UNIVERSITY OF VAASA

SCHOOL OF TECHNOLOGY AND INNOVATIONS

AUTOMATION AND COMPUTER SCIENCE

Matilda Wikar

INTELLIGENT SOFTWARE DEVELOPMENT TOOLS

Master's thesis for the degree of Master of Science in Technology submitted for inspection, Vaasa, 7 September, 2019

Supervisor	Jouni Lampinen
Instructor	David Smedman

FOREWORD

Writing this thesis have been an eye-opening experience that I am very grateful to have gotten the opportunity to do. First, I would like to thank my supervisor, professor Jouni Lampinen for his positive and honest attitude that always motivated me to keep on with this thesis. I would also like to thank my instructor, David Smedman at Wärtsilä for supporting me during this process. To Janne Tarsa and Andreas Bäck, both from Wärtsilä, a special thanks to the both for coming up with the topic of this thesis – to Janne for giving me this opportunity, and to Andreas for providing guidance and advice throughout the process.

I would also like to thank my colleagues at Wärtsilä for an inspiring work atmosphere. And finally, my family and friends for always being there for me, thesis work related or not.

TABLE OF CONTENTS

FOREWORD 2

TABLE OF CONTENTS 3

SYMBOLS AND ABBREVIATIONS 6

ABSTRACT 7

TIIVISTELMÄ 8

ABSTRAKT 9

1 INTRODUCTION 10

1.1 Motivation 10

1.2 Thesis overview 11

2 ARTIFICIAL INTELLIGENCE IN SOFTWARE DEVELOPMENT 13

2.1 Artificial intelligence 14

2.2 Soft computing methods 14

2.2.1 Fuzzy logic 15

2.3 Evolutionary computing 15

2.4 Machine learning 16

2.4.1 Types of machine learning 17

2.4.2 Neural networks 17

2.5 Big data and artificial intelligence 18

2.6 Applying artificial intelligence in software development 19

2.7 Control software development team 20

2.8 Team analysis for AI-adaption 21

3 RESEARCH PLAN 23

3.1 Researching available solutions 23

3.2 Research methodology 24

3.3 Discussions as a methodology 24

3.4 Proof of concept methodology 25

4 INTELLIGENT SOLUTIONS IN SOFTWARE DEVELOPMENT 27

4.1 AI-assisted programming 27

4.1.1 Genetic programming 28

4.1.2 Recurrent Neural Networks programming 29

4.1.3 Low-code tools 30

4.1.4 Example: Java code generated by Bayou 30

4.1.5 AI-based assistance for programming 31

4.2 Bug handling tools 32

4.2.1 Bug identification and analysis 33

4.2.2 Bug patching 34

4.2.3 Example: Repairnator bug fix 36

4.3 Intelligent testing 37

4.3.1 Creating test cases with AI 38

4.3.2 Test analysis with AI 39

4.3.3 Example: Unit test case generation with Diffblue 39

4.4 Recommendations for Wärtsilä's team 41

4.4.1 Recommendation for AI-assisted programming 41

4.4.2 Recommendation for intelligent bug handling 42

4.4.3 Recommendation for intelligent testing 42

5 PROOF OF CONCEPT 44
5.1 Motivation 44
5.2 Automatic unit test generation 45
5.3 Theory and requirements 45
5.3.1 Wärtsilä test platform 47
5.4 Algorithm 47
5.5 Dataset for training 49
5.6 Converting data 49
5.7 The neural network 50
5.7.1 Sigmoid function 52
5.7.2 Neural network operations 53
5.8 Program output 53

6 RESULTS AND EVALUATION 54
6.1 Evaluation 54
6.2 Impact on development 54
6.3 Adaption to test driven development 55
6.4 Improvement suggestions 55

7 CONCLUSIONS 57

LIST OF REFERENCES 59

SYMBOLS AND ABBREVIATIONS

AI	Artificial intelligence
ANFIS	Adaptive neuro fuzzy interference system
ANN	Artificial neural network
API	Application programming interface
AST	Abstract syntax tree
ATG	Automatic test generation tool
CNN	Convolutional neural network
GA	Genetic algorithm
GP	Genetic programming
GUI	Graphical user interface
IDE	Integrated development enviroment
PoC	Proof of Concept
RPA	Robotic process automation
RNN	Recurrent neural network
SVM	Support vector machine
TDD	Test-driven development
WMAP	Wärtsilä Modular Application Platform
WTP	Wärtsilä Test Platform

UNIVERSITY OF VAASA
School of Technology and Innovations
Author: Matilda Wikar
Topic of the Thesis: Intelligent Software Development Tools
Supervisor: Professor Jouni Lampinen
Instructor: MA David Smedman (Wärtsilä)
Degree: Master of Science in Technology
Major of Subject: Automation and Computer Science
Year of Entering the University: 2014
Year of Completing the Thesis: 2019 **Pages:** 67

ABSTRACT

Artificial intelligence can be used to automate various tasks, also within software development. The purpose of this thesis was to research what kind of artificially intelligent software development tools and methods are available, and how they could suit a software development team at Wärtsilä. To illustrate this, a recommendation based on available literature was made on how the Wärtsilä team could possibly take intelligent software development tools into use. Based on the findings, a proof of concept regarding software test automation was developed.

First, different types of artificial intelligence were presented, which after a research plan for how to conduct the literature review and the proof of concept was made. To find the most suitable topic for the proof of concept the literature review was first conducted. Focus was set on three areas – artificial intelligence assisted programming, bug handling tools and software testing. Software test automation seemed the most interesting from a Wärtsilä perspective, thus it was selected as the topic for the proof of concept. A prototype of a neural network that analyses C-functions and recommends a unit test based on the similarity to other functions was developed.

Conclusions drawn are that artificial intelligence-based methods in software development has potential, but no tool was found that would directly suit Wärtsilä at this point. Either the tools need to be developed further to suit the field of embedded software development, or Wärtsilä could adapt their way of working for some tool to be taken into usage or develop their own solutions. The proof of concept illustrates how an own solution could be an alternative, if developed further in accordance to the improvement suggestions on how to make it more efficient.

KEYWORDS: Artificial intelligence, software development, software test automation

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö
Tekijä: Matilda Wikar
Diplomityön nimi: Tekoälyä hyödyntävät työkalut ohjelmistokehityksessä
Valvojan nimi: Professori Jouni Lampinen
Ohjaajan nimi: FM David Smedman (Wärtsilä)
Tutkinto: Diplomi-insinööri
Oppiaine: Automaatio- ja tietotekniikka
Opintojen aloitusvuosi: 2014
Diplomityön valmistumisvuosi: 2019 **Sivumäärä:** 67

TIIVISTELMÄ

Tekoälyn avulla voidaan automatisoida monia tehtäviä myös ohjelmistokehityksessä. Tämän diplomityön tarkoitus oli tutkia minkälaisia tekoälypohjaisia työkaluja ja menetelmiä on olemassa ja miten Wärtsilän ohjelmistokehitystiimi voisi näitä hyödyntää. Suositus siitä, miten Wärtsilän tiimi voisi mahdollisesti ottaa tekoälypohjaisia työkaluja käyttöön tehtiin kirjallisuuskatsauksen perusteella. Sen perusteella kehitettiin prototyyppi mikä liittyy ohjelmistotestaukseen automatisointiin.

Ensin esiteltiin erilaisia haaroja tekoälystä, jonka jälkeen luotiin tutkimussuunnitelma ja toteutettiin kirjallisuuskatsaus ja prototyypin kehittäminen. Kirjallisuuskatsauksessa keskityttiin kolmeen osa-alueseen – tekoälyllä avustettu koodaus, ohjelmointivirheiden hallinta ja ohjelmistotestaus. Näistä ohjelmistotestaus valittiin prototyypin aiheeksi, koska Wärtsilän tiimin tavoitteiden perusteella se koettiin parhaaksi. Prototyyppinä kehitettiin neuroverkko joka analysoi C-kielessä kirjoitetut funktiot ja suosittelee toisen, samankaltaisen funktion perusteella yksikkötestin testausta varten.

Johtopäätöksenä todetaan että tekoälypohjaisilla työkaluilla on potentiaalia, mutta tällä hetkellä ei löydy työkaluja joita voisi suoraan ottaa Wärtsilän käyttöön. Joko työkalut pitäisi kehittää niin, että ne sopisivat myös sulautettujen järjestelmien tuotantoon, tai työtavat pitäisi sopeuttaa työkalujen mukaan, tai kehittää omia ratkaisuja. Työssä esitetään yksi ratkaisu, ja mainitaan parannusehdotuksia, joiden avulla saavutetaan vielä parempi tehokkuus.

AVAINSANAT: Tekoäly, ohjelmistokehitys, automatisoitu ohjelmistotestaus

VASA UNIVERSITET
Enheten för teknik och innovation
Författare: Matilda Wikar
Arbetets titel: Intelligenta verktyg för programvaruutveckling
Handledare: Professor Jouni Lampinen
Instruktör: FM David Smedman (Wärtsilä)
Examen: Diplomingenjör
Huvudämne: Automation- och datavetenskap
Universitetsstudierna inleddes: 2014
Arbetet färdigställdes: 2019 **Pages:** 67

ABSTRAKT

Artificiell intelligens kan användas för att automatisera olika uppgifter, också inom programmering. Målet med detta examensarbete var att undersöka hurudana intelligenta verktyg och metoder som finns, och hur dessa kunde användas av ett mjukvaruutvecklingsteam på Wärtsilä. Baserat på litteratur gjordes en rekommendation för hur teamet på Wärtsilä eventuellt kunde ta sådana verktyg i bruk. Efter litteraturgranskningen utvecklades en prototyp för automatisk generation av mjukvarutester.

Först presenterades olika typer av artificiell intelligens, varefter en forskningsplan gjordes upp för hur litteraturgranskningen skulle genomföras och för hur prototypen skulle utvecklas. För att hitta det mest lämpade användningsområdet för prototypen, krävdes att litteraturgranskningen gjordes först. I den fokuserades det på tre skilda ämnen – programmering, bugghantering och mjukvarutestning assisterade av artificiell intelligens. Intelligent mjukvarutestning verkade mest intressant och lämpligt att utforska mera för Wärtsiläs team. Prototypen som utvecklades var ett neuralt nätverk som analyserar C-funktioner och rekommenderar ett enhetstest baserat på likheterna till andra funktioner.

Slutsatsen av arbetet var att användadet av metoder baserade på artificiell intelligens inom programmering förvisso har potential, men att i detta skede hittades inget verktyg som skulle passa Wärtsilä direkt. Antingen krävs att verktygen utvecklas för att passa industrin, att Wärtsilä anpassar sitt sätt att arbeta eller utvecklar egna intelligenta lösningar. Prototypen som utvecklades illustrerade hur en egenutvecklad lösning kunde vara en möjlighet, om den skulle utvecklas enligt förbättringsförslagen som beskrivs så att den skulle bli mera exakt och effektiv.

NYCKELORD: Artificiell intelligens, programutveckling, automatiserad testning

1 INTRODUCTION

1.1 Motivation

This thesis was written for Wärtsilä's Automation & Controls department, that develops embedded control systems for Wärtsilä engines. Wärtsilä is an energy sector company, whose engines are produced for power plants and the marine business. Today's engine technology requires not only an efficient engine, but a complex control system. Wärtsilä's engine control system is designed to be highly reliable since it is operating in a demanding engine environment. (Wärtsilä 2019.) From a software point of view, it means that the control software is complex and requires a lot of development resources. Automating repetitive tasks and eliminating manual labor could relieve developers of some workload. Therefore, the purpose of this thesis is to investigate software development tools based on artificial intelligence that could assist the developers in the process of developing engine control software. The question to be answered is whether the team at Wärtsilä could use artificial intelligence-based tools to do this.

The reasons for integrating an intelligent tool or method into the software development are many, but the need for alternative solutions to performing repetitive or too large tasks amongst the developers is one. Getting rid of such tasks would allow the developers to focus on the more demanding development work. When short on workforce, it could possibly also increase the efficiency and quality in the development team, if an intelligent bot or software would be able to assist the developer with more manual work, leaving the developer to focus on solving other tasks.

Artificial intelligence has during the recent years become a huge influence in many areas and has reached the point when it sometimes outnumbers human reasoning. Artificial intelligence serves as an "umbrella term" for branches such as machine learning, genetic algorithms and fuzzy logic. Some examples of applications of artificial intelligence are big data analysis, natural language processing and robotics, but it has also been suggested to be implemented within software development. Within the term "automatic

programming", falls applications such as intelligent compilers, verification methods and debugging. (Nilsson 1980). This thesis investigates the theory behind such applications, evaluates the need for such applications, and the suitability for the Wärtsilä software development team. To illustrate how an intelligent tool could work, a proof of concept is developed for the engine control system development team in the second part of the thesis.

1.2 Thesis overview

In the first section of the thesis, general theory about artificial intelligence and how it can be used within software development is presented. Central terminology and concepts within artificial intelligence are presented shortly. The way of working in the Wärtsilä team is presented and analyzed to identify areas that could benefit from implementation of an intelligent development tool.

