Juhan Heamets

Fault Seeding Experiment to Evaluate Effectiveness of System Tests

Metropolia University of Applied Sciences

Bachelor of Engineering

Electrical and Automation Engineering

Thesis

1 January 2020

Metropolia

Author Title Number of Pages Date	Juhan Heamets Fault Seeding Experiment to Evaluate Effectiveness of System Tests 40 pages + 1 appendices 1 January 2020
Degree	Bachelor of Engineering
Degree Programme	Electrical and Automation Engineering
Professional Major	Automation Technology
Instructors	Pekka Alho, Software QA team manager Timo Kasurinen, Senior Lecturer

In this thesis the effectiveness of ABB's system tests was evaluated using fault seeding experiment. Purpose for this thesis was to show how effective ABB's system tests are for finding type of defects for that they are designed for. Work was carried out at the drive software testing team where systems tests are used to evaluate drives behavior. One of the goals of ABB's software testing is to minimize the possibilities of defects reaching the customer, so that end user could receive bug free product.

Thesis introduced testing levels and approaches used in the drive software testing, as well as how effectiveness of testing could be measured in the different testing levels. Work also introduced different types of faults used in the mutation testing.

In the thesis, faults were injected to the source code manually by altering the drive source code using different fault seeding methods. System tests were run with modified drive software and results analyzed to find out does the test cases find the defects in the drive source code as expected.

Results showed that in most cases ABB's system tests were capable of detecting defects as expected. Work brought up locations and types of defects not detected by the system testing and reasons for them. Thesis also gave an overview of systematic mutation testing process.

Keywords	mutation testing, fault seeding, test automation

Tekijä Otsikko Sivumäärä Aika	Juhan Heamets Järjestelmätestien tehokkuuden mittaus mutaatio testauksella 40 sivua + 1 liitettä 1.1.2020
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Sähkö- ja automaatiotekniikka
Ammatillinen pääaine	Automaatiotekniikka
Ohjaajat	Pekka Alho, Software QA tiimin esimies Timo Kasurinen, Lehtori

Insinöörityössä arvioitiin ABB:n järjestelmätestien tehokuutta syöttämällä virheitä taajuusmuuttajan lähdekoodiin. Työn tavoite oli arvioida miten tehokkaat ABB:n järjestelmätestit ovat löytämään tyyppisiä virheitä minkä varten ne ovat suunniteltu. Työ tehtiin taajuusmuuttajan ohjelman testaus tiimille, missä järjestelmätestejä käytetään taajuusmuuttajan ohjelman testaamisessa. ABB:n ohjelmatestauksen yksi tavoitteista on minimoida mahdollisuuksia, missä ohjelmakoodissa oleva virhe joutuisi asiakkaan asti, jotta asiakas saisi aina hyvälaatuisen tuotteen.

Insinöörityö esittelee ABB:llä käytettäviä testaus tasoja, tapoja ja miten niiden tehokuutta voitaisi arvioida. Työ myöskin esittelee erilaisia virheen malleja mutaation testauksessa.

Insinöörityössä virheet tehtiin lähdekoodiin manuaalisesti muokkaamalla taajuusmuuttajan ohjelma, käyttämällä erilaisia virheen syöttö menetelmiä. Järjestelmätestit ajettiin muokatulla taajuusmuuttajaan ohjelmalla ja tulokset analysoitiin. Tuloksista saatiin selville, löytävätkö testit virheitä odotetusti lähdekoodista.

Tulokset osoittivat, että useimmissa tapauksissa järjestelmätestit löysivät virheitä odotetusti taajuusmuuttajan ohjelmasta. Työ kertoo tarkemmin havaitsematta jäänyt virheistä lähdekoodissa ja myöskin syistä miksi ne jäivät havaitsematta järjestelmätesteiltä. Työ antoi myöskin yleiskuvan mutaatio testaus prosessista ja sen eri vaiheista.

Avainsanat	mutaatio testaus, virheen syöttö, testausautomaatio

Metropolia

Contents

List of Abbreviations

ATF	ABB automated testing framework. Testing framework that ABB uses to test drives firmware.
AC	Alternating current
DC	Direct current
IGBT	Insulated gate bipolar transistors
PID	Proportional-integral-derivative controller. A control loop mechanism employing feedback.
DI	Digital input
RO	Relay output

1 Introduction

The topic of the thesis was to evaluate effectiveness of system tests, which are testing drive firmware by doing fault seeding experiment. Thesis was done at ABB. ABB is a global technology firm with more than 147 000 employees in more than 100 countries and a history of innovation spanning more than 130 years. ABB is a global technology leader in electrification, industrial automation, motion, robotics and discrete automation technologies. ABB is a technology firm that is driving the digital transformation of industries. [1.]

Work was carried out at software testing team, which main priority is to test the low voltage alternative current (AC) drives using ABB's automated testing framework (ATF). In addition to that, team also tests drive embedded software manually, as well as creates and maintains ATF test cases and test setups.

Goal for this thesis was to test the effectiveness of current ABB ATF test cases. Test cases were evaluated using fault seeding experiment. Work evaluates how capable ATF tests are for finding type of defects to which they are designed for. ABB does not have at the moment a tool available to measure the effectiveness of ATF tests. ABB's test effectiveness assessment for now is based mainly on feedback from the customers. The less customers report the bugs the more effective testing framework is, as an assumption is that testing will catch bugs which do not reach the customer. Also in the market there are not available tools that could be plugged into the ABB's testing framework to measure the effectiveness of ATF tests.

At the beginning, the thesis looks into different methods, tools and metrics which could be used to asses test effectiveness, and also could potentially be used to inject faults to the software. Based on results, types of faults are seeded to the software which test cases should be capable of detecting using most suitable fault seeding techniques. Thesis shows the effectiveness of ATF test cases, as well as how effective and efficient fault seeding method is for measuring test effectiveness in the ABB's automated testing framework.

Thesis concentrates of evaluating the effectiveness of ATF test cases which are testing software of general purpose family drives. In addition to that, ATF is going through the

update process where physical setups are re-designed and tests updated, so that they will work with new setups. As updating of individual tests is still on going, not all of them are runnable. Therefore, test cases which effectiveness were evaluated were limited to these which are verified to be working with the new ATF setups.

2 Drive

2.1 Drive Benefits

Drive is a device that is used to control electric motor speed and torque by varying voltage and frequency supplied to the motor. When electric motor is controlled by the drive, the motor speed is directly related to the frequency supplied by the drive. Without drive electric motor would always run at full speed without dependency of load. Drive allows to adjust electric motor speed so that motor is providing only the power that process requires and by doing that it saves large amount of energy. [2; 3.]

Processes which do not require for motor to run at constant speed at all times benefit tremendously of usage of drive. In that kind of processes drive allows to change the speed of electric motor at real time to match the requirements of the process and by doing that it will reduce energy consumption and energy cost. In 2014 more than 65% of industries electrical energy is used to power electric motors, and facilities can reduce energy consumption up to 70% by using drives to control electric motors. [2.] For example, ABB drives saved in 2011 about 310 million MWh of energy worldwide, which is equivalent to the electricity produced by around 40 Loviisa nuclear reactors [3.]

Drive consists of three main parts as shown in figure 1, the input AC to direct current (DC) converter section, the intermediate DC bus and the output inverter section. AC drive takes in the fixed frequency AC voltage and converts it to the variable frequency voltage AC power. [4; 5.]

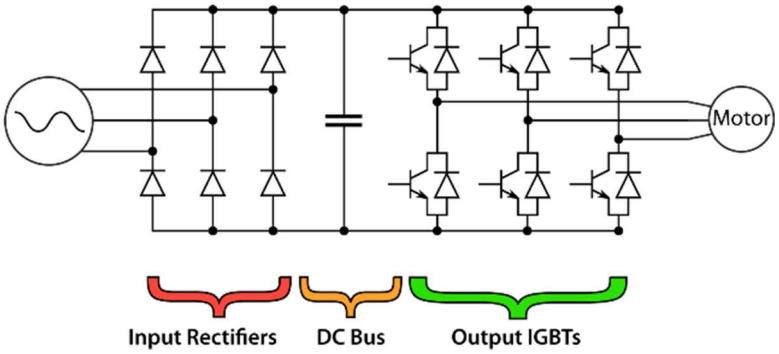

Figure 1 Drive made up of three main parts [4.]

2.2 ABB Low Voltage General Purpose Drive

ABB's general purpose drives shown in figure 2 are for typical light industry applications. Drive is built to control applications like mixers, pumps, conveyers, fans and many other constant and variable torque applications. General purpose drives can be found at food and beverage, plastics and rubber, as well as compressor and blower's industries. For example, Cherry Valley Farms in the UK, world leading breeder of pekin ducks, is using general purpose drives to control air compressors and water pumps in their factories. [6.]

Drive software allows to change hundreds of drive parameters through control panel or tool to fulfill different types of process requirements. It has built-in functions to control different types of electric motors in scalar and vector modes. Drive has functions to monitor, analyze and optimize how drive is operating, as well as built-in process PID controller. General purpose drive also has communication, I/O connection, relay output and auxiliary extension modules available for mounting, which give additional functionalities to the drive [7.]

Figure 2. ACS480 general purpose drive [8.]

3 Software Testing

3.1 Reasons to Test Software

Software testing is a process of evaluating product software to get information about the quality of the software, completeness and correctness. Tests examine different parts of the product to find out whether the developed program has met requirements set to it, and to find defects in the software of the product. Testing can be also used to distinguish errors, missing pieces in existing requirements, as well as to find missing requirements for the product.

Software testing can be divided into two parts, verification and validation. Goal for verification is to ensure that product compiles with requirements, regulations, specifications and conditions set to the product. Validation ensures that product meets the requirements of customers and other stakeholders. [9, 6.]

3.2 Defect and Failure

Defect is a fault or a bug in the software which can cause failure. Failure is a behavior where product behaves differently than it should. Testing does not guarantee that product is free of defects. No matter how much testing is done, there will surely be some defects left. In the dynamic testing like ATF, tests are detecting defects through failures in the product. If software does not proceed to the point where defect exists, it will not cause the failure and therefore defect stays hidden.

Failure in the software is usually caused because of human error in the coding which results bug in the product. There can be many reasons for the human errors like time or budget restrictions, lack of experience or knowledge, unclear requirements for the product or complexity of code infrastructure.

Environmental change can also bring up failures in the embedded software. Phenomena like magnetic and electric field can cause failures that stayed hidden in the normal testing environment. Also running software with different hardware can bring up unexpected behaviors.

3.3 Principles of Software Testing

Software testing allows to show that there are defects in the software, but it cannot prove that all the defects in the software are found and fixed. Testing helps to minimize the number of defects in the software and reduces the risk of failures in the product when using it. Not finding defects in the testing proves more that there are flaws in the testing process, rather than that the quality of the product is extremely good.

In the perfect testing, all the possibilities and features in the product are tested thoroughly in every possible environment, but this is almost impossible in a real world. Test planning requires careful planning so that testing could be limited to a reasonable amount, without compromising the test coverage, quality and reliability. [9, 21.]

In the software testing, defects tend to cluster in certain locations. Studies have showed that typically 20% of whole software contains 80% of all software defects. This usually happens because of the product logic complexity in certain areas. Complex logic raises the chance to make mistakes when creating requirements and writing code. [9, 23.]

Through the lifecycle of any product, there are usually constantly changes made to the software to improve the product and to fix defects. To keep the quality of testing up, it is important to review, update and improve tests after the change in the product, so that they still find defects that they supposed to. This can also help to find areas in the product which are not covered by testing.

Testing a product can greatly differ depending on the product structure and usage. Different testing methods and approaches will benefit various amounts depending on a product. When designing testing process for the product it is important to know the requirements for the product, as well as environment where it will be used. Thoroughly designed testing process can help to predict where most of the defects could be found and therefore how much effort should be spent in different parts of the product.

