Shortening Feedback Time in Continuous Integration Environment in Large-Scale Embedded Software Development with Test Selection

University of Oulu
Degree Programme on Information Processing Science
Master’s Thesis
Jarmo Koivuniemi
Date 14.3.2017
Abstract

Continuous integration is one of the Extreme Programming practices and is used in agile software development to provide rapid feedback and to have a working system at all times. In continuous integration, a developer commits code to projects mainline at least once a day which triggers automated build and tests. Large projects can struggle with continuous integration because with growing code base the number of tests that need to execute after a developer checks in code also gets bigger. The growing number of tests means that the build times can become long. With embedded systems, the problem can be even bigger as the testing is dependent on target hardware. With long builds, feedback becomes longer thus making it harder to practice continuous integration.

Long feedback leads to infrequent integrations which can result in continuous integration becoming a bottle-neck to the development process and can even affect an ability to release software frequently. To shorten the feedback this thesis implements an automated test selection tool to work in continuous integration environment. The test selection tool selects the tests that are relevant for a specific code change instead of executing all tests for each commit. The case company is suffering from long feedbacks and the expensiveness of testing has become a problem. This thesis hopes to solve that problem with the implemented test selection tool. The tool is evaluated using three metrics; feedback time improvement, reduction in number of executed test cases and fault finding capability.

The test selection tool was tested with two test suites with different size by collecting data from continuous integration system. “Shock” suite consisted of about 30 test cases and regression suite about 800 test cases. The average improvement in feedback time for Shock tests was 29.3 % and 55.7 % for regression suite. The test selection tool reduced the number of test cases executed in Shock by 67.1 % and 78.2 % for regression suite. Fault finding capability was measured for Shock suite and the tool was able to find same faults as the full suite in 97.8 % of the cases. Statistical tests show that the test selection has a significant impact on feedback time. By being able to safely shorten the feedback, the test selection tool can potentially help developers in the case company to practice continuous integration.

Keywords
Continuous integration, test case selection, agile software development

Supervisor
Professor Mika Mäntylä
Foreword

I would like to thank my managers for giving me this amazing opportunity. Thanks to you I’m now at the end of one chapter in my life and starting a new one. I’d like to thank my team and my mentors. I’ve learned so much from you about being a professional and about having passion for my craft. I also want to thank my family and friends for being supportive through this journey. I’d also like to thank my supervisor Professor Mika Mäntylä.
Contents

Abstract .......................................................................................................................... 2
Foreword ......................................................................................................................... 3
Contents .......................................................................................................................... 4
1. Introduction ................................................................................................................. 5
2. Research Method and Design ................................................................................... 7
   2.1 Research question ............................................................................................... 7
   2.2 Design Science Research ................................................................................... 9
   2.3 Design science research framework ................................................................. 11
       2.3.1 Environment – Sub-RQ1 ............................................................................ 12
       2.3.2 IS Research .............................................................................................. 14
       2.3.3 Knowledge base ...................................................................................... 15
3. Continuous Integration and Continuous Delivery ..................................................... 16
   3.1 Benefits of Continuous Integration ..................................................................... 16
   3.2 Branching in Continuous Integration ................................................................ 16
   3.3 Build times in large-scale CI – Sub-RQ4, Sub-RQ5 ......................................... 18
   3.4 Growing code-base and CI ................................................................................. 19
   3.5 Scaling Continuous Integration in Literature – Sub-RQ5 ............................... 20
   3.6 Continuous Integration in Embedded Systems development .......................... 24
   3.7 Continuous Integration in target organization – Sub-RQ4 ............................... 25
   3.8 Continuous Delivery – Sub-RQ6 ....................................................................... 29
4. Test optimization techniques – Sub-RQ1, Sub-RQ2 ................................................ 31
   4.1 Software testing ................................................................................................. 31
   4.2 Test suite minimization ...................................................................................... 32
   4.3 Test case prioritization ...................................................................................... 33
   4.4 Test case selection ............................................................................................ 34
5. Implementation of the test selection tool ................................................................. 38
   5.1 The starting point ............................................................................................... 38
   5.2 First steps – collecting the coverage data ....................................................... 39
   5.3 Automating the test selection .......................................................................... 40
   5.4 Implementation of additional features ............................................................. 41
   5.5 Collecting results .............................................................................................. 44
6. Results ....................................................................................................................... 47
   6.1 Feedback – Sub-RQ3 ......................................................................................... 47
   6.2 Number of test cases ......................................................................................... 55
   6.3 Fault-finding capability – Sub-RQ2 ................................................................. 61
   6.4 Summary of the results .................................................................................... 62
7. Discussion & Implications ....................................................................................... 64
8. Conclusions .............................................................................................................. 68
   8.1 Limitations ......................................................................................................... 68
   8.2 Recommendation for future studies ................................................................. 68
References .................................................................................................................... 70
1. Introduction

Since Agile Manifesto was published in 2001, agile software development approaches have been adopted in a wide range of organizations. However, there is a conception out there that agile development practices can work only in small teams and small organizations. To solve the problem of scaling agile development, different frameworks have been introduced in the software industry, Large-Scale Scrum (LeSS) and Scaled Agile Framework (SAFe) are just to mention a few. There are a few reasons that make scaling of agile development challenging but there are practices that hold the uttermost importance when scaling agile software development is continuous integration (CI). For example, one of the founders of LeSS framework, Bas Vodde, stated that continuous integration is the most important practice in adopting agile at scale. (Kircher & Vodde, 2011)

Continuous integration is one of the Extreme Programming techniques and it was mentioned for the first time in the literature by Kent Beck (1999) and it can be seen as an essential practice for scaling lean and agile development (Larman & Vodde, 2010). In continuous integration, a developer checks code into a mainline at frequent time intervals. (Fowler, 2006, Duvall, 2007, Rogers, 2004) The integration process includes automated build and running tests to detect errors as quickly as possible and get rapid feedback. For each commit every test must be executed and the test results should be emailed to team members. (Martin, 2014) Integration should occur at least once a day for it to be considered CI (Fowler, 2006, Duvall, 2007, Rogers, 2004). Integrating in even more frequent intervals such as every few hours can be also seen as desirable. Integration frequency can be determined by three factors: the ability to split large changes, speed of integration and speed of feedback (Larman & Vodde, 2010). In optimal situation, the code should be fit for integration once the code compiles and unit tests pass. A successful integration can be seen as a measure of progress while code sitting in the workstation of a developer does not exist from the point of view of the system and the other developers. (Rogers, 2004).

There is separation between some of the literature as some refer CI as being automated process and some note that CI can be done manually e.g. Fowler (2006). Duvall (2007) on the other hand only discusses automated aspects of CI while still admitting that automation is not a requirement for a process to be considered CI. This research focuses only on automated CI. By automated CI, this research refers to automated build and testing whereas the committing is done manually by a developer which then triggers the building and testing in a CI system.

Test case selection is a test optimization method where a subset of test cases is selected from existing test suite to test the part of the system that was affected by a change. The selection process includes two parts: determining the affected parts of the system and choosing the tests from initial test suite to test those parts. (Biswas, Mall & Satpathy, 2011.) There is a trade-off between test selection and re-testing everything because test selection might not be completely safe in terms of faults find in tests. This trade-off is well-acknowledged by the case company and the results of this study will show whether test selection can be utilized despite this trade-off. The trade-off is also made smaller by the fact that test selection is only applied in branches and not the mainline which means that no untested code will go into production.

This master’s thesis focuses on solving the problem of large-scale continuous integration with test selection tool that is implemented and introduced in this thesis. This thesis was
conducted in a large telecommunications company that develops large embedded systems. There are a few motivations that inspired the topic for this thesis. First, the people working in the organization are facing pain in their development process due to the slow feedback in the integration process. Second, the expensiveness of testing embedded software has created a problem in the target organization as the continuous integration requires continuous testing which has become expensive in terms of testing hardware. Third, the organization is moving towards continuous delivery which can be seen as a continuum of continuous integration (Fowler, 2013.) By that reason, this thesis can also help in solving one of the problems the company is facing in adopting continuous delivery.

This thesis is structured as following, first the research question is introduced along with the sub-questions that are used to guide the researcher to right path in order to answer the main research question. Then, the used research method is explained which gives the reader an overview of the environment, research method itself and knowledge base this research uses and contributes to. After introducing the research method, the literature about continuous integration and test optimization is studied. After that the implementation of the design artefact (automated test selection tool) is explained while supporting design decisions with previous research. Then, the collected results are analysed and discussed. The findings of this research are concluded in the last chapter.
2. Research Method and Design

This chapter is about the research method used for this thesis. The chapter has three subsections; one is about the research question and the second is about the chosen research method; design science research (DSR) (Hevner et al., 2004.) Third chapter describes how DSR framework was applied in this thesis.

2.1 Research Question

The roadblocks in the way of adopting agile in large scale can be traced back to the lowest level, to a developer making changes to code. The feedback loop after a developer pushes a commit can determine how effective a programmer can be and how fast the newly introduced bugs can be found and fixed (Laukkanen & Mäntylä (2015.) Even before commit, if a developer has access to tests that offer fast feedback, it could improve the developer’s personal CI practices and encourage more frequent integration.

The goal of the research is to implement an artefact that can be used to improve test runs in CI environment. By improving test executions, this thesis refers to shortening the time between code is committed and developer getting feedback from CI tool. Data is collected from the CI tool in order to calculate the feedback loop after pushing a commit. Also, the results of testing optimization are studied and whether implemented artefact can shorten the feedback time while still being reliable and trustworthy from developers’ point of view.

There is one research question that is followed by several sub-questions that are used answer the main research questions.

**RQ:** Can automated test selection enable continuous integration at large scale embedded software development?

The main research question aims to answer whether the current non-continuous integration process could be turned around with automated test case selection. The current situation of CI in case company is explained in detail in chapter 4.1.6. The CI has become, at some level, a bottle-neck in case company’s release pipeline. In order to achieve continuous delivery, a company must first have successful continuous integration (Fowler, 2013) and the company has requested an improvement to the feedback cycle in their CI environment. The case company also uses some potentially suboptimal CI strategies, such as long-living branches (Humble & Farley, 2010) that they were, in some sense, forced to adopt due to the slow feedback. This thesis attempts to find out if better strategies could be adopted if the feedback was faster and an artefact is implemented to improve the feedback time. By word “enabling” this thesis refers to the perception that long build times make practicing continuous integration difficult (Duvall, 2007, Brooks, 2008, Rogers (2004) and by shortening feedback from CI build would allow developers to practice continuous integration. Also, currently the case company is not doing CI by its definition (Duvall, 2007) which can be partly traced to long build times (Rogers, 2004). Enabling continuous integration would mean that the improvements that come from the test selection tool would remove some of the things blocking the organization from practicing CI.

**Sub-RQ1:** What test optimization techniques are there?
This sub-question attempts to find out the potential test optimization techniques presented in the literature. Those techniques will be evaluated and reflected against requirements by the case company. The test decision on which test optimization technique is used in this research is based on the findings of this question. Literature and case company’s ongoing test optimization initiatives factor in the choice of the technique for test optimization.

**Sub-RQ2: Can automated test selection be used safely without compromising fault finding capability?**

The usual problem with test optimization techniques is whether their benefits are higher than their costs (Elbaum et al, 2014.) Fast feedback could be seen as a benefit but the potential drawbacks are reduced fault finding capability and if the automated test optimization technique takes as much time to execute as the tests themselves. This question is answered using literature and by evaluating the designed artefact.

**Sub-RQ3: Can automated test case selection in embedded software development make shorter feedback loops?**

The case company is currently suffering from long and tedious feedback loops which make practicing CI difficult (Duvall, 2007, Brooks, 2008, Rogers (2004.) They are also facing problems with expensiveness of the test environments as the tests are executed in real hardware. This question aims to answer whether test selection could make feedback loop shorter and ease the load of test machines. The aim is to execute the test cases that are relevant to changed code for each commit. Feedback loop is calculated from time between a commit is made and when a developer can see the results of the build. The measured times are compared to the current feedback times. The designed artefact and the results of its evaluation are used to answer this question.

**Sub-RQ4: How does slow feedback affect developers and organizations?**

Slow feedback is one of the things that people bring up in the case company when talking about pain points. This question aims to answer that in what ways slow feedback, in continuous integration environment, can affect development and organizations. This question is answered using literature.

**Sub-RQ5: Can test case selection enable better continuous integration strategies?**

This question answers the question of whether test case selection can improve continuous integration strategies and practices. The answers for this questions are mostly gotten from literature and previous research from the point of view of how long build times affect a continuous integration practice (Chapter 3.3). The effects of long builds are studied from different sources such as books, research articles and observing the atmosphere in case company. Slow feedback has driven CI strategy to its current state and the possibilities of how test case selection could be used to improve the CI strategies are studied.

**Sub-RQ6: Can successful continuous integration help the organization to achieve continuous delivery?**

The case company is moving towards continuous delivery and there are several roadblocks on the way. To answer this question, literature is studied to find out whether successful continuous integration practices can help an organization move towards continuous delivery. This question is answered by using literature.
2.2 Design Science Research

This study adopts Design Science Research (DSR) from Hevner et al. (2004) and this chapter goes through how DSR is used in this research. The seven guidelines for DSR will be introduced from theoretical point of view and in the context of this research. Hevner et al. (2004) points out that not all guidelines are mandatory to follow and that creative skills and judgement should be used when deciding which guidelines are applied. Also, this research uses the guidelines as they are seen fit for this research.

**Design as an artefact:** A design-science research should create an IT artefact that will address an important organizational need. An IT artefact could be a construct, a model or an instantiation. (Hevner et al., 2004.) In this design-science research the artefact is the automation and integration of a test case selection algorithm into continuous integration environment. The main purpose of the artefact is to shorten the feedback loop in CI environment and solve some of the problems brought by expensiveness of testing environments in embedded development.

**Problem relevance:** The objective of research in information systems is to gather knowledge and understanding that enable the development and implementation of technology-based solutions that solve important business problems that, until that point, have remained unsolved. Design-science research wants to solve the problem by construction of innovative artefacts. The definition of a problem can be deducted by looking the gap between a goal state and the current state of a system.

A goal for information system is basically to create revenue by improving the ratio of money going in and out of the company. (Hevner, 2004) To give meaningful motivation for this research it should be approached from larger context. Competition in software industry is fierce and managing to stay ahead of competitors could be seen as a never-ending task. The case company is adopting continuous delivery in order to meet the demands of their customers as there are customers that would like to take in the newest software in more frequent time frames. Teams that use continuous integration are able to release software frequently (Humble & Farley, 2010) so for trying to find roadblocks in the way of adopting CD it was suggested that the feedback in CI was too slow for continuously releasing the software. Slow feedback was one of the reasons that forced the case company to adopt things like use of permanent branches in CI environment which then has worsened communication between teams. Also, it is one of the pain points that developers in the case company consistently point out. More detailed numbers on feedback are shown in chapter 6.1.

The matter of feedback seems to be important on both organizational level but also at the lowest level. The problem is something that needs to be solved for the company in order to optimize the continuous integration practices. CD takes CI practices to even a step further so it could be seen as essential to have functioning CI in order to adopt CD. (Laukkanen et al., 2015) More on CI and CD in chapter 3.

**Design evaluation:** It is essential for DSR process that the artefact is evaluated. The utility, quality and efficacy of a design artefact can be demonstrated via well-executed evaluation methods. The evaluation is based on the requirements set by the business environment. The artefact can be evaluated in terms of functionality, completeness, consistency, accuracy, performance, reliability, usability, fit with the organization, and other attributes that are relevant to give information about the quality of the artefact. If the metrics for evaluation are appropriate, it is possible to mathematically evaluate the artefact. (Hevner et al., 2004.)
Since design as an activity is an iterative and incremental activity, the evaluation phase provides essential feedback to construction phase as to the quality of the design process itself and the design product under development. The completeness and effectiveness of a design artefact can be determined by its capability to satisfy the requirements and constraints of the problem it is meant to solve. (Hevner et al., 2004.)

