the "Hello" part. This was very important as many strings were automatically generated and inserted in the production code by Swift. By locating how MiLu handles the strings when reading strings from source files we were able to locate the problem. We found that the file *ASTUnit.c* in MiLu handled this. MiLu missed the strings because if it found a quote it kept reading until the next quote and then ignored the rest even if the next character was a quote. We changed this so that MiLu continued to read for quotes and all characters between quotes were added to a big string. This corrected the problem and made MiLu able to handle concatenating strings in the same way specified in ANSI C standard.

When running Swift on mutants we found a problem with constant declaration. MiLu changed the function input parameter in some cases. Let us look at the example below:

```
int * const p
const int * p
```

These two looks similar but they make a big difference. In the first line the pointer is constant and in the second line the value is constant. MiLu switched places on the const as shown in the example. This was only done for function definitions not for the function declarations. This gave us a compilation error when running Swift since the declaration and definition did not match. By changing how MiLu handled const we were able to fix this problem. One problem with this is that it is a limited solution and might not work for all cases.

Even if we made some improvements to MiLu there were remaining issues. We had to make some small changes to the production code otherwise MiLu would break it. One specific problem was with function pointers like in the example below.

```
variable = ( void (*) ( void*))0;
```

If we changed this to

```
variable = 0;
```

we got correct results from Swift and MiLu is able to handle it.

To be able to start with the bash script we needed to know how MiLu structured the generated mutants. By running a test with a production file we noticed that the mutated files are stored in subdirectories one for each mutant and they are named in a discrete fashion starting from one and for each mutants one is added. If MiLu would generate three mutants of *file_under_test_1.c* we would get a directory structure like this:

```
MiLu_output/
`-- mutants
   |-- 1
   |   `-- src
   |       `-- file_under_test_1.c
   |-- 2
   |   `-- src
   |       `-- file_under_test_1.c
   `-- 3
       `-- src
           `-- file_under_test_1.c
```

MiLu applies a syntactic normalization function on the production code based on information from the abstract syntax tree. This is from where MiLu generates mutants. It is appropriate to compare the mutants with the normalized version of the production code since the original and the normalized will differ. We made the decision to call the normalized version of the production

code a *zero mutant*. This zero mutant is the production code as MiLu interpreted it but without any mutations.

By comparing the zero mutant with the other mutants, we could get information about where in the file the mutation is located and what kind of mutation it is. We used a program called *Diff* which presents basic information about the difference between two files. In figure 6 on the next page we can see what the result would look like.

```
--- zero_mutant.i
+++/file_path/src/file_under_test_1.i
@@ -5902,3 +5902,3 @@
 Function_name((p -> Slices[aIdx ]),TimeSlice,((sizeof (timeSlice)
-+ 1) >> 1));
+- 1) >> 1));
 TimeSliceS        *    timeSlice_p    =    &    timeSlicePair.timeSlice[aIdx];
```

Figure 6 Printout from diff which tells the difference between the zero mutant and one of the mutants. This tells us that the mutation is at line 5902 and that the plus sign is replaced by a minus sign.

The preprocessor adds functions and statements to the production file which MiLu was able to mutate. This was a problem since the test suites are only designed to check for errors in the places were the original production code is located. To solve this we decided only to run Swift if the mutation was below a specified line in the code. If the mutation was located above the specified line, we would just exclude it from the testing and add it to the number of *excluded mutants*. This makes the mutation score more credible since the tests are not designed to find errors in the code above the specified line.

Given the number of mutants generated by MiLu, the amount of mutants the Swift tests are able to kill and the number of mutants we exclude from the testing, we can calculate a mutation score. To tell which mutants are still alive we keep a list of these. We can see an example of the printout below:

```
Mutants still alive:  41 48 54 55 57 58 63 64 75 76 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92
Mutants excluded: 31
Mutants killed: 52
Mutants total: 109
Mutant score: .66666666666666666666
```

The interesting part of mutation testing is to inspect surviving mutants as the mutants killed are found by the test suite. To make this process a little easier we also print the difference between the zero mutant and the mutated file. This will make it faster to check if the mutants are equivalent or not. It will also make it easier to find the correct line where the mutation exists since the line number is also printed. We can see the line numbering in figure 6 between the double “@” signs.