3.4 Importance of Software Testing

Testing in any project can feel like a burden at first, as it can require big part of projects human resources and budget, but in a long run it has positive impact especially when

testing is started at the early stages. Usually finding and fixing bugs in the earlier stages of the project costs much less compared to later part of the process. Starting testing late can lead to situations where program is built on top of the faulty program which can lead to situation where costly modifications are needed to fix the product.

Lack of testing or too rushed testing in a project can lead to the product with bad quality and therefore cost a company a lot of customers and revenue. In an open market with a lot of products to choose from, quality of the product is very important. It does not take a lot from the customer to choose the product from another manufacturer, because of lack of quality in the product. Product in the market with defects can lead to situation where it is recalled from the market to be fixed and because of that company could lose revenue on sales. For example, in 2014 car manufacturer Nissan had to recall over a million cars due to software failure in the airbag sensory detectors [10.]

Software testing is especially important for products which can harm customer physically in case of faulty behavior. Defects in the software can be extremely costly and harmful in high power electrical devices. There is always some risk of getting hurt when using any electric device and adding defect to the software will increase that chance. For example, between 1985-1987 radiation therapy patients got overdose of radiation from the Therac-25 radiation-treatment device, because of defects in the device software. Patients received as much as 100 times more radiation than intended and it was cause of death for at least three patients. [11.]

3.5 Test Approaches

3.5.1 Black-Box Testing

Black-box testing is a testing technique where products external behavior is tested without taking a look into the code structure, how it is built and designed. Test cases are based of product documents which describe system specifications and requirements. Test cases are checking how product under test behaves in specific conditions and compares behavior to expected behavior gathered from the documents. In black-box testing test coverage measurement is based on different test objectives identified from the test basis and how thoroughly these objects have been tested. Black-box testing uses techniques like equivalence partitioning, boundary value analysis, two- and three-value technique, decision table testing, state transition tables and used case testing. [9, 7-9.]

One of the advantages of the black-box testing is that it does not require programming skills from a tester to test the product. By knowing how product under test should act in a certain situation, tester can verify manually does the product behave as it should. For example, when pressing start button, does the product start.

Using only black-box testing in the project can also have some negative effects. As black-box testing does not analyze product's code structure, there can be situations where some parts of the product are left untested. This can happen for example because some product features are described unclearly in the documents. It can also lead to situations where unnecessary test cases are created. Studying the code structure of the product could bring up situations where some parts of the product could have been tested with less amount of test cases. [9, 9.]

3.5.2 White-Box Testing

White-box testing is testing based on the knowledge of how system is designed and built. Test cases are designed based of a code structure to cover the whole program as thoroughly as possible. White-box testing includes testing techniques like statement, decision, condition and path testing. [9, 28.]

White-box testing techniques help to verify that when the input command is given, the command flows through code in correct path and gives the expected output. It also helps to find situations where some specific input causes command to flow through unexpected path, which can cause for example security holes in a product.

While white-box testing can assure that all the possible paths in the product software are covered by testing, it also can be very time consuming, complex and expensive. Complex product source code could have large number of opinions for commands to flow through and covering them all can take a lot of time and effort. White-box testing also requires from testers to have good programming skills and detailed overview of the products code structure.

3.6 Test Levels

3.6.1 Test Levels in Testing

Test levels refer to stack of test activities which are similar to each other, in sense that goal is to find same kind of flaws and they are monitored and managed together. [9, 11.]

In the software testing, software evaluation process is divided into four main testing levels unit, integration, system and acceptance testing level. Every testing level has its own set of objectives, goals and ways how product's compliance is checked. How much time is spent in different levels and how many levels are used in a testing project can vary greatly depending on the complexity and size of the product under test.

3.6.2 Unit Test

Unit testing is the first level of the testing process and this is done during the development of the product. In most cases unit testing is done by developer. Testing level is focused on testing the smallest individual parts of the software, like module and components to show that specific module works as expected and fulfill requirements. [12.]

Main benefit of unit testing is that it can be started at the same time as products development. It allows to find coding errors done by developers at the early stages and this diminish chance of creating errors on top of the other errors. For example, as illustrated below in figure 3 simple calculation function called Add is created to take in two values and to return sum of these values. Unit test illustrated in figure 4 verifies that logic behind function is correct and can be used to prevent coding errors like illustrated in figure 5 where mistakenly values are subtracted.

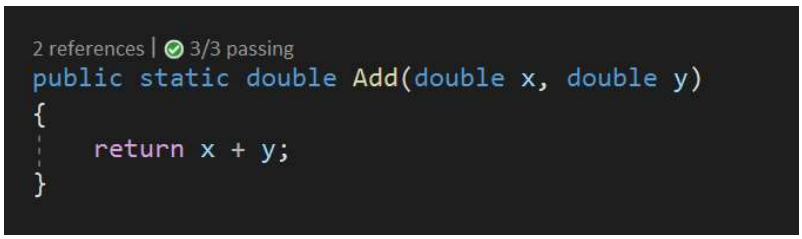

Figure 3. Method to sum values.

```
[Theory]
[InlineData(4,3,7)]
[InlineData(21, 5.25, 26.25)]
[InlineData(double.MaxValue, 5, double.MaxValue)]
0 references
public void Add_SimpleValuesShouldCalculate(double x, double y, double expected)
{

    // Act
    double actual = Calculator.Add(x, y);

    // Assert
    Assert.Equal(expected, actual);
}
```

Figure 4. Unit test.

```
2 references | 1/3 passing
public static double Add(double x, double y)
{
    return x - y;
}
```

Figure 5. Method with mistake.

3.6.3 Integration Test

Integration testing follows the unit testing in the testing process and focuses on testing the interfaces between different parts of the software. Testing verifies that different parts of the software work together as expected.

Integration testing can be divided into two parts, component and system testing. Component testing focuses on testing the communication between internal components, while system testing focuses on testing the functionality of the software and its interfacing systems. Component testing is usually done by developers simultaneously with unit testing and system testing is done by testers. [9, 15.]

Most used test strategies in the integration testing are Bottom-Up, Top-Down and Big-Bang. Bottom-Up strategy starts testing components communication between each other at the lowest level and moves up till the interfaces between the highest parts are tested. Top-Down testing is done as Bottom-Up testing, but only in reverse order. In the Big-Bang strategy, the interaction between parts are tested after the entire system is put together. [9, 17.] Finding defects in the Big-Bang strategy can be complicated, because there can be a lot of interfaces which can cause faults. Finding defect source in the Big-Bang can be as difficult as finding a needle in the haystack.

3.6.4 System Test

System testing is part of black-box testing, where external behavior of the product is evaluated. System testing is used to test the functionality of the entire system and testing is usually carried out in the production-like environment. In this stage of the process product is basically complete and testing verifies that product as a whole is ready to use and meets the requirements. [9, 19.]

System testing verifies that all the functionalities exist and work as expected, as well as that hardware and software work together as expected. When input command is given to the product, the system testing verifies that product behaves as expected.

System testing ensure that product can handle stress of real-life environment and is capable of recouping in unexpected situations. Half-done system testing can lead to situations where product cannot handle stress caused by real environment. For example, in the opening of the Heathrow Terminal five, the baggage handling system failed to work properly which caused around 42 000 bags not to travel with owners and over 500 flights were canceled over the span of the 10 days. Failure was caused because testing process did not cover some real-life scenarios for the system which resulted in unexpected behavior in the system in real environment. [13.]

One of the most used types of system tests is regression testing. In the regression testing tests are re-executed after the change. Regression testing ensures that changes made in the source code have not affected other parts of the product unexpectedly. It ensures that product works after the change as expected, as well as that changes have not created new defects.

3.6.5 Acceptance Test

Acceptance testing is usually last level of the testing process. Goal of the acceptance testing is not to find defects, but to verify that product is ready to use and usable. Testing is done from the customer point of view without taking a look to the software structure. Acceptance testing is usually tested by future users of the product. [9,22-26.]

Most used acceptance tests are user acceptance, operational acceptance, contractual and regulatory acceptance tests, as well as alpha and beta tests. User acceptance

testing is done by likely future end users and it verifies that product is usable in intended environment and accommodates requirements set do it. Operational acceptance testing verifies that product maintains the functionality when something unanticipated happens and this is usually done by system administrators. In alpha and beta testing, product is tested first in a manufacture environment and later in the end-user environment, to get feedback from potential users about the product before it is released to the market. [9,24]

3.7 Automated Testing

Automated testing is a process where product under test is tested by running automated tests to find defects in the system. In the automated testing, usually testing system automatically runs, analyzes test results and reports results to the tester who verifies that failed tests are caused by actual defects.

Creation of automated testing framework can take a lot time and effort at first, but in a long run it reduces resources needed for testing and gives more accurate tests results. In most cases running automated tests does not require human presence, therefore tests can be run for example at nights which reduces testing time, as well as testers needed for the project. Automated tests allow to evaluate product with same data and environment. This ensures reliable constant results from the tests. One of the big benefits of test automation is reusability. It allows for example to easily verify that bug is fixed correctly by running same tests again. It also allows to re-use entire testing system in similar future projects and this can reduce budget and effort needed for the testing in the future projects. In most cases creation of tests and test environment is most time consuming and expensive part of test automation.

One of the burdens in test automation is maintainability. While automating testing can reduce time and effort needed for testing, keeping the testing environment up to date can be very complex and fragile process. Automating testing needs constant maintenance to ensure that tests give reliable results. Not keeping them up to date can lead to situations where tests are failing. Finding reasons for failures can be very complex and time-consuming process. Not maintained testing process can lead to situations where tests which should not be passing are actually passing and defects are left unnoticed.

4 Test Effectiveness

4.1 Reasons to Evaluate Effectiveness of Tests

For testing process, knowing how effective individual tests are is as important as test coverage measurement. While test coverage shows how well product is covered by tests, the test effectiveness shows the quality of tests to find defects. Both are equally important to guarantee high quality in the testing, as well as to make sure that testing is efficient based on how much time and resources are spent to it.

Effective test is a test case that brings up defects from the product to which it is designed for. It is important to constantly evaluate test effectiveness to make sure that tests are capable of finding expected defects. Proving that they are effective can be complex and time-consuming process. As product is improved and changed through its lifecycle, it is important to keep the quality of testing up by updating test cases and verifying that they are working after the change in the product's source code. For every test, as important as quality of finding defects is also that tests describe faults clearly and this is especially important in test automation. Test case should give a clear explanation for the failure. Not keeping tests up to date can cause situations where tests are passing there they should not. This can create false promise of product's quality and test's effectiveness.

4.2 Effectiveness Based on Test Coverage

4.2.1 Test Coverage

Test coverage is important part of the software testing, as it displays quality, progress and effectiveness of tests. Test coverage is a technique which is used to determine whether the test system covers the whole product and how much code of the product source code is exercised when tests are executed. Test coverage helps to find flaws in the requirements and tests, as well as detect defects at the early stages of the test process, which reduces risk of building code on top of the defects. Test coverage can also be used for prioritizing the testing process. [13.]

While test coverage measurement does not give straight answer about the effectiveness of individual test cases, overall effectiveness of testing could be assessed using test

coverage measurement results. Test coverage metrics can be divided into three parts code-level, feature and application level metrics. Metrics can be used to ensure that all the functionalities, features and lines of code are covered by tests, and therefore ensure that testing process is capable of finding all the expected defects. [18.]

