



FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Juhani Wilén

**Code Change Based Selective Testing in
Continuous Integration Environment**

Master's Thesis
Degree Programme in Computer Science and Engineering
May 2018

Wilén J. (2018) Code Change Based Selective Testing in Continuous Integration Environment. University of Oulu, Degree Programme in Computer Science and Engineering. Master's thesis, 64 p.

ABSTRACT

Continuous integration (CI) is a software engineering practice in which new code is integrated to existing codebase continuously. Integration testing ensures that the changes in code function as intended together with the other parts of the code. The number of tests tend to grow and at some point performing them all becomes infeasible due to limited time between consecutive test executions. Therefore, the traditional retest-all approach becomes inoperative and test optimization techniques are required. Test selection is one of those techniques and it encompasses selecting tests which are relevant to recent changes in the code.

The purpose of this thesis is to analyze existing test selection methods, and to implement an initial continuous test selection method in CI environment that reduces duration of integration testing stage and provides faster feedback. The method is aimed to be safe that no additional faults are let through the testing. The test selection is based on changes submitted to version control system (VCS), which are compared with source code file coverages of different hardware variants reported by compilers. In addition, other possible dependencies between variants and code changes are investigated. Those are related to test codes and interfaces. Now the testing of change independent variants can be ignored, and only testing change dependent variants is conducted.

At the beginning the implemented test selection method was used in a single software development branch for testing purposes. The results indicate that utilizing the method accomplished slight but statistically significant reduction of integration testing duration with significance level of 0.05. The mean of the testing duration was decreased by 15.2% and the median by 22.2%. However, the implementation still has some inaccuracies in dependency detection, and further improvements are needed to make the test selection method more efficient.

Keywords: integration testing, test selection

Wilén J. (2018) Koodimuutoksiin perustuva valikoiva testaus jatkuvan integraation ympäristössä. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 64 s.

TIIVISTELMÄ

Jatkuva integraatio on ohjelmistotuotannon käytäntö, jossa muutokset ohjelmakoodiin integroidaan osaksi jo olemassa olevaa ohjelmistoa jatkuvasti. Integraatiotestauksella varmistetaan, että muutokset koodiin toimivat sen muiden osien kanssa kuten on tarkoitettu. Suoritettavien testien määrä usein kasvaa ajan mittaan, ja jossakin vaiheessa niiden kaikkien suorittaminen ei ole enää järkevää, koska perättäisten testiajojen välinen aika on rajallinen. Siksi perinteinen kaikkien testien uudelleenajaminen tulee haastavaksi ja tarvitaan testien optimointitekniikoita. Testien valinta on yksi näistä tekniikoista. Se sisältää sellaisten testien valinnan, jotka ovat oleellisia testaamaan viimeaikaisia muutoksia koodiin.

Tämän diplomityön tarkoituksena on analysoida olemassa olevia testien valintamenetelmiä ja luoda alustava toteutus jatkuvasta testien valintamenetelmästä jatkuvan integraation ympäristössä, millä vähennetään testien kestoajaa integraatiotestausvaiheessa ja nopeutetaan palautteen saamista. Tavoitteena on, ettei testauksen läpäisseiden vikojen määrä kuitenkaan kasva. Testien valinta perustuu versionhallintajärjestelmään toimitettuihin muutoksiin, joita verrataan kääntäjien raportoihin lähdekoodikattavuuksiin eri laiteversioille. Lisäksi laiteversioiden riippuvuus testikoodien ja rajapintojen muutoksiin tutkitaan. Ne laiteversiot, jotka eivät ole riippuvaisia mistään muutoksista, jätetään testaamatta, ja ainoastaan muutoksista riippuvaisten laiteversioiden ohjelmakoodit testataan.

Testien valintaan toteutettu menetelmä otettiin käyttöön aluksi yhdessä ohjelmistokehityshaarassa sen toiminnan testaamiseksi. Saadut tulokset näyttävät, että menetelmän hyödyntämisellä saavutettiin vähäinen mutta tilastollisesti merkittävä integraatiotestauksen kestoajan lyheneminen merkitsevyystasolla 0,05. Testauksen keston keskiarvo laski 15,2% ja mediaani 22,2%. Toteutuksessa on vielä epätarkkuuksia riippuvuuksien havaitsemisessa, ja sitä pitää kehittää paremman tehokkuuden saavuttamiseksi.