The third chapter presents a research plan for investigating the different artificial intelligence-based methods and technologies that can be applied in software development. For the proof of concept development, a software development plan is selected and described.

In the fourth chapter, already existing solutions are presented and investigated with the purpose of deciding what would be the most promising solution. Three main areas are investigated – AI-based programming, bug handling methods and testing. The chapter is concluded with recommendations for how the Wärtsilä software development team could take AI-based solutions into use.

After reviewing the different areas, the most promising one, intelligent testing was selected for further investigation for the proof of concept topic. Chapter five describes the theory behind the proof of concept, which is a neural network that could serve as an automatic unit test generation tool.

Chapter six describes the result of the development and evaluates the outcome. The impact on the development is evaluated as well as the suitability of the prototype. Improvement suggestions are also made, should the work be continued with the proof of concept implementation after this thesis.

The thesis is concluded in the final chapter, with results and observations. The stage of artificial intelligence software development tools for the embedded industry is discussed, as well as the suggestions for how the team should continue with intelligent software development tools.

2 ARTIFICIAL INTELLIGENCE IN SOFTWARE DEVELOPMENT

In this chapter, the use of artificial intelligence in software development, its background and current state is presented. Some artificial intelligence techniques that are central when applying artificial intelligence in software development are presented shortly. The current way of working in the engine control system software development team at Wärtsilä is also described and investigated, with the purpose of mapping the development areas that could benefit from an AI-based solution.

With artificial intelligence, the purpose is to replicate human intelligence, so that a machine or software can act in a way that a human would describe it as intelligent (Jones 2003: 1-2). Artificial intelligence has in the past years grown to be a common component in many areas. It can be found in various fields, such as medicine, the game industry and economics, and of course in computer science. (Kulkarni & Padmanabham 2016.) Therefore, it is natural that it would also be used in software engineering and development. Integrating intelligent assistants for software developers is however not a recent idea. Intelligent assistants for software developers were discussed and proposed already in the early 1970's, when programming was nowhere nearly as developed as it is today (Winograd 1973). Winograd (1973) presents what he called "obvious examples" of how a programmer could benefit from an artificially intelligent assistant. He suggested that error checking and debugging would become more automated and intelligent, as well as the way of searching for information, and an improved overview of the program so that the programmer would not have to keep everything in mind themselves. Today, his predictions have become true in a sense – we have powerful debuggers and Integrated Development Environments (IDEs) that can suggest how to resolve errors and generate functions with automatic parameter handling.

Automating repetitive tasks is both effective time-wise, but also a good way of securing quality. Human errors can be avoided if a reliable computer is put to work instead, as the computer won't get bored – and will also sometimes perform the much faster than a human (Xie 2013). A study performed amongst software developers showed that software developers find the need for assistance within three main areas of the software

development (Rech, Ras & Decker 2007). There were multiple reasons for wanting an intelligent assistance tool – the main one was the complex nature of software development. Programming, designing and handling requirements were examples of time consuming and challenging tasks. The first purpose for intelligent assistance tools was to automate simple and repetitive tasks, the second need was for assistance with visualization of the system under development and the third one was related to the interaction between different parties within the development team. (Rech et al. 2007.)

2.1 Artificial intelligence

Different methods of implementing artificial intelligence have different benefits and thereby are more suitable for some applications. There are many branches within artificial intelligence, such as machine learning, data mining, evolutionary computing and fuzzy logic. (Jones 2003; Shi 2011.)

The following subchapters describes some of these branches that have been found beneficial in applying artificial intelligence in software development, with examples of how they can be applied. The concept of big data is also reviewed, since using big data together with artificial intelligence is a common approach. (Kersting & Meyer 2018)

2.2 Soft computing methods

Soft computing is an umbrella term for artificial intelligence methods such as fuzzy logic, neural networks and probabilistic reasoning. The common factor for soft computing methods is that the methods tries to replicate the human minds way of thinking and reasoning. (Zadeh 1996.)

2.2.1 Fuzzy logic

Fuzzy logic resembles human reasoning; instead of something being simply true or false, there is an infinite amount of variations between one and zero. A fuzzy set has a membership function, that shows where on the range between true and false something lies. (Jones 2003: 235; Nguyen, Walker & Walker 2019: 2-3). Fuzzy logic has proven to be suitable for usage in control and expert system and in replicating human thinking. (Jones 2003: 254).

Fuzzy logic could efficiently be integrated to software development processes to help complete and define for example poorly defined requirements and situations when information is lacking. Another example of how fuzzy logic can be implemented is to model software reliability. (Harman 2012.)

2.3 Evolutionary computing

Evolutionary computing is a subfield of artificial intelligence, in which the behaviour of nature – evolution and natural selection is mimicked. Evolution can be viewed as a cycle, starting with new generations being born, resulting in populations of species that have their own traits. Only the fittest individuals survive, resulting in their genes being passed on to the next generation, and so the loop continues. (Siddique & Adeli 2013: 183-185.)

Genetic algorithms are an evolutionary computing method that follow a few steps (see figure 1). First, the population is initialized where the population is created. The population consists of chromosomes, that are created with random traits. Evaluation means that the chromosomes are evaluated based on how well they solve a given problem. (Jones 2003: 115-120.)

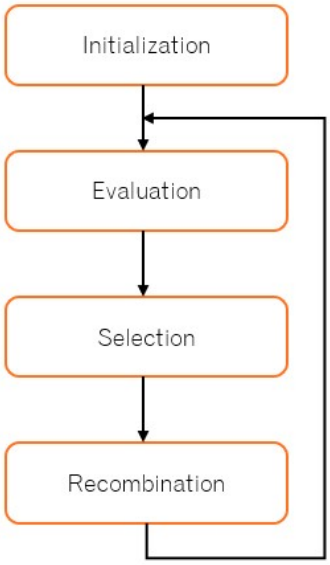

Figure 1. Basic architecture of a genetic algorithm – initialization, evaluation, selection and recombination of the genes in the population (Jones 2003: 116).

To measure how well a chromosome performs, a fitness value is calculated for each chromosome. Based on this fitness, the most fit chromosomes are selected for recombination, where chromosomes are recombined. The recombination can happen with for example methods like crossover or mutation. (Jones 2003: 115-120.)

2.4 Machine learning

With machine learning, a system is trained upon a set of data, to perform some activity. When the system has learnt how to deal with known data, it can act upon unknown, new data and thereby perform complex tasks (Loudiras & Ebert 2016).

Machine learning can also be used within software development processes for cost and time planning, and even to predict faults and thus it can serve as technology for a bug handling tool (Harman 2012; Jonsson 2018). There have also been studies where machine learning has been used to predict the resources a certain software development project

would need, so machine learning can also be used to plan activities (Wen, Li, Lin, Hu & Huang 2011).

2.4.1 Types of machine learning

Machine learning can generally be divided into supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning. (Mohri, Rostamizadeh & Talwalkar 2012: 7)

In supervised learning the algorithm processes a training dataset that has been labelled so that the algorithm can learn from it. Supervised learning is a typical solution when classifying and ranking problems. Unsupervised learning works in the same way, making predictions, but without labelled data, making it more unreliable. (Mohri et al 2012: 7.)

Semi-supervised learning is a combination of supervised and unsupervised learning, where the dataset introduced to the algorithm is a mix of both labelled and unlabelled data (Mohri et. Al. 2012: 7).

Reinforcement learning learns from its environment, meaning that the algorithm learns from what it has experienced continuously by receiving feedback from the environment (Mohri et. Al. 2012: 8, 313-216). Reinforcement learning algorithms can be found in technologies such as robotics and self-driving cars (Fumo 2017).

2.4.2 Neural networks

A neural network tries to imitate functions in the human brain. A brain has neurons, that are linked together via axons. At the end of the axon, a synapse serves as a transmitter for the signals to other neurons. In an artificial neural network, a neuron is called a perceptron (see figure 2). ANNs are structured in the same way – only much simpler than a brain. This way, ANNs can process data and learn from it. (Jones 2003: 83-85; Shiffman 2012.)

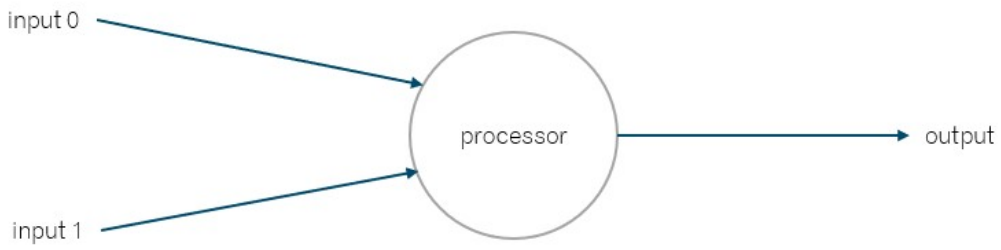

Figure 2. The simplest neural network is one single perceptron. A perceptron is an artificial representation of the neuron in the brain. (Shiffman 2012.)

A perceptron consists of inputs, one or multiple, a processor and an output. Each input has a weight so that it can be evaluated. (Shiffman 2012).

2.5 Big data and artificial intelligence

The term big data can be explained as large volumes of data, generated at high speed and that consists of different types of data (Marr 2015: 79-80). Big data and artificial intelligence can be applied together, as artificial intelligence can be applied on big data for various implementations, like recognizing patters in the data or the data serving as a training data for machine learning algorithms. (O'Leary 2013). Any data is unusable until analysed or investigated further, so artificial intelligence is an excellent tool for extracting something valuable out of a big data set (Ghahramani 2015).

Big data collected in software development processes can be used as a base for machine learning to be trained upon, resulting in optimized employee efficiency or identifying bottlenecks in the software development process (Varhol 2017).

2.6 Applying artificial intelligence in software development

Software development can traditionally be viewed in four main phases – planning, development, testing, and release (see figure 3) (Harman 2012). Artificial intelligence, integrated as a development tool or technique into any of these phases, can have different purposes and benefits. In practice, it would be most beneficial to implement artificial intelligence in generating specifications, programming, debugging and testing (Xie 2013).

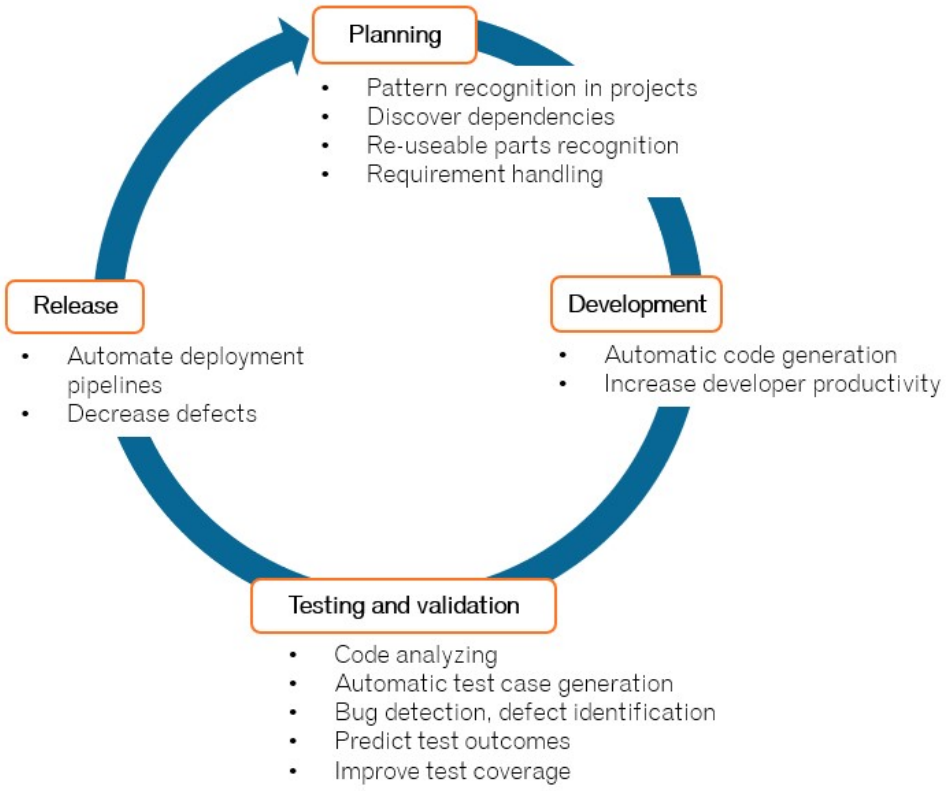

Figure 3. Software development in four phases – planning, development, testing and release. In each phase, artificial intelligence can be used for a different purpose (Harman 2012).

According to Giudice (2016), the testing phase is where artificial intelligence would be most beneficial. Test cases written in the traditional way, by humans, are dependent on the intelligence of the programmer. Test cases generated by artificial intelligence can be

created against the testing requirements, increasing the reliability (Xie 2013). Artificial intelligence can analyse code, generate test cases automatically and increase the test coverage. It can also recognize bugs and predict the outcome of tests. (Giudice 2016; Harman 2012)

Generating code with artificial intelligence would mean that the program can create itself by the specifications and requirements made. Using artificial intelligence to generate code, would mean that the user would only have to provide the program synthesis, and the syntax is then handled by the artificially intelligent tool. (Xie 2013.)