4.2.2 Code-Level Metric

Code level metric can be used to get an overview of how much of the product code structure is covered by tests. To figure out how many lines of code is executed, white-box testing techniques like statement, path and decision coverage are used. Based on results code coverage can be calculated using equation 1 below. This makes sure that all the lines are covered by tests, as well as that there is no dead code in the product source code. [13.]

$$Percentage\ of\ lines\ covered = \frac{\text{Number of lines executed by tests}}{\text{Total number of lines}} * 100\% \quad (1)$$

Test execution coverage equation illustrated below is another code level metric. It displays how many tests are executed out of total number. While calculation shows progression of overall testing process, it does not show the quality of testing process or product under test. [18.]

$$Percentage\ of\ executed\ tests = \frac{\text{Number of tests run}}{\text{Total number of tests to be run}} * 100\% \quad (2)$$

4.2.3 Feature Level Metric

Feature level metrics are used in the projects to verify that all the features and requirements set to the product are covered by tests. Equation 3 can be used in the projects to display test creation process and to verify that at the end of the testing process all the requirements are covered by the tests. [13.]

$$Percentage\ of\ requirements\ covered = \frac{\text{Number of requirements covered}}{\text{Total number of requirements}} * 100\% \quad (3)$$

Another feature level testing metric is test cases by requirement illustrated in table 1, which displays all the requirements, test cases and testing outcomes. As in most cases

multiple tests are needed to verify that product fulfills specific requirement and this metric allows easily to follow which tests failed and to which requirement it was related to. It also allows to show how fully specific requirements are covered by testing, as well as which requirements are most problematic based on failures. [13.]

Table 1. Test cases by requirement metric.

Requirement	Test case	Test result
Requirement 1	Test case 1	Pass
Requirement 1	Test case 2	Failed
Requirement 2	Test case 3	Incomplete

4.2.4 Application Level Metric

One of the application level metrics is defect density. Metric is used to calculate defect density in the specific parts of the application. Metric results can be used to find areas which are most defect prone and based on that prioritize how thoroughly specific areas of product should be covered by tests. [13.]

$$Defect\ density = \frac{\text{Number of found defects}}{\text{Size of software entity}} \quad (4)$$

4.3 Effectiveness Based on Customer Feedback

4.3.1 Metrics

One of the ways to measure the effectiveness of testing is to use information gathered from the customers. Metrics based on customer feedback can be divided into two main categories customer satisfaction and defect measurement. [14.]

4.3.2 Customer Satisfaction Measurement

One of the ways to assess the quality and effectiveness of testing is to use feedback from the customers. Feedback about the product can be collected for example through surveys, help-desk calls and referrals. Assessment is based on whether the customer is satisfied with the product. High customer satisfaction with the product means that the manufacturer can be satisfied with the quality of product and testing process. [14.]

While customer is satisfied with the quality of the product this does not guarantee that the quality of the product development and testing process is actually good. It is hard to make sure or even possible that customers report all the unpleasant situations and weird behaviors about the products, as well as that feedback collection process remains consistent. Customer feedback can differ a lot, as customers could be using product in different ways and environments. In customer feedback where is always possibility of misunderstandings, as well as details can be lost through communication process. [14.]

Another problem of assessing the testing quality based on customers satisfaction is that it does not separate the quality of product development from the quality of testing process. When customer do not report any bugs, it can be because of quality of the development or because of quality of the testing, but it would not verify either. This situation can also lead to false belief of great quality of the testing process. For example, as illustrated in figure 6, high quality product tested with low quality testing process could bring up same number of bugs, as good quality testing process with high quality product. [14.]

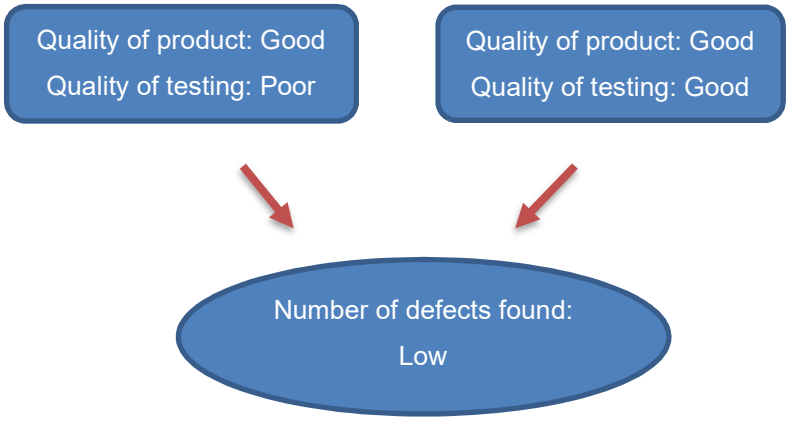

Figure 6. Quality of product and testing.

4.3.3 Defect Measurement

Testing process effectiveness can be calculated based on number of defects found in the testing and reported by the customers. For any testing process, one of the main goals is always to minimize the possibilities which would allow defects to reach the customer. Defect reaching the customer proves that there are flaws in the testing process. Equation 5 can be used to display how effective testing process is, as it gives percentage of defects found by testing from the total number of defects found. [14.]

$$Defect\ removal\ efficiency = \frac{\text{Defects found in testing}}{\text{Defects found in testing+defect reported by customer}} \quad (5)$$

In most cases to get 100% defect removal efficiency is impossible and not even necessary. Trying to reach high effectiveness level can create too complicated and endless testing process. At the end of the day producing and developing the product must make sense to the manufacturer. Trying to set too high-quality goals for the product can lead to situation where product will never be ready. Product should satisfy customer needs and trying to find all the defects in most cases is not necessary for that. Acceptable test effectiveness level can vary a lot depending on which kind product is tested. For example, finding all the defects from the product in the medical field is far more important than finding all the faults from the simple calculator application.

While test effectiveness assessment based on number of defects gives some kind of overview of testing effectiveness, it is still very heavily dependent on customer feedback, as was customer satisfaction measurement. In ideal world it would be perfect equation to calculate the effectiveness of testing, but in a real-world accuracy of equation can vary a lot. There is big chance that customers would not report all the bugs, as well as same bug can be reported multiple times where root cause of the bug is the same defect. Assessing what is a defect can vary a lot as well, as one can think that customer used the product in a wrong way while another can think it is a defect. To get accurate value from equation 5, defects reported by customer need thorough analyzing before making calculation to limit false and duplicate defects.

4.4 Evaluation of Test Effectiveness Using Fault Seeding

4.4.1 Fault Seeding

Fault seeding is a technique where defects are intentionally injected into the software. Fault seeding is used to evaluate the effectiveness of testing, as well as to test how product performs under stress.

From test effectiveness point of view, faults are injected to the product to verify does the individual tests notice defects to which they are designed for. Using fault seeding to measure test effectiveness requires first to understand which kind of defects individual test cases are designed to detect. Fault seeding can also be used to find requirements

which are not covered by testing process. Faults can be injected to the software so that product would not work as specified in the requirement and lack of requirement coverage is verified when all the tests are run and none of them find the defect.

Systematic fault seeding testing process is called mutation testing. In the mutation testing different versions of original source code are created where every mutated software version has single fault injected to them. Both original and mutant software are tested, and results compared to measure the effectiveness and quality of testing.

Fault injections can be divided into two main groups based on when fault is injected, compile-time and run-time fault injections. [15.]

4.4.2 Compile-Time Fault Injection

Compile-time fault injection is a technique where fault is seeded straight to the product's source code. Errors are seeded to the product by modifying, removing or by adding code to the source code. Faults injected to the product's source code can be divided into three main categories based on how they affect the product, value, decision and statement mutations. [15.]

Value mutation illustrated in figure 7, is a type of change where variable value is changed in the source code.

```
// Original
real32 speed = 5.2f;

// Value mutation
real32 speed = 1.3f;
```

Figure 7. Value mutation.

Decision mutation is a change in the source code which causes code to run through different path in specific condition. It is achieved for example by changing conditions when code runs through some loop in the source code. This can be done by replacing arithmetic and relation operators, like illustrated in figure 8.

Figure 8. Decision mutation.

Statement mutation is a method where part of the code is modified or removed so that it returns different outcome in specific conditions. For example, mutation can be generated by changing return statement of a method or by removing else part from the else-if statement, as illustrated in figure 9. [16.]

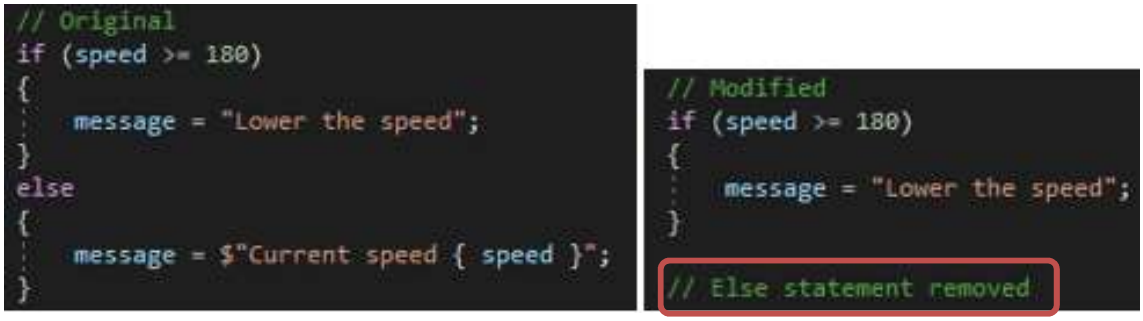

Figure 9. Statement mutation.

In the statement mutation also extreme mutation strategy could be used. In the extreme mutation whole method or logic is replaced by a nullable block. For example, in figure 10, instead of creating mutant software against every statement in the method one mutant is created out of original source code. Extreme mutation is good for preliminary analysis to strengthen the test case before evaluating the tests effectiveness against statements inside the method. [16.]

```
public static string Speed(int speed)
{
    string message;

    if (speed < 50)
    {
        message = "Speed is under 50.";
    }
    else if (speed <= 50 && speed >= 100)
    {
        message = "Speed is between 50 and 100.";
    }
    else
    {
        message = "Speed is to high";
    }

    return message;
}
```

```
public static string Speed(int speed)
{
    return null;
}
```

Figure 10. Extreme mutation.

For example, below in figure 11 is a method called CheckPalindrome and test case CheckShouldBeTrue_CheckPalindrome. Method takes in a word and checks is a word palindrome or not, and test case is designed to verify that method correctly detects palindromes. In figure 12 decision mutation and statement mutation are used to create type of defects which test case should detect. In this example test case noticed defects in the method as expected and therefore test case is effective.

```
3 references
public static bool CheckPalindrome(string word)
{
    bool output = false;

    char[] reverse = word.ToCharArray();
    Array.Reverse(reverse);

    if (word == new string(reverse))
    {
        output = true;
    }

    return output;
}
```

```
[TestMethod]
0 references
public void CheckShouldBeTrue_CheckPalindrome()
{
    // Arrange
    string word = "ebe";

    // Act
    bool actual = Palindrome.CheckPalindrome(word);

    // Assert
    Assert.AreEqual(expected:true, actual);
}
```

Figure 11. Original method and test case.

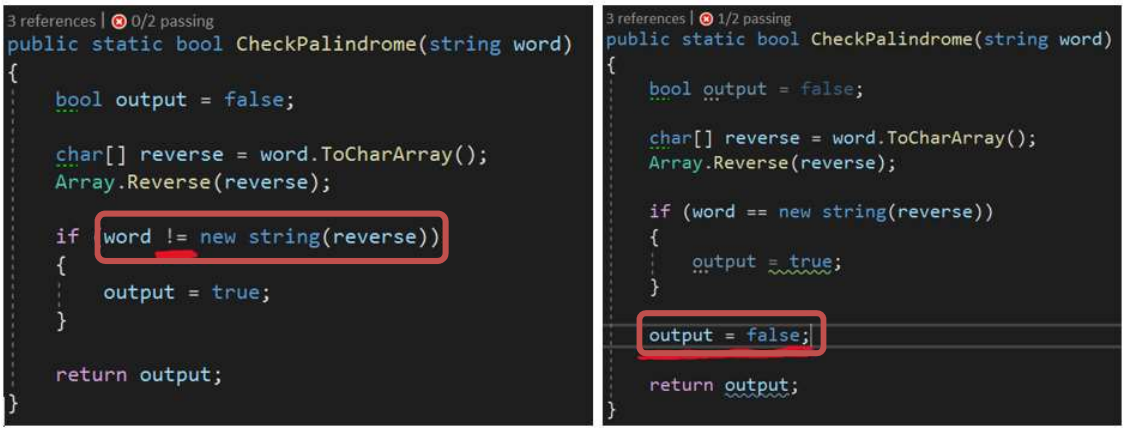

Figure 12. Decision mutation and statement mutation.