In this research the main requirement is to shorten the feedback in CI. The feedback is evaluated by comparing feedback when using test selection tool to current system that runs every test for each commit. In context of test selection, term safe is used to describe the selection technique’s ability to detect faults. Safe test selection technique means it will catch same faults as re-test all technique. Safety is another requirement even though it was agreed by the case company that a certain level of risk is acceptable. Still, the tests should be reliable and it should be possible to use those tests as a safety net. That is, because in the end if the feedback is wrong, it doesn’t matter how fast the feedback is. Metrics of evaluation will be explained in chapter 5 along with their limitations.

**Research contributions:** For DSR to be effective it should provide clear contributions in the areas of the design artefacts, design construction knowledge, and / or design evaluation knowledge. It must answer the question of “what are the new and interesting contributions?” DSR has three potential types of research contributions and at least one of them should be found in the research.

The design artefact is the most common contribution of design-science research. The artefact itself should provide a solution for unsolved problem. It can extend the existing knowledge base or apply new and innovative ways. Foundations are another contribution for design-science research. Foundations can be creative development of novel, appropriately evaluated constructs, models, methods or instantiations that extend and improve existing foundations in design-science knowledge base. Third type of research contribution in DSR are methodologies. They can be creative development and use of evaluation methods and new evaluation metrics. Both evaluation methods and metrics are essential for DSR. Beyond these three types, research must provide a clear contribution to business environment, solving an important problem that is yet to be solved.

The main contribution of this research comes along with the designed artefact as it aims to prove that test case selection can be used to enable continuous integration at large scale. It also provides evidence that test selection can be useful in embedded systems development. It also extends previous knowledge base and provides the readers way to help the practice of CI in large enterprises. By offering a solution to a problem it extends the existing knowledge base on how to scale continuous integration in embedded software development. This research also looks at the problems that organizations are having at adopting continuous delivery and discusses if the artefact could be used to in the adoption of continuous delivery.

**Research rigor:** Rigour addresses the way in which the research is conducted. DSR requires that the use of rigorous methods used to construct and evaluate designed artefact. DSR often uses mathematical formalism in its description of constructed artefact. Focusing too heavily on research rigour and formalism can however hurt the relevance of a research. This caused by the fact that in order to formalize something one might have to abstract their solution so much that it loses its relevance. (Hevner et al., 2004.)

Theoretical foundations and research methodologies form the knowledge base for the research of which the rigour is derived from. It is researcher’s responsibility to select a set of appropriate techniques to build a theory artefact and appropriate methods to
evaluate it. The evaluation of the artefact is usually based on performance metrics. (Hevner et al., 2004.)

This research attempts to use as much existing literature as possible in constructing the artefact. However, the literature on the subject is scarce and coming up with clear methods that are suitable for this research were hard to find. One could argue that some of the research on test case selection is abstracted so much that it is hard to apply the methods in their presented form. Still, previous research offer ways to evaluate the artefact. Evaluation of the artefact is based on metrics presented in the literature along with the metrics given by the case company. The main goal of this research is to reduce the feedback time in continuous integration environment so the main metric is the feedback time itself. Also, another important metric for evaluating the artefact is the number of test cases selected. Both feedback and number of test cases are compared to the current situation where every test case is executed for each commit. One thing that must be recognized when considering research rigor in this research is that the literature, that is used to answer some of the research questions, are not all scientific by nature.

**Design as a search process:** The design process should be done in an iterative fashion and it can be seen as a search process to discover effective solutions to a problem. Abstraction and representation of appropriate means, ends and laws are crucial factors in DSR. It is appropriate to simplify the problem by representing only a subset of the relevant means, ends and laws. The simplified representation, while not being realistic enough to be implemented as it is, can offer a good starting point for solving the problem. (Hevner et al., 2004.)

In this research the design of the artefact is iterated until it satisfies the requirements of the case company. It is possible that more requirements will rise between iteration cycles as the artefact is observed in real environment and both the researcher and the case company can see what is good about it and also the shortcomings of the current solution. The metrics that are used to evaluate the artefact can also change during the research as the artefact evolves. That is, because it is possible that the case company comes up with more metrics that are important to them that were not known to them when the initial requirements were set.

**Communication of research:** The research in design-science must be presented in a form that is understandable for both technical audience but also to management-oriented audience. The technical audience need to understand how to implement the artefact in appropriate organizational context. By enabling others to implement the artefact the artefact itself can be further evaluated and analysed. Management-oriented audience must understand the problem that the artefact can solve and how it could benefit their organization. (Hevner et al., 2004.)

In this research the original problem that the research attempts to solve is relevant from both technical and management-oriented point of view. The implementation of the artefact is explained but also its implications for the whole product that the organization is building.

### 2.3 Design Science Research Framework

The framework for DSR by Hevner et al. (2004) is used in this research as a general construct. This chapter will give the reader an overview of the environment the research is conducted, relevance of the research to business and also for its contributions for
scientific community. Also, the knowledge base used in this research is explained. Figure 1 shows gives an overview of how the framework is applied in this research.

![Design science research framework applied in this research](image)

**Figure 1**: Design science research framework applied in this research (Hevner et al., 2004)

### 2.3.1 Environment – Sub-RQ1

The environment in DSR framework is constructed from three actors: people, organizations and technology. People can have different roles, capabilities and characteristics that can have an impact on the environment. Organizations’ strategies, structure, culture and different processes are also something that make the environment. The technical environment is defined by describing the different technologies used in the environment such as infrastructure, applications, communications architecture and development capabilities.

**People**: Users of the constructed artefact are mostly software developers as they are the ones that are checking in code changes and currently suffering from slow feedback of the CI system. The hope is that the artefact will eventually result in less painful integration of software as faster feedback could potentially enable the software developers to practice continuous integration. Developers working in the target organization are fairly opinionated about the current situation of the CI environment. The state of CI in the target organization is explained in more detail in chapter 3.1.7. Also, the management are constantly worried about slow feedback times and testing hardware resources.

**Organization**: The product developed in a case company is a telecommunications product and its development has been distributed to multiple sites that are located in multiple countries. The product is not new and it’s been out there for several years by the
time of this research. It is composed of different system level components of which development is also distributed for multiple sites. This research is conducted in a line organization inside the case company and it develops one of the system components. There are 8-10 teams working on the system component on the site that this research takes place. Each team have about eight people and the teams are built in cross-functional manner as there are test engineers, specification engineers and software engineers in each team. Teams work mostly on maintenance, optimisations and new features.

The organization is going through a transformation as they are adopting Scrum and trying to do it at scale. The current teams are called Scrum teams or feature teams and some of the Scrum practices have been adopted. Some of the agile software engineering practices have been adopted as well. Pair programming and test-driven development are not the norm but the engineers are encouraged to practice them and, to some extent, people are. They also have continuous integration system in place but the way continuous integration is practised could be seen as sub-optimal. Chapter 3.1.7 has more detailed description about the state of CI in the target organization.

This research could benefit the organization in a couple of ways. The first one includes all the benefits that a successful continuous integration can bring. The organization is moving towards continuous delivery which was a strong motivation for this research as the continuous delivery can be seen as a continuum of continuous integration (Fowler, 2013.) Because the organization is somewhat struggling with continuous integration, this research could help them in the process of moving towards more frequent deliveries. The organization is also moving towards having real feature teams and collective code ownership which means there will be more sites and more teams involved in developing the system component which itself requires functioning CI system. Another organizational motivation is that the continuous integration system is heavily relied on testing environments and because of that a large number of hardware resources are required to maintain the CI system. The problem with that is that the testing machines are expensive and they are limited in numbers. By replacing the current, re-test all, policy that requires huge amounts of hardware to run every test on every commit, with a test selection tool, tests wouldn’t be executed unless they’re relevant to the committed code change. This could have a huge impact for their current situation as they wouldn’t have to execute unnecessary tests.

**Technology:** A few different technologies are involved in this research. The software product the case company is making is a large-scale embedded system written in C++. The tool for continuous integration is Jenkins that handles the execution of different builds, tests and static analysis. The feedback from Jenkins can be seen from its graphical web interface. Jenkins is a cross-platform, open-source continuous integration and continuous delivery application that can be used to build and test software projects. (Jenkins-ci.org) The CI system is visualized in Figure 2.

The test selection used in this research is done for system component tests (SCT) that run on a target. The SCT uses RobotFramework as a testing framework. It is a generic and open-source test automation framework for acceptance testing and acceptance test-driven development. It uses keyword-driven testing approach and it can be extended with Python or Java. (Robotframework.org) The RobotFramework is used as a test runner as well as a simulator of other system components.

The test selection artefact constructed in this research doesn’t start from scratch. The reason for this is that there was already an initiative for such work as a developer in the company had implemented an algorithm that could be used to check manually that which
of the test cases test a certain file. However, that script could be only executed manually and it wasn’t effective in terms of how much time had to be spent to run the script. This research uses that same algorithm to select test cases but automates it and makes it usable in CI environment.

![Diagram of CI environment](image)

**Figure 2**: CI environment

### 2.3.2 IS Research

The design artefact of this design-science research is an automated test selection tool that works in continuous integration environment. The implementation of the artefact doesn’t start from a scratch as the code instrumentation and test coverage collecting scripts were written before this thesis started. However, they were not fit to work in continuous integration environment in their current format and features had to be added to help in the automation of the selection process. The metrics that are used to evaluate the artefact are feedback time, number of test cases selected and its fault-detecting capability compared to running a whole test suite. The data is gathered by different tools that collect statistics from continuous integration tool Jenkins. The safeness of the test selection technique will be evaluated by measuring its fault-detecting capability which is calculated from faults that gets through the test selection. For example, if a test case fails in whole suite, and it’s not executed in the selected sub-set, it means the artefact wasn’t able to detect that fault.

The justification to the need for such artefact is argued by the studying benefits of fast feedback in continuous integration. Also, the problems of slow feedback are used to justify the need for the artefact. The benefits of a functioning CI system are also used to highlight the importance of the artefact for the target organization and how big of a role CI plays in the process of scaling agile software development.
2.3.3 Knowledge Base

The knowledge base for this research is a combination of existing literature and previous scientific research. The literature used in this research consists of subjects such as test selection techniques, feedback times in CI, scaling agile software development and CI in embedded applications. The metrics to evaluate the artefact are chosen by consulting existing literature on test selection and what metrics are commonly used to evaluate a test selection technique and by prioritizing of what metrics are matching the interests of the case company.
3. Continuous Integration and Continuous Delivery

This chapter gives an overview of the continuous integration (CI) and continuous delivery (CD). While shortening the time of your test runs and getting faster feedback are important part of the software development, there are also organizational aspects to CI and CD and the problems with adopting them are not purely technical. This chapter will explain the concepts of CI and CD and how they are both strongly tied to the goals of this research. Both CI and CD are discussed in general terms but the focus for both will be from large organizations’ point of view. After reviewing literature, the state of CI in the target organization will be discussed and reflected against findings in the literature.

3.1 Benefits of Continuous Integration

One of the benefits of CI is the rapid feedback it provides. The feedback about the state of the project is provided several times a day when integration is done continuously. The rapid feedback allows developers to find about defects quickly after it’s been introduced thus it can reduce the time between when a defect is found and when it’s fixed. In other words, doing CI can improve the overall quality of the software. (Duvall, 2007) The constant feedback also enables software development practices like refactoring and test-driven development (TDD) as the small changes can be quickly integrated with the whole system and a developer can get feedback shortly after each commit. Also, just as much CI enables TDD it can be also viewed that it works also the other way around. TDD helps in splitting large changes into smaller ones thus making it easy to integrate changes in small pieces. If TDD is practiced it is possible that the changes can be integrated even as frequently as after each TDD cycle. (Larman & Vodde, 2010) However, as it is the subject of this research, the feedback times can increase as the build times go up. Rapid feedback can be seen as a benefit of CI but it also possible to lose that benefit when builds take up hours. The long-lasting builds and scaling CI will be discussed further in chapter 3.1.3.

One of the main benefits of continuously integrating is to move some of the communication between developers to the code level. If integration is done frequently developers can communicate to each other at which part of the system they’ve touched thus find out about the conflicts in their code sooner rather than later. A conflict occurs when two or more developers have changed the same few lines of code of a file (TortoiseSvn). According to Fowler (2006) the key to fixing problems quickly is to find them quickly. The earlier you find out about a conflict, the easier it is to fix. On the other hand, if the integration is done e.g. every two weeks it is possible to have undetected conflicts hiding in the code without developers knowing. Those types of conflicts that stay undetected for weeks or even longer can be very difficult to solve (Fowler, 2006). Finding out about conflicts soon is not the only communication related benefit of CI. Being informed of someone else touching the same code file as you are, can open a communication between two developers. In that sense, CI works as a trigger of communication between people and it enables people to interact and work together.

3.2 Branching in Continuous Integration

The idea in CI is to have everyone committing to same mainline. However, some projects distribute their development to one or more branches. There are a few reason why this is done. Branching could be done by feature, release or by team. The reason could also be physical as the different files, components or subsystems are branched. The branching
can also be done of a system’s functional configuration or it can be based on the build and runtime environment. There could also be organizational motivations for branching when branches are created for activities/tasks, subprojects, roles and groups. The reason might be also procedural when branches are created to support various policies, processes and states. Larger organizations might have branches for different components and products. (Duvall, 2007.)

Branching in CI is a somewhat controversial topic. That is because by the definition, working on a branch is not continuously integrating because developers are not integrating with mainline. The changes in a side branch are moved to the mainline by merging. The merging means taking the changes from the branch that is being worked on and applying them to mainline. If the branch is not merged to mainline at least daily, it cannot be called CI. (Duvall, 2007.) Also, Humble and Farley (2010) say that submitting to a mainline is the only way of doing CI which can interpreted that developing in long-living branches is not actually CI. It is apparent that there are different views about what qualifies as CI. Fowler (2006) recommends to use branches as little as possible. Though he does mention some useful use cases for branches e.g. feature branches are useful when a developer (or a team) can work on a feature in isolation while still pulling the current version from the mainline. Using feature or team branch the developers can decide which of the features are going to be released and merged to the mainline. They can also be used for complex refactoring where some of the core elements of the system are changed. This could be for example making capacity improvements or rearchitecting layers of the system. Even though branches can be useful in such situations, another approach could be to make the changes in small incremental steps while keeping all the tests passing (Humble & Farley, 2010). Larman & Vodde (2010) also highlight the importance of integrating to the mainline. They claim that making changes in a branch means that the integration with the mainline is delayed. Their view on branches is that they shouldn’t live longer than hours and they should be overall avoided.

The conflict problem was discussed earlier in previous chapter and branching can bring a new level to that problem. The problem with branching is that the probability of conflicts emerging when working in separate branches increases. If no conflicts arise during the merge, the process is fast and easy. However, the problem with branches and merges comes around when the changes made in a side branch are in conflict with mainline or the other way around. (Duvall, 2007) If merges are not done frequently, a conflict can exist for a long time between the mainline and a branch without anybody knowing. Working on branches can potentially lead to very difficult merging and integration process at the end of the project (Humble & Farley, 2010). A real-life example can be gotten from Larman & Vodde (2010) where they worked in a project that used branches for different configurations and used those branches as development for a year and then another half-year just to merge the changes to mainline. Duvall (2007) has similar experiences as he points out that the merges will be more painful if they are delayed.

Another problem with long-living branches is that the fear of merge conflicts can hold developers back from refactoring the code. They can be afraid to move code between classes or renaming methods because they don’t want to deal with merge conflicts once the branch is merged to mainline. To give an example from the case company, there has been times where a developer renames a single variable and then spends rest of the day fixing broken builds and solving merge conflicts. The lack of refactoring can lead to increasing amount of technical debt. (Naik, 2015.) The unwillingness to refactor means that developers in that organization cannot follow “The Boy Scout Rule” by Martin (2009) in which you leave the codebase in better state than you find it. Doing little clean
ups for each code check-ins and keeping codebase clean could be made impossible by using branches because of the fear of merge conflicts they cause.