Harman (2012) divides the usage of artificial intelligence in software engineering into three different general techniques – computational search and optimization techniques, fuzzy and probabilistic methods for reasoning, and classification, learning and prediction. Harman (2012) also describes some challenges that AI in software development must overcome to reach its potential. One challenge within AI for software development is, that if used to resolve problems, the AI technique cannot fully provide a strategy for solving an issue. Also, it can be questioned whether AI techniques should be adapted to an already working process, if not necessary. (Harman 2012.)

2.7 Control software development team

The control software development team consists of both application and system developers. The team has adapted an agile way of working and is implementing DevOps practices into their development philosophy. This means that the team is striving to change their processes to increase software release time by adapting better practices regarding testing, solving issues and managing projects. (Atlassian 2019.)

The team is responsible for developing and maintaining control applications and configuring engine software configuration. Application developers develop control applications with a specific purpose, like speed and load controller, fuel mode control and ignition control, that are needed to control an engine. The control applications are integrated in a

software package, that can be configured to fit a specific engine. Applications are written in C language or generated to C code from a MATLAB Simulink model. An application developer is responsible for the application development, unit and white-box testing and documentation related to the application. (Wärtsilä 2019b.)

A system developer, or a system integrator, integrates the application components into a software package, which can be configured specifically for an engine. A system developer also has the responsibility of managing the engine configuration project, create the necessary documentation, and release the configuration so that it can be deployed in the field. A Wärtsilä UNIC system is a complex system that includes safety configuration, instrument configuration, communication and user interfaces. (Wärtsilä 2019b.)

The control software development team is hereafter in the thesis referred to as “the team”.

2.8 Team analysis for AI-adaption

To analyse and find areas within the engine control system development team, each development activity was investigated with the purpose of finding a suitable artificial intelligence-based solution to implement (see figure 4). This model was derived from the illustration of how software development could benefit from artificial intelligence assistance tools, but the analysis focused on what seemed most relevant for the team.

Within the implementation and testing phases, both the application programming and the testing could be aided by intelligent software solutions. Also requirement handling, documentation and the release process could benefit from automated tasks, however, such solutions may not have to be artificially intelligent, but could be solved by Robotic Process Automation (RPAs) for example.

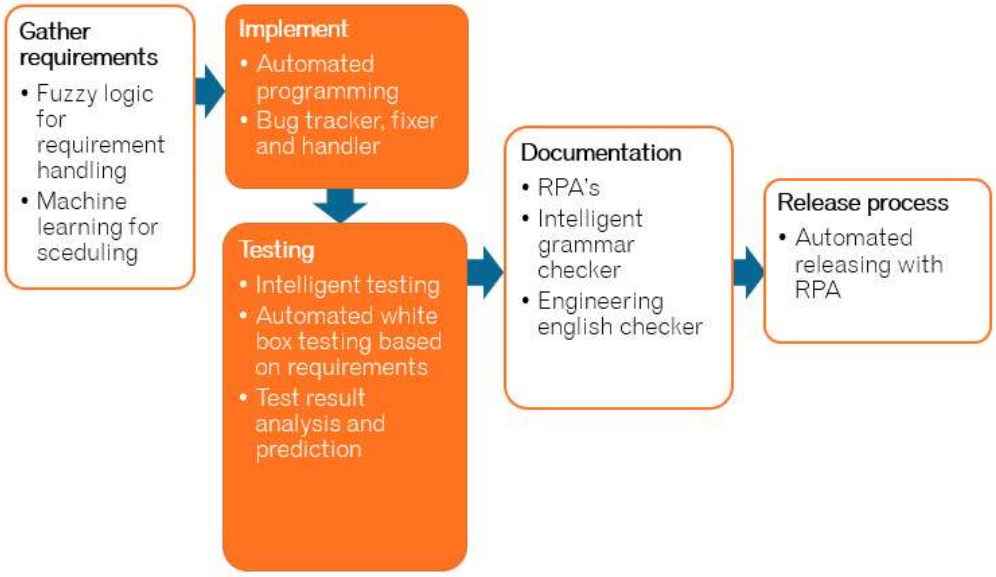

Figure 4. The different development phases for an application developer in the engine control system development team. In each phase, some examples of how artificial intelligence could be used is found. The examples in orange are those which seem especially suitable for the Wärtsilä software development team.

The team is adapting a test-driven development attitude, meaning more efforts will be focused on testing the engine control system software. The next chapter will investigate three different artificially intelligent based solutions to provide for better software – AI-assisted programming, bug handling, and intelligent software testing methods. The proof of concept developed and presented later in this thesis is related to some of these areas, depending on the research findings and which of the alternatives seem to bring most value to Wärtsilä's software development process.

3 RESEARCH PLAN

This chapter describes the workflow of the following chapters, so that the research stays within scope and the outcome of the thesis remains clear. The methodologies used in the research are described. It also describes the way the research is carried out, starting from researching the available solutions and how they could be implemented by the Wärtsilä software development team. Based on the findings of the researched technologies and the conducted analysis of the team, a proof of concept is designed and implemented.

3.1 Researching available solutions

The research conducted in this thesis is carried out as follows. First, a review of the available solutions and how they could be implemented at Wärtsilä is made. This research is based on the findings in various scientific articles, magazines, web-pages and other relevant material. The aim is to investigate three main areas within artificial intelligence in software development. These three areas are selected based on the analysis of the Wärtsilä software development team in the previous chapter. Automatic programming, software bug handling and intelligent testing are the fields that will be described further.

The technologies presented shall be suitable for the team to implement, and it shall be kept in mind that the aim of this research is to provide the team with recommendations on whether AI-based software development tools are something to investigate and pursue further. For each technology investigated, the question of why such a technology should be used shall be answered. Some background information of why it would be beneficial to implement such an AI-based solution, so that usage of the solution into use is well-motivated. The advantages and possible disadvantages of each technology is presented and discussed.

The theory behind each technology shall be presented. Preferably, different solutions of the implementation of one technology shall be compared so that it becomes clear what the available approaches to each solution are. After the theory behind each method is

presented, examples of already existing frameworks and commercial tools that implements the technology described. The existing tools may not be fully ready for integration with the way of working in the team, but in that case a recommendation on how it could be integrated to their process shall be made. Each subchapter shall be concluded with a recommendation for the team on how to progress with the presented technology.

3.2 Research methodology

The methodology mostly used in this thesis is reviewing literature found on the topic. The right sources for the literature review are found by searching for scientific articles, books and other sources on the internet and at the university library. The search for appropriate material starts with finding literature related to the three main areas – automatic programming, bug patching technology and intelligent testing. Using the university library's own database and google.com and scholar.google.com, books and articles, both in physical form and online can be found. Screening these articles, some important keywords can be sorted out and further searching is done based on these. Such keywords and search words are for example *"genetic programming", "artificial intelligence + bug patching",* and *"generating test cases + software engineering + machine learning".*

For each main area, the goal is to find a tool that can be investigated further to better visualize how each technology can be applied. Familiarizing with the tools gives a better understanding of how the they work and perform. Thus, each technology can be evaluated according to Wärtsilä's needs.

3.3 Discussions as a methodology

Discussions, or unstructured interviews, will be used as a methodology for finding the most promising topic for the proof of concept product. The discussions will take place as meetings with the instructor from Wärtsilä, and possibly also other parties from Wärtsilä that can provide insight to the processes followed by the team. The discussions will

provide better understanding of Wärtsilä's needs, their processes and way of working, and keep this project up to date on in what direction the development is heading. This way an informed decision on what to pursue as a topic for the proof of concept developed later can be made.

3.4 Proof of concept methodology

After the different methods and technologies have been investigated, the implementation of an application related to one of these areas is demonstrated. The topic for the proof of concept product is chosen based on what seems most beneficial for the team to implement.

Regarding the methodology for developing the proof of concept product, the PoC will be developed in a cyclic model, that will be inspired by the extreme programming software development methodology (see figure 5).

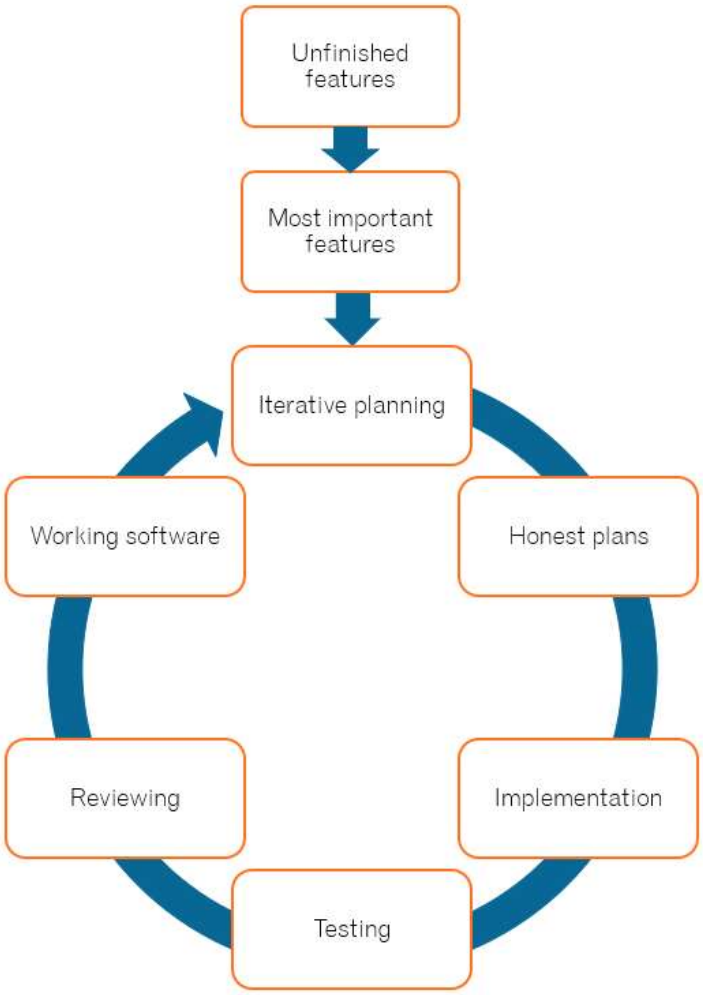

Figure 5. Development process for the proof of concept influenced by the extreme programming method (Wells 2013).

The extreme programming methodology can be scaled down to a single developer project, which will suit this thesis project as it will be done independently, with a clear deadline and loose requirements. This will suit the idea of developing a proof of concept, as the prototype can be improved further and further if time remains. Another benefit of the extreme programming method is that it allows for adjustments of the requirements while developing the product (Wells 2013). This will allow for constant improvement of the proof of concept until the deadline for the project is reached. The prototype must not be a fully usable tool, but it shall demonstrate how an artificial intelligence-based solution would fit the needs of the Wärtsilä software development team.

4 INTELLIGENT SOLUTIONS IN SOFTWARE DEVELOPMENT

This chapter presents the theory behind the three application areas for artificial intelligence solutions within software development. Methods for artificial intelligence assisted programming tools and other "intelligent" ways to generate executable code are investigated. Bug handling, from detection to patching, with help of some AI-method is reviewed, as well as different methods for intelligent testing of software. Other AI-based solutions suitable and interesting for the engine control system development team are shortly reviewed in the last section. Why and how such solutions would be applied, and the benefits and disadvantages of different methods are presented for each area of application. Each subchapter contains examples of methods and tools for implementation, and the chapter is concluded with recommendations for if, and how the engine control system development team could utilize the solutions. The topic for the proof of concept developed later in this thesis is selected based on the findings of this chapter.

4.1 AI-assisted programming

Predictions have been made, that in twenty years, technology will have advanced so that human programmers could be replaced. Automatic code generation tools are already available and to some extent usable (Billings, McCaskey, Vallee & Watson 2017). Unintelligent code generation tools like MATLAB Simulink allows the user to create code from a model, but the next step would be having artificial intelligence creating code on its own based on the user input. Some businesses have already invented this kind of technology, with tools like Bayou, that uses AI-methods like neural networks to generate Java API code according to the users input (Bayou 2018).

There are different ways of generating code with artificial intelligence, which are described in the following chapters.

4.1.1 Genetic programming

Genetic programming is a form of automatic programming with artificial intelligence. It is based on evolutionary algorithms that can generate solutions to problems automatically (Rouse 2005). Genetic programming has also been used to optimize existing code (Langdon & Harman 2015).

Genetic programming is an evolutionary algorithm, that aims to evolve a program so that it can produce a solution to the problem. One benefit of genetic programming algorithms is that the user does not have to be able to formulate the program specifically, but the program evolves itself to be able to produce a solution. This would make genetic programming suitable for automatic code generation, as the program should generate code based on the user input that may not be more than roughly specified. (Poli, Langdon & McPhee 2008.)