When seeding faults to the products software, it is also important to understand the source code. Not all the injected errors cause change in the behavior of the product, like illustrated in figure 13. These seeded faults are called equivalent mutations. Equivalent mutations can be caused because of dead code or because triggering injected fault is restricted by other logic elsewhere in the code. Some injected defects also can cause situations where only the performance of the product is affected. Equivalent mutations usually bring up unnecessary parts of the source code which can be simplified or deleted. [16.]

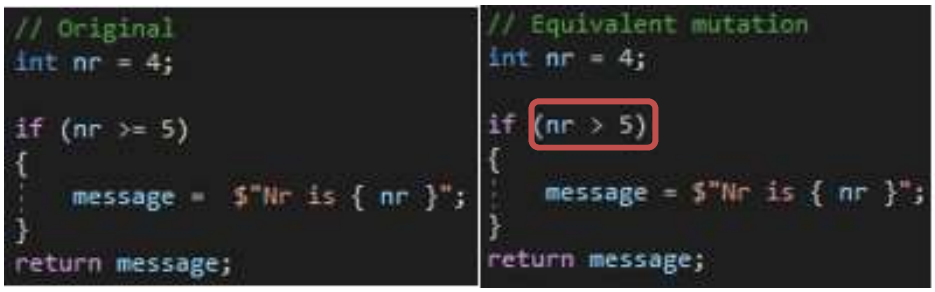

Figure 13. Equivalent mutation.

4.4.3 Run-time Fault Injection

Run-time fault injection is a technique where faults are seeded into the running product. Corrupting memory space, system call interposition and network level methods are used to seed run-time faults into the products software. Corrupting memory space method is used to corrupt main memory and process registers, while network level method is used to inject fault real-time into the network interface. System call interposition is a method where system calls made by user-level software are intercepted by real-time fault injections. [15.]

4.4.4 Metric

Fault detection rate equation can be used to calculate the effectiveness of test case or overall testing process, as it shows how many injected faults tests were able to detect. For example, when 10 faults are seeded to the source code and testing process finds 8 of them, when effectiveness of testing is 80%. [15.]

$$Fault\ detection\ rate = \frac{\text{Discovered seeded faults}}{\text{Seeded faults}} * 100\% \quad (6)$$

4.4.5 Advantages

Fault seeding is a very powerful way to measure the effectiveness of individual test cases and overall testing process. By understating which kind of defects specific tests should find, the effectiveness could be evaluated by seeding errors to the source code. This directly shows is test capable of finding defects to which it is designed to and based on that, overall testing quality and effectiveness can be assessed.

As mentioned in customer feedback measurement, quality and consistency of a customer feedback can vary a lot which affects accuracy of test effectiveness measurement. Fault seeding evaluation process removes dependency to customer and therefore can give more accurate measurement and constant process.

Test process effectiveness evaluation can be started right away when the first test cases are created, and this allows to keep the quality and effectiveness up from the beginning. Early started test effectiveness evaluation process reduces the chance that product's software is built on top of the defects.

While fault seeding cannot verify that testing process could find all the defects it can get close to that, assuming that in the designing process all the requirements for the product are brought up. It allows manufacturer to produce reliable and stable product, as well as ensures that customer receives quality product with least number of defects. As quality of product is one of the most important ingredients for successful product, the measurement of quality and effectiveness of testing process should be one of the top priorities of any product development process as well.

4.4.6 Disadvantages

Fault seeding testing can be a very complex process. It requires good understating of how individual test cases work and which type of defects they are designed to detect. In addition of knowledge of tests, fault seeding requires good understanding of product source code and knowledge of coding. To generate specific faults in the product it is essential to understand how product should operate in the normal conditions and how behavior could be altered in the source code. This can be very difficult process for products with complex source code and situations where there is lack of documentation about products requirements and functionalities.

In addition to complexity, fault seeding testing process can be very time consuming and could require a lot of effort. Big projects could contain thousands of test cases and this requires creating a lot of mutants out of the software to cover all the defects the test cases should find. In the systematic fault seeding testing this can take a lot of time, especially when it is done manually. Fault seeding testing also requires running tests multiple times, first with original software and after with mutant software to compare results and this can take a lot of time especially if test case has multiple assertions.

When evaluating effectiveness of test cases which are used with physical device, seeded faults can do a lot of harm for the test setup. Seeding faults to the source code without careful planning can lead to situations where test setup is damaged. There is also possibility that small unnoticeable damages are done to the physical device which can at some point later alter the test results.

4.4.7 Tools to Evaluate Test Effectiveness

To lower the time taken by fault seeding process automation tools can be used to inject faults to the source code, as well as to run mutated software against original test cases. Open source tools available in the market are mostly only suitable for evaluating the effectiveness of unit tests.

Stryker is one of the most popular open source mutation testing tools. Stryker seeds automatically errors to the software and runs and evaluates effectiveness of tests. It is easy to implement and is commonly used to test the effectiveness of unit tests. It supports C#, Java, Scala programming languages and allows to configurate settings to fulfill

specific mutation testing needs. For example, Stryker allows to specify which kind of faults to seed, as well as which kind of methods, files, mutations exclude from mutation testing. [17.]

Below in figure 14 is an example of Stryker's mutation testing report. It displays how effective test cases were of finding injected defects. Tool also allows to take a closer look, as illustrated in figure 15, to see which kind of defects were injected, survived or were not covered by testing.

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	76.92	10	1	0	2	0	0	**10**	**3**	**13**
Calculator.cs	83.33	5	0	0	1	0	0	**5**	**1**	**6**
Converter.cs	66.67	2	0	0	1	0	0	**2**	**1**	**3**
Example.cs	75.00	3	1	0	0	0	0	**3**	**1**	**4**

Figure 14. Stryker mutation testing report.

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
Calculator.cs	83.33	5	0	0	1	0	0	5	1	6

Killed (5) NoCoverage (1) Collapse all

```
namespace DemoLibrary
{
    public class Calculator
    {
        public static double Add(double x, double y)
        {
            return 0 x - y ~~x + y~~;
        }

        public static double Subtract(double x, double y)
        {
            return 1 x + y ~~x - y~~;
        }

        public static double Multiply(double x, double y)
        {
            return 2 x / y ~~x * y~~;
        }

        public static double Divide(double x, double y)
        {
            if (3 y == 0 4 !(y != 0) ~~y != 0~~)
            {
                return 5 x * y ~~x / y~~;
            }
            else
            {
                // Custom business logic for divide by zero
                return 0;
            }
        }
    }
}
```

Figure 15. Detailed test result.

5 Testing of Drives Software at ABB

5.1 Testing in ABB

In ABB drive software is continuously developed by hundreds of developers in all over the world and because of that importance of testing is very high. ABB is releasing updated version of the drive software to the market multiple times in a year. Updated

version includes bug fixes, improvements on current functionalities, as well as new functionalities for the drive. As software structure grows, the complexity of code increases and therefore coding errors are easier to come by. To see also how change in one-part affects software functionalities in all the other parts gets more difficult as development continuous. With complex and large project, high quality testing process is needed to cover all the features and functionalities. As ABB software testing process is developed and maintained by developers all over the world, ensuring and proving that the quality and effectiveness of testing is high can be very time consuming and difficult process.

One of the goals of ABB's software testing is to minimize the possibilities of defects reaching the customer, so that end user could receive bug free product. As drive has hundreds of parameters and endless combinations for them, as mentioned in chapter 2.2, quality of testing must be high and constantly improved to achieve this goal.

5.2 Testing Process

In ABB drive software is tested daily. Every time when changes are made to the software repository tests are executed automatically. Testing makes sure that added changes work correctly and changes did not break any other parts of the software.

ABB's automated testing framework called ATF is a software solution for test automation. ATF has been developed and used for testing in the ABB for more than ten years. ATF is a system testing framework which is using black box testing techniques to test drives functionalities and features. ATF has thousands of test cases which are testing drive's individual functionalities. ATF runs on computer which controls and monitors the drive where software is loaded. When all the tests are executed, results are gathered and reported.

ABB is using continuous integration server for testing process illustrated in figure 16. When changes are committed to the software repository, server automatically executes tests starting with unit test and static code analyses. If software is passing them, then testing continues with basic ATF tests called Smoke tests. Smoke tests evaluate workings of basic drive functionalities, like start and stop commands. Smoke testing verifies that software can be loaded to the drive and main functionalities still work properly after

the change. When software is passing the smoke tests, it means that software version under test is stable and changes made to the software can be kept.

Once a day all the ATF tests are run with the latest stable software. There are hundreds of tests for one product family and running them all can take more than 20 hours. Because of that it is not efficient to run all the stable ATF tests after every change in the software repository. Continuous integration server triggers automatically test run with all the suitable ATF test cases once a day. Test run starts after working hours and runs through the night so that testing setups can be free to use by developers through working hours.

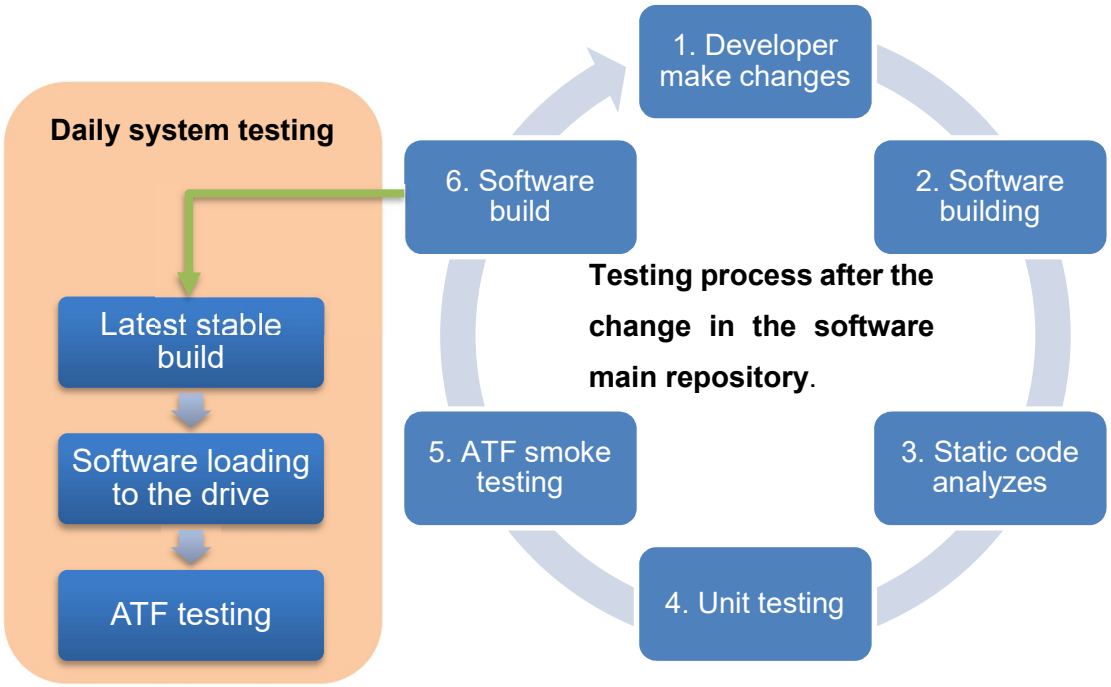

Figure 16. Testing process

5.3 Measuring Effectiveness of ATF Tests

ABB does not have at the moment systematic way to evaluate effectiveness of ATF tests. ABB uses customer feedback to assess effectiveness of overall testing process. As mentioned in chapter 4.3.3, information can be used to calculate overall effectiveness using equation 5, as well as to improve testing and find areas in the software which are not covered by testing. But as goal for ABB is to reduce possibility of defects reaching the customer, especially critical defects, this is not best solution. This process is always one

step behind, as in case of customer reported bugs, testing process is improved after the feedback from the customer about found bug. To improve this situation fault seeding experiment could be used. Fault seeding allows to simulate same or similar defects like reported by the customers and therefore allows to improve the testing process before defect is reached the customer.