Fowler (2009) states that using branches takes out the aspect of human communication from CI which he sees as bad because communication can be seen as a key factor in software development. If software is developed on mainline developers can pick up each other’s changes any time (Humble & Farley, 2010) If not, other developers are not able to pick up the changes quickly enough. This can cause code reuse and duplication efforts. For example, there have been cases in target organization where a team was working on an algorithm that another team was also working on without teams not being aware of each other’s work.

3.3 Build Times in Large-Scale CI – Sub-RQ4, Sub-RQ5

Laukkanen and Mäntylä (2015) did a literature review on effects of build waiting times in continuous integration environments. They found three articles that were about build waiting times. This means also this thesis must rely on only a few sources of information on this subject as there haven’t been new studies made on the subject since Laukkanen & Mäntylä (2015) study.

Large projects can struggle with CI. The concern when the code-base consists of millions of lines of code is that the build times are going to be long. (Duvall, 2007) The amount of time a developer has to wait for build to finish can be critical to developer’s work flow. For example, Beck (2000) says ten minutes is the maximum time for build to finish. However, with large code base, a good suite of tests is needed to verify the correctness of the software. This presents a problem in a sense that large number of tests needs to be ran but at the same time a short build time is desired. Still, stopping the development work just to wait for feedback can slow down the rhythm of development for everyone working on that project. Slow builds can even impact how people perceive CI as a practice and they can be the reason why a developer is integrating infrequently. Thus, getting rapid feedback is very important. (Duvall, 2007.)

Brooks (2008) compared effects of build waiting time based on his experience with two software projects. One project had a build taking 2 minutes and the other 20 minutes. He makes several findings from those experiences. According to his experience long build waiting times resulted in infrequent code check-ins, avoidance of small commits, harder fixing of a broken builds, interrupted development flow and reduced developer satisfaction. He found that long builds would make people picky when it came to choosing their time to integrate. It varied from not wanting to check the code in until they had enough time, to the point where people did weekly check-ins and just accepted the amount of work it required. Long build times also affected the developers’ willingness to refactor. That caused all kinds of problems with the team as the lack of refactoring led to lots of technical debt. The improvement ideas were put on the backlog and they were supposed be done when there was time and priority allowed. On other hand, the team with two minute builds were more willing to do major refactoring thus they weren’t overwhelmed by the technical debt neither did they suffer for any previously mentioned problems.

Rogers (2004) states that long builds lead to infrequent integration whereas short builds lead to very frequent integration. Medium builds make developers integrate several times a day. In his paper, he refers long builds as builds taking more than 30 minutes, medium builds as builds taking 5-10 minutes and short as builds taking less than two minutes. If the builds are taking more than 30 minutes the integration is likely to happen once a week
and a whole day for the integration process will be allocated. He also points out that long builds make integration effort higher and when that happens the whole integration process becomes an interruption which forces developers to schedule their integration.

Rasmusson (2004) also reflects his experiences on build times in his paper. He has found out that short builds provide the developer with fast feedback whereas long builds cause slow feedback. The slow feedback itself can cause pain to the developers. Waiting for the build to finish can also be distracting and if the waiting time is long so the developer is likely to do other things while waiting. He goes to as far as claiming that long builds lessen team morale.

Long builds can have effects that are strictly CI specific but they can also have cognitive and emotional effects. Build time affects commit sizes and frequency, build down time and integration effort. They affect development flow and feedback time and can also have an impact on developer satisfaction and team morale. (Laukkanen & Mäntylä, 2015) It was mentioned earlier that CI enables agile software development practices like refactoring and test-driven development. This might not be the case when the build times are too long. Based on conclusions of Laukkanen & Mäntylä (2015) one can argue that long builds can affect how a developer writes code and how it can take away some of the benefits that CI would ideally provide.

3.4 Growing Code-Base and CI

When continuous integration is introduced to larger projects there is a chance that it will turn out to be a burden. If a developer is getting email notifications of a broken build multiple times a day due to someone else breaking the build, it could potentially become just continuous noise. Unstable mainline is hard environment to make progress in. It should be noted though that CI doesn’t cause instability but it exposes it. (Poole, 2008.)

According to Rogers (2004) the growing code base itself does increase the compilation time but the compilation time itself isn’t as big as of a problem as the growing number of tests. The build time can potentially increase exponentially as the number of tests goes up. The problems related to growing code base are highlighted when more people are added to a project. The first problem is that the volume of code being produced goes up which makes the problems with growing code base even bigger. The other problem with growing number of people working on a same code base is that there are more and more people who depend on a working build, thus breaking a build will affect lots of people. In CI committing on top of the broken build is prohibited. (Rogers, 2004) According to Martin (2014) it is important to keep CI tests running at all times and if someone breaks the build, the whole team needs to stop and focus on fixing the build. In reality though in some cases people can ignore the broken build and commit their changes despite it is not allowed. This can cause several problems as it is impossible to verify the correctness of the software if the build is already broken. If this kind of behaviour becomes common, it is possible for developers to consider the broken build as a free-for-all situation where everyone just dump their changes on top of a broken build. (Rogers, 2004)

Despite the previously mentioned ‘always green’ mentality in CI, there are some controversy surrounding that mind-set. Larman & Vodde (2010) talk about their experiences working with organizations where the policy of always keeping a working build led to a situation where breaking the build led to shaming of the people who broke it. This resulted in developers delaying their integration out of fear of being shamed which meant that despite constantly having working build, they were not doing CI.
Still, a broken build is a serious problem in a development process. It is likely to undermine productivity and morale of the developers. According to Rogers (2004) it is a common approach to force a strict pre-commit procedure that will make it as hard as possible to break a build. This kind of pre-commit procedure can consist of running the whole integration build on a local machine and only if the build passes, committing the changes is allowed. This is an effective way to keep a working build but it also increases the integration time. For some perspective, that kind of mind-set can be compared to idea of Larman & Vodde (2010) where they suggested that integration could be done as often as each TDD cycle. Obviously each TDD cycle can’t consist of running the whole integration build so these kind of pre-commit policies can reduce the integration frequency or make it impossible to integrate frequently. Because of these negative effects of pre-commit policies Rogers (2004) states that it should be acceptable to break the build because the integration server is there to provide that feedback if something goes wrong with a code change. He also says that turning every developers’ local machine into integration machine is an ineffective way to do integration as it can hurt the productivity and morale of developers. Testing too much stuff on local machine before commit can become a bottleneck for the whole development and CI process (Elbaum, Rothermel & Penix, 2014).

3.5 Scaling Continuous Integration in Literature – Sub-RQ5

The literature surrounding continuous integration at scale is somewhat scarce. Especially academic publications about large-scale continuous integration are very few in numbers. There are some real-life examples of companies that have successfully adopted CI at large-scale and there are also guide-lines and books written by experts that will give some guidance how it can be done. One of the biggest problems of these guidelines given by literature is that they are general whereas the CI problems can be product-specific which means they are not very helpful or applicable in every situation. Also, the lack of scientific research on scaling CI is somewhat troubling.

One of the most referenced paper is Rasmusson (2004) as it provides the list of general solutions for making builds faster. Larman & Vodde (2010) used that list in their book and added some of their own insight for each solution. Also, Rogers (2004) and Duvall (2007) offer some general strategies for successfully scaling CI. Whittaker et al. (2012) have published a book of how Google tests software and they bring some insight of how Google managed to implement CI in truly large-scale software development process. Still, even they don’t go into the specifics of how they did it.

Before going into how to solve problems in scaling CI, Rogers (2004) brings up a point that could be seen as a key factor in keeping builds fast. He recommends that a maximum build length should be established at the start of any project. Just by doing this people are forced to come up with strategies to keep the build times where they need to be. It also makes organizations determine how long is too long and when a long build will have negative effects for whole product. It is possible that if the maximum build time is established and everyone agrees on it, it is easier to allocate resources for keeping the build times in agreed state.
<table>
<thead>
<tr>
<th>Root Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow hardware</td>
<td>Acquire faster hardware</td>
</tr>
<tr>
<td>Poorly written test</td>
<td>Re-visit original intent of test</td>
</tr>
<tr>
<td>Testing at the wrong architectural layer</td>
<td>Write tests at proper architectural layer</td>
</tr>
<tr>
<td>Network Intensive Tests</td>
<td>Stubs and mocks</td>
</tr>
<tr>
<td>Large code base</td>
<td>All above</td>
</tr>
<tr>
<td></td>
<td>Break application into sub domains</td>
</tr>
<tr>
<td></td>
<td>Run tests in parallel</td>
</tr>
<tr>
<td></td>
<td>Serialize the check-in process</td>
</tr>
</tbody>
</table>

**Table 1: Long Build Trouble Shooting Guide (Rasmusson, 2004)**

Table 1 shows Rasmusson’s (2004) Long Build Trouble Shooting Guide that shows root cause of problems and possible solutions. Adding more hardware or acquiring faster hardware is the easiest solution to speed up the build. (Larman & Vodde, 2010, Rasmusson, 2004) Just by adding more computers, extra memory or a faster network connection, a build can be speeded up with only spending some money. Larman & Vodde (2010) observed a 50% improvement in compilation time by just adding memory when they were working on a telecom product. They recommend it as the easiest and best practice to speed up builds. However, Rothermel (2016) raised up a point that the more resources you add, the faster the amount of testing increases and you will end up using all the previously acquired resources. Also, there is a potential problem with buying better hardware if the product being worked on is some kind of embedded system. If tests are executed in a real target, acquiring faster hardware might not be an option. On the other hand, if code is compiled and tested in the cloud, it is possible to just use more powerful hardware to speed up the build.

Another solution that is somewhat related to adding hardware is to parallelize and distribute the build. It is not as easy and straight-forward as just acquiring better computers as it requires redesigning build scripts, changing tools or building new tools. So just comparing to hardware acquiring, parallelization requires more effort but it can also be helpful method to make builds faster as it’s been used successfully in practice. (Larman & Vodde, 2010). For example, the case company executes some of the largest regression suites in parallel.

Larman & Vodde (2010) present an idea where software could be built incrementally. Building incrementally means that only changed components are compiled and tested. According to them it is easy to do in theory but in practice things tend to get more complicated. The hard part of incremental builds comes from dependencies between components, changes in interfaces or incompatible binaries because they make determining what to compile hard. For the same reason choosing the right tests is hard. It is also pointed out that incremental builds are not completely reliable so if one decides to do incremental builds, it’s a good practice to keep a clean daily build.

Refactoring the tests can also speed up build times. (Rasmusson, 2004, Rogers 2004) Refactoring test code is usually needed as some developers care less about test code than
production code which leads to bad structured tests and causes them to run slow (Larman & Vodde, 2010). Both Rasmusson (2004) and Larman & Vodde (2010) were able to gain notable results from refactoring test code. Rasmusson used one test as an example where he was able to take a test that took 30 minutes to execute and make it finish in 3 minutes. Larman & Vodde (2010) shared their experiences about a time when they spent a half-day refactoring tests and were able to speed up the build by 60%.

According to Whittaker et al. (2012) Google approached the problem of solving how to scale CI from a different angle. In traditional CI, the process goes as follows: 1) get latest copy of the code 2) run all tests 3) report results 4) repeat 1-3. They admit that this works for small code bases when builds run fast and it is possible to run all tests quickly. However, as code base grows, each clean build lasts longer and each test run will have more changes to test. If something breaks it can be difficult to find out which change actually broke the system. Google wanted to have a CI system that would always tell the exact change that broke the build. They came up with a CI system that would do a dependency analysis for each commit and only choose tests that need executing based on the changes.

In Google, everyone submits their code to the mainline but with so many commits coming in they couldn’t afford having their build broken all the time. According to Rothermel (2016) they have split the submit process into two phases; pre-submit and post-submit. Before changed is integrated with mainline, a set of tests is selected and ran and the selection is based on dependencies of the changed code. If the selected tests pass, the code is then integrated with mainline. Rothermel (2016) also suggests his own idea in order to provide as fast feedback as possible while maintaining reliability for test results. He suggests that test selection technique is used to select tests for pre-submit phase and test prioritization is used in post-submit phase. In post-submit phase every test is executed but in prioritized order where the tests that are most likely to fail, will be ran first. That way a developer is more likely to find out if his / her changes broke the build. The test selection is somewhat related to what Larman & Vodde (2010) said about incremental builds where only necessary parts of the system would be built and tested but they are not using test selection as the term for this. Still the idea behind their approach is similar to test selection as only necessary tests are ran in incremental build. Test selection techniques will be discussed in more detail in chapter 4.4.

Like mentioned earlier in this chapter, there is a lack of scientific literature on how to scale continuous integration. Larman & Vodde give simple do’s and don’ts for doing CI in their website (LessWorks) based on their findings in Larman & Vodde (2010). Their advices are shown in Table 2.
<table>
<thead>
<tr>
<th>DO</th>
<th>DON’T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit after every TDD Cycle (e.g. every 5-10 minutes)</td>
<td>Develop on branches</td>
</tr>
<tr>
<td>Build quality in to the code to support optimistic integration</td>
<td>Have a policy to review code before commit</td>
</tr>
<tr>
<td>Make it easy to fail fast, stop &amp; fix and learn from ‘mistakes’</td>
<td>Have a “don’t break the build” policy; don’t have blaming and shaming culture</td>
</tr>
<tr>
<td>Stop and fix when build breaks</td>
<td></td>
</tr>
<tr>
<td>Use visual management showing the state of the build</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: DO’s and DON’Ts of CI (LessWorks)

Even though being against branching in CI, Humble & Farley (2010) offer some advice how and when to do it. He gives tips for branching by release, feature and by teams. From those three, branching by team is the most relevant for this research. Branch by team pattern attempts to address the situation where there are a large team working on multiple work streams while maintaining a common mainline that is kept releasable. The merging process in this pattern is that a team branch should be merged to mainline when branch is at stable state and each time a team branch is merged to mainline, other team branches should pull the newest mainline. The mainline should be merged to team branches daily. An example of branching by team is shown in Figure 3.

<table>
<thead>
<tr>
<th>Trunk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team 2</td>
</tr>
<tr>
<td>Team 1</td>
</tr>
</tbody>
</table>

Figure 3: Branching by team

Testing in team branches and mainline should be done differently. Testing should be more robust in mainline than in team branch. For example, in a team branch every commit should trigger unit and acceptance tests whereas in mainline all tests are ran including integration tests. (Humble & Farley, 2010.)

Larman & Vodde (2010) talk about multi-stage CI for legacy systems or systems that are just big in size. The different stages could be at fast component level, slow component level, product stability level, feature level, system level and stability-performance level
stage. They point out that having all these in one pipeline is unusual and it’s more likely for a company to choose the stages that fit their needs. They all differ in complexity as for example fast component level stage works at very low level running unit tests, code coverage, static analysis and complexity measurements. This level should be very fast and the feedback should also be fast. At stability-performance stage the stability and performance tests can take days or even weeks.

3.6 Continuous Integration in Embedded Systems Development

Until this point continuous integration has been discussed only in general terms without paying much attention to the environment the CI is practiced. It should be noted that the literature cited in previous chapters are about CI in sort of general context and specifically not embedded systems. Since this research takes place in embedded system development, a special focus is to be given of how embedded environment affects practicing of CI. This chapter will introduce the problems embedded systems can cause for the practice of CI and how CI in embedded systems is represented in literature.

Mårtensson, Ståhl and Bosch (2016) studied two companies that make embedded systems, one a telecommunications company (company A) and the other company (company B) develops airborne systems, main product being Gripen fighter aircraft, and their support systems. In their study company A had implemented an advanced system of automated build and test to support CI. Build, test and different types of analysis were ran based on events and fixed schedules. They had a wide range of physical target systems and simulators that were used to test their system. Company B practiced CI as they had automated tests, private builds and integration builds servers. Mårtensson et al. (2016) also mentioned that company B had one common mainline where every team commit their code.