When running the script for different mutation operators we found some problems with some of them. One was the statement deletion (SSDL) which seemed to be broken since it did not generate any mutants at all. When trying to run this operator in a combination with others it broke the others as well and we ended up with no mutants at all. The other one was the required constant replacement (CRCR) mutation operator. It generated some mutants that were no different from the production file. The CRCR operators will replace the constant with one, zero and negative one. It will in addition to this increment and decrement the constant value by one. This will create mutants identical to the production code in some cases. For instance, a constant with the value one will be switched for the value one. Because of this we also added a check in the script to see that we had diffs between the zero mutant and the mutant. If no diff was detected, we skipped running Swift for that mutant. We counted it as an excluded mutant.

4.3 Evaluation

We chose to use three different combinations of mutation operators from the ones MiLu supports. Offutt et al. (1996) stated five operators that were better than others to use and MiLu has support for some of these but the naming is different. *Absolute value insertion* operator is not present, *arithmetic operator replacement* is called OAAN, *logical connector replacement* is called OLLN, *relational operator replacement* is called ORRN and *unary operator insertion* could be seen as OLNG and OCNG since they negate expressions. We called these operators *Big five*. MiLu is using three default operators if we do not specify any. These operators are OAAN, OAAA and ORRN. We called these operators *MiLu default*. The last combination is with all the mutation operators supported by MiLu, which we called *ALL*. MiLu will handle these combinations in the same way as if we would only use a single mutation operator. It will first generate mutants based on the first mutation operator and then move on to the second operator. MiLu will always base the mutations on the zero mutant and never on previously generated mutants.

4.3.1 Number of generated mutants

To get an idea of how different operators affect the number of generated mutants we have run MiLu on each file for each operator. In addition to the ordinary mutation operators, we have also run it on the three combinations we introduced earlier. The table below shows the numbers we got from each file and mutation operator. As long as the *files_under_test_#* does not change, we will get the same result every time we run MiLu for these files. There is no randomness in how MiLu generates mutants

Mutation Operator	file_under_test_1	file_under_test_2	file_under_test_3	file_under_test_4
OAAN	52	40	48	80
OBBN	2	3	2	0
OAAA	0	0	0	0
OBBA	0	0	0	0
ORRN	30	35	35	25
OLLN	0	1	0	0
OLNG	0	3	0	0
OCNG	5	5	5	4
CRCR	312	276	312	432
OIDO	3	6	3	0
SBRC	0	0	0	0
ALL	404	369	405	541
MiLu default	82	75	83	105
Big five	87	84	88	109

Table 3 Table of the number of mutants the different mutation operators generate.

4.3.2 Time Factor

The execution time is calculated by the program *Time*. We executed time with our script as argument with the "-p" flag.

```
/usr/bin/time -p ./mt_test.sh source_path filename line_number
```

The flag will make the result from time a little easier to interpret since it looks something like this:

Real 14.71
User 0.96
Sys 7.44

The *real* is the actual running time in seconds and the *user* and *sys* are for how long the program was in user versus kernel mode. In this thesis we only looked at the real value from the program time.

Table 4 shows the total running time in seconds for the mutation script to run from start to finish. The script generates mutants and runs Swift on each mutant. The time factor for the mutant generation is negligible in this context. This is the reason why we do not present statistics for each file and mutation operator. For reference we can look at the CRCR operator on *file_under_test_4* for which MiLu generated 432 mutants in under 15 seconds (real 14.71).

Operators	Source file	Running time (seconds)
All	file_under_test_1	1686.75
All	file_under_test_2	1639.22
All	file_under_test_3	1973.5
All	file_under_test_4	2315.77
MiLu default	file_under_test_1	324.62
MiLu default	file_under_test_2	259.79
Mliu default	file_under_test_3	371.94
MiLu default	file_under_test_4	503.21
Big five	file_under_test_1	398.97
Big five	file_under_test_2	350.75
Big five	file_under_test_3	409.33
Big five	file_under_test_4	598.94

Table 4 Table of running time for the mutation script evaluating different combinations of mutation operators.