In unit testing using and implementing code coverage or mutation testing tools like Stryker is easy and does not require a lot of additional work. This is not the case when effectiveness of testing is evaluated in the system testing. As ATF testing is black-box testing, it means that software is tested without taking a look into the source code or having access for modification in the testing environment. Mutation testing and code coverage tools available in the market are only suitable for the white-box testing like unit testing where source code is available for modification in the testing environment. While there are no mutation testing tools available for system testing, fault seeding process can be used in ATF. To evaluate the effectiveness of ATF tests with mutation testing, evaluation process can be divided into two. Fault seeding can be done in the drive software environment and test evaluation with mutated software in the testing environment.

6 Evaluation of ATF Tests

6.1 Clean Run with Original Software

As mentioned in introduction, test cases were selected based on which are working with the new ATF testing setups. For the general purpose drives ATF has around 250 individual tests from which 79 are suitable to run with updated setups as shown in figure 17. Test cases are divided into groups based on which drive parameter groups and functionalities they are evaluating. Every test case is executed multiple times using different drive settings. Tests under evaluation in this thesis are testing behavior of drive parameters in group 6, 10, 12, 13, 20, 21 and 96.

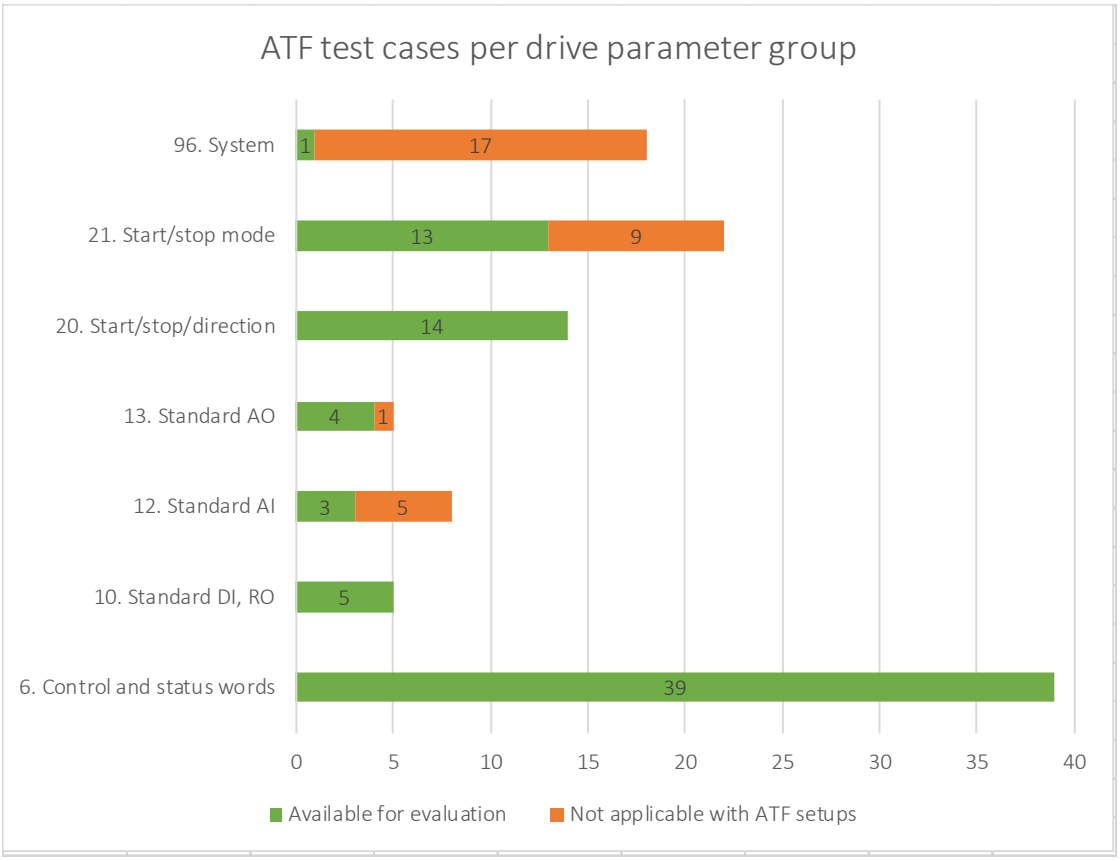

Figure 17. ATF test cases per drive parameter group.

To get baseline, test cases were run with clean original code to ensure that software under test is bug free, as well as that test cases are passing as expected with clean software.

Kaikki testit

Package	Kesto	Epäonnistui	(muutos)	Ohitettiin	(muutos)	Pass	(muutos)	Yhteensä
DC_Heating	18 min	0		0		13	+13	13
DC_Hold	6 min 1 sec	0		0		8	+8	8
DriveLogic	6 min 2 sec	0		0		1	+1	1
DriveStatusWord1	10 min	0		0		13	+13	13
DriveStatusWord2	14 min	0		0		15	+15	15
EnableToRotate	2 hr 43 min	0		0		16	+16	16
MainStatusWord	16 min	0		0		19	+19	19
PostMagnetization	26 min	0	-27	0		37	+27	37
Setup	4 min 20 sec	0		0		2	+1	2
Standard_DI	4 min 8 sec	0		0		8	+8	8
Standard_RO	48 min	0		0		63	+63	63
StartInhibitStatusWord	8 min 21 sec	0		0		9	+9	9
StartStopDir	2 hr 59 min	0		0		182	+182	182
UnitSelection	6 min 11 sec	0		0		1	+1	1

Figure 18. Test results with clean run.

6.2 Fault Injection and Mutation Testing Process

In this thesis fault seeding method was used to evaluate the effectiveness of individual ATF tests. Fault injection process illustrated in figure 22 started by studying tests to see which kind of defects specific tests are looking for. Modifications were made manually to the software source code one at the time to mimic specific defects that tests are looking for. As goal was to test the effectiveness of specific tests, inserting random faults using some fault seeding tool was not efficient, as only part of tests were used. Inserting random errors would have required to go over injected faults to ensure that they are in the correct places and do not harm the drive when tests are executed.

Faults were generated using decision and statement fault seeding methods described in the chapter 4.4.2. In most cases faults were injected by modifying if statements in the source code, by commenting out if statement, or by changing conditions when if statement is executed. For example, to evaluate the effectiveness of test case number 1 in appendix 1, both decision and statement mutations were used to alter behavior of enabled bit shown in figure 19. In the statement mutation code line is commented out, which made bit to stay at false state at all times. In the decision mutation, condition when if loop is executed is changed which made bit to stay at wrong state at all times.

Original:

```
//-------------------------------------------------------------------------------------------
// Drive status word 1
//B0-B3
//-------------------------------------------------------------------------------------------
if (clDCPStatusWord.bfENABLED)                                    u16Temp |= DRIVESW1_Enabled;
```

Statement mutation:

```
//-------------------------------------------------------------------------------------------
// Drive status word 1
//B0-B3
//-------------------------------------------------------------------------------------------
//if (clDCPStatusWord.bfENABLED)                                  u16Temp |= DRIVESW1_Enabled;
```

Decision mutation:

```
//-------------------------------------------------------------------------------------------
// Drive status word 1
//B0-B3
//-------------------------------------------------------------------------------------------
if (!clDCPStatusWord.bfENABLED)                                   u16Temp |= DRIVESW1_Enabled;
```

Figure 19. Statement and Decision mutations.

Before testing it was verified manually that change in the software made drive to behave differently. For that mutated software was loaded to the drive and behavior of the drive was monitored using ABB's tool called drive composer. For example, enabled bit modified in figure 19 should be in active state when drive parameters 20.12 and 20.19 are selected, meaning they are active as shown in figure 20 [18, p. 169]. Both mutations in figure 19 always set bit to be at wrong state like shown in figure 21.

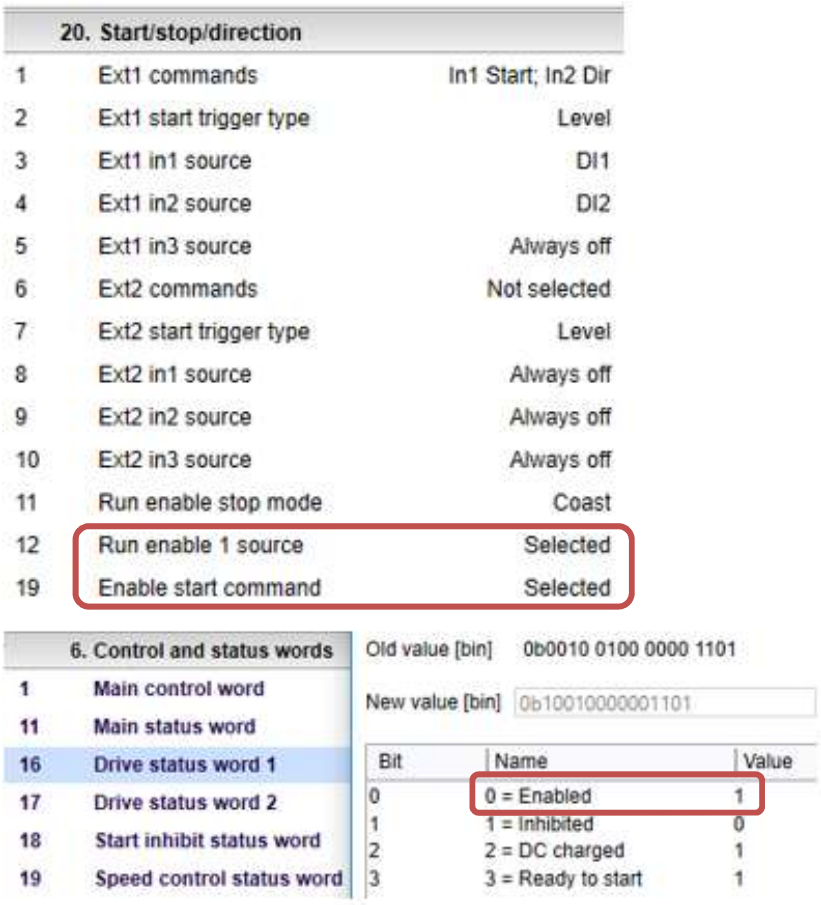

Figure 20. Enabled bit correct behavior with clean software.