Mårtensson et al. (2016) listed problems the two companies faced in practicing CI in embedded environment. Some of there are more relevant for this research than others. For example, they talk about usability as the company B had users directly interacting with their products. Then again with company A, the company didn’t have users directly using the product and the usability from customer’s point of view is mostly the connection speed and reliability. Keeping that in mind, usability in embedded systems is not relevant for this research. One thing that they stated that was common for both companies and could potentially apply to other embedded systems as well, is that number of test environments is limited as testing hardware can be expensive. That also means that as testing of the product is dependent of number of hardware resources, also practicing is dependent on testing hardware. Their findings on build waiting times matches the findings mentioned in previous chapters as long build times lessen the willingness to integrate frequently.

Large embedded systems bring up interesting aspects to even the most basic rules of continuous integration. The common idea in CI is that after a commit, 100 % of the tests must pass. Mårtensson et al. (2016) question the definition of this rule in a large embedded system. A large number of hardware configurations increase the test effort as different configurations demand their own test environments. They point out that having many different configurations increase the risk of having flaky tests in a test suite as there are more test environments to maintain. Given that, and the fact that in embedded software development tests must be executed on a real target, most likely in multiple different deployments, 100 % of the tests passing requires a lot of testing and a risk of breaking the build can increase because of the flaky tests.
According to Mårtensson et al. (2016) practicing CI in embedded systems is made difficult by the need to have enough test environments to execute the tests after each commit. They argue that simulated environments cannot fully solve this problem because of the uncertainty whether the same tests will pass in a real hardware.

3.7 Continuous Integration in Target Organization – Sub-RQ4

This chapter describes the current situation with CI in target organization. The main focus is on the status of CI in the line organization where this research was conducted. The information represented here is gathered from conversations with different individuals, daily stand-up discussions, email conversations and meetings considering this thesis.

CI in target organization works like a pipeline where each system level component has their own trunk. Once a trunk is promoted it moves up in the pipeline and is integrated with other system components. This goes on until the whole product is in place and the system is ready for testing. A developer working on a system component only sees the trunk for that specific system component. This research is not focused on the whole pipeline but on CI on a system component level. There is a specified team in the line organization to take care of the integration of changes coming from other system components. Their job includes for example taking care that an interface change in another component don’t break the build in their own trunk. This kind of CI pipeline is similar to what Larman & Vodde (2010) introduced as a multi-stage CI. The CI pipeline is shown in Figure 4.
A couple years ago the decision to move development to team branches was made due to the lack of promotions from the trunk. Until that point every commit went to the mainline and was compiled and tested there. However, there was a problem with builds being broken for long periods of time and feedback was very slow for each commit. Slow feedback also resulted in one test execution testing several commits together which would make finding out the fault-introducing commit difficult. Slow feedback also led to situations where the whole CI pipeline was affected by broken trunk as the trunk wasn’t getting promoted to the next stage in CI. The situation with state of the trunk resulted in some developers being afraid to commit their code to trunk and would delay their integration for the fear of breaking the build. By using team branches the management and developers wanted to make sure that no untested code will go to trunk.

There is a policy that if a build gets broken in trunk, there is a 15-minute window for a responsible person to use CI system’s ‘claim’ feature to inform everyone that he was responsible for breaking the build. Also, the person who broke the build should be informed immediately of trunk’s status in case that nobody has claimed the broken build. If the build is still broken after 30 minutes anyone has the right to revert the faulty commit.
and inform the person responsible for the commit that his changes have been reverted. However, even though it is a policy written in the main page of the CI system’s web interface it is rarely, if ever, followed. There are several potential reasons for the unwillingness to follow these rules. Maybe the biggest is the state of the testing environment. Test results are not trusted and it is a common practice to re-start the test to verify the correctness of the test result. Re-running the test will take more than 15 minutes which means following the first part of the policy is not possible. Sometimes even re-running the test isn’t enough to proof that the original result was correct. It is possible that the test equipment has some trouble updating itself or there is a problem with network connection. In those cases, a developer must manually force the CI system to run the tests on a different machine. Constant communication with the CI team is often needed. Another reason for unwillingness to follow the policy are the human factors of reverting someone else’s changes. Reverting someone’s changes can be perceived as a personal offence and not everyone is comfortable doing it. More common practice is just to ask someone to revert their own changes via email.

There is also an e-mail policy to some extent in a sense that the person responsible for breaking something is automatically mailed a report provided by CI system. However, it seems that at some point these emails lost their meaning due to the fact there are constantly broken builds and tests and everyone still commit their code despite the state of the trunk. This means that everyone who makes a commit to broken trunk receives emails accusing them for breaking the build even though their commit wasn’t the actual cause for the broken build. This has resulted in situation where emails are sometimes completely ignored and even filtered out by developers.

There has been also some juggling with email policy and sometimes emails are not automatically sent. To some people this could be seen as a relief because now their inbox isn’t getting flooded with emails from CI tool. For others, though, it has caused more trouble as they still rely on the CI system to inform them of the status of the build and not being aware of the changing email policy can make broken builds go unnoticed. Disabling emails from CI means that the only way to get feedback from CI is to watch the CI system’s web interface and see if the all tests will turn green. There has been some effort in preventing the trunk from getting broken. For example, there is an auto-revert that will automatically revert the commit that broke the build. This however doesn’t always work or it can frustrate developers by unnecessarily reverting commits. There have been situations where the code that was submitted contained only small changes in scripts that weren’t tested by CI but still the acceptance tests failed due to the flakiness of the tests and the failing tests then triggered the auto-revert.

Moving to team branches has caused some internal disputes between those who believe in true continuous integration (true in a sense that CI is truly CI only when everyone is submitting to trunk) and management who are concerned about the amount of daily promotions of the trunk. It should be noted though, that there are many developers who prefer team branches because they feel more secure and comfortable submitting their code. So, in a sense migrating to team branches has helped developers to continuously integrate their changes but it has hurt CI process at system component level and even the later CI stages.

By looking at the commit history it could be argued that currently the line organization is not doing CI if the definition of CI from Duvall (2007) is used. He states that if a branch is not merged daily to mainline, it is not CI. At the moment, a team branch is merged to trunk whenever the team sees fit. Trunk is automatically merged to every team branch once a day. The rate of which merges are made is hard to measure as they can be done
manually from command line but also the CI system has a job where a button press will trigger a merge. Frequency in which merges are made is sparse which often causes the two branches to conflict with each other. This requires a developer to solve the conflicts manually and then perform a manual merge. Manual merges are hard to recognize as merges because they are seen as a regular commit in the commit log which makes measuring the frequency of merges difficult. There is some differentiation how often teams integrate their branch with trunk but most don’t do it daily and there have even been situations where a team merged their changes that had been sitting in their branch for a couple of months.

One possible solution for avoiding conflicts and improving communication at the code-level would be automatically merging team branch to trunk whenever tests are passing. The decision for not to automate merges to trunk was made for several reasons. One was that conflicts wanted to be solved by people and not by computer. Conflicts are common with infrequent merges so it is assumed that there would be conflicts if merges were automated. Another and maybe even a more important reason is the lack of resources. Every commit would trigger a full build including every test which means that the team would run out of testing equipment if so many merges would be going back and forth between branches and trunk. Also, currently the regression testing is taking hours to run and teams want to make sure that their merge won’t introduce new regressions in trunk. Only way to be sure of that would be to wait until regression tests are passing.

Moving to branches did not remove all the problems with CI and there are still consistently problems getting promotions from trunk. The practice is, that when trunk is broken for a long period of time, the version control gets locked so no other changes are allowed in order to get tests passing again and to get trunk promoted. Once the trunk is promoted, it is unlocked again. The decision to lock the trunk is made by the team responsible for adaptation and a line manager. During the last couple of months, a policy was adopted where committing to trunk was prohibited and only commits allowed were the ones that had sufficient prefix in the commit message i.e. “PROMOFIX”. Also, merges from branches are allowed. Adopting this policy has not been very successful as some developers still don’t want to submit to team branch. Developers have created temporary branches for themselves and used them to submit their changes straight to trunk to go around the new restrictions.

As for testing, there are some differences between trunk and team branches. Generally, the same tests that are ran in branches are also ran in trunk. Trunk additionally has more thorough and frequent testing. Test jobs in trunk are also given a higher priority than the ones in team branches. The promotion criteria tests are triggered by every commit. At this stage of the CI pipeline commit triggers unit tests, component tests, acceptance tests and different types of static code analysis. Acceptance tests are running in a real hardware so running every test for each commit is eating a lot of resources. This can also affect developers as they might have to wait for the test machine to free up. Especially at the end of the day the feedback can be very slow as every test machine is reserved.

There are also lots of flaky tests that fail randomly or due to a problem in the system running the test. The flakiness is also caused by hidden dependencies between tests and timing in tests. That causes uncertainty and frustration amongst developers because a failing test causes them stop and investigate the errors even though their change had nothing to do with the test failing. It can also lead to falsely accusing someone for breaking a test even though just re-running the test will make it pass again.
3.8 Continuous Delivery – Sub-RQ6

This chapter offers basic knowledge about continuous delivery based on the literature made on the subject. Even though the designed artefact is not related to CD, it was one of the biggest motivations for this thesis as the company is moving towards more frequent releases. Also, the goal of this chapter is to argue that continuous delivery can be achieved through successful continuous integration which makes the designed artefact also relevant for CD.

Continuous delivery is a set of practices and principles to release software faster and with higher frequency. (Krusche & Alperowitz, 2014) It keeps teams producing software in short cycles and it keeps the software in releasable condition at any time. It can be achieved by having discipline and by automating the delivery itself, including building, testing and deploying the software. (Laukkanen et al., 2016) Continuous delivery should not be confused with continuous deployment which takes the practice of CD to the extreme. As it is stated above, in CD the emphasis is on the fact that software is in releasable condition whereas in continuous deployment the software is continuously released. The distinction between the two is that in continuous delivery a company can decide whether they want to release the software when it’s in releasable condition. (Laukkanen et al., 2016.)

The promise of CD is that it enables organizations bring service improvements to market and stay ahead of competition in rapid, efficient and reliable form. While this is something to be desired for every organization, it can be challenging to implement CD. Especially for large enterprises that already have existing environment for releasing it is challenging practice to adopt. (Chen, 2015.) Also, being able to release continuously is often blocked by different bottlenecks in the delivery process.

Even though adopting CD can be challenging, it has been adopted successfully in some organizations. (Rahman et al., 2015) There have also been companies that have found adopting CD challenging. (Laukkanen, Paasivaara & Arvonen, 2015) Laukkanen et al. (2016) came in to a conclusion that even though there have been reports about problems in CD adaptation there hasn’t been research that focus on causal mechanisms of those problems. They studied a bottom-up adaptation of CD and reported the problems the people involved faced during that process.
Like shown in Figure 5, slow feedback is listed as a direct sign of dysfunctional CD practices. There are three root causes that lead to several process mechanisms that cause slow feedback. The root causes of slow feedback can be tracked down to stage-gate development process and lack of testing strategy. In this case those have led to delayed integration, slow builds and limited hardware resources. Limited hardware resources and delayed integration can be tracked to use of multiple branches. Slow builds are caused by duplicate testing which stems from lack of testing strategy. For the purpose of this thesis the findings of Laukkanen et al. (2016) are particularly interesting as they are tied also to the findings of this research and they are also problems that this research attempts to solve.

Figure 5: Overview of the direct signs, root causes and mechanism in between (Laukkanen et al., 2016)
4. Test Optimization Techniques – Sub-RQ1, Sub-RQ2

This chapter is about different techniques on test optimization in software testing presented in literature. The motivation for attempting to optimize how software is tested comes from the changing software industry. Nowadays organizations must be able to adapt to constantly changing requirements and be able to release software with new features faster and more often. Software testing is a common bottle-neck for companies’ continuous integration process. (Knauss et al., 2015.) According to Nilsson, Bosch & Berger (2014) efficient testing is a key factor for working CI and still most companies tend to struggle managing the complexity of testing activities. They claim that inadequate overview on companies’ testing leads to double-work, slow feedback loop, high quantity of post-release faults, disconnected organizations and unpredictable release schedules.

Challenge in managing testing activities is to know when to run tests and what tests should be executed. The simplest solution to software testing is a method called retest-all (Yoo & Harman, 2012.). Many view testing as it is better to test everything just in case and even test more than it is needed (Knauss et al., 2015.) This approach can work well if the code-base is relatively small and the tests don’t take a lot of time to run to begin with. However, as software grows it can become more expensive to run entire test suite. (Yoo & Harman, 2012.) By test suite, this thesis refers to set of test cases executed together. Running every test can also provide an organization with false confidence in quality of the product as well as it can eat up lot of resources (Knauss et al., 2015). Even a company like Google avoids running all tests despite their massive computation resources (Whittaker et al., 2012).

There are three distinguishable approaches used for test optimization; test suite minimisation (or test suite reduction), test case selection and test case prioritisation (Yoo & Harman, 2012). Each technique will be discussed and looked from the point of view of needs of the target organization. It should be also noted that this research focuses of automated test optimization techniques. In the literature, these techniques are discussed with same terms but there is no clear distinction between automated and manual techniques.

Test suite minimisation and test case prioritisation are not used for the technical implementation part of this research and for that reason this thesis will not discuss them in depth. Test case selection is the technique selected for this research for its suitability for the target organization’s needs so it will be the main focus of this chapter. The used test case selection technique does not involve manual work as it is fully automated in the continuous integration environment.

4.1 Software Testing

Since software testing is tightly related to continuous integration and the topic of this thesis, it feels appropriate to give a brief overview of the software testing. This chapter will go through the different methods for testing and different levels of testing. Also, as this research is conducted in embedded software development environment, aspects of testing embedded programs are at the centre of focus. It should be also mentioned that this research is only focused on automated software testing and will not discuss manual testing. The topic of software testing is vast but this chapter will not cover it in depth. The
purpose is to give context for this research as in what type of software testing is involved in this research.

Black-box testing is a testing method that describes a testing that is not aware of the internal structure, design or implementation of the system under test. Black-box tests can be functional or non-functional, even though the functional tests are the most usual type of testing in black-box testing. The levels of testing in black-box testing are integration, system and acceptance testing. It is useful method when system under test is big and complex. (SoftwareTestingFundamentals.)

White-box testing is the opposite of black-box testing because in white-box testing, the internal structure of the system is known to tester. The testing levels for white-box testing are unit, integration and system testing even though it is mainly applied in unit testing. (SoftwareTestingFundamentals.)

Gray-box testing is a combination of white- and black-box testing methods. That means that the system under test is partially known to tester. The testing is conducted at black-box level where the bigger system is tested. It is mainly used for integration tests even though it is applicable for other levels too. (SoftwareTestingFundamentals.)

The three most distinguishable levels of testing in case company are unit tests, integration testing and system testing. Unit tests test individual units of source code and they aim to isolate each part of the program and show that individual parts are correct. Integration testing means testing of combined parts of an application to determine if they function correctly. System tests test the system which means all the components are integrated. (TutorialsPoint.)

As it was said in chapter 2.3.1, this research focuses on testing at system component level. SCT aims to find defects from a system component that has been integrated into the system. When system component testing is conducted, the whole product is not yet in place and some of the other system components have fake implementations. The point of SCT is to test one system component against external interfaces. Literature doesn’t offer a description for SCT as in the case company it is used to describe functional, capacity, performance, stability and functional tests that test only one system component.

System component tests resemble gray-box testing most from the three testing methods described above. The part of the system that is visible to the tester is the system component under test. Still, it involves other system components that are not visible.

Regression testing is a type of software testing that is carried out to ensure no new errors have been introduced into previously working system. Regression testing can be done at unit, integration or system level. (Biswas et al., 2011.) Regression tests in this research are done at system component level which means they run in real target hardware.

4.2 Test Suite Minimization

Software testing and maintenance is a process that uses a lot of resources for organizations. The size of the test suites play important role in the whole process. The test suites change and grow due to changing requirements and growing code-base and the amount of test code and test cases tend to increase over time. (Yoo & Harman, 2012).