4.3.3 Mutation score

We have the Mutation score to reflect excluded mutants like this:

$$mutation\ score = \frac{killed\ mutants}{total\ number\ of\ mutants - \ excluded\ mutants}$$

Table 5 shows each of the mutation operators that generated mutants based on the information from table 3. We skipped the ones that did not generate any mutants for the selected set of files since they would not generate any usable data.

Operators	Source file	Total Mutants	Excluded Mutants	Killed Mutants	Alive Mutants	Mutation Score
OAAN	file_under_test_1	52	20	12	20	0.3750
OAAN	file_under_test_2	40	20	9	11	0.4500
OAAN	file_under_test_3	48	20	9	19	0.3215
OAAN	file_under_test_4	80	20	39	21	0.6500
OBBN	file_under_test_1	2	0	2	0	1.0000
OBBN	file_under_test_2	3	0	2	1	0.6667
OBBN	file_under_test_3	2	0	2	0	1.0000
OBBN	file_under_test_4	0	NA	NA	NA	NA
ORRN	file_under_test_1	30	10	13	7	0.6500
ORRN	file_under_test_2	35	10	16	9	0.6400
ORRN	file_under_test_3	35	10	16	9	0.6400
ORRN	file_under_test_4	25	10	10	5	0.6666
OCNG	file_under_test_1	5	1	3	1	0.7500
OCNG	file_under_test_2	5	1	3	1	0.7500
OCNG	file_under_test_3	5	1	4	0	1.0000
OCNG	file_under_test_4	4	1	3	0	1.0000
CRCR	file_under_test_1	312	106	80	126	0.3883
CRCR	file_under_test_2	276	98	34	144	0.1910
CRCR	file_under_test_3	312	116	88	108	0.4490
CRCR	file_under_test_4	432	144	186	102	0.6458
OIDO	file_under_test_1	3	0	2	1	0.6667
OIDO	file_under_test_2	6	0	4	2	0.6667
OIDO	file_under_test_3	3	0	0	3	0.0000
OIDO	file_under_test_4	0	NA	NA	NA	NA

Table 5 Mutation statistics for each file and mutation operator.

Table 6 presents results from test runs with the three combinations of mutants. This gives us a better picture of how this would look like in a more realistic environment since the testing will probably be conducted with combinations of different operators.

Operators	Source file	Total Mutants	Excluded Mutants	Killed Mutants	Alive Mutants	Mutation Score
All	file_under_test_1	404	137	112	155	0.4195
All	file_under_test_2	369	129	72	168	0.3000
All	file_under_test_3	405	147	119	139	0.4612
All	file_under_test_4	541	175	238	128	0.6503
MiLu default	file_under_test_1	82	30	25	27	0.4808
MiLu default	file_under_test_2	75	30	25	20	0.5555
MiLu default	file_under_test_3	83	30	25	28	0.4717
MiLu default	file_under_test_4	105	30	49	26	0.6533
Big five	file_under_test_1	87	31	28	28	0.5000
Big five	file_under_test_2	84	31	32	21	0.6038
Big file	file_under_test_3	88	31	29	28	0.5088
Big five	file_under_test_4	109	31	52	26	0.6666

Table 6 Mutation statistics for each test subject for the three mutation operator combinations.

Table 5 and 6 give us a good indication on the mutation score for different operators.

4.3.4 Mutated code

We have selected some of the mutations to present here since the complete list for each file is too comprehensive to fit in the result section. We had to make this decision since we got many alive mutants from different mutation operators. The code snippets are the mutated code and in some cases with a couple of rows above and or below. The minus and plus signs indicate the mutation. The minus sign points out the original and the plus the mutated line. To make it easier to identify the mutation we have highlighted it with grey. We have anonymized the code since it could be possible to infer some hardware-specific function calls on the Ericsson proprietary hardware.

We will start by looking at some examples from the OAAN operators. This operator generates somewhere between 30 to 60 testable mutants. The operator switches an arithmetic operator for another arithmetic operator. In the first example of a surviving OAAN mutant the plus sign has been replaced with a minus sign:

```
Copy(&(tw->Slices[Idx]), &timeSlicePair, ((sizeof(timeSlicePair)
-+ 1 ) >> 1 ) ) ;
+- 1 ) >> 1 ) ) ;
```

Code example 1.