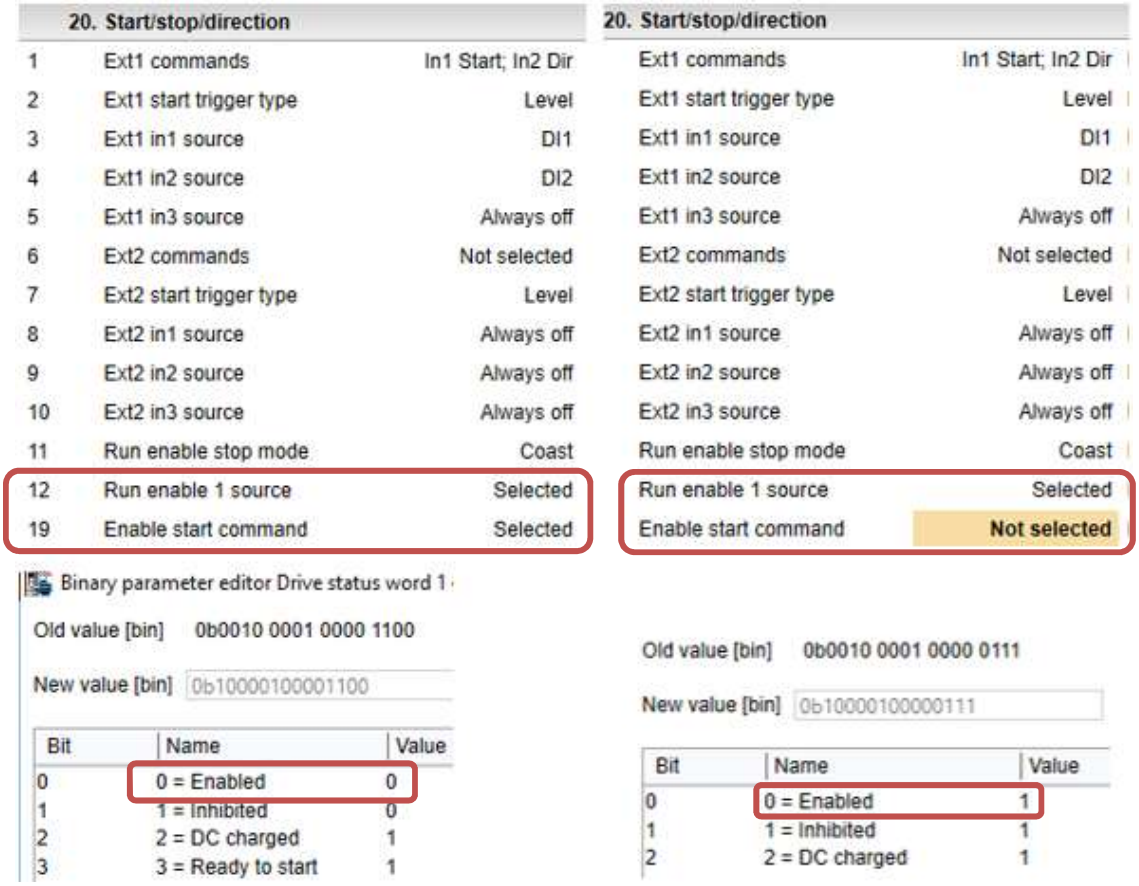

Figure 21. Enabled bit behavior with mutated software.

When drive acted differently as expected, testing was started. For the test case evaluation process, continuous integration server was used, first to generate software package for the drive out of mutated software and then for running ATF tests. Using continuous integration server allowed to use same test setups which are used in the normal daily system testing, as well as allowed to run tests against mutated software after working hours, which speeds up overall test evaluation process.

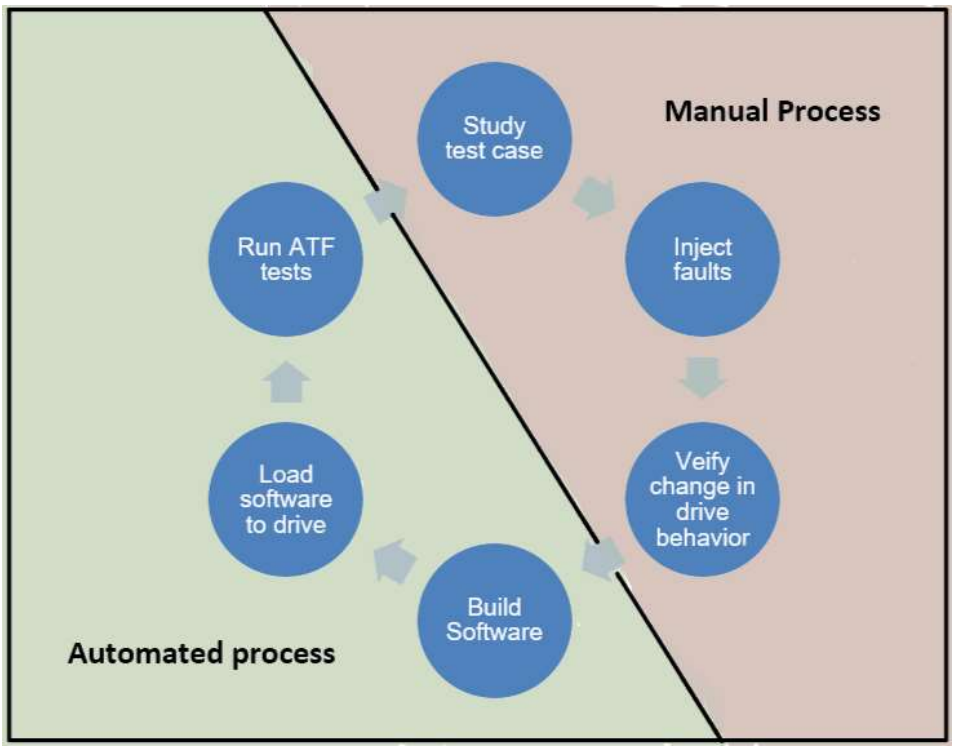

Figure 22. Test case evaluation process.

6.3 Results

Total 124 defects were injected to the software to evaluate the effectiveness of 79 tests. In most cases defects were injected one at the time to get most accurate results. Faults were generated to places in the source code based on information gathered when analyzing individual tests. Modification were made to the behavior of parameters or functionalities which generated faulty behavior that individual test cases were looking. In the system testing, tests are evaluating only the external behavior of the product in the specific situations and in the source code this behavior could be altered multiple ways and places. In this thesis same wrong drive behavior that tests were looking was generated only one way, as creating same behavior change in the different place of the source code would not have given any more benefit for the tests evaluation process in the system testing. Both decision and statement mutations were used to evaluate the effectiveness of test cases. Beginning of tests evaluation process mutations were done against every test once to ensure that all tests were evaluated at least once in this thesis. After that as many tests as possible were evaluated at least once more by making drive parameter or functionality to behave wrongly in some other way.

Test cases were able to detect 121 defects out of the total number shown in figure 23. Defects which were left undetected were run multiple times to ensure that change in software were actually left unnoticed.

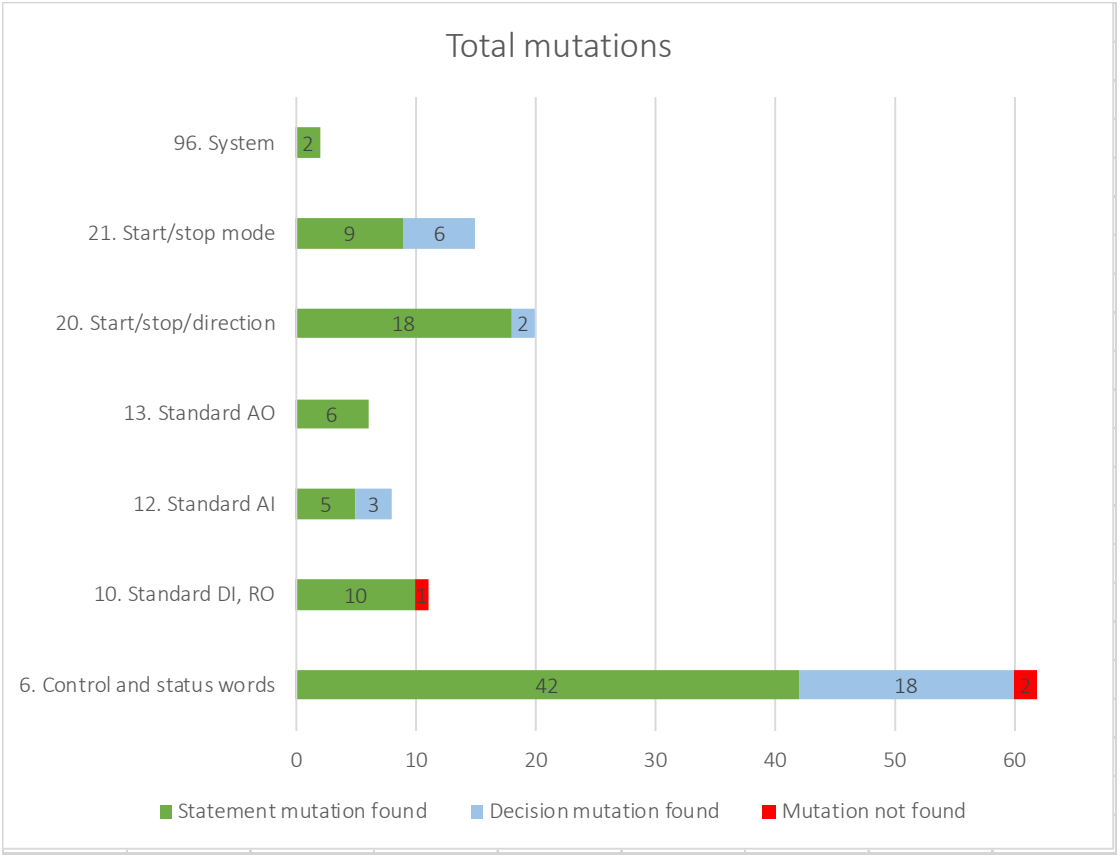

Figure 23. Total faults seeded per drive parameter group.

In all cases where test case found the defect, assertion message gave clear reason why test was failing. For example, test 1 in appendix 1 with mutations shown in figure 19 gave correctly and clearly reason why test failed as shown in figure 24.

```
** Assertion failure! **
  Message: "3. Drive is Enabled when both Run Enable and Enable Start Command is on"
  Expected: True
  Actual:   False
```

Figure 24. DWS1_Bit0_Enabled test case assertion message.

One defect that was not caught by testing was related to behavior of parameter 10.1 digital input (DI) status. Defect was done in similar fashion like shown in figure 19 and it made DI status parameter bits to stay at false state, shown in figure 26. Defect was expected to be found by test case number 40 in appendix 1. By test case naming

convention and description, it was expected to verify correct behavior of the parameter 10.1 DI status. Closer analysis of test case showed that it was actually evaluating workings of parameter 10.2 DI delayed status. In the normal conditions DI status parameter bits are always active when delay parameter bits are active as shown in figure 25, but in the software source code their state is altered by different methods. Later re-run was also done with the same mutated software, but in this case all the tests were used shown in appendix 1 and also in this case defect was not found. Test case number 40 in appendix 1 was also evaluated with the mutated software where parameter 10.2 DI delay status behavior was changed as shown in the figure 27 and in this case, test noticed change in the drive behavior. To detect defects related to the parameter 10.1 DI status small changes would be needed to be done for the test case. It would require adding some assertions to the test which takes a lot less time and effort compared to the situations there this defect is found by customer.

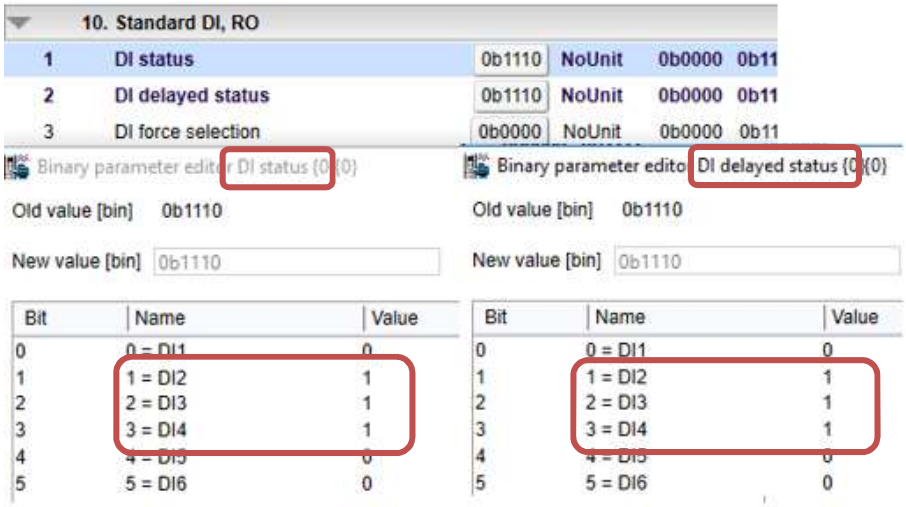

Figure 25. Parameters DI status and DI delayed status behavior with clean software.