The generation program resembles a genetic algorithm. First, a set of random programs are created. Then the programs are run and evaluated based on how well they perform. Fitter programs are bred by crossover and mutation. Instead of representing the program under development as code, genetic programs are often represented in the form of a syntax tree, where each decision branches off and every choice is represented as a branch (see figure 6). (Poli et. Al. 2008:2.)

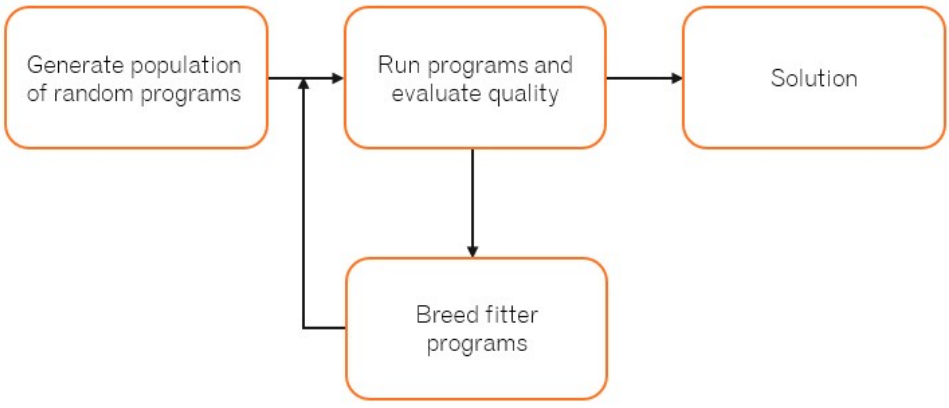

Figure 6. General workflow of genetic programming. The “survival of the fittest” method is used to find solutions. (Poli et. Al. 2008:2.)

Regarding the performance of genetic programming, Igwe & Pillay (2013) developed and tested a genetic programming algorithm for generating code that was successful in solving basic programming problems.

4.1.2 Recurrent Neural Networks programming

Recurrent neural networks (RNN) differ from regular neural networks in the sense that they have loops, meaning that they consider what they learned previously by analysing. This means that each step is linked together (see figure 7). (Banerjee 2018.) RNNs have been successful in speech recognition, modelling languages and generating images, but they have also been suggested as a tool for generating code (Spirina 2018).

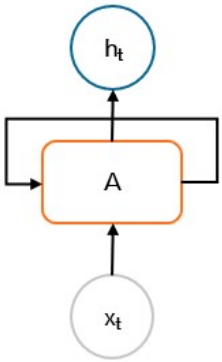

Figure 7. A recurrent neural network works in loops, allowing it to consider what is already has learnt before moving on (Banerjee 2018).

Using RNNs to generate code is a young but promising technology. RNN-generated code has been successful in designing graphical user interfaces (GUI). In a solution called pix2code, the code for the GUI was generated using a mock-up picture of the GUI, by a combination of RNNs and Convolutional Neural Networks (CNN). (Beltramelli 2017.)

4.1.3 Low-code tools

Low-code development tools remove the need for actual programming by having the developer create the software with help of a graphical user interface, to structure the data and logic as it would be in the developed application. In addition to this, low-code tools can also include some programming to for example integrate the application to a system or make it backwards compatible. Low-code tools are already on the market for both novices and professional use. (Boulton 2019.)

Low-code tools themselves are not intelligent, even though they can reduce the developer workload by automating the hand-written code development. However, should artificial intelligence be added to the low-code tool, the tool could aid with solving more challenging issues on its own, for example by suggesting the design of more complex actions (Bloomberg 2017.)

4.1.4 Example: Java code generated by Bayou

Bayou is a program that takes calls for Java APIs and type specifications as inputs and then generates a function that uses the called function with the specified type. Bayou uses neural networks to learn how to generate correct Java-code, more specifically neural sketch learning, which trains a Gaussian encoder-decoder network on a large set of code. (Bayou 2018).

After processing the input (see figure 8), the tool generates code that can be executed to to read and close the file (see figure 9).

```
1 import java.io.File;
2 public class Test {
3     void read(File file) {
4         {
5             /// call:readLine call:close
6         }
7     }
8 }
```

Figure 8. Input to Bayou with two API-calls, to read lines from a file and then close it (Bayou 2018).