Test suite minimization (TSM) seeks to reduce the size of the test suite by eliminating the redundant test cases from the test suite. Another name for TSM is test suite reduction.
There is a slight difference with TSM and test suite reduction as in test suite reduction the elimination of redundant test case is permanent. (Yoo & Harman, 2012.) There are two types of redundancy: semantic and syntactic redundancy. Semantically redundant test case is a test case that by removing it, effectiveness of fault detection does not change. Syntactic redundancy means there are duplication in the test code. (Singh, Mishra & Yadav, 2011.) Harrold et al. (1993) defined TSM as follows:

**Given:** A test suite TS, a set of test case requirements \( \{r_1, r_2, ..., r_n\} \) that must be satisfied to provide the desired testing coverage of the program and subsets of TS, \( T_1, T_2, ..., T_n \), one associated with each of the \( r_i \)s such as any of the test cases \( t_j \) belonging to \( T_i \) can be used to achieve requirements \( r_i \).

**Problem:** Find a representative set, \( T' \), of test cases from \( T \) that satisfies all \( r_i \)s.

The testing is sufficient when every test requirement in \( \{r_1, r_2, ..., r_n\} \) is satisfied. A test requirement \( r_i \) is satisfied by any test case \( t_j \) that is part of the \( T_i \) which is a subset of \( T \). By that logic the representative set of test cases is the that test \( T_i \)s. For minimisation to be effective, \( T' \) should be the minimal set of tests that test \( T_i \)s. However, this definition assumes that each test case \( r_i \) can be satisfied by a single test case. In reality though this might not be true if the requirement is functional and not structural. If the requirement is functional more than one test case could be needed to satisfy the requirement. This could be solved by dividing the requirement to smaller sub-requirements that can be matched with a single test case. (Yoo & Harman, 2012.)

Multiple studies have been made on the TSM subject and different approaches have been used to achieve the wanted result. A lot of them are concerned with the risk that reducing tests bring to the table. There have been multiple efforts in making TSM efficient while preserving a capability of finding faults. The most used approaches for TSM are those that involve use of some kind of code coverage for finding redundant test cases. (Yoo & Harman, 2012.) This research uses tests’ code-coverage for the test selection and some of that coverage data could potentially be used for test case minimisation. However, this research is focused on test case selection and TSM will not be discussed in depth. It is hard to make general statements about TSM since different studies use different approaches and techniques and since this research doesn’t implement solution for TSM, they are not particularly important from this thesis’ point of view. Even though TSM is not used in this research, its importance should not be undervalued as the target organization has ongoing initiatives involving finding and removing redundancy from the test code. TSM also fits to Rasmusson’s (2004) point about refactoring tests as one way to successfully scale continuous integration. The drawback of TSM is that it is despite some of the research implying that TSM techniques don’t hinder the fault revealing capability of test suite, it is still difficult to apply the technique while being completely certain that the minimized test suite will catch same number of faults as full suite. (Rothermel et al., 2001.)

### 4.3 Test Case Prioritization

Test case prioritization (TCP) attempts to maximise early the desired outcome such as rate of fault detection. It is used to find out about faults faster than it would be by running the tests in unprioritized order. (Yoo & Harman, 2012.) TCP is defined by Rothermel et al. (2001) as:
Given: a test suite, $T$, the set of permutations of $T$, $PT$, and a fault-detection rate function from $PT$ to real numbers, $f: PT \rightarrow \mathbb{R}$.

Problem: to find $T' \in PT$ such that $(\forall T'') (T'' \in PT) [f(T') \geq f(T'')]$.

However, the test case prioritization does not reduce the number of executed test cases. All cases are executed in the order of the permutation that the TCP produces. (Yoo & Harman, 2012.) Even if the TCP doesn’t select any test cases, it could potentially inform about faults earlier thus the feedback would also be faster. If the fault is found early, a developer can terminate the test if there is no need to run rest of the tests as the developer is only interested in the failing case. This scenario doesn’t always apply as it could be beneficial to run all the tests despite having a test case fail early.

This research doesn’t use TCP as its technique for test optimization as it uses test case selection. It should be noted though that test case prioritization could be beneficial as described by Rothermel (2016) where pre-submit testing would include selected set of test cases and post-submit that would run the whole test suite with its test cases being in prioritized order. Using one test optimization technique doesn’t exclude others and they can be used together to achieve as fast feedback as possible.

Test case prioritization doesn’t have the drawbacks of test suite minimisation and test case selection as it doesn’t reduce the number of executed test cases thus it doesn’t change fault revealing capabilities (Rothermel et al., 2001).

4.4 Test Case Selection

In test case selection, a subset of test cases is selected from existing test suite to test the part of the system that was affected by a change. The selection process includes two parts: determining the affected parts of the system and choosing the tests from initial test suite to test those parts. (Biswas, Mall & Satpathy, 2011.) A more formal definition of test case selection comes from Rothermel and Harrold (1994): “Let $P$ be an application program and $P'$ be modified version of $P$. Let $T$ be the test suite developed initially for testing $P$. An RTS (regression test selection) aims to select a subset of test cases $T' \subseteq T$ to be executed on $P'$, such that every error detected when $P'$ is executed with $T$ is also detected when $P'$ is executed with $T'$.”

Biswas et al. (2011) in their survey reported that there are many different techniques for test case selection reported in the literature. This research also focuses heavily on techniques that have been used in the context of continuous integration and automated test case selection. There have been many studies conducted on this subject but some of them are focused on manual test case selection or the testing itself is manual. Also, it should be noted that only two of the presented studies that implement test selection technique into a CI system of an embedded software project.

In continuous integration where every code change triggers execution of test cases, especially when it comes to embedded systems, executing every test case can be very expensive in terms of hardware resources. In that sense the common retest all strategy might not be the optimal strategy. Even with companies like Google, that have an enormous server cloud farms for their test executions, they have to select test cases that are relevant for a change (Whittaker et al., 2012). With embedded systems, it is not possible to have limitless hardware for testing so it could be argued that test case selection is even more important when the developed system is an embedded system.
Three researches were found that were specifically relevant to this study by Elbaum et al. (2014), Knauss et al. (2014) and Vöst and Wagner (2016). They were relevant because they used methods that were not limited to a specific domain or programming language and they were implemented to work in continuous integration environment. All three studies provide useful information for the designing of the artifact but the design is not solely based on any of the three studies. Those studies will be explored in this chapter and their applicability for this research will be discussed.

Elbaum et al. (2014) came up with their solution of test case selection for regression testing. They divided the commit process to two parts, pre-submit and post-submit phases. They applied test case selection to pre-submit phase were a subset of test cases were executed before the code was committed to source control repository. They argue that traditional test case selection techniques are hard to apply in CI environment as they rely in code instrumentation or complex analysis which means using them in CI environment wouldn’t be cost-effective as the analysis itself can require a lot of time to execute. They claim that the data gathered by code instrumentation is rendered useless by fast-changing code. Because of their view on short-comings of code instrumentation, they used history-based selection technique. In their solution, the execution history of test suites determines which tests suites should be selected. It should be noted that their solution focuses on test selection at suite level, not test case level. Their selection technique focuses on test suites that have been shown to reveal failure in the past. They use a “failure window” to select test suites that have revealed failures within a specified number of executions. For example, if a test suite had revealed a failure two builds ago, it would be selected to test also the new code change. Another thing they use in selection process is an “execution window”. That means that a test suite that hasn’t been executed within a specified number of executions, it will be selected to “refresh” it. For example, if a test suite hasn’t been executed within last five builds, it will be selected. Their algorithm for test suite selection is following:

Parameters:
- Test Suite $T$
- Failure Window $W_f$
- Execution Window $W_e$

for all $T_i \in T$ do
  if $\text{TimeSinceLastFailure}(T_i) \leq W_f$ or $\text{TimeSinceLastExecution}(T_i) > W_e$ or $T_i$ is new then
    $T' \leftarrow T' \cup T_i$
  end if
end for
return $T'$

The metrics to evaluate their algorithm were percentages of test suites selected, the percentage of execution time required and percentages of failures detected related to baseline technique. Their results show that number of selected test suites increased aggressively with larger execution window. The number also grew rapidly when failure window was grown until the window size reached 12 after which the growth leveled out. They didn’t do much comparison between their technique and other test selection techniques and only used random test suite selection as a comparison. Compared to random selection their technique, random selection performed approximately six times worse in failure detection. Their technique was able to detect 70-80% of failing test suites when compared to running all the tests. The failing suites that were left out by their algorithm because they had been executed recently but hadn’t failed inside the failure window.
Elbaum et al. (2014) base their assumption of uselessness of instrumentation data in test selection to Elbaum, Gable, & Rothermel (2001) study on impact of software evolution on code coverage information. In their study Elbaum et al. (2001) compared code coverage information between released software versions. In their study, they found out that if the change in code is small, coverage information doesn’t change that much and that a large change in source code affects the validity of coverage information more than a small change. The changes they observed were big indeed, at some cases approximately ten thousand lines of code. However, Elbaum et al. (2014) used that to state that coverage information gets quickly old and redundant. It could be argued that statement might not be accurate as ten thousand lines of code doesn’t change very quickly.

Elbaum et al. (2014) statement that code coverage data gets old quickly could be seen to conflict with their approach that solely rely on historical data. Both code coverage based selection techniques and their approach use the past state of the system to select a subset of tests. The conclusions of Elbaum et al. (2001) would suggest that a small change in source code has only a small effect on code coverage information. To avoid coverage information becoming redundant or imprecise the coverage information could be updated frequently. It should also be noted that a state of software product could potentially affect the usefulness of coverage information. A new green field software project software can change very quickly in short amount of time as new features are constantly added. In an older system where the majority of work that goes into it, is maintenance work, coverage information might be more stable.

Elbaum et al. (2014) do provide useful design ideas for test selection technique. The idea of ensuring that the previously failed cases are executed is something that is also experimented with in this thesis. Their metrics for evaluating a test selection technique were also found useful and they are part of the knowledge base of the DSR framework.

Knauss et al. (2015) presented a test selection technique to support continuous integration. The level of testing they use in their research is done at higher level than the testing in this research. As this research focuses on system component level testing, their focus is on system level testing. Their method is based on analysis of correlations between failed test cases and source code changes. They use a heatmap to visualize the relationship between failed test cases and code changes. From that data, they used a statistical model to suggest a subset of test cases that should be executed.

Vöst and Wagner (2016) introduced a test selection technique for a product in automotive industry. In terms of this research Vöst and Wagner (2016) study is interesting as it includes both CI and embedded software development and it takes place within an industry that seems to be moving towards agile development. Their reason for wanting to be more selective when it comes to testing stems from expensiveness of testing equipment. Still despite testing being expensive they see regression and integration testing improving software quality which means they see continuous testing as important part of developing.

Their work relied on function web, a graphical representation of a distributed embedded system that implements a certain function. It consists of different Electronic Control Units (ECU) on which the function is deployed and the communication channels between them. An ECU means a single piece of hardware consisting of a central processor, connections to several other ECUs and / or sensors / actors and a software component designed for a specific task. The function web is derived from system architecture that describes how signals are processed and forwarded. After that,
keywords are traced. Each keyword is linked to a concrete implementation of the system. A keyword trace can tell how ECUs are related during its execution. After that keyword trace is done for whole test case where each keyword used in test case is traced. The selection process is based on a mapping table which is generated by iterating all nodes in function web and map them with every test case and their traces.

Vöst and Wagner (2016) state that their approach is highly maintainable and scalable. The selection process needs to be re-run if their three inputs change. Keyword traces and test case traces need to be re-executed if test suite or keyword implementation change. Only a change in system specification requires the re-execution of the whole process. They didn’t automate the steps in their study but they state that other than forming a function web, the preparation steps are easy to automate. Their findings revealed that their approach were able to reduce the test for 50 % for changes that affected 11-12 ECUs. For changes that involved only 8 ECUs the amount of test cases was even 90 % lower compared to running every test case. They used the saving in test case executions as a metric but did not consider their approach’s fault revealing capability. However, their finding that test case selection can be beneficial in testing of embedded system. Some perceived weaknesses in their approach was that they did not address the issue how changes in system specification, keyword implementation and full test suite are tracked and if there would be process to handle those changes regarding to the test selection. For this research, it would have been interesting to see whether the fault revealing capability changes if either of the three inputs are changed. Even though this research doesn’t implement the Vöst and Wagner’s (2016) solution, it provides a good point of that trace-based selection need to consider the changes in system. This could potentially be applied to coverage-based solutions where changes in systems must be taken into account and coverage-data must be frequently updated in case of changes in system.

Asadullah et al. (2014) used a call trace based technique for regression test selection, SoRTEA. Their approach relied on both static and dynamic analysis. The approach has similar characteristics as the selection technique applied in this study as they used instrumented code to determine what test cases should be selected for a code change. They used their tool for two systems. With the first system, the tool averaged on selecting 27 % of the full suite and with the second system the tool averaged selecting 34 % of the full suite. SoRTEA was able to detect 93 % of the faults reported by full test suite. They did not measure the feedback or test execution time in their research.

Di Nardo et al., (2015) used coverage-based test case selection technique. Their technique didn’t provide very significant results. Their selection technique led to less than 2 % savings in execution time even though they stated that the nature of changes made to the system affected the results. The changes made in their analysis were big and concerned the core parts of the system which naturally made the tool select most of the test cases. They stated that smaller changes might result in more savings in execution time.
5. Implementation of the Test Selection Tool

This chapter is about the implementation of the design artefact constructed for this research. The relevant literature has been introduced in the previous chapters and this chapter will refer to those publications as they are used to justify the design decisions. At this point the reader should have a clear picture of the environment in which this research is conducted. Also, it should be clear at this point, what are the potential benefits of optimizing continuous integration system from point of view of developers, management and scaling agile development. This chapter will go through the steps in the development of the artefact. After explaining the implementation, result collecting is described.

5.1 The Starting Point

As it has been mentioned before, the designing of the artefact didn’t start from scratch. There had been an initiative for writing a tool for test selection and a developer in the case company had started it by writing a tool for instrumenting code and collecting code coverage data from the application code. The reason why building such a tool was initially started, and not using an existing tool, was because the instrumented code is executed in memory and CPU constrained environment. This means that by writing tool of their own, it would be easier to control tool’s behaviour. Also, the solution needed to work for all compilers used in case company’s products which itself puts some restrictions in choosing an existing tool.

In some test selection techniques, the test selection is done at function level but in this one, each source file is instrumented. Part of the decision why the coverage is measured at file level and not function level, was due to the fact that the coverage info collecting had to be executed in real target that has limited amount of memory and by measuring the coverage for each function would mean that the memory consumed would be too high. By doing instrumentation only for files, the whole process remains light-weight and can be executed in target. There was however a concern whether the technique would be effective enough and if most of the files choose every test. Also by doing the selection based on files makes the results very dependent on a size of a changed file.

The way that the tool was developed was that its user had to first run the instrumentation script to instrument each source file. After the instrumentation, a set of test are executed and another tool is used to collect the coverage data from test execution. Each test case is given an id and every time a test case executed code in a source file, the script would write the test case id to the source file. After that a user could run the test selection tool and give it file names as arguments and it would print out the tests that visit that source file.