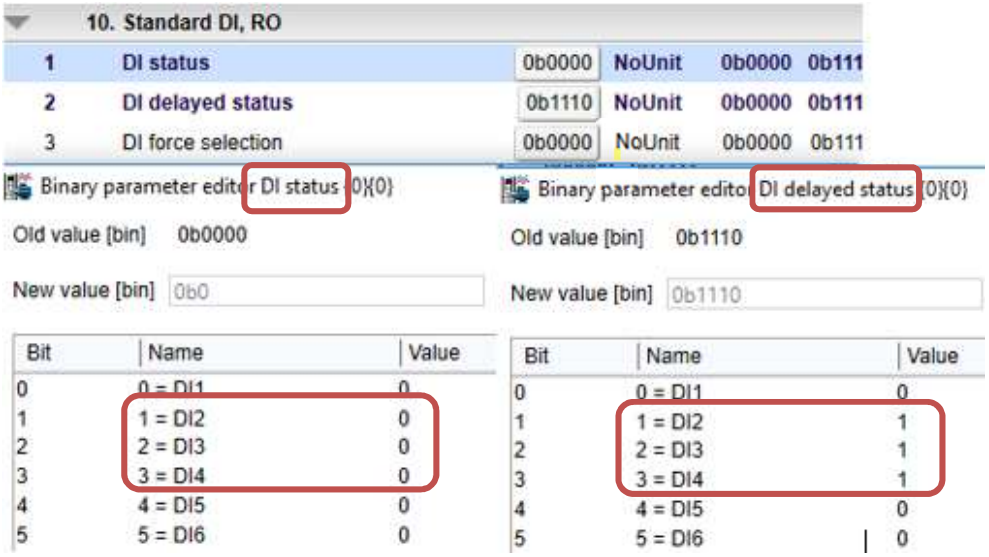

Figure 26. Parameters DI status and DI delayed status behavior with mutation done to status bit.

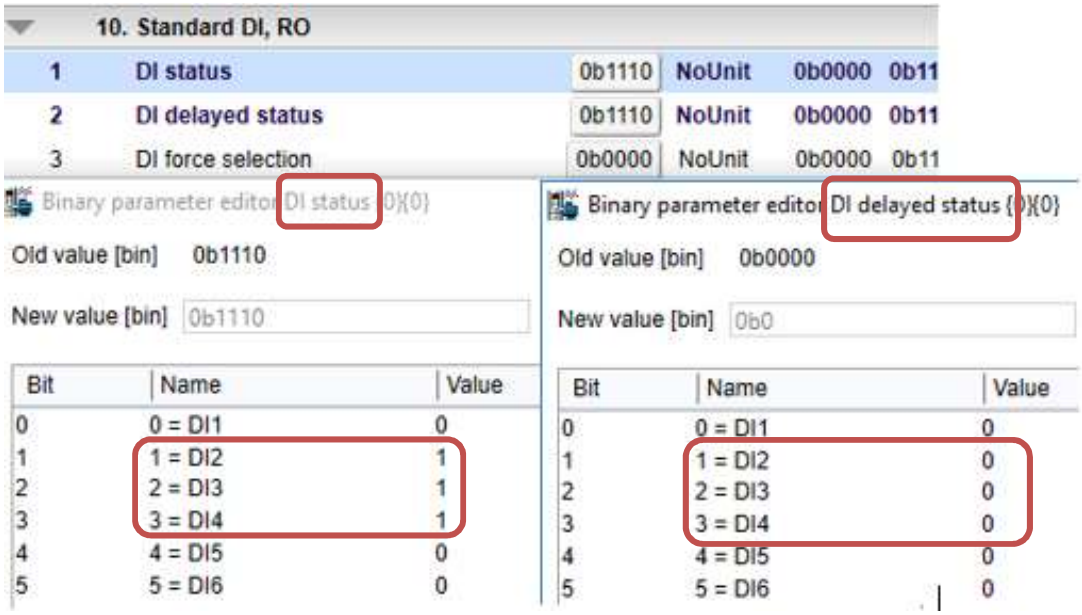

Figure 27. Parameters DI status and DI delayed status behavior with mutation done to delay bit.

Two other defects not found by testing were related to the behavior of bit called speed control which is set to true when speed control mode is active. In scalar motor control mode drives operation mode is always speed control, while in the vector motor control mode it can either be speed or torque. Speed control bits behavior is checked with the test case 40 in appendix 1. By closer analysis of the test case, it showed that test is checking bit behavior only in the vector motor control mode. In the drive source code bit is activated in different places depending on which motor control mode drive is using. Test case did not notice behavior change shown in figure 28 where fault was injected to

the loop which was altering the bit state in the scalar control mode. Re-run was also done with all the available test cases shown in appendix 1 against same mutated software and no test found defect in the source code. Test case correctly found defects related to the speed control bit behavior in the vector motor control mode. To make sure that detect is found by testing, small changes are needed to be done to the test case which should not take a lot of time and effort.

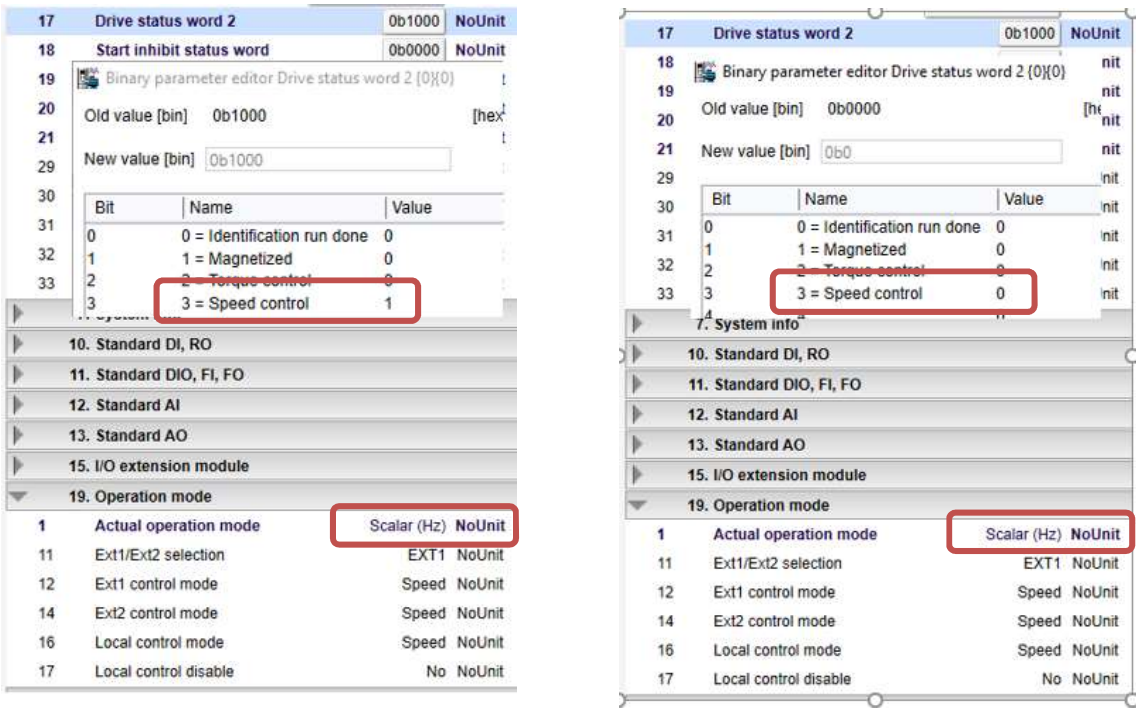

Figure 28. Speed control bit behavior in original source code vs in mutated software.

Test evaluation process also brought up three test cases which were not running in the daily system testing. Test cases number 58, 75 and 76 in the appendix 1 were removed from their test groups, but there was no clear reason why. Testing showed that tests number 58 and 76 in the appendix 1 worked correctly with ATF setups. With clean source code tests passed and with the mutated software they failed as expected. Run with test case number 75 in the appendix 1 showed that it is not applicable to run with the updated ATF setups. Defect seeded to evaluate test case number 75 was found by the test case number 76 in the appendix 1.

When using equation 6, fault detection rate, it shows that effectiveness of current available ATF tests is 97.6%. While there is no exact result what should be achieved to be satisfied with test effectiveness, in common sense the better result from the equation the more satisfied one could be with testing. In case of this thesis, all the defects injected to

the source code made drive behave wrongly and should have been found by tests, and therefore result shows pretty accurately effectiveness of tests under evaluation. Goal for any testing process should be to get as close as possible for 100%, as it gives best chance to catch defects in the testing. Any real fault left unnoticed requires fix at some point and in most cases, later the fault is found the more effort and time is needed to fix the bug.

7 Conclusion

In this thesis, the effectiveness of ATF tests were evaluated using fault seeding method. 79 general purpose drive tests out of 250 were selected for the evaluation. In test cases evaluation process, first tests were reviewed to understand which kind of defects they should find and then faults were injected to the source code in most cases one by one. Tests were run against mutated software using continuous integration server which allowed easily to generate software package for drive and run tests nightly.

Faults were seeded to the drive source code manually by altering source code using decision and statement mutations. In most cases finding where to inject faults was not as complicated as expected first. Variable naming conventions in the drive software and testing environment were similar, which eased finding how to alter drive behavior correctly. Most worrying part of the test evaluation process was ensuring that mutation would not do harm to the drive under test and this was limited by injecting faults manually and trying out mutated software before testing.

Total 124 faults were seeded to the source code to evaluate the effectiveness of tests. Thesis showed that in most cases ATF tests were capable of finding most of the injected defects. Work brought up type of defects which should have been found by ATF test cases under review but were not, as well as reasons why defects were left unnoticed.

Work process showed that fault seeding is very effective way to evaluate the effectiveness of ABB's ATF tests. Fault seeding allowed to generate exact defects that test cases should find and therefore show how effective tests are. While it can be very effective way to evaluate the effectiveness, it can also be endless process. For against every test case there is possible to do a huge number of different mutations, and therefore it is important to make plan how tests are evaluated like is done in normal testing process. In case of

system testing, to create same external behavior change in the different places on the source code does not give any more benefit for the mutation testing, as system tests only evaluate external behavior of the product, not how this behavior was created in the source code.

In ABB, to automate test evaluation process using fault seeding should not require a lot of work, as only fault seeding part needs to be automated. ABB already has all other parts automated and in the open market there are available open source tools which can potentially be implemented for fault seeding. Automated process would reduce time and effort needed from the process and it would be most beneficial when all the unit and ATF tests are available, as it would allow to insert random faults to the source code.

In both manual and automated process, the time and effort needed could be reduced by injecting faults only to the most critical places in the software. These could be places where fault can do most harm or areas where most of the bugs are reported, which could be calculated using equation 4, defect density rate. The biggest benefit for the manual process is that it allows generate exact defects needed for the evaluation, while in automation process this can be harder to achieve, but automated process would take less time and effort in a long run when huge number of tests are evaluated systematically.

Systematic fault seeding test evaluation process could be very powerful and effective way to evaluate the effectiveness of both unit and ATF tests, as well as to find parts of software which are not covered by testing. This would improve testing quality and minimize the possibilities of defects reaching the customer, which is one of the goals of testing in ABB.

References

1 About ABB. Online. ABB Oy. <https://new.abb.com/about>. Accessed 6 December 2019.

2 Hartman, Craig. 2014. What is a VFD. Online. VFDs.com. <https://www.vfds.com/blog/what-is-a-vfd>. Accessed 26 December 2019.

3 ABB:n taajuusmuuttajilla 40 ydinvoimalan energiansäästö. Online. ABB. <http://www.abb.fi/cawp/seitp202/6f770aa684e40007c1257a1a00368559.aspx>. Accessed 30 December 2019.