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.FileReader;
6  public class TestIO {
7    void read(File file) {
8      {
9        BufferedReader br1;
10       FileReader fr1;
11       String s1;
12       try {
13         fr1 = new FileReader(file);
14         br1 = new BufferedReader(fr1);
15         s1 = br1.readLine();
16         br1.close();
17       } catch (FileNotFoundException _e) {
18       } catch (IOException _e) {
19       }
20       return;
21     }
22   }
23 }
```

Figure 9. The code generated on the queries in figure 8. (Bayou 2018)

Bayou would perhaps help a skilled Java programmer that knows what APIs to call and how to structure them, but at this stage it does not aid the developer much. Further development may however progress the tool.

4.1.5 AI-based assistance for programming

Microsoft is currently working on an AI-assisted programming for C++, C#, Java, JavaScript and XAML. The tool, called IntelliCode, is integrated as an extension in the Visual Studio IDE. IntelliCode features and builds upon the code completion tool called IntelliSense, that has previously have been available for Visual Studio. IntelliSense predicts and suggests APIs based on the programmers already developed code, instead of providing suggestions in a classical alphabetical list. (Microsoft 2019; Lardinois 2019.) The AI behind IntelliSense has been trained upon highly rated open-source code on Github (Lardinois 2019).

4.2 Bug handling tools

Bugs, meaning faults in the software – are unavoidable in software development. Bugs are not only costly, but also time-consuming, as they require a lot of resources (Hammouri, Hammad, Albadhan & Alsarayah 2018). Bug handling is a process that starts with the bug being found and reported. Then it is analysed by for example the software developers, that fix the bug to eliminate the reported issue. Thereafter, the bug fix needs to be verified, and when the solution is accepted, the software needs to be re-commissioned to the customer. (Jonsson 2019: 8-9.)

Predicting and identifying software faults is an application that can be implemented with artificial intelligence and that can serve as a useful tool in software development. The main benefits of predicting software bugs are that it can increase the system quality, predict the need for software maintenance and make the software development process faster as it serves as a code review tool at the same time (Erturk & Sezer 2014). Software fault prediction software has been implemented using machine learning, specifically artificial neural networks (ANN), Naïve Bayes, Decision Trees, Adaptive Neuro Fuzzy Interference System (ANFIS) and support vector machine (SVM) (Erturk & Sezer 2014; Hammouri et al. 2018.)

Patching faulty code by using automatic bug patching software may reduce the time spent on manually fixing the code. Bug patching bots are designed to help developers become more efficient and to increase software quality (Urli, Seinturier, Yu & Monperrus 2018). Both Jonsson (2018) and Tufano (2018) suggest machine-learning based solutions, where a bug fixer software is trained upon already made bug fixes, so that it can generate fixes for new bugs. A project conducted by Urli et al. (2018) resulted in a bug repair software called Repairnator that identifies and suggest bug fixes to the developer.

4.2.1 Bug identification and analysis

Before artificial intelligence can be used to fix a bug in the code, the bug must be predicted, detected or identified. A common approach to preventing bugs is keeping code reviews, where other members of the team go through the code and gives feedback. Reviewing the code can also be automated with so-called static analysis tools.

DeepCode is a code review tool that differs from traditional static code analysis tools as it uses artificial intelligence to analyse the code. The technology behind DeepCode is based on machine learning algorithms and is not bound to any specific programming language (see figure 8). (DeepCode AG 2019.)

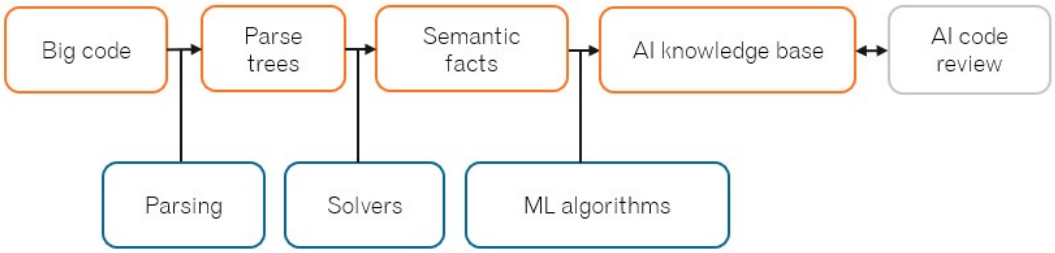

Figure 10. The workflow of the DeepCode code analysing tool (DeepCode AG 2019).

DeepCode converts the "Big code" by parsing them into so-called parse trees, meaning that the code loose is no longer bound to any specific language but represents the structure of the code, in the same way an abstract syntax tree would (DeepCode 2019; Miller & Ranum 2014). The code is then analysed in the format of parse trees with a traditional static code analysis. Based on the facts found in the static analysis, machine learning algorithms are run upon to allow the tool to understand structure, function and the purpose of the code, resulting in an AI-assisted code review. The machine learning algorithms are linked together with the AI code review, forming what is referred to as the "AI knowledge base". (DeepCode AG 2019.)

4.2.2 Bug patching

Jonsson (2018) has developed a bug handling tool that uses machine learning. Weimer, Forrest, Le Goues and Nguyen (2010) describe an automatic program repair using genetic programming.

The repair program described by Weimer et. Al. (2010) uses genetic programming to find fixes to bugs. Their software takes three inputs; the C code containing the bug, the failed test case that describes the desired functionality of the code and some other test cases as reference to the program functionality. The program then makes variants of fixed code – so called individuals. Each program variant is represented as an abstract syntax tree (AST), that loosely describes the functionality of the code via a visual representation of the paths and conditions within. Each program variant also has a weighted path – a pair of weights that describes the relationship between the negative test case and the code. Mutation and crossovers happen between the individuals, and the fitness is evaluated. The least fit program variants are deleted by selection when the population is cut in half. The repair is minimized with structural differencing and delta debugging so that the produced code fix will be human interpretable. (Weimer et. Al. 2010.)

The Repairnator project conducted by Urli et. Al. in 2018 combined two different bug patching methods. Genetic programming was used and derived from the evolutionary repair program Genprog (Le Goues, Forrest & Weimer 2019). A program called Nopol was used to fix faulty conditions, like broken if-else statements. (Xuan, Martinez, DeMarco, Clément, Marcote, Durieux, Le Berre & Monperrus 2016).

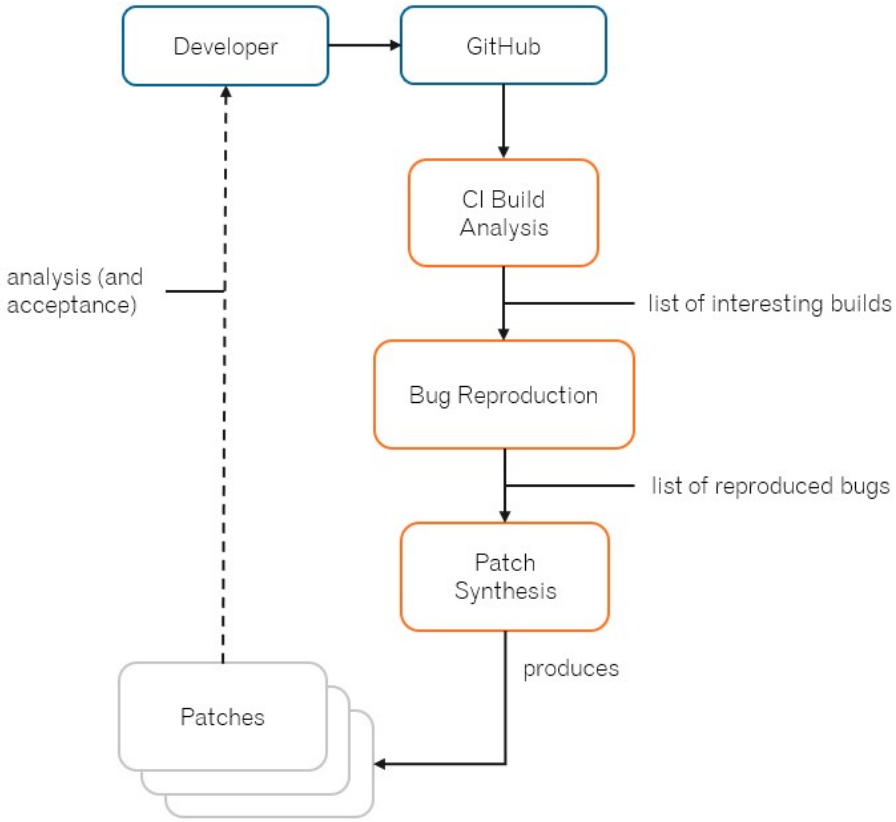

Figure 11. Simplified workflow of the Reparnaitor bug patcher (Urli et. Al. 2018).

The purpose of the Reparnaitor project was to generate bug fixes that would be accepted by human developers, as that would prove the quality of the bug patcher. In the spring of 2018, the Reparnaitor robot succeeded to generate five patches that were accepted by other developers on the Github platform. The other developers were not aware of the Reparnaitor being a robot, but believed it to be another developer, thus they were not biased to rank the patches provided by Reparnaitor any different than they would with a human developer. (Monperrus, Urli, Durieux, Martinez, Baudry, Seinturier 2018.)

4.2.3 Example: Repairnator bug fix

Reparnaitor can be added as a contributor to a project on Github to fix bugs.

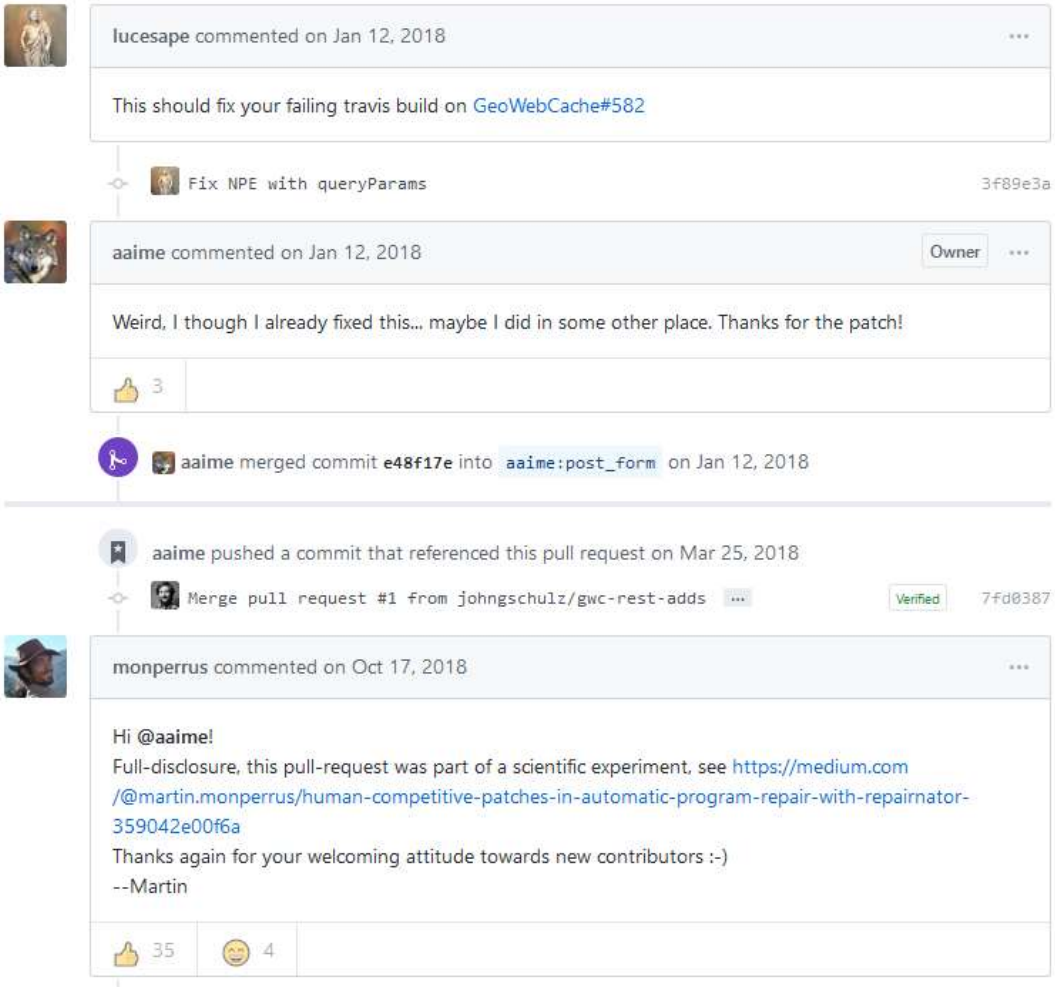

Figure 12. Developers on Github have been unaware of Repairnator contributing as the bot has been working under a fake profile “lucesape”.

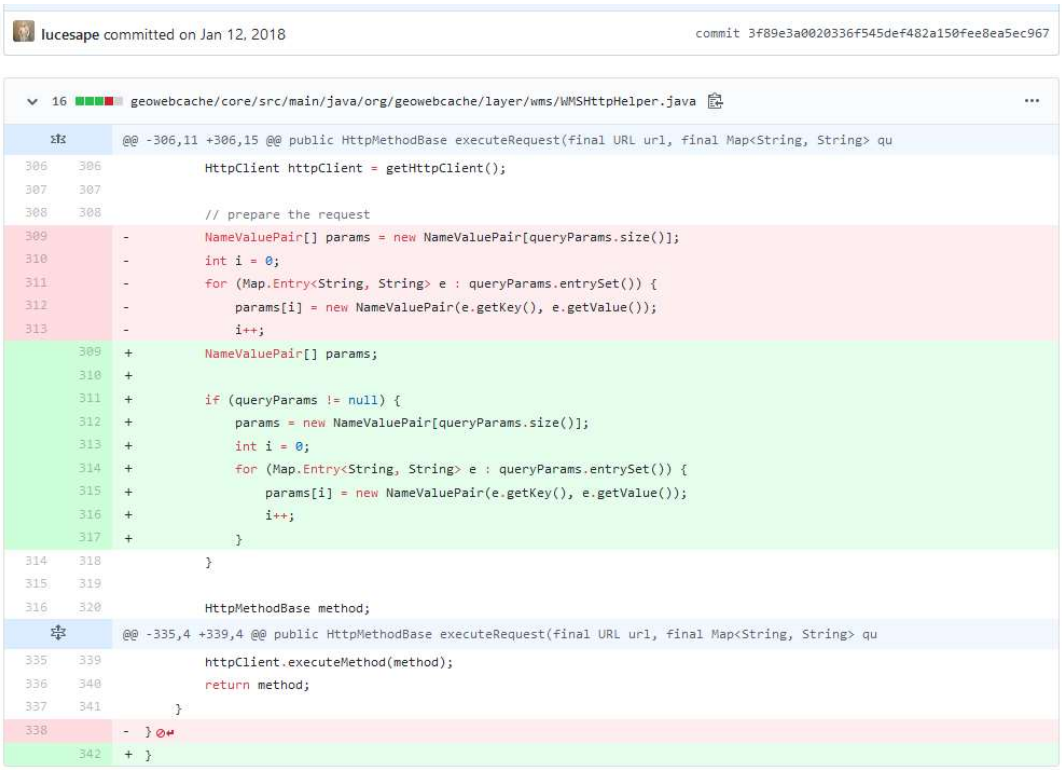

Figure 13. A bug fix proposed by the Github-user lucesape/Repairnator bot.

4.3 Intelligent testing

Software testing itself is a complex activity, which can generally be divided into static and dynamic test techniques. Dynamic test techniques are for example white box and black box testing. (Homès 2013: 143.) Module testing, or unit testing, means testing the different parts of a program separately (Myers, Corey & Badgett 2011: 85). These practices have been adapted in the Wärtsilä software development team, and therefore intelligent tools and methods for automating these practices are investigated further.

Automated software testing means that the testing is automated, so that some repetitive tasks of running the software tests are made obsolete and instead a framework will handle it. However, such tools have their limitations and that is where artificial intelligence can assist. With artificial intelligence, unit tests can be generated, as well as input and the parameters. (Kirilenko 2018; Merrill; Maruti Techlabs). Artificial intelligence can

improve the test coverage, and as the input data is generated, it will be more thorough than a human mind. Based on the analysis of the code, the AI can also determine where there have been changes in the code and thereby test the necessary parts. (Kirilenko 2018.)

There are also a variety of AI-based software testing tools on the market or under development (Colantonio 2017). Many of these are intended for testing of visual systems, or mobile apps or websites, making them unsuitable for testing embedded system software. The technology behind is however interesting, as it could likely evolve in the future.

Creating test cases automatically is one of the areas within software testing where artificial intelligence has been promoted actively. Another area where artificial intelligence can aid developers is analysing test results. (Stadlbauer 2018.)

4.3.1 Creating test cases with AI

Creating test cases means writing the test case functionality, but also selecting or creating proper test data.

Mantere and Alander (2005) describe generating test cases with genetic algorithms and evaluating the test case fitness with help of a neural network. The genetic algorithm generated the test cases, which were passed through the neural network for fitness evaluation, as it communicated with the software under test. Other studies have suggested to generate test data, to increase the code coverage and to detect the faults at the same time (Zhang & Gong 2014).

Evolutionary computing methods have been used for integrating artificial intelligence to software testing, as well as fuzzy logic and machine learning (Mantere & Alander 2005; Last, Kandel & Bunke 2004). Several solutions for generating test data with artificial intelligence have been presented. Genetic algorithms, ant- and bee colonies and swarm optimization can be used to generate test data (Kire & Malhotra 2014; Keyvanpour, Homayouni & Shirazee 2011).

4.3.2 Test analysis with AI

ReportPortal is a test report system that shows and analyses test results. ReportPortal uses a machine learning algorithm to find and predict failures in tests, so-called auto-analysis. With help of the auto-analysis, the cause of the failed test can be detected. The tool can classify the failed test by defect type, link the defect to a bug tracking tool like for example Jira, and generate a comment to the issue. (ReportPortal.) Data from the project is used by the machine learning algorithm to keep it accurate (Colantonio 2, 2018).

4.3.3 Example: Unit test case generation with Diffblue

To illustrate how unit tests can be generated by an existing tool, a short Java-program without no further purpose but to demonstrate how well the selected tool performs, was written. The Java-program is not an engine control application, but a simple program that calculates the average for four integers, and makes a subtraction based on some conditions in the code. A third function sets an "actioncode" based on the outcome of the other functions. The testcode is not part of any engine control software, as no such code or resembling code is to be leaked outside of Wärtsilä.

Diffblue is a research project that aims to create a software that uses artificial intelligence to generate unit test cases for Java-code. Diffblue has a playground open for free testing of code up to 200 lines, which was therefore selected to illustrate how automatic unit test generation could look like.

Diffblue created 12 unit test cases for the tested Java-program. For the function "CalcDiff" (see figure 14), which was an interesting function due to the functions many branches, it created 6 test cases. One issue with the generated test case (see figure 15) is that it looks messy and disorganized, as Diffblue puts the function under test and its parameters in place by its own help-functions. A human programmer can determine this themselves and does not need to fetch parameter datatypes by any APIs. Therefore, the test cases may look disorganized at first glance.

```java
static double CalcDiff(int a, int b, int c, int d) {
    double difference = 0;
    if (a > b && c < d) {
        difference = a-b;
    }
    if (b > c && c < d) {
        double e = c+d;
        if (e > a-c){
            difference = b-a;
        }
        else{
            difference = a-c;
        }
    }
    return difference;
}
```

Figure 14. The observed function inserted to Diffblue. (Diffblue 2019)

```java
// Test written by Diffblue Cover.
@Test
public void CalcDiffInputNegativeNegativeNegativePositiveOutputNegative() throws NoSuchMethodException, IllegalAccessException, InvocationTargetException {

    // Arrange
    final int a = -914_358_279;
    final int b = -982_941_703;
    final int c = -1_006_632_960;
    final int d = 1_157_627_904;
    final Class<?> classUnderTest = Reflector.forName("Calculations");
    final Method methodUnderTest = classUnderTest.getDeclaredMethod("CalcDiff", Reflector.forName("int"), Reflector.forName("int"), Reflector.forName("int"), Reflector.forName("int"));
    methodUnderTest.setAccessible(true);

    // Act
    final double retval = (double) methodUnderTest.invoke(null, a, b, c, d);

    // Assert result
    Assert.assertEquals(-0x1.05ap+26 /* -6.85834e+07 */, retval, 0.0);