This solution was however, not fit to work in continuous integration environment in its current form. Firstly, running the instrumentation and then the whole test suite just to select test cases would be counterproductive as it would take longer than just testing code changes with the full test suite. This was also one of the problems mentioned by Elbaum et al. (2014). Secondly, the files had to be given manually which meant that to automate the process, changed files had to be automatically given to the tool.
5.2 First Steps – Collecting the Coverage Data

First thing to do was to store the coverage information so that it could be accessed and used by CI tool. When deciding how this should be done, first the format of the data had to be decided. Since the test selection would be based on incoming code changes, the logical way to determine changed files was to inspect the commit log. The case company uses Subversion as their source code control tool. With Subversion changed files for a specific commit can be retrieved by command:

```
svn -r REVISION NUMBER --verbose --xml > changed_files.xml
```

This command prints out the commit log with the files that have been changed. Subversion has an option to format the log into XML form so `--xml` is added to the command in order ease the parsing of the commit log. The output of the command is then stored into a file that could be parsed in the future. The changed files could be parsed in the form of:

```
Path/to/file/SourceFile.cpp
```

Now that the format of the changed file was known, the format to save the coverage data could be decided. Different database solutions were briefly considered but the simpler solution was to simply save the coverage data into files and store them on a server. The files that stored the coverage data were named in the format such as:

```
Path_to_file_SourceFile.cpp.testcase
```

This was seen as valid format to name the files as each name would be unique as there can’t be files within a same directory with identical names. The stored file withheld lists of test names separated by line break:

```
Test 1
Test 2
Test 3
```

The coverage can become imprecise as it was pointed out by Elbaum et al. (2014) and for that reason the updating of the coverage information was automated so that every night the coverage would be collected by executing the steps described above by the continuous integration tool. The coverage is collected for two test types of tests. The other one, “Shock” tests consist of only about 30 test cases and the other one is a large regression test suite consisting of about 800 test cases. Each test suite has its own job in Jenkins. Both Shock and regression tests use RobotFramework as testing framework. Shock tests the overall functionality of the product and its purpose is to quickly test that the product still works. Passing the Shock test is one of the promotion criteria which means that to promote a branch to next level in the CI pipeline Shock test must pass. This means that if a developer wants to make his changes visible in trunk, he must wait for the promotion jobs to pass before he can merge a team branch to trunk.

The drawback of this coverage collecting method is that it isn’t suitable for all the products in the case company. The coverage collecting script consumed a lot of memory and as the embedded system can have restricted amount of memory, it was only the newest product that had a sufficient amount of memory to execute tests using instrumented code and collect coverage data. However, there are other ways to collect the coverage data, for example the coverage collecting could be executed in simulated environment. Still, as it was stated by Mårtensson et al. (2016), there lies some uncertainty
with simulated environments as it is hard to be sure that tests executed in simulated environment would pass in real target. Also, it was stated by the case company that their main interest is in the newest product as it is and will be at the top of focus for the company.

To summarize, at this point we have reached a point where we have coverage data for each source file saved in a server. Each file is named after its file path and consists of every test case that executes code in that source file.

### 5.3 Automating the Test Selection

This chapter describes the process of how tests are selected when a code change is committed to a code repository. The CI server starts by compiling and building the code which then triggers test jobs. When the job with selected subset of tests is triggered the coverage data files in the server are downloaded to the machine executing the test cases. Figure 6 shows where the test selection takes place in the CI system.

**Figure 6:** Selection process overview

The test selection tool in its simplest form is constructed as following:

```plaintext
changed files = create change list
selected tests = select test cases for changed files
if selected tests is not empty:
    create test list of selected tests
```

The script takes the XML formatted commit log and coverage data files as an input and outputs new test list if any tests were selected.

To use the coverage data files, the selection tool parses the XML formatted commit log and puts each changed file in a list, first formatting it to match the format of the filename. Example of XML formatted commit log:
<log>
  <logentry revision="123">
    <author>developer</author>
    <date>2016-01-01T00:00:00.12345</date>
    <paths>
      <path prop-mods="false" text-mods="true" kind="file" action="M">
        /path/to/file/sourceFile.cpp
      </path>
      <path prop-mods="false" text-mods="true" kind="file" action="M">
        /path/to/file/sourceFile.hpp
      </path>
    </paths>
    <msg>to be done</msg>
  </logentry>
</log>

Now create_change_list turns the commit log into a list of:

[path_to_file_sourceFile.cpp, path_to_file_sourceFile.hpp]

That is list is then passed to function select_test_cases that will parse the list and look for test cases from coverage data file.

Function select_test_cases

selected_tests = list of unique test cases
for each file in changed files
  iterate files in coverage data files
    find match for file and add tests to selected tests
return selected_tests

The test selection was implemented in three team branches, each team working on different tasks. The idea was to expose the test selection tool for teams with different kind of tasks to see whether the teams’ tasks affect the effectiveness of the tool. At first it was discussed whether the tool should be implemented in Trunk but it came apparent that most incoming changes in Trunk were mostly merges from team branches. As it was stated in chapter 3.1.7, merges are infrequent and usually large in size and since the tool analyses code changes at file level, big merges are most likely to render the tool useless. As it was discovered by Di Nardo et al., (2015), large changes can lessen the impact of test selection tool. Team branches were seen as a better fit because developers practicing continuous integration are frequently integrating small changes to their team branches.

5.4 Implementation of Additional Features

Now the created test list can be executed in Jenkins by giving it as an argument to the script that executes system component tests. At this point the selection tool worked fine but there were couple of problems that appeared. First was a problem with regression suite always having failing cases and it raised a question of whether those failed cases should be executed even though they weren’t selected for the subset of test cases. It was discussed whether the selected subset should report the total amount of errors in code base or give feedback to a developer whether his/her changes broke anything. In literature,
history data has been used in test selection. For example, Elbaum et al. (2014) used history data to populate subset of tests and it has been shown to be useful.

It was requested by the case company to add a feature to the selection tool that would add failed cases from previous build to the selected test list. To do this the failed cases for previous build had to be downloaded and parsed. In order gain access to history data, the test execution script had to be modified to populate list of failed test cases. First attempt to acquire the failed test cases was to parse console log of the previous build. Jenkins shows the output of the commands it’s given and that log could be downloaded from URL CI_server_name/job/job_name/lastCompletedBuild/console. The console log was however too inconsistent for it to be sensible to parse it. More suitable solution was to modify to test execution script and add --xunit option for the RobotFramework. That command produced XML consisting of numbers of test cases executed, number of failures and name of every test case and information whether it failed or not. Jenkins stored the file to a job’s build artefacts that are different files passed to downstream jobs in Jenkins and stored to the machine executing the Jenkins job. The option was added for test selection tool to call it with an argument CI_server_name/job/job_name/lastCompletedBuild/artefact/test_results.xml and the algorithm was modified into

Function select_test_cases:
    selected_tests = list of unique test cases
    if link for previously failed cases is given
        parse failed cases from xml
        add failed cases to selected tests
    for each file in changed files
        iterate files in coverage data files
            find match for file and add tests to selected tests
    return selected_tests

Fairly quickly it was obvious that the regression suite was too unstable for this to be a sensible decision. The main purpose of adding such feature was to be able to track the total number of regressions. However, most failing tests were flaky which resulted in unusable test results. For example, if 50 test cases failed in previous build, only 10 of them would fail in the next build even though the new code change didn’t fix anything. It was also impossible to track the fault revealing capability with so many flaky tests in the test suite. It also slowed the feedback unnecessarily if the amount of failed cases in previous build was high. For example, the selection tool itself might not find any test cases to execute but it would choose over a hundred test case for new test list from previously failed cases. After discussing this issue with company representatives, it was decided to abandon the added feature and maybe add it later if the regression test suite ever got to a stable state. Even though the implementation of utilization of history data was eventually removed from the tool, all the work didn’t go to waste as the XML produced for this feature, could be used for collecting test result data. The data collecting is more thoroughly explained in chapter 5.5.

Another problem faced after implementing the test selection tool was related to test executions consisting of high number of tests. The full regression suite is about 800 tests and for a number that big it would take a very long to execute those tests on one machine. This problem has been solved in the case company by distributing the test cases to multiple machines. At the moment, the full suite is divided between seven machines or “sections". It seemed like a good solution to add the same argument to test execution command also for selected subset and use seven machines to execute the selected subset of tests. It ended up being a good solution for executions where the test selection tool selected a large number of test cases but for the cases that had for example 50 tests it was
wasteful to reserve so many machines to execute such little amount of test cases. Another
problem relating to number of test cases was that for team branches’ regression tests are
given lower priority. That is because the CI team wants to ensure that there are always
machines available for test executions in Trunk and other more important branches.
Again, this seemed to be good solution for large test sets but rather unnecessary for lower
number of test cases. Both, the number of sections and priority are given as an argument
to the test execution command, for example:

```bash
./run_tests.sh --selected_tests --sections 7 --priority 2
```

Another feature was added to test selection to determine the number of sections and
priority level. The new feature parsed the test list of selected subset and counted number
of tests in the list. The algorithm determined the number of sections based on the number
of tests

Function: get number of sections needed

```plaintext
sections_needed =
  (number of test cases - number of test cases % 100) / 100
  if number of test cases <= 100
    then return 0
  elif number of test cases % 100 == 0
    then return sections_needed
  else if sections_needed < 7
    then return sections_needed + 1
  else
    return MAX_NUMBER_OF_SECTIONS(=7)
```

The algorithm for determining priority was based also on number of test cases

Function: get priority

```plaintext
return "--prio 2" if number of test cases > 200 else return ""
```

The output was a string consisting of "--sections + get number of sections needed
+ get priority"). If get number sections returned 0, the output was an empty string.
Now the amount of resources needed for each test execution could be dynamically
determined based on number of tests in test list by giving the script as an argument for
test command.

```bash
./run_tests.sh --selected_tests get test parameters
```

Executing the test command above triggers test and it reserves a test machine and loads
the binaries to the machine which itself can take a long time. However, if no tests are
selected, it is unnecessary to even execute the test command as it would just delay
feedback to developer and waste resources. A condition was added for executing the
selection. As it can be seen from the algorithm there is condition for creating a test list
where the test list is not even created if there are no selected tests. The condition to restrict
Jenkins executing the test command was added by checking whether test list file exists.
If the test list exists, then Jenkins will execute the given test command.

Another problem worth mentioning that came up during the implementation, surfaced
when there were changes from multiple commits to be executed in the test job for selected
tests. Looking back at the subversion command `svn -r REVISION NUMBER --verbose
--xml > changed_files.xml` it can be seen that it produces names and paths of changed
files for that specific commit. However, the code might be built with changes from
multiple commits and the given command would only pick up last of the revision
numbers. The command had to be changed to `svn diff -r
To do this the revision from last successful build had to be downloaded and parsed in order to extract the revision number. Now the changed files would contain every file changed between the last successful build and new build. The || true had to be added because the last successful revision number might not exist and it was undesirable that the build would fail because of that command.

### 5.5 Collecting Results

The test selection tool is evaluated by measuring three metrics; feedback time, number of test cases and fault revealing capability. Since the artefact had to be observed it being integrated to the continuous integration system, it would have been too tedious of a task to collect the results for all three metrics by hand. For each metric, the collecting of the data was automated and different tools had to be written to do the task. Giving the task of collecting the result data to an automated tool might present a threat to the validity for this research. However, manually collecting results can also be prone to mistakes. Also, every tool used for result collecting have been unit tested and manually tested to verify the correctness of information. The flakiness of the regression test suite causes another threat to validity as the coverage is collected by executing the full test suite that can be unstable. If the test execution in coverage collecting fails and it goes unnoticed, it is possible that test selection is based on incomplete data which could make the selection unreliable. For example, if a fatal problem occurs in the target machine which causes every test to fail, it would mean that none of the test cases after the target failure would record any coverage information which then would result in worse coverage than successful test execution. This however is a risk that had to be taken for the sake of this research. When the selection is fully adopted, the Jenkins job that handles the coverage data collecting could be given higher priority and it would be observed more carefully. It could be maybe even made part of the promotion jobs in order to force the attention to it.

The feedback is measured for both Shock and regression suites but the case company’s interest is in feedback for regression suite as Shock tests are rather fast already. The number of test cases are measured for both Shock and regression but the case company’s interest is mostly in regression because regression tests requires the most resources. Fault finding capability is only measured for subset of Shock tests. This is due to the flakiness of the regression suite. The fault finding in this research is done by comparing full suite execution and execution of selected subset. If a fault occurs in full suite but not in selected subset, or the failing test is not executed in selected subset, it can be seen as a failure to detect errors. This kind of comparison was attempted for regression suite but it became apparent very quickly that the results gained were almost completely useless. Almost each test execution had failing test cases in full regression suite which were not part of the selected subset. After investigating further, it was obvious that the failing tests were failing due, not to the error in submitted code, but due to flaky nature of the test suite.

In order to measure fault finding capability and number of test cases, a tooling was needed to collect the data. The results had to be collected almost real-time because the Jenkins executions were not stored for long periods of time. For that reason, the collecting of the test results had to be automated and stored somewhere they could be accessed and analysed. A tool was implemented for collecting the results for each Jenkins job executions. Every time a test job would finish, the result collecting job is triggered in Jenkins. The test job passes its name as a parameter to the result collector tool and the tool would then download the result XML form URL CI_server_name/job/job_name/lastCompletedBuild/artefact/test_results.xm
ml. It uses the same XML that was used to determined previously failed test cases. An example of test result XML:

```xml
<testsuite failures="1" tests="2" errors="0" name="Tests" skip="0">
    <testcase classname="Test suite name" name="Test case name 1"
        time="Execution time in seconds">
    </testcase>

    <testcase classname="Test suite name" name="Test case name 2"
        time="Execution time in seconds">
        <failure message="Failure message" type="Error type">
    </testcase>
</testsuite>
```

As it can be seen, the testing framework can produce highly usable data that tells us the number of test cases, number of failures and information about failed test cases. The XML itself doesn't offer anything that would identify it to the software version used when executing tests. There happened to be an existing implementation where Jenkins would automatically store a file for each build, containing the revision number along with other commit information. That meant that the revision number could be parsed by downloading artefact from URL `CI_server_name/job/job_name/lastCompletedBuild/artefact/revision_file.txt`.

Now after each Jenkins execution, a tool parses the result XML and revision file in and forms a Python dictionary that contains a revision number, number of test cases executed and number of failures. That information could be then converted into JSON and stored in a file on a server in the format of `JOB_NAME.json`. The tool downloads existing JSON file, matching the job name it's given as an argument and adds new result to it before uploading the updated version back to server. In order to measure fault finding capability, the same XML is parsed to collect failed test cases and every executed test case. This JSON file contains the revision number, names of failed test cases, and name of every executed test case. Now, that data could be used to compare failed cases in full suite execution and find a matching revision number in results of selected subset of tests and compare whether the failures in full suite appear in failures or executions of selected subset. This automatic data collecting method creates another threat to validity of these results because the XML is produced by the RobotFramework and if the test list, created by the selection tool, is empty the RobotFramework won't start the test execution. This means the XML is not generated which means that if the result collecting tool is attempting to download the contents of the URL, it faces an HTTP error. This problem is handled in the tool so that if an HTTP error occurs, the tool assumes that the test list was empty and marks zero as the number of test cases. This is rather a small problem since network problems are relatively rare in the company but it is possible that download fails due to a problem in network connection and the results falsely report zero as the number of test cases executed.

The feedback times are collected by an existing system the case company has implemented where the feedback times for each Jenkins job are stored into a database. The feedback time in their implementation is calculated by parsing the commit timestamp and then parsing the end time from the console log. For purposes of this research, this type of calculation is a suitable solution as this research is concerned with feedback in CI so by measuring the feedback, and not just the execution time of the tests, the direct effects of test selection to feedback time are visible. The jobs are executed as a stream where the
Shock test are executed after compilation and it triggers a regression suite. For test selection, the selected shock tests are executed after compilation and it triggers selected regression suite.

The fact that feedback statistics were available from the start meant that they could be observed from the very beginning. However, there was some much change in the test selection tool which means the not all results could be used to evaluate the tool. Especially with regression suite the test selection tool had to be modified multiple times which rendered previously acquired results redundant. When test selection was applied to regression suite for the first time, the initial results were very promising. However, it was later found out that the test command for selected tests wasn’t the determining prioritization level. That meant that since the full regression suite, that the selected regression suite was compared to, used lower priority, it would have made results biased in favour of the selection tool. This meant that the test command had to be changed which wasn’t a problem itself but it rendered previously collected results useless.

After the priority for test execution was modified, the previously mentioned requirement for executing previously failed cases in selected set of tests was presented. This meant again that the previously collected had to be abandoned because adding the option for executing previously failed cases meant that the feedback time was bound to go up. However, after this feature was removed for the reasons explained previously in this chapter, it also meant that the already collected results were no longer valid which meant that again the result collecting had to be started all over again.