4 Glampe, Mike. How Pulse Width Modulation in a VFD works [online]. Keb America. <https://kebblog.com/pulse-width-modulation-in-vfds/>. Accessed 26 December 2019.

5 What is an AC drive?. Online. ABB. <https://new.abb.com/drives/what-is-a-drive>. Accessed 26 December 2019.

6 ABB drives save £25,000 on energy at duck processing plant. Online. ABB. <https://new.abb.com/drives/media/abb-drives-save-25-000-on-energy-at-duck-processing-plant>. Accessed 1 January 2020.

7 ABB general purpose drives. Online. 2019. ABB Oy. < https://library.e.abb.com/public/663a085159d64c0c8f0927522feef0b4/ACS480_general_purpose_drives_catalog_3AUA0000204668_RevE_EN.pdf>. Accessed 2 March 2020

8 ABB's new ACS480 drive for energy efficiency and effortless operation in pumps, fans and compressors. Online. 2019. ABB Oy. <http://www.abb.com/cawp/seitp202/90a1982a0575bcb6c12580580046aeb8.aspx>. Accessed 2 March 2020

9 Knowit Oy. 2019. ISTQB Foundation certificate on software testing. Course material.

10 Jensen, Christopher. 2014. Nissan Recalls 990,000 Vehicles for Air Bag Malfunction. Online. The New York Times. <https://www.nytimes.com/2014/03/27/automobiles/nissan-recalls-990000-cars-and-trucks-for-air-bag-malfunction.html>. Accessed 3 January 2020.

11 Leveson, Nancy; Turner, C.S. 1993. An investigation of the Therac-25 accidents. IEEE Computer. Vol. 26, p. 18-41.

12 Eriksson, Ulf. Differences Between the Different Levels of Testing. Online. ReQtest. <https://reqtest.com/testing-blog/different-levels-of-testing/>. Accessed 7 January 2020.

13 Why Test Coverage is an Important part of Software Testing?. Online. Simform. America. <https://www.simform.com/test-coverage/>. Accessed 9 January 2020.

14 Craig, Rick. 2000. Measuring test effectiveness: how good is your testing?. Online. International Conference on Software Testing Analysis & Review. <https://www.stickyminds.com/sites/default/files/presentation/file/2013/00STRWR_WG1.pdf>. Accessed 10 January 2020.

15 Fault Injection in Software Engineering. Online. GeeksforGeeks. <https://www.geeksforgeeks.org/fault-injection-in-software-engineering/>. Accessed 12 January 2020.

16 Rijo, Pedro. 2019. An intro to Mutation Testing. Online. Pedrorijo.com. <https://pedrorijo.com/blog/intro-mutation/>. Accessed 13 January 2020.

17 Introduction. Online. Stryker. <https://stryker-mutator.io/stryker-net/>. Accessed 17 January 2020.

18 ACS480-vakio-ohjausohjelma. Online. ABB Oy. <https://library.e.abb.com/public/300401693e9d477cb33951c1a9a9b6a4/FI_ACS480_ctrl_prg_FW_C_A5.pdf>. Accessed 12 February 2020.

Test Results

Decision mutation:	D	Mutation not found:	
Statement mutation:	S	Mutation found:	

Test case	Parameter group	Mutation	Behavior change
1	6. Control and status words	D	Drive status word 1, enabled bit behavior reversed
		S	Drive status word 1, enabled bit at false state
2	6. Control and status words	D	Drive status word 1, inhibited bit behavior reversed
		S	Drive status word 1, inhibited bit at false state
3	6. Control and status words	D	Drive status word 1, ext1 active bit behavior reversed
		S	Drive status word 1, ext1 active bit at false state
4	6. Control and status words	D	Drive status word 1, ext2 active bit behavior reversed
		S	Drive status word 1, ext2 active bit at false state
5	6. Control and status words	S	Drive status word 1, start request bit at false state
6	6. Control and status words	S	Drive status word 1, ready to start bit at false state
7	6. Control and status words	S	Drive status word 1, running bit at false state
8	6. Control and status words	S	Drive status word 1, started bit at false state
9	6. Control and status words	S	Drive status word 1, modulated bit at false state
10	6. Control and status words	D	Drive status word 1, limiting bit behavior reversed
		S	Drive status word 1, limiting bit at false state
11	6. Control and status words	S	Drive status word 1, local control bit at false state
12	6. Control and status words	S	Drive status word 2, ID run done bit at false state
13	6. Control and status words	S	Drive status word 2, magnetized bit at false state
14	6. Control and status words	D	Drive status word 2, start delay active bit behavior reversed
		S	Drive status word 2, start delay active bit at false state
15	6. Control and status words	S	Drive status word 2, speedControl bit at false state in vector m
		S	Drive status word 2, speedControl bit at false state in scalar m
		S	Drive status word 2, speedControl bit at false state in scalar m
16	6. Control and status words	D	Drive status word 2, safe ref active bit behavior reversed
		S	Drive status word 2, safe ref active bit at false state
17	6. Control and status words	D	Drive status word 2, last speed active bit behavior reversed
		S	Drive status word 2, last speed active bit at false state
18	6. Control and status words	D	Drive status word 2, emg stop failed bit behavior reversed
		S	Drive status word 2, emg stop failed bit at false state
19	6. Control and status words	S	Main status word, ready to switch on bit at false state
20	6. Control and status words	S	Main status word, ready run bit at false state
21	6. Control and status words	D	Main status word, above limit bit behavior reversed
		S	Main status word, above limit bit at false state
22	6. Control and status words	D	Main status word, user bits bit behavior reversed
		S	Main status word, user bits bit at false state

Decision mutation:	D	Mutation not found:	
Statement mutation:	S	Mutation found:	

Test case	Parameter group	Mutation	Behavior change
23	6. Control and status words	S	Main status word, ready ref bit at false state
24	6. Control and status words	S	Main status word, tripped bit behavior reversed
		S	Main status word, tripped bit at false state
25	6. Control and status words	S	Main status word, Off2 Inactive bit behavior reversed
		S	Main status word, Off2 Inactive bit at false state
26	6. Control and status words	S	Main status word, Off3 Inactive bit behavior reversed
		S	Main status word, Off3 Inactive bit at false state
27	6. Control and status words	S	Main status word, switch on inhibit bit at false state
28	6. Control and status words	S	Main status word, warning bit at false state
		S	Main status word, warning bit behavior reversed
29	6. Control and status words	D	Main status word, at setpoint bit behavior reversed
		S	Main status word, at setpoint bit at false state
30	6. Control and status words	S	Main status word, remote bit at false state
31	6. Control and status words	S	Start inhibit status word, Ctrl loc changed bit at false state
32	6. Control and status words	D	Start inhibit status word, EmOff1 bit behavior reversed
		S	Start inhibit status word, EmOff1 bit at false state
33	6. Control and status words	S	Start inhibit status word, EmOff2 bit at false state
34	6. Control and status words	D	Start inhibit status word, EmOff3 bit behavior reversed
		S	Start inhibit status word, EmOff3 bit at false state
35	6. Control and status words	S	Start inhibit status word, auto reset inhibit bit at false state
36	6. Control and status words	D	Fault reset bit behavior reversed
		S	Start inhibit status word, fault reset bit at false state
37	6. Control and status words	D	Start inhibit status word, lost start enable bit behavior reverse
		S	Start inhibit status word, lost start enable bit at false state
38	6. Control and status words	D	Start inhibit status word, lost run enable bit behavior reverse
		S	Start inhibit status word, lost run enable bit at false state
39	6. Control and status words	D	Start inhibit status word, STO bit behavior reversed
		S	Start inhibit status word, STO bit at false state
40	10. Standard DI, RO	S	DI status bit at false state
		S	DI delay bit at false state
41	10. Standard DI, RO	S	RO toggle counter values not updating
		S	RO toggle counter values incorrect
42	10. Standard DI, RO	S	RO1 ON delay not updated
		S	RO2 ON delay not updated
		S	RO2 OFF delay not updated
		S	RO's ON delay not updated
43	10. Standard DI, RO	S	RO source selection not updated
44	10. Standard DI, RO	S	Warning not raised
		S	Warning raised reverse times
45	12. Standard AI	S	AI force value is never checked
		S	AI force value always 0
		S	AI force value always 8

Decision mutation:	D
Statement mutation:	S

Mutation not found:	
Mutation found:	
Test case not applicable with ATF setups:	
Test case not used in daily system testing:	

Test case	Parameter group	Mutation	Behavior change
46	12. Standard AI	D	AI scaled value incorrect, when value is under maximum
		D	AI scaled value incorrect, when value is over minimum value
		S	AI scaled value always 0
		S	AI scaled value always 5
47	12. Standard AI	D	AI scaled value incorrect, when value is under maximum
48	13. Standard AO	S	AO filter time not updated
49	13. Standard A0	S	AO filter time not updated
		S	AO 1 filter time not updated
		S	AO 2 filter time not updated
50	13. Standard AO	S	AO force data not read
51	13. Standard AO	S	AO filter time not updated
52	20. Start/Stop/Direction	D	Warning raised reverse times
		S	Warning Run Enable not raised
53	20. Start/Stop/Direction	S	Enable to rotate not allowed
		D	Enable to rotate allowed wrong time
54	20. Start/Stop/Direction	S	Enable to rotate not allowed
55	20. Start/Stop/Direction	S	Drive not starting in In1 Start Fwd, In2 Start Rev mode
56	20. Start/Stop/Direction	S	Drive not starting in In1 Start mode
57	20. Start/Stop/Direction	S	Drive not starting in In1EdgeStart In2Dir mode
		S	Drive not rotating reverse in In1EdgeStart In2Dir mode
58	20. Start/Stop/Direction	S	Drive not rotating reverse in In1LevelFwd In2Rev mode
59	20. Start/Stop/Direction	S	Drive not starting in In1 Start mode
60	20. Start/Stop/Direction	S	Drive not starting in In1LevelStart In2Dir mode
		S	Drive not rotating reverse in In1LevelStart In2Dir
61	20. Start/Stop/Direction	S	Drive not starting in In1PStart In2Stop mode
62	20. Start/Stop/Direction	S	Drive not rotating reverse in In1PStart In2Stop mode
63	20. Start/Stop/Direction	S	Drive not starting reverse in In1PStartFwd In2PStartRev In3Stop mode
		S	Drive not start foward in In1PStartFwd In2PStartRev In3Stop
64	20. Start/Stop/Direction	S	Start command not set to Ext2
		S	Start command not set to Ext1
65	20. Start/Stop/Direction	S	Ext1 never set to active
66	21. Start/stop mode	S	DC hold status bit behavior reversed
67	21. Start/stop mode	S	DC hold status bit at false state
68	21. Start/stop mode	S	DC hold can not be stopped by DC hold request
69	21. Start/stop mode	S	DC hold status bit at false state
70	21. Start/stop mode	S	DC heating bit at false state
71	21. Start/stop mode	S	DC heating bit at false state
72	21. Start/stop mode	S	Post magn bit behavior reversed
73	21. Start/stop mode	D	Post magn timers behavior reversed
		S	Emergency ramp stop not activated when requested
74	21. Start/stop mode	S	Post magn bit behavior reversed

Decision mutation:	D
Statement mutation:	S

Mutation not found:	
Mutation found:	
Test case not applicable with ATF setups:	
Test case not used in daily system testing:	

Test case	Parameter group	Mutation	Behavior change
75	21. Start/stop mode	D	Emergency ramp stop not activated when requested
76	21. Start/stop mode	D	Emergency stop not activated
		D	Emergency stop activation logic reversed
77	21. Start/stop mode	D	Post magnetization bit at false state
78	21. Start/stop mode	D	Post magnnetization not activated
79	96. System	S	US units in default region
		S	Default units in US region

Metropolia