}
```

Figure 15. A test case for the function “CalcDiff” (Diffblue 2019).

The test case in figure 15 was generated for the function CalcDiff with the purpose of testing the function with negative inputs. Another five testcases were generated with positive inputs, both negative and positive inputs, zeros as inputs and negative and positive inputs mixed.

4.4 Recommendations for Wärtsilä's team

The three areas that were investigated have showed that the different technologies are still in varying stages of development. As the purpose of this thesis is to make a recommendation for how Wärtsilä could integrate artificial intelligence in their software development, some recommendation based on the findings are made. The recommendations are indicative and provide suggestions for what the team could keep in mind and look out for as technology progresses.

4.4.1 Recommendation for AI-assisted programming

It seems that there are not many existing tools for fully automatic code generation that could be applicable in the Wärtsilä software development team. The tools found were either focused on generating Java-functions (Bayou 2018) or GUI programming, neither of which applies to the work of developing embedded software for engines. However, the descriptions of methods for successfully generating code are likely to be developed further in the future. Perhaps frameworks like these, using GP, RNNs or machine learning will be more common and commercialized in the future, making tools like that also applicable for the heavy industry.

Instead of focusing on automatically generating code successfully with artificial intelligence, as the area is not that mature yet, AI could provide assistance to the person writing the code. This is perhaps a more realistic approach to using AI to create software at this point. Should a tool similar to Microsoft's IntelliCode become available for C-code, it could be tested out in the correct development environment. However, as Wärtsilä have created their own APIs that are mostly used when programming engine control software,

it would require the tool to be able to identify those APIs to provide the correct suggestions.

Even though Wärtsilä may not benefit from integrating AI-based tools into their way of working as of today, the team should actively keep researching what tools are becoming available on the market. The field of AI is progressing rapidly, and with the technologies found, it is no impossibility that similar tools more suitable for the embedded industry could become available.

4.4.2 Recommendation for intelligent bug handling

Artificially intelligent bug handling technology need to be further developed before it could be trusted completely. However, it is a promising area, as there are already tools on the market that can analyse code, as well as open-source patching tools under development.

The team is already using CodeSonar as a static analysis tool for code, so the next step could be testing out a tool that uses artificial intelligence for analysis, like for example DeepCode, which was described earlier.

4.4.3 Recommendation for intelligent testing

As none of the existing AI-assisted testing tools really focus on embedded software testing Wärtsilä could use some of the proposed methods for intelligent software testing as a stand-alone component. Alternatively, they could evaluate whether a testing tool designed for another type of software than embedded software, could still perform well enough to be taken into used by the team.

As the intelligent software testing tools on the market have been focused on web technology, user interfaces and more visual systems, the next generation of AI-assisted software testing tools will perhaps target more complex system testing, like embedded software. This kind of tool could be evaluated to gain knowledge about how well they perform, so that it can be concluded whether it would be wise to invest in in the future.

Another issue with the tools in the market is that none really focused on C as a language. Even though the automatic unit test generating tool DiffBlue that was described seems like an interesting technology, it would not be applicable for Wärtsilä's software, as switching to Java as a programming language is not an option due to Wärtsilä embedded systems not being compatible with other languages than C at this point. Therefore, it is again to be said, that a suitable unit test generation tool is still not found for Wärtsilä. The technology is however interesting, and perhaps some AI could be integrated to Wärtsilä's existing unit test environment WTP (Wärtsilä Testing Platform). Implementing and integrating artificial intelligence into WTP are described in the following chapters, where a proof of concept is developed to evaluate this further.

5 PROOF OF CONCEPT

To illustrate how the tools and methods presented in earlier chapters can be adapted by the team, this proof of concept was designed and implemented. The proof of concept was selected to focus on testing, for which the motivation is also presented.

The purpose of the proof of concept is to give Wärtsilä an idea of how they could continue with the recommendations made in the previous chapter. The proof of concept is not meant to be a product ready to take into use by the team, but it is a prototype that illustrates where Wärtsilä could start their own development of a similar automatic test case generation tool, as well as the architecture of such a tool.

5.1 Motivation

The reasons for selecting to further investigate intelligent testing methods out of the main areas investigated are mainly three. Each reason is hereby presented.

In the team analysis, implementation and testing were selected as the most interesting areas to investigate further. Based on this analysis, three alternatives were investigated and presented further – automatic programming, bug patching technology and intelligent testing. In the early analysis, the area of testing already has the most alternatives for implementation.

During the literature review and the research outcomes of that, the most mature solutions were found for testing. The development and the market of bug patching technologies and automatic programming seem to be in a much earlier stage than intelligent testing, at least in terms of suitability for Wärtsilä.

Also, when discussing the current development needs and situation with the Wärtsilä team's software architect, the team manager and a project manager at Wärtsilä, it became

clear that testing was the most suitable area for further investigation, as the team is focusing on improving their test procedures constantly. (Smedman, Tarsa & Bäck: 2019.)

5.2 Automatic unit test generation

Automatic unit test generation means that the unit tests are created automatically based on the code. Wärtsilä have up to this point approached this by creating a system that generates the test function skeleton that places the right function name and parameters when a new test environment is set up for an application. The Wärtsilä test platform can also to some extent generate test stubs. By introducing artificial intelligence into the process of generating a test structure, the process of writing unit test cases could be further automated.

Laurence Saes (2018) presents in his thesis a similar idea, generating unit tests with machine learning. Saes research presented a solution that transforms Java methods to test methods (2018). If a similar method could be applied on C-code, it could be an interesting solution for Wärtsilä. No tool found focused on C-language, but as the likelihood of Wärtsilä changing their development language is very small, it seemed most beneficial to set up a system of their own.

5.3 Theory and requirements

The purpose of the proof of concept prototype is to suggest a structure of a unit test based on the similarity between the function under test and other Wärtsilä C-functions. The prototype reads a C-function, converts it so that it can be processed by the neural network, which then calculates the similarity to other functions in the dataset.

Different applications have different purposes, but the structure of some of the functions still resembles each other. For example, functions with switch-statements or classic if-else statements, that write something to another place are typical. By identifying the

common factor in a group of functions it could be possible to classify functions and group them. Based on this, the correlating structure of a unit test can be suggested to the user.

Figure 16. Workflow of the prototype. A function is analysed by a machine learning algorithm that classifies the read function and based on that a recommendation for how the unit test case could be written is made.

Saes (2018) proved that Java functions can be linked to test cases using machine learning. The idea behind this proof of concept is that a neural network should be able to find the similarities between the structure in C-functions, based on the syntax. Similar functions may have different parameters, API-calls and function naming, but the structure of various functions is similar, meaning the algorithm could measure the similarity between functions based that instead of unique values.

Identifying the structure of a function would allow the neural network to match it to a similar unit test, for example a function with multiple if-else statements can be tested equally exhaustively as a function with the same amount of if-else conditions. The more if-else conditions, the more test case scenarios needed and so forth.

The suggested unit test may not be directly working but should give the test developer a vision of how the test case could be programmed. Also, the prototype does not need to read a full application but isolated functions from applications.

5.3.1 Wärtsilä test platform

As a test framework, Wärtsilä's own test platform (WTP) was considered (Wärstilä 2018). The prototype is not compatible with WTP but was developed with the test framework in mind.

Wärtsilä testing standards require the unit test function to be named after the function it is designed to test. Hence, a logical link is formed between the function under test and the unit test. (Wärtsilä 2018.)

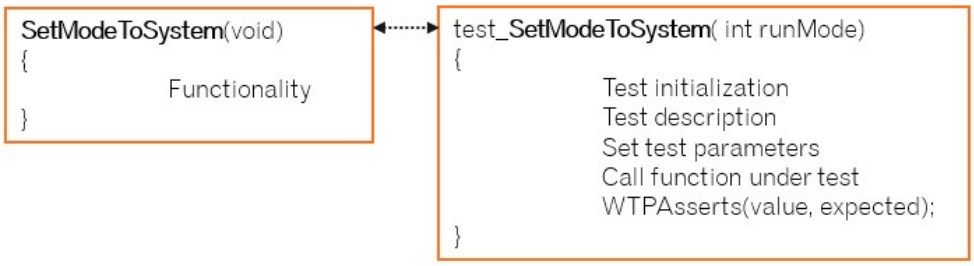

Figure 17. A logical linking logic can already be found between functions under test and corresponding unit test, as the unit test is named after the function with the extension "test_-" before the function name.

When setting up a new unit test environment, WTP generates empty test functions and test stubs automatically. This could be developed further with the tool also generating the structure of the test cases based on what the neural network suggests. Therefore, the proof of concept was developed with functions and test cases already compatible with WTP.

5.4 Algorithm

The algorithm can be viewed in three phases – data conversion, the neural network and the linking and adaption of the matching testcase (see figure 18). In this proof of concept, major focus is on the neural network that finds similarities between the functions. The

second priority is to get the algorithm to illustrate the whole process from preprocessing to delivering a testcase to an unknown function.

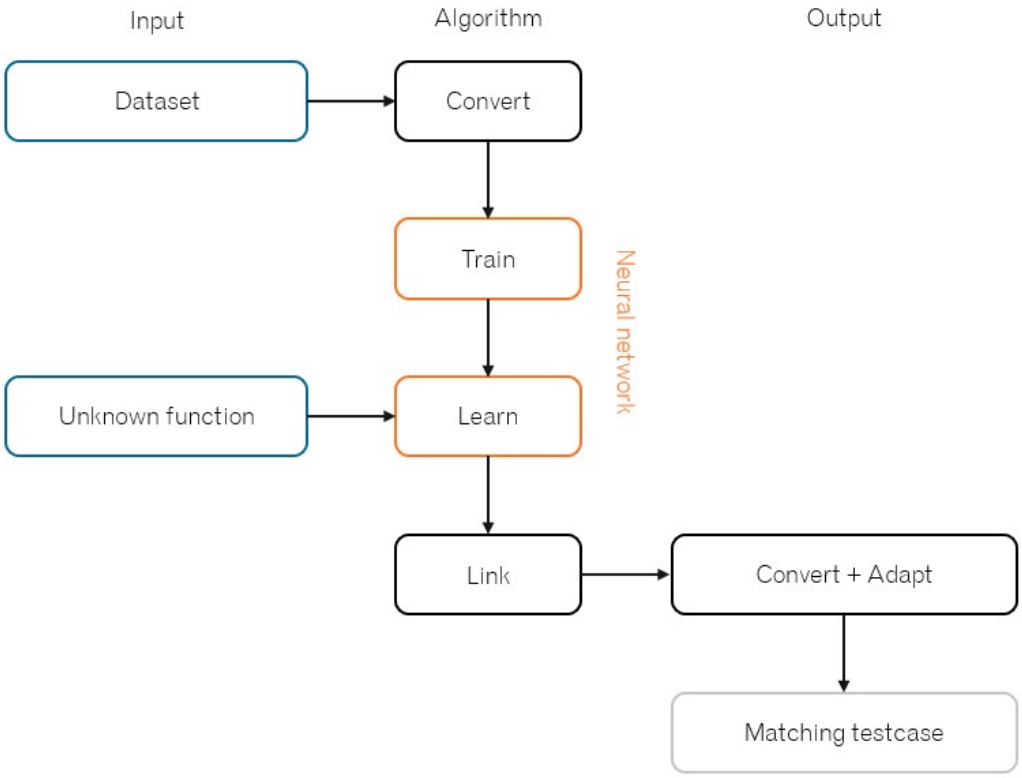

Figure 18. Workflow of the program that consists of data conversion, the neural network, linking and testcase adaption.

The prototype is programmed using Python and the Python package NumPy, which is suitable to use when programming mathematical operations such as those in the neural network.

5.5 Dataset for training

The training data set consists of short isolated C-functions that have been pre-processed and converted (see figure 19). The corresponding unit test case is found from a separate folder, after the algorithm has mapped out the closest test case for the function.

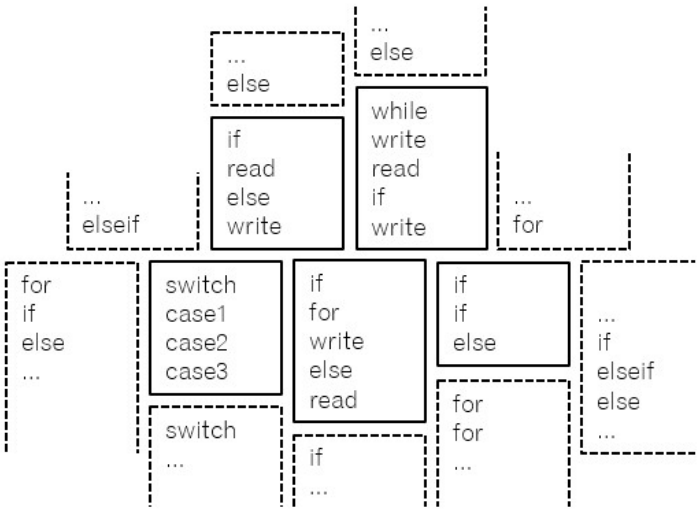

Figure 19. Visualization of the dataset and format. The data set can be scaled up to train the neural network further.

As this prototype is custom made for Wärtsilä, the dataset consists of the functions from Wärtsilä applications only. The neural network was tested with training on 9 functions per time, from a collection of the available Wärtsilä applications.

5.6 Converting data

For the neural network to be able to process the C-functions, data conversion is needed. Ideally, the neural network should operate the functions on binary code level, but as the Wärtsilä applications are dependent on the custom platform WMAP (Wärtsilä Modular Application Platform) for compilation, no binary code was used. Instead the .c-files were

processed from text to decimal numbers between 0 and 1. This proved to work well with the matrix multiplication in the neural network. Other options that were investigated were ASTs and vectorizing the .c-files, neither of which were suitable.