The above function call is common in all of the files we have reviewed. While examining the file generated from the mutation testing script we can also notice that many OAAN mutations made to this function are missed by the test cases.

Another example of common OAAN mutations that the test cases miss is an assignment. The following assignment is an example of common mutants that are missed by the tests.

```
U16 size = (U16) ((sizeof(TimerU)
-+ 1 ) >> 1 ) * timeSlice.nofAlloc;
+- 1 ) >> 1 ) * timeSlice.nofAlloc;
 TimerU * timers_p = malloc(size);
```

Code example 2.

OBBN operator switches the bitwise operator for other bitwise operators. The number of mutants is quite few only around two in each file. Only one mutation of this kind slipped through the test cases. In code example 3 below we can see that the AND bitwise operator has been switched for the OR bitwise operator.

```
-if ((((p->Obj_p)->Grp &(1UL)))) {
+if ((((p->Obj_p)->Grp |(1UL)))) {
 trace ();
```

Code example 3.

The ORRN operator is the third most productive considering the number of mutants generated. This operator generates around 20 testable mutants per file, and switches relational operators for other relational operators.

In the example below the greater than (>) operator has been switched for a less than (<) operator, which was not caught by the tests.

```
-if (timeSlice_p->nofActive > 0) {
+if (timeSlice_p->nofActive < 0) {
 Copy(…);
```

Code example 4.

In the example below we have a case where a less than has been replaced by a less than or equals operator.

```
-while ((iter  < end) && (iter->timer != timer->timer))
+while ((iter <= end) && (iter->timer != timer->timer))
```

Code example 5.

The OCNG operator is among the ones that generate few mutants and most of them are killed by the test cases. The OCNG operator negates an expression inside an *if-statement*. If the expression is true it will be false and vice versa. We have selected one of the two mutants not killed by the test cases since these two are almost identical. We can see that the expression is negated inside the if statement by the exclamation mark.

```
-if (  timeSlice_p->nofActive > 0) {
+if (!(timeSlice_p->nofActive > 0)){
 Copy(…);
```

Code example 6.

The CRCR operator is the one that generates the most mutants. It generates somewhere between 176 to 288 testable mutants for our files. This mutation operator replaces constants with other constants.

In the first example we have a one replaced by a zero. This will have the same effect as the OAAN which replaced the plus with a multiplication. This mutation and those similar often pass through the tests undetected. With similar we mean those where a one is replaced by a two and a negative one.

```
-Copy(&Pair, (void*) & (p->Slices_p[idx]), ((sizeof(Pair) +1 ) >> 1 ));
+Copy(&Pair, (void*) & (p->Slices_p[idx]), ((sizeof(Pair) +0 ) >> 1 ));
```

Code example 7.

In the third example we have an *if-statement* where the greater than zero has been changed to greater than negative one.

```
-if (timeSlice.nofAlloc > 0 ) {
+if (timeSlice.nofAlloc > -1) {
```

Code example 8.

The CRCR operator generates more than 100 mutants that the test suite misses and many of those are the same statements or similar statements. We will not show more examples since the examples would be a repetition of the same as the ones already shown.

The OIDO operator generates quite few mutants around three to six. The operator changes increment and decrement for both postfix and prefix.

The example in code example 9 is a for loop. In this case the pre increment has been switched for a post decrement.

```
for(U16 i = 0;
-i < timeSlice.nofActive ; ++i)
+i < timeSlice.nofActive ; i--)
```

Code example 9.

All of the mentioned examples are mutants from the production code that the test suites did not catch. By analyzing these code snippets we can get an idea of what kind of errors pass through the tests undetected.

5 Discussion

To be able to draw any conclusions about the result in this thesis we need to analyze our data. We will begin with discussing the results that stand out and continue with a discussion about our methodological decisions.