At this point it seemed that the tool would stay stable from that point forward. However, at this point it was noticed that the test commands had to be determined dynamically which affected the number of test machines the test execution was distributed to and the priority. The implementation of this feature is described in chapter 5.4. Since this affected the feedback, the previously collected results had to be abandoned. This was last modification made for the tool and every result since is fit to analyse and evaluate the artefact.
6. Results

In this chapter collected results are analysed and visualized. The chapter is divided into four sub-sections, first three covering one evaluation metric for the test selection tool and fourth that summarizes the results. As the selection tool was implemented into three team branches, the team branches are analysed first separately and then the overall result for each metric is calculated. Each team branch will be referred as Team1, Team2 and Team3.

6.1 Feedback – Sub-RQ3

For Team1’s Shock tests, the feedback was measured for 136 commits / Jenkins job executions, three of which were considered outliers because test executions took more than 500 minutes. Since Shock tests are usually very fast, it seemed pointless to include such a rare case and it also helps reading the visualizations. Each branch is followed by a table that shows the numerical measures for feedback times in that branch. The measures were done for mean, median, standard deviation and percentiles (P5, P10, P25, P75, P90 and P95).

![Boxplot presentation of Shock feedback in Team1 branch](image)
Figure 8: Bar chart presentation of Shock feedback in Team1 branch

Average feedback for whole Shock suite was 24.2 minutes and 18.2 for suite selected by the selection tool. The average improvement in feedback between to test sets was then 24.6%. Figure 7 and Figure 8 show the difference in feedback.

For Team1’s regression suite the feedback was measured for 135 commits. For regression suite, each execution was included even though there were some test executions that stood out by their unusually long feedback. This is because very long executions are not that uncommon with big regression suite.

Figure 9: Boxplot presentation of Regression feedback in Team1 branch
Figure 10: Bar chart presentation of Regression feedback in Team1 branch

Average feedback for full regression suite was 237 minutes and 118 for suite selected by the selection tool. The average improvement in feedback between two test sets was 50.2%. Figure 9 and Figure 10 show the difference in feedback. Table 3 shows the numerical measures for Team1 feedback.

<table>
<thead>
<tr>
<th>Job</th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>P5</th>
<th>P10</th>
<th>P25</th>
<th>P75</th>
<th>P90</th>
<th>P95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback in selected Shock</td>
<td>18.2</td>
<td>16.8</td>
<td>107.4</td>
<td>9.4</td>
<td>10.1</td>
<td>12.2</td>
<td>21.1</td>
<td>27.8</td>
<td>34.8</td>
</tr>
<tr>
<td>Feedback in full Shock</td>
<td>24.2</td>
<td>20.1</td>
<td>15.0</td>
<td>15.7</td>
<td>16.4</td>
<td>17.7</td>
<td>24.0</td>
<td>31.4</td>
<td>52.2</td>
</tr>
<tr>
<td>Feedback in selected Regression</td>
<td>118</td>
<td>62.5</td>
<td>191.3</td>
<td>0.0</td>
<td>10.5</td>
<td>16.4</td>
<td>138.0</td>
<td>213.5</td>
<td>593.9</td>
</tr>
<tr>
<td>Feedback in full Regression</td>
<td>237</td>
<td>193.8</td>
<td>82.6</td>
<td>83.4</td>
<td>110.1</td>
<td>144.5</td>
<td>238.0</td>
<td>337.0</td>
<td>374.8</td>
</tr>
</tbody>
</table>

Table 3: Team1 numerical measures for feedback in minutes

For Team2’s Shock feedback was measured for 92 commits, one of which was ignored as an outlier due to the exceptionally long feedback time.
Average feedback for full Shock suite was 26.8 minutes and 20.1 for suite selected by the test selection tool. The average improvement in feedback between two test sets was 25%. Figure 11 and Figure 12 show the difference in feedback.
Feedback for Team2’s regression was measured for 92 commits.

**Figure 13:** Boxplot presentation of Regression feedback in Team2 branch

**Figure 14:** Bar chart presentation of Regression feedback in Team2 branch

Average feedback for full regression suite was 191 minutes and 83 for suite selected by the selection tool. The average improvement in feedback between two test sets was 56.5%. Figure 13 and Figure 14 show the difference in feedback. Table 4 shows the numerical measures for feedback in Team2 branch.
<table>
<thead>
<tr>
<th>Job</th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>P5</th>
<th>P10</th>
<th>P25</th>
<th>P75</th>
<th>P90</th>
<th>P95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback in selected</td>
<td>20.1</td>
<td>19.8</td>
<td>59.9</td>
<td>10.1</td>
<td>11.3</td>
<td>14.9</td>
<td>22.7</td>
<td>26.7</td>
<td>34.2</td>
</tr>
<tr>
<td>Shock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feedback in full</td>
<td>26.8</td>
<td>21.4</td>
<td>16.3</td>
<td>15.7</td>
<td>16.9</td>
<td>18.7</td>
<td>26.2</td>
<td>39.6</td>
<td>61.8</td>
</tr>
<tr>
<td>Shock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feedback in selected</td>
<td>83.0</td>
<td>62.5</td>
<td>191.3</td>
<td>0.0</td>
<td>10.5</td>
<td>16.4</td>
<td>138.0</td>
<td>213.5</td>
<td>593.9</td>
</tr>
<tr>
<td>Regression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feedback in full</td>
<td>191.0</td>
<td>193.8</td>
<td>82.6</td>
<td>83.4</td>
<td>110.1</td>
<td>144.5</td>
<td>238.0</td>
<td>337.0</td>
<td>374.8</td>
</tr>
<tr>
<td>Regression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Team2 numerical measures for feedback in minutes

For Team3’s Shock feedback was measured for 180 commits.

**Figure 15:** Boxplot presentation of Shock feedback in Team3 branch
Average feedback for full Shock suite was 29.1 minutes and 19.1 for suite selected by the selection tool. The average improvement in feedback between two test sets was 34.4 %. Figure 15 and Figure 16 show the difference in feedback.

Feedback for Team2’s regression was measured for 180 commits.

**Figure 16:** Bar chat presentation of Shock feedback in Team3 branch

**Figure 17:** Boxplot presentation of Regression feedback in Team3 branch
Average feedback for full regression suite was 193 minutes and 75 for suite selected by the selection tool. The average improvement in feedback between two test sets was 61.1 %. Figure 17 and Figure 18 show the difference in feedback. Table 5 shows the numerical measures for feedback in Team3 branch.

<table>
<thead>
<tr>
<th>Job</th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>P5</th>
<th>P10</th>
<th>P25</th>
<th>P75</th>
<th>P90</th>
<th>P95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback in selected Shock</td>
<td>19.1</td>
<td>16.3</td>
<td>10.7</td>
<td>8.5</td>
<td>9.1</td>
<td>11.4</td>
<td>21.6</td>
<td>33.2</td>
<td>42.2</td>
</tr>
<tr>
<td>Feedback in full Shock</td>
<td>29.1</td>
<td>21.1</td>
<td>18.9</td>
<td>15.4</td>
<td>16.4</td>
<td>18.9</td>
<td>30.1</td>
<td>59.1</td>
<td>79.2</td>
</tr>
<tr>
<td>Feedback in selected Regression</td>
<td>75.3</td>
<td>33.7</td>
<td>170.8</td>
<td>11.6</td>
<td>12.3</td>
<td>14.4</td>
<td>73.1</td>
<td>128.1</td>
<td>179.1</td>
</tr>
<tr>
<td>Feedback in full Regression</td>
<td>193.1</td>
<td>173.8</td>
<td>78.0</td>
<td>76.6</td>
<td>87.5</td>
<td>135.5</td>
<td>226.0</td>
<td>307.9</td>
<td>346.2</td>
</tr>
</tbody>
</table>

Table 5: Team3 numerical measures for feedback in minutes

Overall 404 commits were used to collect feedback data. The overall average feedback for the whole Shock suite was 27 minutes and 19.1 minutes for selected subset. The average of overall improvement in Shock feedback would then be 29.3 %. The overall average feedback for whole regression suite was 212 minutes and 94 minutes for selected suite. The overall improvement in regression feedback is then 55.7 %. Table 6 shows the results of statistical tests made for feedback and it shows that the test selection tool’s impact on feedback is significant. T-test’s p-value shows that the impact is more significant between selected subset of regression tests and full regression suite. While the significance is lower between selected subset of Shock and full Shock suite, the impact is still significant as it is lower than the significance level (0.05).
### Table 6: Statistical tests for feedback times between selected subset and full test suite.

<table>
<thead>
<tr>
<th>Job</th>
<th>Student’s t-test (two-tailed, .05 significance level)</th>
<th>Effective size cohen’s d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t-value</td>
<td>p-value</td>
</tr>
<tr>
<td>Team1 selected / full shock</td>
<td>-3.81</td>
<td>.000174</td>
</tr>
<tr>
<td>Team1 selected / full regression</td>
<td>-4.57</td>
<td>&lt; .00001</td>
</tr>
<tr>
<td>Team2 selected / full shock</td>
<td>-3.37</td>
<td>.00093</td>
</tr>
<tr>
<td>Team2 selected / full regression</td>
<td>-7.99</td>
<td>&lt; .00001</td>
</tr>
<tr>
<td>Team3 selected / full shock</td>
<td>-6.18</td>
<td>&lt; .00001</td>
</tr>
<tr>
<td>Team3 selected / full regression</td>
<td>-7.78</td>
<td>&lt; .00001</td>
</tr>
</tbody>
</table>

6.2 Number of Test Cases

For Team1 Shock tests, 102 test executions were measured. Figure 19 below shows the number of test cases executed for each commit. You can see there are lonely bars without a bar for selected tests beside it. That means there were no tests selected for that specific revision.
Figure 19: Executed test cases in Team1 Shock

As Figure 19 shows, the number of tests in full suite is relatively stable. The average number of test cases executed in full suite is 32.4 but for selected subset the number is 8.6. If we ignore the test executions that hold no tests, the average of test cases executed in selected subset is 27.4. From 102 commits, only 32 commits selected any tests which explains why the average number is so low compared to full suite. However, if we accept this, the selection tool averagely selected 73.5 % less tests than in full suite. If we ignore the cases where the number of selected tests is zero, the selection tool selects 15.4 % less tests than in full suite. Figure 19 shows the number of test cases executed in full suite and selected subset.

Number of test cases for Team1 regression was measured for 53 test executions. Now the figure will show completely blank sections which is result of neither full nor selected suite executing any test cases for that particular commit. This due to the fact that regression suite takes a long time to execute as it could be seen in chapter 6.2. The Jenkins job for selected subset is executed more times than full regression suite because a test execution will have had to finish before starting to test other commits. If the selected subset doesn’t select any or many test cases, the Jenkins job has time execute and complete itself multiple times during the execution of full regression suite. This behaviour is expected for regression suite for other teams branches as well. However, because the point of evaluating this metric is to compare how many test cases are selected compared to full suite and as we know that the current system would execute every test case for each commit, it can be seen as appropriate to ignore the zeroes in full suite executions and only count the real full suite executions. There is another difference between results of Shock and regression suites as number of Jenkins job executions is very close to the number of commits whereas with regression, the number of Jenkins job executions doesn’t match the number of commits for the reasons mentioned before.
Figure 20: Executed test cases in Team1 regression suite

Average number of executed test cases in Team1 full regression suite is 827.6 and 172.9 for selected subset. When we ignore the zeroes in selected subset, the average is 538.9. The difference between two suites is 79.1 % if we don’t ignore the zeroes in selected suite. If we do ignore them, the selected suite averages 34.9 % less tests executed than in full suite. From 53 test executions, 36 executions didn’t select any tests which means that 32 % of the test job executions selected any tests. Figure 20 shows the number of test cases executed in full suite and selected subset. Table 7 shows numerical measures for executed test cases.

<table>
<thead>
<tr>
<th>Job</th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>P5</th>
<th>P10</th>
<th>P25</th>
<th>P75</th>
<th>P90</th>
<th>P95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test cases in selected Shock</td>
<td>8.6</td>
<td>0.0</td>
<td>13.4</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>22.0</td>
<td>32.0</td>
<td>33.0</td>
</tr>
<tr>
<td>Test cases in full Shock</td>
<td>32.4</td>
<td>32.0</td>
<td>0.5</td>
<td>32.0</td>
<td>32.0</td>
<td>32.0</td>
<td>33.0</td>
<td>33.0</td>
<td>33.0</td>
</tr>
<tr>
<td>Test cases in selected Regression</td>
<td>172.9</td>
<td>0.0</td>
<td>302.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>248.0</td>
<td>823.0</td>
<td>824.0</td>
</tr>
<tr>
<td>Test cases in full Regression</td>
<td>827.6</td>
<td>845.0</td>
<td>55.9</td>
<td>731.2</td>
<td>806.5</td>
<td>854.0</td>
<td>846.0</td>
<td>846.0</td>
<td>846.0</td>
</tr>
</tbody>
</table>

Table 7: Team1 numerical measures for number of test cases

For Team2 shock tests 48 commits were measured. Chart below shows the number of executed test cases for each revision.
Figure 21: Executed test cases in Team2 Shock

The average number of test cases executed in Team2’s full shock suite is 32.2 but for selected subset the number is 19.1. If we ignore the test executions that hold no tests, the average of test cases executed in selected subset is 30. The difference between the two suites is 40.7 % if the empty test executions are included. If the zeroes are ignored the difference is 6.8 %. From 48 commits, 20 didn’t select any tests which means 62 % of the test job executions picked any tests. Figure 21 shows the number of test cases executed in full suite and selected subset.

For Team2 regressions, the number of executed test cases was measured for 20 test executions. Average number of executed test cases in Team2 full regression suite is 841.8 and 293.2 for selected subset. If again, we ignore the zeroes in selected subset, the average is 586.4. The difference between two suites is 65.2 % if we don’t ignore the zeroes in selected suite. If we do ignore them, the selected suite averages 33.3 % less tests executed than in full suite. From 20 test executions, 10 test job executions didn’t select any tests which means the selection rate was 50 %. Figure 22 shows the number of test cases executed in full suite and selected subset. Table 8 shows numerical measures for executed test cases in Team2 branch.
**Figure 22:** Executed test cases in Team2 Regression

<table>
<thead>
<tr>
<th>Job</th>
<th>Mean</th>
<th>Median</th>
<th>Standard deviation</th>
<th>P5</th>
<th>P10</th>
<th>P25</th>
<th>P75</th>
<th>P90</th>
<th>P95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test cases in selected Shock</td>
<td>19.1</td>
<td>22.0</td>
<td>15.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>32.0</td>
<td>33.0</td>
<td>33.0</td>
</tr>
<tr>
<td>Test cases in full Shock</td>
<td>32.2</td>
<td>32.0</td>
<td>0.4</td>
<td>32.0</td>
<td>32.0</td>
<td>32.0</td>
<td>32.0</td>
<td>33.0</td>
<td>33.0</td>
</tr>
<tr>
<td>Test cases in selected Regression</td>
<td>293.2</td>
<td>6.5</td>
<td>371.4</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>748.5</td>
<td>820.6</td>
<td>826.5</td>
</tr>
<tr>
<td>Test cases in full Regression</td>
<td>841.8</td>
<td>845.0</td>
<td>13.7</td>
<td>830.3</td>
<td>844.5</td>
<td>845.0</td>
<td>845.3</td>
<td>846.0</td>
<td>846.5</td>
</tr>
</tbody>
</table>

**Table 8:** Team2 numerical measures for number of test cases

The number of test cases executed in Team3 branch shock tests was measured for 145 commits. The average number of test cases executed in Team3’s full shock suite was 31.3 and for selected subset the number was 9.2. If we ignore the test executions that hold no tests, the average of test cases executed in selected subset is 27.7. The difference between the two suites is 70.6 % if the empty test executions are included. If the zeroes are ignored the difference is 11.5 %. From the 145 commits, 97 didn’t select any tests which means that in 33 % of the test executions any tests were selected. Figure 23 shows the number of test cases executed in full suite and selected subset.
The measurement of test cases executed in Team3 branch regression tests was measured for 99 test executions. Average number of executed test cases in Team3 full regression suite is 833.4 and 163.7 for selected subset. If again, we ignore the zeroes in selected subset, the average is 648.1. The difference between two suites is 80.4 % if we don’t ignore the zeroes in selected suite. If we do ignore them, the selected suite averages 22.2 % less tests executed than in full suite. From 99 test executions, 74 didn’t select any tests which means that in 25 % of the test executions any tests were selected. Figure 24 shows the number of test cases executed in full suite and selected subset. Table 9 shows numerical measures for executed test cases in Team3 branch.