```
function = open('function.c','r').read()
#only save the whitelisted words for each function
keep_list = ["if", "else", "for", "switch", "case",
"read", "write"] #whitelist
infile = "function.c"
keptfunc = "" #create empty string to store data in
filein = open(infile)

for line in filein:
  for word in keep_list:
    if (word in line):
      keptfunc+=str(word) #save only whitelisted words
      keptfunc+=str("\n")
filein.close()

# convert whitelisted functions into machine readable
# DEC and divide by 255 to get range 0 to 1
temp=[ord(c) for c in keptfunc]
fixedfunction = [x / 255 for x in temp]
```

The above code illustrates how the data is cleaned up and converted for a file in the training set. First, the program reads the C-function in text format and removes the words that won't have any significance when calculating the similarity. Then, the remaining words are converted into decimal numbers between 0 and 1. The result is then used in the matrix multiplication in the neural network.

5.7 The neural network

The neural network created in the prototype is a simple neural network that has one hidden layer (see figure 20). The development of the neural network started with simply setting it up to predict the outcome of an array consisting of zeros and ones. After experimenting with the data conversion and format, the neural network was converted to process the converted functions in decimal format.

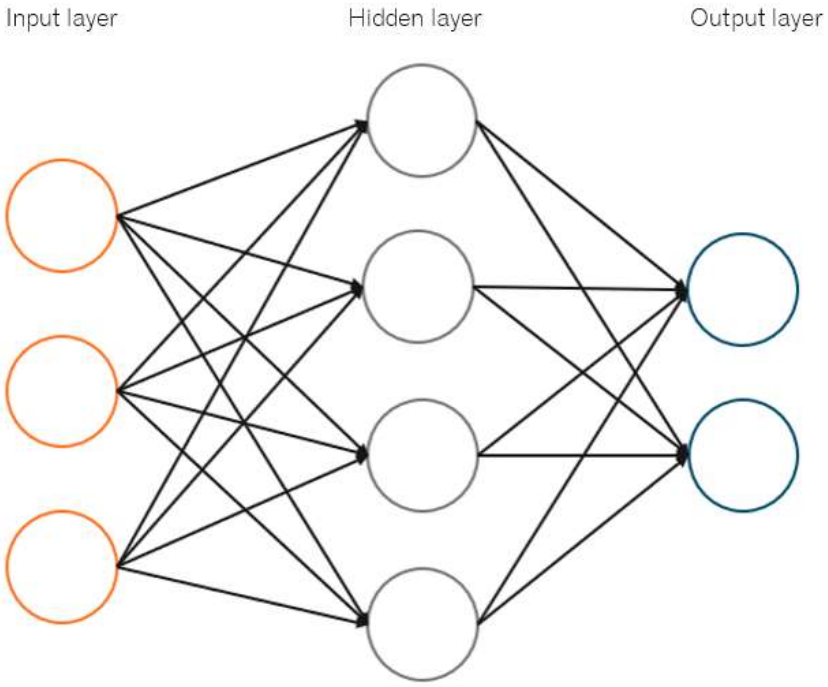

Figure 20. The neural network in the prototype has one input layer, one hidden layer and one output layer (Larose & Larose 2014: 189-190).

The essential features of the neural network topology in short were:

- Input: Wärtsilä C-functions
- Output: Resemblance values for compared to unknown C-function
- 8 neurons
- 1 hidden layer
- 10000 iterations found suitable by testing
- Validation by functions not used for training

5.7.1 Sigmoid function

The Sigmoid function is used as an activation function for the neural network. The Sigmoid function is popular as an activation function, as it is removes linearity from the neural network (see figure 21). The Sigmoid function formula is

$$\sigma(x) = \frac{1}{1+e^{-x}} \ldots (1).$$

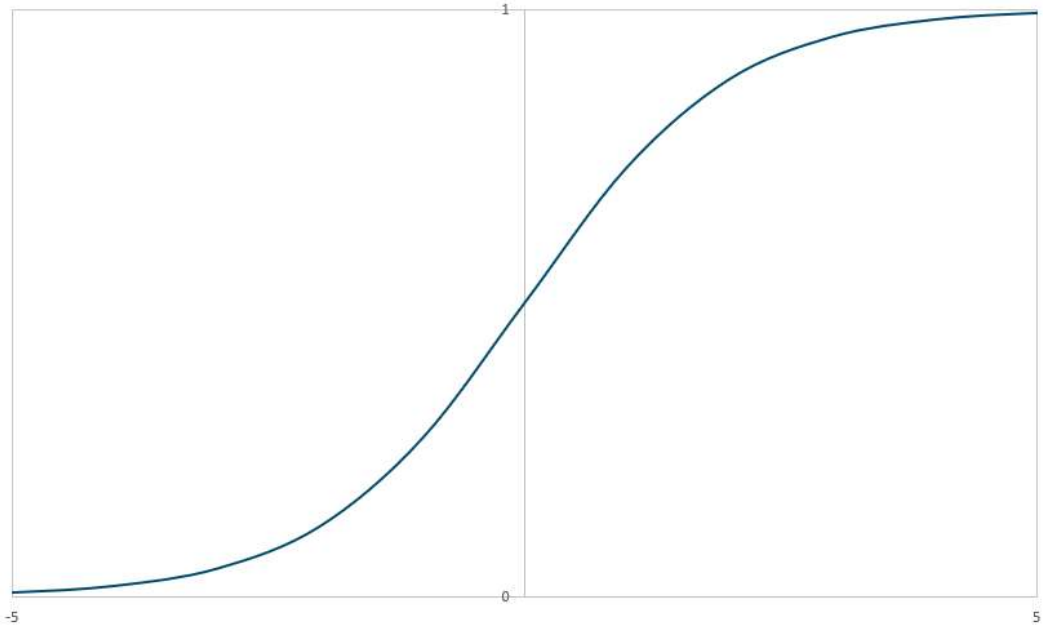

Figure 21. Plot of the Sigmoid function (Larose & Larose 2014: 193).

```
def __sigmoid(self, x):
        return 1 / (1 + exp(-x))

def __derivative(self, x):
        return x * (1 - x)
```

Above is the Sigmoid function and its derivative programmed in Python (Pal 2019).

5.7.2 Neural network operations

First, the training data set is pre-processed and converted before they are assigned weights. Then, random weights are assigned to a matrix that is the size of the amount of training data times the length of the converted functions (Pal 2019).

```
self.synaptic_weights=2*random.random((length,size))-1
```

These are then used in the training, and are adjusted each iteration. To adjust the weights, error calculation is used. The error is the difference between the expected output and the real output, which is then adjusted by a factor with help of the Sigmoid function derivative, and then used in the next iteration and so on. (Pal 2019.)