5.1 Result

Ericsson measures the quality of their test suites with coverage score as metric. A high coverage score should indicate fewer bugs than a lower coverage score. High code coverage in Swift should demonstrate high quality of the tested code. If we look at table 1 we can see that Swift gives us a result of 100% statement coverage in three of our four files, the fourth file has a coverage of 97%. The 97% means in this case that the test does not visit one line of code, to be specific line 87. If we look at table 2 we can get an idea of how extensive the tests are. The production code comprises 77 to 118 lines of code per file. The test files for these files range from 153 lines to 298 lines. However, we must take into account that *file_under_test_3* and *file_under_test_4* use the same test file for their tests. The ratio between the number of lines in the files and their test files are in the range from 1.5 to 1.9. We can clearly see that there are more lines of code in the test suites than in the production code. This tells us that to be able get a high coverage we need quite a lot of test code in comparison to the tested code. But what about the quality of the tests themselves, this is hard to know by only looking at the coverage score.

5.1.1 Generated mutants and time factor

By studying table 3, we get an idea of the difference between the mutation operators. The more mutants we generate the longer running time we should get. In addition it should also create more work for the individual developer which will take additional time. This is because if we generate more mutants, then more mutants have the chance to survive the tests. If we look at table 5 we can clearly see that with more mutants we get more surviving mutants. We can also see some tendencies in the mutation score that with more mutants we get a lower score. However, we cannot claim anything about this because we have not done any deeper studies of the cause of this. If we look at the live mutants in table 5 and take for example CRCR for *file_under_test_2* we get 144 mutants. If we assume that the developer needs three extra lines for each mutation to be able to understand the context of the code we would get 432 lines of code. It would take some time to analyze this amount of code.

If we look at table 4 we can see that the running time increases with the amount of mutants. If we start by looking at the statistics for all the mutation operators we have a running time of between 27 and 39 minutes. This in addition to the time it takes to analyze the surviving mutants might not be viable to do every time a test or a production file is updated. This is if we assume that all the surviving mutants need to be analyzed by a developer. This will probably not be the case since many alive mutants will be from the same expression. If a mutated expression slips through the tests, there is a high probability that another mutation from the same mutation operators also slips through. If we look at the Big five operator in table 4 we get running times of between six and ten minutes. In addition if we look at table 6 we can see that we only have between 21 and 28 alive mutants. This would give the developer somewhere between 63 to 84 lines of code to

check. This should not take that long for a developer to check. Since Offutt et al (1996) claim in their study that these operators are among the best ones to use we can assume that this will generate mutants worth investigating. We will not be able to state anything about the effect this would have for the individual developer since we have not made any studies regarding the time a developer spends on test development or how big the changes are between commits. We are able to tweak the execution time by changing the mutation operators to match specific errors the test suite should find.

5.1.2 Mutated code and tests

To get a picture of how the mutations could affect the program we will discuss all the code examples from the result section. All of the errors are based on the competent programmer hypothesis since they are from simple mutations done to the production code. We can then assume that they all are quite simple and developers could easily make these mistakes.

In code example 1, that switched a plus for a minus (OAAN), we get an error that is quite easy to make and could be hard to find. This mutation is problematic since it will make a copy function miss some data. It will shrink the size of the how much to copy with one instead of increasing it by one. If the plus one is important we could get some bugs that might be hard to find. If we look at code example 7 we have a similar error since it belongs to the same function. Instead of changing the plus for another operator, we have switched the plus one for a plus zero. This would also make the program miss some data in the copy process.

Code example 2 is an example of memory allocation code. The variable size is the size of memory that is going to be allocated. In this case we will get a smaller memory allocation than expected. Memory allocations are problematic since it is hard to test for it. One advantage on using Swift is that we could utilize other linux programs like Valgrind that can check memory allocations.

Code example 3 is an error that could be easy to make but hard to find. In this case the condition will instead of doing a bitwise AND comparison it will do a bitwise OR. This will probably make the condition true when it is supposed to be false. This will make the function to run when it is not supposed to. We have a similar error in code example 6 where an exclamation mark has been inserted so that the evaluated value inside the condition is reversed. Because of this the trace function will run when it is not supposed to.

Another interesting error, which is easy to make, is code example 4. In this error the *if-statement* will always evaluate to false since we check if an unsigned value is less than zero. Code example 8 is similar to this since it also evaluates to the same result but in this case to true. The most interesting aspect of code example 8 is that we have a comparison with one unsigned and one signed value. To be able to compare these we need a type conversion in this case the unsigned value is converted to a signed. We will then always compare something greater than or equal to zero to a value less than zero.