Figure 23: Executed test cases in Team3 Shock

Figure 24: Executed test cases in Team3 regression
6.3 Fault-Finding Capability – Sub-RQ2

For Team1 branch fault finding capability was measured for 94 commits, three of which failed to reveal a fault. That means a test case failed in full suite but it was not executed in the selected subset. Each case will be looked at individually because by looking at a fault, it can be deduced with a fair amount of certainty whether a failure was a shortcoming of the test selection tool or a failure of the testing environment.

In the first case, selection tool failed to select 20 cases that failed in full suite but were not executed in selected subset. Selected subset had selected 13 tests for that commit, none of which failed in the execution for selected subset. By investigating further and executing the full shock suite using that specific revision, it turned out that it was a failure in test environment because executing the same tests with same code passed all the tests. It can be concluded that in this case, the test selection tool did not fail to detect a fault.

In the two other cases, the test selection tool came up with no tests but still one test failed in the full shock suite. By looking at the changes made in the commits it became apparent that the most likely explanation for the failure is that the test case was unstable. In one of the two cases the changed file was a unit test file and in the other it was for test list for another deployment which means neither of these changes could have affected the shock tests. For Team1 shock tests it seems that the selection tool did not fail to detect a fault in the 94 commits because all three could be explained by a failure in the test environment.

For Team2 branch, fault finding capability was measured for 42 commits one of which failed to reveal a fault as there was a commit that reported a fault in full suite while selected subset reported no faults. For that commit the test selection tool hadn’t selected any tests but every test full suite failed. In order to decipher whether full suite reported a real fault or not, the same tests had to be re-executed for that revision. It turned out that all the tests passed in the re-execution. This means that for Team2, the selection tool did...
not fail to detect faults and that the reported failures were due to instability of the environment.

Team3 branch had the most commits, 142 and there were 12 cases where a fault slipped through the test selection tool. Each were examined further to determine whether the faults were real.

One case was identical with the one on Team1 branch where every 32 tests failed in full suite but selection tool selected only 15. However, none of those 15 failed when they were executed which hints that the failures in full suite were caused due to instability. This was confirmed by re-executing the full suite using that revision.

There were four cases where there was a single failure in the full suite while the selection tool selected no tests for that commit. Three of the four were false-positives as they all passed when re-executed. However, one turned out to be a real fault that wasn’t detected by the test selection. The change that caused the fault was a change in the test material.

Six times every test failed in full suite. After re-running the tests using the appropriate revision, one turned out to false-positive. The remaining five were legitimate faults that the selection tool failed to detect. Each of the give commits didn’t touch the source code but the test material which explains the failures.

Overall, 278 commits were analysed and six times the selection failed to reveal a fault. That means that for 97.8 % of the selected shock test executions, the selection tool came up with same results than the full shock suite.

### 6.4 Summary of the Results

Tables 10 and 11 show a summary of the results.

<table>
<thead>
<tr>
<th></th>
<th>Team1</th>
<th>Team 2</th>
<th>Team3</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback improvement</td>
<td>24.6%</td>
<td>25%</td>
<td>34.4%</td>
<td>29.3%</td>
</tr>
<tr>
<td>Reduction in number of</td>
<td>73.5%</td>
<td>40.7%</td>
<td>70.6%</td>
<td>67.1%</td>
</tr>
<tr>
<td>executed test cases</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fault detecting capability</td>
<td>100%</td>
<td>100%</td>
<td>95.8%</td>
<td>97.8%</td>
</tr>
</tbody>
</table>

**Table 10**: Percentage of improvement between selected subset of shock suite and full shock suite

<table>
<thead>
<tr>
<th></th>
<th>Team1</th>
<th>Team2</th>
<th>Team3</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback improvement</td>
<td>50.2%</td>
<td>56.6%</td>
<td>61.1%</td>
<td>55.7%</td>
</tr>
<tr>
<td>Reduction in number of</td>
<td>79.1%</td>
<td>65.2%</td>
<td>80.4</td>
<td>78.2%</td>
</tr>
<tr>
<td>executed test cases</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>executed test cases</td>
<td>Fault detecting capability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 11:** Percentage of improvement between selected subset of regression suite and full regression suite
7. Discussion & Implications

This section will present the findings and the answers for the research question and the sub-questions. The sub-questions will be answered first and then the main question will be discussed. Both the findings from previous literature and findings from artefact evaluation will be used to answer the questions.

Sub-RQ1: What test optimization techniques are there?

As it was discovered the three most popular test optimization techniques are test suite minimization, test suite optimization and test case selection. Test case selection was found to be the best suited for the problem at hand. Also, it turned out that there was an initiative about test selection inside the organization. That initiative played a big role in the designing of the artefact.

Sub-RQ2: Can automated test optimization be used safely without compromising fault finding capability?

Elbaum et al. (2014) found that random selection technique performed approximately six times worse in failure detection than their history-based selection algorithm. Their technique was able to detect 70-80% of failing test suites when compared to running all the tests. Findings in this study are positive from the point of view of fault finding capability. The selection tool did not miss any faults in the source code but it did not detect six faults caused by changes in test environment related code. This was expected as the coverage data used by the selection tool covered only the production code. As no changes in the source code was missed by the selection tool, the Elbaum et al. (2014) claim stating coverage data will go imprecise quickly, is not in line with the observations of this thesis. It seems that automated test case selection can be used to test the changes in source code but in its current state it has no means of detecting faults introduced by changes in test code.

Sub-RQ3: Can automated test case selection in embedded software development make shorter feedback loops?

The results of this research provide very promising results in terms of acquiring faster feedback with automated test selection. With smaller and overall faster test suite (Shock) the effects are not very impactful. Especially, the difference in feedback between executing 20 cases and 30 cases seemed to be relatively small because in such small test suite, majority of time seemed to go into setting up the test environment. With Shock tests the best results were gotten from commits that did not trigger any tests as it didn’t require setting up the test environment. This is also positive from testing hardware resources’ point of view as less hardware needs to be reserved with the selection tool.

Compared to shock test, selected subset of regression suite provided great results in terms of feedback. By cutting the feedback time in half, the test case selection tool could have a great impact in terms of continuous integration practices and hardware resources needed in testing embedded systems.

By comparing to other techniques reported by the literature, in terms of reduced feedback, test selection tool implemented in this research did fairly well. Di Nardo et al., (2015) tool had less than 2 % reduction in execution time whereas the selection tool implemented in this thesis averaged 29.3 % improvement with selected subset of shock suite and 55.7
% improvement in selected subset of regression. It was difficult to find studies that had similar enough setting to make a meaningful comparison. Most test selection related research considered only percentage of test cases selected by a test case selection technique. However, only measuring number of test cases was not sufficient enough to evaluate the implemented tool because one of the requirement was to make feedback faster. Number of executed test cases doesn’t tell the whole truth because their execution times are not identical. Still number of test cases selected by a selection tool is commonly used metric so it was also analyzed for the implemented tool.

Vöst and Wagner (2016) results show that their technique selected 50 % of the full suite when testing higher number of units and 90 % when the number of units was smaller. Asadullah et al. (2014) saw 73% and 66 % reduction in number of test cases when applying their technique to two systems. In this research the average reduction in number of test cases was 67.1 % for selected subset of shock suite and 78.2 % for regression suite. It should be noted that a large part of analyzed commits selected no tests which explains why the reduction in number of test cases is so big. This however is acknowledged by the case company that it is expected that commits are different and it will affect the results.

Tables 7-9 and figures 19-24 show that there are a lot of test job executions where no tests are selected. If fact, it seems that majority of commits changes parts of code that is not real application code but rather, for example, test code.

Sub-RQ4: How does slow feedback affect developers and organizations?

Findings and observations made in the case company are in line with each other. Slow feedback seems to harm the practise of continuous integration. Slow feedback causes infrequent integration and other practices that are not optimal in continuous integration. Slow feedback was one of the reasons why the case company drove the development work into team branches.

Slow feedback can affect individuals by disturbing their workflow but it can also have an impact of how they think about CI as a practise (Duvall, 2007). It can also be linked to growing technical debt (Brooks, 2008). Slow feedback can also have a negative effect on developers at cognitive level as it can impact developer satisfaction and team morale (Laukkanen & Mäntylä, 2015).

Sub-RQ5: Can test case selection enable better continuous integration strategies?

It is difficult to give a straight answer for this question. This research has shown that test case selection can improve the feedback time in continuous integration environment which turns the question into whether feedback improvements made by test selection can enable some of the benefits of continuous integration. For one, it can be a step towards to real continuous integration in the target organization. It can lessen the need to have team branches and encourage developers to commit their changes more frequently. The findings of Brooks (2008) supports this conclusion because he found out that faster feedback means that a developer is more likely to integrate more frequently.

Problems of large-scale CI was studied and one of the reported problems of scaling of CI was slow running tests. Rogers (2004) said that the most problematic thing about growing code base is that the number of tests goes up as the code base grows. Test case selection can limit the number of test cases that need to be executed for each commit so in that sense test case selection can help removing some of the pain points that are related to CI in large code base.
Sub-RQ6: Can successful continuous integration help the organization to achieve continuous delivery?

This thesis offers only a very shallow answer to this question because the actual state of continuous delivery was not studied in this thesis. The main interest from point of view of the case company was to know whether successful continuous integration at system component level can benefit in the transformation towards continuous delivery. The problems faced in adopting continuous delivery in Laukkanen et al. (2016) are very similar to the problems the case company in this thesis is facing in continuous integration. It seems that the problems are similar in both and by solving the problems of continuous integration could help greatly in adoption of continuous delivery.

The main research question was; can automated test selection enable continuous integration at large scale embedded software development? This thesis has shown that other big organizations, for example Google utilize some kind of test selection in their continuous integration (Whittaker et al., 2012). It has also been argued in this thesis that test case selection is particularly important when it comes to continuous integration in embedded systems. The expensiveness of the testing is a big problem with embedded systems but still to be able to practice continuous integration, and benefit from it, it is essential to be able to verify the correctness of the software after every commit. The test selection tool could solve some of the problems in target organization. As it was stated before, they were forced to distribute development into team branches due to slow feedback and constantly having broken builds in the mainline. The team branches ensured that no code would go to mainline untested. With automated test selection, the case company could start moving towards continuous integration as the tests would be faster to execute and less unnecessary tests would be executed.

By looking at the average feedback for shock tests in each team branch it seems that the feedback is around 20 minutes or less in each team branch while feedback being around 25-29 minutes with full shock suite. This doesn’t reach the bar set by Beck (1999) which sets the maximum limit of build time to ten minutes. By looking at Roger’s (2004) view of how build times affect integration rate, it can be roughly estimated that with the average feedback gotten from selected subset of shock suite, the feedback would allow a developer to integrate approximately in daily basis. This however doesn’t apply on feedback gotten from full regression suite. While the feedback improvement gotten from selected subset of regression suite being >55% it doesn’t reach a number that would suggest that the feedback is sufficient for daily integration. However, it was mentioned in chapter 5.1 that a developer must wait until promotion jobs (e.g. Shock test) finishes before a team branch can be merged to trunk. It was said by Duvall (2007) that a branch has to be merged at least daily for it to be considered CI. Having quicker feedback available might encourage more frequent merges to mainline thus making the integration more continuous. This could be a similar workflow that Elbaum et al. (2014) mentioned where testing had two phases, pre-commit and post-commit testing. In the context of the case company a team branch could work as a pre-commit phase where test selection techniques are used and when tests pass the branch is merged to trunk where code is tested by executing full test suites.

It should be noted though that this thesis is heavily focused on feedback and test selection is used to solve part of the problem of continuous integration in the case company. It became clear during the implementation of the selection tool that the problems with testing doesn’t end even if the problem of slow feedback is solved. Testing large embedded system is challenging and executing less tests doesn’t solve the whole problem. The reason why the evaluation of the designed artefact was limited to Shock tests in terms
of fault finding capability was because the testing environment and the tests themselves were too unstable to gather any useful data out of them. Also, the testing strategy itself could be reviewed because so much of the feedback gotten from CI relies on tests that are executed in target. The perceived consensus between developers in the case company is that future work should heavily lean on writing good enough unit test suite that execute far more quickly than system component tests.

This thesis has focused on continuous integration focusing on what happens after a developer pushes a commit. However, to utilize the tool in its full potential it should be made easily available for developers to quickly run the selected set of tests from their local workstations. Fast feedback should be something that also available before committing code changes. This would be easy to implement and it is something that will be done but it is not included in this thesis.

This thesis contributes to existing literature and research by implementing an artefact that can help in scaling continuous integration when there are limited resources set by the environment. Not having to execute tests when it’s not necessary can have a big impact financially for software companies. It shows that test selection can be used to successfully detect faults in continuous integration environment and it can improve the feedback time. The artefact also shows some proof that while re-test all is the safest of testing strategies, it might not be necessary as test selection can offer the same results for less effort. In this thesis, the importance of continuous integration in large-scale agile development is argued and that can be used as a justification to initiate tasks to improve the continuous integration environment such as test selection.
8. Conclusions

This thesis studied large-scale continuous integration and implemented an automated test selection tool to support continuous integration. The test selection tool was piloted in real continuous integration environment in a large telecom company. The company is adopting agile software practices at large scale and practices like continuous delivery which worked as motivation for this thesis. Literature was used to justify the need for the test selection tool. The problem was looked from point of view of continuous integration and what are the things that hinder the ability to benefit from practicing continuous integration. It was discovered that continuous integration is one of the most important things affecting successful adaptation of agile software development at scale.

The automated test selection tool provided promising results as it cut the feedback time by 55.7% in 404 commits. The selection tool was relatively safe as only six faults slipped through it in 278 commits and it was well-acknowledged that those kinds of faults would slip through as they were caused by changes in test code, not source code.

8.1 Limitations

The data that was reported in this study about feedback, number of test cases and fault finding capability were limited as they were collected in fairly short amount of time. There were two reasons for this. The tool changed so frequently so it was hard to gather results that used the same version of the tool. The second one was that the metrics of how to evaluate the artefact weren’t clear from the start and they came up later as did the means to make data available.

There were a couple limitations with the artefact itself. First, it covered only the production code which resulted in failures of detecting faults that were caused by changes in test code. The second limitation was that we were unable to study the fault-finding capability for selected subset of regression suite. Also, even though it seemed appropriate to use data gathered in continuous integration environment, it might have not been the best place to evaluate fault finding capability. The fault-finding capability was only evaluated by comparing the found faults to faults found by the full shock suite. The overall number of faults detected by the selection tool was not studied.

8.2 Recommendation for future studies

In the future, some of the limitations should be addressed. For example, also covering the test code in some manner thus improving the fault-finding capability would improve the test selection tool immensely. Also, as the test selection tool works at file level, in the future it could be improved so that it would detect changes in specific methods and only select tests that cover that method.

During the initial testing of the selection tool it became apparent that the number of tests selected for each test execution is heavily dependent on the changed file. Some files naturally picked up more test cases than others. That information could however have some architectural implications, if the most frequently changed files are most heavily tested also. It’s possible that those files that take more tests, are also bigger in size. This observation springs a new metric to analyse. The size of the file and the frequency of it is changed and their relation to the number of tests is something that could be beneficial to
study. It might also help explain some of the results gotten out of the test selection tool. If the relationship between commit size and feedback could be shown it might even encourage developers to keep their commits small and practice continuous integration.
References


Jenkins. Retrieved February, 10, 2017 from https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins


LeSS. Retrieved February, 10, 2017 from https://less.works/less/technical-excellence/continuous-integration.html


Software testing levels. Retrieved February, 3, 2017, from https://www.tutorialspoint.com/software_testing/software_testing_levels.htm