```
error = outputs - output
factor=dot(inputs.T,error*self.__derivative(output))
self.synaptic_weights += factor
```

The neural network is then ready to process the unknown function for which a test case is expected and goes on to deliver the outcome of the program.

5.8 Program output

The output of the neural network is an array of decimals, where the unknown C-function is compared to the functions in the training set. The lower the number, the more the functions resembles each other. The prototype recommends the unit test of the most similar C-function (from the dataset), as base for the unit test for the unknown function, and outputs it. In this prototype, the recommended unit test is not further processed, but could be changed to match the function under test to relieve the developer of even more work. This is further described in the next chapter where several improvement points are made.

6 RESULTS AND EVALUATION

6.1 Evaluation

As an alternative for automatic unit test generate, a neural network is an option when wanting to use artificial intelligence. Other machine learning algorithms could perhaps also be used, either on their own but also to support the neural network algorithm. Using a neural network both has its advantages and disadvantages.

Such a small simple neural network as the one created may not be too precise in its calculations but demonstrated the main idea of automatic unit test generation with help of a neural network quite well. Should the neural network be scaled up to process larger and more functions and test cases, the developer could at least be supported by getting to an example of how the test case should be structured to reach a high coverage on unit tests.

6.2 Impact on development

To be able to use an automatic test generation tool, the way of writing code would perhaps need to be restructured. An automatic test generation tool is likely to perform better the easier the code it processes is. This means complex structures of functions would need to be simplified, so that functions do not become too long or complex to test. There are many features of programs that impacts the complexity and thereby the testability. Besides the length of the program, features like the amount of decisions, equalities and inequalities and measurements like McCabe's cyclomatic complexity all have an impact on how easy the code is to test (Ferrer, Chicano & Alba 2013). A practical example of this is functions with a great number of if-else statements – gaining a high test coverage on such a function demands many test cases, which may be hard to create with an automatic test generation tool. Therefore, if using an automatic test generator, the complexity of the code should be taken in account when writing it, and a low complexity should be aimed for to increase the success rate of the tool.

6.3 Adaption to test driven development

Test driven development (TDD) means that the developer starts programming with a test, that sets the requirements for which later the actual function that is to be written (Koutifaris 2018). Wärtsilä's software development team is striving towards a more agile organization, where TDD would be adapted. This unfortunately means that the kind of tool illustrated by the proof of concept prototype would perhaps not be part of the development process. However, should the function and test case switch places in the prototype, so that the developer would have written a test case that could be understood by the neural network so that it could recommend a function that could be further developed instead, it could fit the TDD work process. This kind of tool would be a sort of automatic programming tool.

6.4 Improvement suggestions

To develop the algorithm further would need a development team that would focus on three main areas. First, the data conversion would need to be optimized. Data should be easy to translate, meaning it should contain nothing more than necessary. The code should be clean and easily understandable.

Representing code as either parse trees or ASTs is a method that has been used in many of the tools and techniques described in chapter 4, such as the DeepCode code analyser (Deepcode 2019) and the repair program described by Weimer (2019) that used genetic programming to find bug fixes. Therefore, it could also suit this case.

The neural network itself could be optimized further to achieve better accuracy. It could be investigated how many layers would be needed to gain higher correctness. As the program quickly became heavy to run on a normal engineering laptop, it could be investigated if the neural network could be deployed on some server to relieve the computer of some of the load.

The third area of improvement would be outputting the test case. To really achieve automatic test case generation, the test case recommended for use by the algorithm should also be processed to match the function under test. Replacing function names, parameters and API calls would make the recommended test case suit the new function better. To achieve this, another AI-algorithm could perhaps be used. The algorithm would be trained to recognize what characterizes a function or parameter name, how parameters are passed to functions and how an API-call looks. After recognizing the characteristics of the new function, the corresponding function and parameter names could be transferred to the test case to match the function.

7 CONCLUSIONS

The alternatives for implementing artificial intelligence as an assistance tool or solution within software development are many. Deploying an artificially intelligent component to the software development process requires resources, time and training for the developers who would eventually benefit from such a solution. Therefore, the benefits of implementing artificial intelligence-based tools and solutions into an already working development process should be carefully weighed against the amount of time and resources it would take to start using it.

Technologies, like artificial intelligence for requirement handling, or automatic code generation based on inputting the requirements, may seem like appealing solutions – and should it perform as well as expected it could be a suitable investment. However, out of the tools investigated in this thesis, not all are yet in such a stage that they would be practical to implement. Most of the tools were focused on visual systems, which are not at all suitable for the embedded industry.

On the other hand, artificial intelligence may bring a lot to some processes. The need for testing grows as the complexity of the system increases, and thereby automating the testing and increasing the test coverage is desirable. If one can identify the correct area that would benefit from an AI-based solution, and deploy the solution successfully, AI could be considered. The proof of concept showed that if implemented successfully, artificial intelligence could be a great complement to the work of any decent software engineer, but also that it takes a lot of resources before a working product is set up.

The research on how to use artificial intelligence in software development is progressing, as well as the research on AI itself. Artificial intelligence is right now a “trend”, that will perhaps someday be replace by some other buzzword in science and technology, but regardless of that, artificial intelligence is likely to play a great role in the future, in many areas – also within software development. Current AI-based software tools may be more applicable on other fields of software engineering, like web technology and apps, but as research progress it may very well also be suitable for the embedded industry.

Regarding the proof of concept developed according to the team's needs, it succeeded to demonstrate the idea and architecture of an automatic test generation tool but the question of how it could be integrated to the current software development process remains. The purpose was to show an example of how artificial intelligence could be used in the engine control system development team, meaning it was out of scope to fully integrate it to the way of working at Wärtsilä. There are still major improvements that should be done regarding the product before it could be used in practice. Another outcome of the proof of concept was the realization of how much effort developing such a tool would require. Wärtsilä would probably benefit both cost-wise and with quality in mind to buy a similar tool instead of developing their own.

LIST OF REFERENCES

Atlassian (2019). DevOps: Breaking the Development-Operations barrier [online]. [Cited 18.3.2019] Available: https://www.atlassian.com/devops

Banerjee, Suvro (2018). *An Introduction to Recurrent Neural Networks*. [online]. Medium.com. [Cited 10.4.2019]. Available: https://medium.com/explore-artificial-intelligence/an-introduction-to-recurrent-neural-networks-72c97bf0912

Bayou (2018). *What is Bayou?* [online] Bayou [Cited 28.3.2019]. Available: https://info.askbayou.com/

Beltramelli, Tony (2017). *pix2code: Generating Code from a Graphical User Interface Screenshot.* [online] Copenhagen, Denmark: UIzard Technologies. [Cited 10.4.2019]. Available: https://arxiv.org/pdf/1705.07962.pdf

Billings, J. J., A. J. McCaskey, G. Vallee & G. Watson (2017). *Will humans even write code in 2040 and what would that mean for extreme heterogeneity in computing?* [online]. New York: Cornell University. [Cited 22.3.2019]. Available: https://arxiv.org/abs/1712.00676

Bloomberg, Jason (2017). *The Low-Code/No-Code Movement: More Disruptive Than You Realize.* [online]. Forbes.com. [Cited 8.5.2019]. Available: https://www.forbes.com/sites/jasonbloomberg/2017/07/20/the-low-codeno-code-movement-more-disruptive-than-you-realize/#684c994722a3

Boulton, Clint (2019). *What is low-code development? A Lego-like approach to building software.* [online]. Cio.com. [Cited 8.5.2019]. Available: https://www.cio.com/article/3263392/what-is-low-code-development-a-lego-like-approach-to-building-software.html

Colantonio, Joe (2017). *8 Innovative AI Test Automation Tools for the Future: The Third Wave*. [online] Testguild.com. [Cited 10.4.2019]. Available: https://testguild.com/7-innovative-ai-test-automation-tools-future-third-wave/

DeepCode (2019). *DeepCode Tecchnology.* [online]. DeepCode AG. [Cited 3.6.2019]. Available: https://www.deepcode.ai/tech/

Erturk, E. & E. A. Sezer (2015). A comparison of some soft computing methods for software fault prediction. *Expert Systems with Applications*. 11(2), 1872-1879. Available: https://doi.org/10.1016/j.eswa.2014.10.025

Ferrer J., F. Chicano & E. Alba (2013). Estimating software testing complexity. *Information and Software Technology,* 55(12), 2125-2139. Available: https://doi.org/10.1016/j.infsof.2013.07.007

Fumo, David (2017). *Types of Machine Learning Algorithms You Should Know.* [online]. Towardsdatascience.com. [Cited 24.4.2019]. Available: https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861

Pal, Vivek (2019). *Implementing Artificial Neural Network training process in Python.* [online]. Geeksforgeeks.com [Cited 2.8.2019]. Available: https://www.geeksforgeeks.org/implementing-ann-training-process-in-python/

Ghahramani, Zoubin (2015). Probabilistic machine learning and artificial intelligence. [online]. *Nature,* 521, 452-459. Available: https://doi.org/10.1038/nature14541

Giudice, Diego Lo (2016). *How AI Will Change Software Development And Applications* [online]. Massachusetts: Forrester, 2.11.2016. [Cited 5.3.2019]. Available: https://www.nhaustralia.com.au/documents/AI_report.pdf

Hammouri, A., M. Hammad, M. Albadhan & F. Alsarayah (2018). Software Bug Prediction using Machine Learning Approach. [online]. *International Journal of Advanced Computer Science and Applications.* 9(2), 78-83. Available: https://doi.org/10.14569/IJACSA.2018.090212

Harman, Mark (2012). *The Role of Artificial Intelligence in Software Engineering*. Paper presented at the First International Workshop on Realizing AI Synergies in Software Engineering *(*RAISE). Zürich 5.6.2012. Available: https://doi.org/10.1109/RAISE.2012.6227961

Homès, Bernard (2013). *Fundamentals of Software Testing*. USA: John Wiley and Sons Inc. ISBN: 987-1-84821-324-1.

Igwe, K. & N. Pillay (2013). *Automatic Programming Using Genetic Programming.* Paper presented at the Third World Congress on Information and Communication Technologies (WICT): Hanoi 15-18.12.2013. Available: https://doi.org/10.1109/WICT.2013.7113158

Jones, M. Tim (2003). *AI Application Programming.* USA: Charles River Media. ISBN: 1-58450-278-9.

Jonsson, Leif (2018). *Machine Learning-Based Bug Handling in Large-Scale Software Development.* Doctoral thesis, Linköping University, Sweden. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-147059/

Kersting, K. & U. Meyer (2018). From Big Data to Big Artificial Intelligence? Künstliche Intelligenz, 32(1), 3-8. Available: https://doi.org/10.1007/s13218-017-0523-7

Keyvanpour, M. R., H. Homayouni & Hasein Shirazee (2011). Automatic Software Test Case Generation. [online]. *Journal of Software Engineering,* 5(3), 91-101. Available: https://scialert.net/fulltextmobile/?doi=jse.2011.91.101

Kire, K. & N. Malhotra (2014). Software Testing using Intelligent Technique. [online]. *International Journal of Computer Applications,* 90(19), 22-15. Available: http://dx.doi.org/10.5120/15829-4637

Kirilenko, Igor (2018). *What is Artificial Intellgence in Software Testing?* [online]. Parasoft. [Cited 18.3.2019]. Available: https://blog.parasoft.com/what-is-artificial-intelligence-in-software-testing

Kulkarni, Rajesh H. & Palacholla Padmanabham (2016). Integration of articifial intelligence activities in software development processes and measuring effectiveness of integration. *IET Journals,* 11(1), 18-26. Available: https://doi.org/10.1049/iet-sen.2016.0095

Koutifaris, Andrea (2018). *Test driven development – what it is and what it is not.* Freecodecamp.com [Cited 2.8.2019] Available: https://www.freecodecamp.org/news/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2/

Lardinois, Frederic (2019). *Microsoft's IntelliCode for AI-assisted coding comes out of preview.* TechCrunch.com. [Cited 7.5.2019] Available: https://techcrunch.com/2019/05/06/microsofts-intellicode-for-ai-assisted-coding-is-now-generally-available/

Larose D. T. & C. D. Larose (2014). *Discovering Knowledge in Data: An Introduction to Data Mining.* USA: John Wiley & Sons, Inc. ISBN: 978-0-470-90874-7.

Last, M., A. Kandel & H. Bunke (2004). *Artificial Intelligence Methods in Software Testing.* Singapore: World Scientific Publishing Co. ISBN: 981-238-854-0.

Le Goues, C., S. Forrest & W. Weimer (2019). *Genprog. Evolutionary Program Repair.* [online] Github homepage. [Cited 11.4.2019]. Available: https://squares-lab.github.io/genprog-code/

Loudiras, P. & C. Elbert (2016). Machine Learning. [online]. *IEEE Software,* 33(5), 110-115. Available: https://ieeexplore.ieee.org/document/7548905/

Mantere, T. & J. Alander (2005). Evolutionary software engineering, a review. [online]. *Applied Soft Computing,* 5(3), 315-331. Available: https://doi.org/10.1016/j.asoc.2004.08.004

Maruti Techlabs. *What are the advantages of artificial intelligence in testing?* [online] Maruti Techlabs. [Cited 18.3.2019]. Available: https://www.marutitech.com/artificial-intelligence-in-testing/

Marr, Bernard M. (2015). *Big Data: Using SMART Big Data, Analytics and Metrics to Make Better Decisions and Improve Performance.* United Kingdom: John Wiley & Sons, Incorporated. ISBN: 978-1-118-96578-8.

Merrill, Paul. *5 ways AI will change software testing*. [online] Techbeacon.com. [Cited 18.3.2019]. Available: https://techbeacon.com/app-dev-testing/5-ways-ai-will-change-software-testing

Miller, B. & D. Ranum (2014). *Problem Solving with Algorithms and Data Structures.* [online]. Interactivepython.org. [Cited 4.6.2019]. Available: https://interactivepython.org/runestone/static/pythonds/Trees/ParseTree.html

Mohri M., A. Rostamizadeh & A. Talwalkar (2012). *Foundations of Machine Learning.* USA: MIT Press. ISBN: 978-0-262-01825-8.

Monperrus, M., Urli S., Durieux T., Martinez M., Baudry, B., Seinturier, L. (2018). Human-competitive Patches in Automatic Program Repair with Reparnaitor. *Software Engineering Research at KTH Royal Institute of Tehcnonoly & INRIA.* Available: https://arxiv.org/abs/1810.05806v1#

Myers J., C. Sandler & T. Badgett (2011). *The Art of Software Testing*. USA: John Wiley and Sons. ISBN: 987-1-118-03196-4.

Nilsson, Nils J. (1980). *Principles of Artificial intelligence.* USA: Morgan Kaufmann Publishers, Inc. ISBN 0-934613-10-9.

Nguyen Hung T., Carol L. Walker & Elbert A. Walker (2019). *A First Course in Fuzzy Logic*. USA: Taylor and Francis Group, CRC Press. ISBN: 987-1-138-58508-9.

O'Leary, Daniel E. (2013) Artificial Intelligence and Big Data. [online] *IEEE Intelligent Systems,* 28(2), 96-99. [Cited 11.3.2019] Available: https://ieeexplore.ieee.org/document/6547979

Poli, R., W. B. Langdon & N. F. McPhee (2008). *A Field Guide to Genetic Programming*. [online]. USA: Creative Commons. [Cited 11.3.2019] Available: http://www0.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/poli08_fieldguide.pdf

Rech, J., E. Ras & B. Decker (2007). Intelligent Assistance in German Software Development: A Survey. *IEEE Software,* 24(4), 72-79. [Cited 5.3.2019]. Available: https://ieeexplore.ieee.org/document/4267606

Rouse, Margaret (2005). *Definition: Genetic programming.* [online] Techtarget.com [Cited 28.3.2019]. Available: https://whatis.techtarget.com/definition/genetic-programming

Saes, Laurence (2018). *Unit Test Generation using Machine Learning.* Master's thesis. University of Amsterdam, the Netherlands. Available: http://www.scriptiesonline.uba.uva.nl/document/660538

Siddique, Nazmul & Hojjat Adeli (2013). *Computational Intelligence Synenergies of Fuzzy Logic, Neural Networks and Evolutionary Computing.* United Kingdom: John Wiley & Sons. ISBN: 978-1-118-33784-4.

Shi, Zhongzhi (2011). *Advanced Artificial Intelligence*. Singapore: World Scientific Publishing Co. Ptd. Ltd. ISBN: 978-981-4291-34-7.

Shiffman, Daniel (2012). *The nature of code*. [online]. Natureofcode.com [Cited 25.4.2019]. Available: https://natureofcode.com/book/

Smedman, D., J. Tarsa, A. Bäck (2019). Chief Engineer/General Manager/Project Manager, Wärtsilä Finland. Discussion, Vaasa 15.4.2019.

Spirina, Katrine (2018). *How Artificial Neural Networks Can Code Smarter Than GUI Programmer.* [online]. Blog-post at Hackernoon.com. [Cited 10.4.2019]. Available: https://hackernoon.com/how-artificial-neural-networks-can-code-smarter-than-gui-programmer-1cdfaecb4851

Stadlbauer, Reginald (2018). *Froglogic Announced 2018 Plans for AI-Driven Test Automation Solutions.* [online]. Froglogic.com. [Cited 16.4.2019]. Available: https://www.froglogic.com/news/froglogics-plans-ai-driven-test-automation-solutions/

Tufano M., C. Watson, G. Bavota, M. Di Penta, M. White, D. Poshyvanyk (2018). *An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation.* Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. Montpellier, France, 3.7.2018. Available: https://doi.org/10.1145/3238147.3240732

Urli, S., L. Seinturier, Z. Yu & M. Monperrus (2018). *How to Design a Program Repair Bot? Insights from the Repairnator Project.* [online]. 40th International Conference on Software Engineering, Track Software Engineering in Practice. Gothenburg, Sweden, 27.5-3.6.2018. Available: https://hal.inria.fr/hal-01691496/document

Varhol, Peter (2017). 10 ways machine learning can optimize DevOps. [online] Techbeacon.com. [Cited 7.3.2019]. Available: https://techbeacon.com/enterprise-it/10-ways-machine-learning-can-optimize-devops

Weimer W., S. Forrest, C. Le Goues and T. Nguyen (2010) Automatic Program Repair with Evolutionary Computation. [online]. *Communications of the ACM*, 53(5), 109-116. [Cited 1.4.2019] Available: https://doi.org/10.1145/1735223.1735249

Wells, Don (2013). *Extreme Programming: A gentle introduction.* [online]. Available: http://www.extremeprogramming.org/

Wen, J., S. Li, Z. Lin, Y. Hu & C. Huang (2011). Systematic literature review of machine learning based software development effort estimation models. [online]. *Information and Software Technology,* 54(1), 41-59. [Cited 14.3.2019]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584911001832

Winograd, Terry (1973). Breaking the Complexity Barrier again. [online]. *ACM SIGIR Forum - Proceedings of ACM SIGPLAN - SIGIR interface meeting,* 9(3), 13-30. [Cited 18.3.2019]. Available: https://dl.acm.org/citation.cfm?doid=951761.951764

Wärtsilä (2019). *Wärtsilä UNIC engine control system for diesel engines*. [online]. Available: https://www.wartsila.com/services-catalogue/electrical-automation-services/wartsila-unic-engine-control-system-for-diesel-engines

Wärtsilä (2018). *WTP Manual*. Internal document of Wärtsilä.

Wärtsilä (2019b) *Wärtsilä Application Development Environment Guide.* Internal document of Wärtsiä.

Xie, Tao (2013). The Synenergy of Human and Artificial Intelligence in Software Engineering. [online]. *2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. San Fransisco, USA, 25-26.5.2013. [Cited 18.3.2019]. Available: https://ieeexplore.ieee.org /document/661519

Xuan, J., M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, M. Monperrus (2017). Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. [online]. *IEEE Transactions on Software Engineering,* 43(1), 34-55. [Cited 11.4.2019]. Available: https://dx.doi.org/10.1109/TSE.2016.2560811