Another error that could introduce bugs is in code example 5. In this case we would probably run the loop one time too many since we added an equals to the less than operator. This could be a common error made by developers. Code example 9 is also interesting, in the loop the increment

has been switched for a decrement and will make the variable `i` flip from zero to the value of max int. This will make the loop run for one iteration instead of the number of times stated by the variable `nofActive`.

All of these errors simulate bugs in the system and if not detected in time they could be hard to find. We find that all of these errors except for memory allocation should be detectable with different tests. For instance in the different copy examples we would miss some data, we need to check that the copied value is equal to the original value. By adding tests that compares the original value with the copied value, we would easily kill all mutants that mutated this function. This could of course be hard in some cases. In the other cases we need to check that the correct functions are run when we expect them to since most errors are from predicate expressions in conditional statement evaluations. All of these errors are very small, even experienced developers could make them, and it should not be that hard to make tests for them. Another aspect is that if these small errors could slip through we might have bigger and more severe bugs in the code. We cannot tell if there are bugs in the production code, but if these mutants slip through the tests, there is a risk that there are bugs and errors in the production code.

We can see that with the use of mutation testing we are able to check if there are errors that the test suite does not catch. This is in line with what we discussed earlier in the theory chapter where Assylbekov et. al (2013) and Smith & Williams (2009) argue that mutation testing can help improve test quality. For the mutants that did slip through the test suite the developers can get an indication on which tests that needs to be added to the test suites.

5.2 Method

To be able to tell if the thesis has a good academic standard in relation to the used method we need to evaluate it from the three different criteria: reliability, validity and replicability. If we start by looking at the replicability someone at Ericsson should be able to do this study again and get the same result. Everything in the results are based on tools and files only available at Ericsson. In the case with reliability and validity we argue that all our empirical data is within the field of mutation testing and we have made these measurements as correct as possible. We have based the empirical data collection on techniques that are used in papers about mutation testing. Some of the tests are hard to get the exact same results from, like time to run an application since it is dependent on the current load on the machine. This should not have a big impact on the reliability on the results in general since nothing is time critical. It is also important to take into consideration that if the production code files are changed, we would get other mutation values and if the tests are modified, we could get other results.

The split into three parts were a good thing since it made it easier to keep everything separated. That we started with a quite thorough study of the different fields was a good thing since the implementation required it. We realized this when we started with the implementation, without the theory the implementation would have taken longer time and would probably been less efficient and effective.

An issue with this thesis is the small number of test objects. The files we have chosen to run our test on were chosen for their good Swift testability. These files have unit tests written for them,

which is required for running with Swift and they have very good coverage. This also makes the empirical part of this thesis narrow. Because of this we could get low generalizability but we can show that mutation testing is a good tool for increasing the quality of test suites. If this study would have been conducted at another company, with similar tools (Swift and MiLu) the results might have been the same since we have based our assumptions on related work.

5.2.1 Source criticism

All the sources in this thesis have been carefully evaluated with respect to their credibility. Source credibility was established by controlling the authors and number of citations. In the case of the authors we have checked their field of expertise and earlier publications. However, the number of citations has been difficult to take in consideration for newly published (2012 - 2014) sources. The majority of the sources for this thesis are first hand scientific papers. A couple of the sources are younger than five years and some are old and well-known papers.

6 Conclusion and future work

After our deeper analysis of our empirical data with related work in mind, we have come to some conclusions related to the purpose of this thesis.

We addressed the first research question, *is mutation testing viable in the Ericsson context of baseband development*, by discussing the time aspects and number of surviving mutants. We can see that by using a limited number of mutation operators we get a time frame of below 10 minutes for mutant generation and testing. From the files we have tested we got few surviving mutants. This makes mutation testing a viable method for Ericsson's baseband development.

We addressed the second research question, *is common code coverage a good measure of code quality*, by discussing the coverage score in comparison to mutation score and surviving mutants. We could clearly see that even if the coverage score was 100% some errors slipped through the test suite. This indicates that even with a perfect coverage score there could exist bugs and errors in the code. This makes the coverage metric somewhat misleading since it does not give a good estimate of the quality of the test suite. Papers support this as they show that coverage metrics alone is not a good indication of test quality.

We addressed the final research question, *could mutation testing increase the quality of the test suites*, by discussing the surviving mutants. If we would add tests to our test suite that would detect the live mutants we would keep our coverage score and also kill these mutants. We claim that this would increase the quality of the test suite since it will detect more errors and possible bugs. This will of course require more time from the developers. Since the later in the production chain a bug is found the higher the cost, we think that that the time spent is worth it in the end.

As future work, one of the most interesting parts to continue with is to make a new version of MiLu that is able to handle the code in a better way than it does at the time being. We found some of the problems but we are sure that there are more. Two of the problems we corrected are not handled in a correct way. If we would add support for the special extensions from DSP-C and Ericsson we would get a powerful tool that would be useful in the quality verification of test suites. In addition to this it would be interesting to study the impact this tool would have on the single developer, if it would increase the workload or make it easier to develop high quality test suites. To make MiLu work with the DSP-C extension we would need a modified version of LLVM that supports the types introduced in DSP-C. In addition we would need to modify MiLu so that it is able to handle these types in a correct way.

To minimize the number of mutants even more we could add another line limit in addition to the existing one to the mutation script. This would make the testing only focus on a small part of the code. If we have done some small changes to the test suite for detecting some mutants this would speed up the testing.

An addition to the testing of mutants we could add some programs to check for syntactical errors or warnings. In the case where we have a comparison between unsigned and signed values we would be able to generate a warning. This is already done on the code in later stages at Ericsson. If we would kill or mark these mutants we would get even fewer surviving mutants.

7 References

Agrawal, H., Demillo, R. A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E. W., et al. (1989). *Design of mutant operators for the C programming language*.

Ammann, P. & Offutt, J. (2008). *Introduction to software testing*. Cambridge: Cambridge University Press.

Assylbekov, B., Gaspar, E., Uddin, N., & Egan, P. (2013). Investigating the correlation between mutation score and coverage score. *15th International Conference on Computer Modelling and Simulation,* pp. 347-352.

Beizer, B. (1990). *Software testing techniques*. (2. ed.) New York: International Thomson Computer Press.

Budd, T., & Angluin, D. (1982). Two notions of correctness and their relation to testing. *Acta Informatica, 18*(1), 31-45.

Copeland, L. (2004). *A practitioner's guide to software test design [Elektronisk resurs]*. Boston, Mass.: Artech House.

DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer, 11*(4), 34-41.

Hamill, P. (2009). *Unit Test Frameworks [Elektronisk resurs] : Tools for High-Quality Software Development.* Sebastopol: O'Reilly Media, Inc.

IEEE standard for software and system test documentation.(2008). *IEEE Std 829-2008,* , 1-150.

Inozemtseva, L., & Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. *In Proceedings of the International Conference on Software Engineering* (ICSE), pp. page To appear.

Kernighan, B.W. & Ritchie, D.M. (1988). *The C programming language*. (2. ed.) Englewood Cliffs: Prentice Hall.

Myers, G.J., Badgett, T., Thomas, T.M. & Sandler, C. (2004). *The art of software testing [Elektronisk resurs]*. (2nd ed.) Hoboken, N.J.: John Wiley & Sons.

Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., & Zapf, C. (1996). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology, 5*(2), 99-118.

Smith, B. H., & Williams, L. (2009). Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software, 82*(11), 1819-1832.

Yue Jia, & Harman, M. (2008). MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference,* pp. 94-98.

Yue Jia, & Harman, M. (2011). An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on, 37*(5), 649-678.

Appendix A

```
#!/bin/sh
#Author Niclas Norman
#
######Run tests on all mutants################
#Run with arguments
#First argument:
# path the the source file to be tested
#Second argument:
# name of the file to be tested
#Third argument:
# From which line to test mutants
##############################################


######Functions#################################
function pause(){
   read -p "$*"
}
##############################################


######Variables#################################
nr_of_mutants=0
killed_mutants=0
not_killed_mutants=
mutant_score=0
mutated_line_nr=0
excluded_mutants=0
date=`date`

milu_path=`pwd`
source_path=$1
filename=$2
lower_line_nr_exclude_limit=$3
output_file_name="mutation_result_"$filename".txt"
zero_mutant_name="zero_mutant.i"
##############################################


######Create mutants by running Milu###########
#pause "Press ENTER to start generating mutants"
`env LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/app/clang/3.4/LMWP3/lib/ ./milu --
mut-operators=operators.op -v --debug=src  $source_path/$filename >
$zero_mutant_name`

nr_of_mutants=`find . -type d | grep src | wc -l`
##############################################


#####Pause and show generated mutants###########
echo $nr_of_mutants Mutants generated
#pause "Press ENTER to start running test on all the mutants"
```

```
################################################


#####Backup the original file###################
`cp $source_path/$filename $source_path/$filename.bak`
################################################


#####Run swift for every mutant and check if it is killed#####
for i in `seq 1 $nr_of_mutants`;
do
    echo Mutant under test: $i
    `cp $milu_path/milu_output/mutants/$i/src/$filename $source_path/`

    #Check the line number of the mutant
    cd $milu_path
    mutated_line_nr=`diff -u0 $zero_mutant_name
$milu_path/milu_output/mutants/$i/src/$filename | grep @@ | egrep -o '[0-9]+'
| head -n 1`

    #Check that we got return value from diff
    re='^[0-9]+$'
    if ! [[ $mutated_line_nr =~ $re ]] ; then
    mutated_line_nr=0
    fi


    #Only run the test if the line number is equal or greater than the
specified
    echo Mutated Line Number: $mutated_line_nr
    echo lower Exclude limit: $lower_line_nr_exclude_limit

    if [ $mutated_line_nr -ge $lower_line_nr_exclude_limit ]; then

    cd $source_path
    swift


    if [ $? = 0 ]; then
            not_killed_mutants="$not_killed_mutants $i"
    else
            ((killed_mutants++))
        echo mutant killed
    fi
    else
    ((excluded_mutants++))
    echo mutant skipped
    fi
done
################################################################


#####Restore the original file#################################
`cp $source_path/$filename.bak $source_path/$filename`
`rm $source_path/$filename.bak`
################################################################
cd $milu_path
```

```
######Generate test output file##############################
mutant_score=`echo "$killed_mutants / ($nr_of_mutants - $excluded_mutants)" |
bc -l`
`printf "%s %s %s \n" "#### Time and date of test:" "$date"  "####" >
$output_file_name`
`printf "%s \n" "#### Mutant number is based on the directory name ####" >>
$output_file_name`
`printf "%s\n" "#### The mutated code are the line below the --- #### " >>
$output_file_name`
`printf "%s %s %s\n" "Name of the mutated file: --->" "$filename" "<---"  >>
$output_file_name`
`printf "\n%s %s %s\n" "##########"  "Mutation statistics" "##########" >>
$output_file_name`
`printf "Mutants still alive: %s\n"  "$not_killed_mutants"  >>
$output_file_name`
`printf "Mutants excluded: %s\n" "$excluded_mutants" >> $output_file_name`
`printf "Mutants killed: %s\n" "$killed_mutants" >> $output_file_name`
`printf "Mutants total: %s\n" "$nr_of_mutants" >> $output_file_name`
`printf "Mutant score: %s\n"  "$mutant_score"  >> $output_file_name`
`printf "\n\n%s %s %s\n\n"  "##########" "Diffs for the mutants still alive"
"##########"  >> $output_file_name`

IFS=', ' read -a array <<< "$not_killed_mutants"
for index in "${!array[@]}"
do

    #echo "$index ${array[index]}"
    `printf "##### Mutant number %s #####\n" "${array[index]}" >>
$output_file_name`
    `diff -u1 $zero_mutant_name
$milu_path/milu_output/mutants/${array[index]}/src/$filename >>
$output_file_name`
    `printf "\n%s\n\n" "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@" >>
$output_file_name`
done



############################################################


#####Print information about mutants#########################
echo Mutants still alive:  $not_killed_mutants
echo Mutants excluded: $excluded_mutants
echo Mutants killed: $killed_mutants
echo Mutants total: $nr_of_mutants
#mutant_score=`echo "$killed_mutants / ($nr_of_mutants - $excluded_mutants)"
| bc -l`
echo Mutant score: $mutant_score
############################################################
```