Avainsanat: integraatiotestaus, testien valinta

TABLE OF CONTENTS

ABSTRACT

TIIVISTELMÄ

TABLE OF CONTENTS

FOREWORD

ABBREVIATIONS AND SYMBOLS

1. INTRODUCTION	8
2. SOFTWARE TESTING AND DEVELOPMENT	9
2.1. Software testing	9
2.1.1. Levels of software testing	9
2.1.2. Testing methods	11
2.1.3. Testing types, techniques and tactics	12
2.2. Agile software development	12
2.2.1. Scrum	14
2.3. Continuous software engineering practices	16
2.3.1. Continuous integration	17
2.3.2. Continuous delivery	18
2.3.3. Continuous deployment	19
2.4. Techniques for test optimization	20
2.4.1. Test selection	21
2.4.2. Test suite minimization	22
2.4.3. Test prioritization	23
3. APPROACHES FOR TEST SELECTION IN CI	24
3.1. Test execution history based selection and prioritization	24
3.2. Faster fault finding by test optimization	26
3.3. Code-churn based test selection	27
3.4. Discussion	28
4. ENVIRONMENT & TOOLS	30
4.1. Continuous integration environment overview	31
4.2. Release CI	31
4.3. Common tools, items, and principles	34
4.3.1. CI core application and database	35
4.3.2. Metrics	36
4.3.3. Test sets, test suites and test cases	36
4.3.4. Version control system	37
4.3.5. Branching	37
4.3.6. Jenkins automation server	38
4.4. Microcontroller and signal processing CIs	39

4.4.1. Test coverage analyzer	40
5. IMPLEMENTATION	41
5.1. The proposal	41
5.2. Product file coverage	42
5.3. Changed files between revisions (delta)	43
5.3.1. Defining revisions for delta: first attempt	43
5.3.2. Defining revisions for delta: second attempt	44
5.4. Finding dependencies	45
5.5. Putting the steps together	47
6. RESULTS AND VALIDATION	49
6.1. Data collection	49
6.1.1. Threats to validity	50
6.2. Results	51
6.2.1. Detecting outliers	52
6.2.2. Tests of statistical significance	56
6.2.3. Analyzing the results	57
6.3. Discussion	57
6.3.1. Future work	58
7. SUMMARY	60
8. REFERENCES	61

FOREWORD

My five-year journey as a university student is coming to an end. These years have taught me a lot, and that knowledge I will need in my future endeavors. Writing this thesis was the last challenge on this journey. I want to thank the master's thesis work provider, a telecommunications company, for giving me this opportunity. I want to thank my team in the company for sharing their knowledge and for all the support during these months. Specially I want to thank my technical supervisor Tapio Paananen, and also Pekka Tuuttila, for their guidance and feedback regarding this thesis.

From the University of Oulu, I want to thank my principal supervisor professor Juha Röning for guiding me through the thesis process, and the second examiner of the thesis, professor Mika Mäntylä. And lastly, I want to thank my parents and other family members, along with my friends, for supporting me over the past few years.

Oulu, Finland May 31, 2018

Juhani Wilén

ABBREVIATIONS AND SYMBOLS

ASD	adaptive software development
CCTS	code-churn based test selection
CD	continuous delivery
CI	continuous integration
CPI	continuous platform integration
HTML	Hypertext Markup Language
IDE	integrated development environment
IQR	interquartile range
MAD	median absolute deviaton
OOP	object-oriented programming
ORM	object-relational mapping
Q-Q	quantile-quantile
RPC	remote procedure call
SD	standard deviation
SSH	secure shell
TBD	trunk-based development
URL	Uniform Resource Locator
VCS	version control system
XP	extreme programming
<i>B</i>	branch
<i>b</i>	build object
<i>C</i>	low level system component
<i>D_i</i>	data set
Δf	added, modified or deleted files between revisions (delta)
<i>F, F_i</i>	frame object
<i>f</i>	function
<i>H₀</i>	null hypothesis
<i>H₁</i>	alternative hypothesis
<i>L</i>	lowest level system component
<i>M</i>	microcontroller system component
<i>M_i</i>	module
<i>m</i>	metric object
<i>P</i>	set of permutations, program
<i>p, p_i</i>	product object
<i>p</i>	probability value
<i>Q_i</i>	quartile
<i>R</i>	release matrix object
<i>r_i</i>	test requirement
<i>r_i</i>	revision number
<i>T</i>	test, test suite
<i>T_e</i>	mean based threshold
<i>T_d</i>	median based threshold
<i>T_q</i>	interquartile range based threshold

t_i	epoch timestamp
U	signal processing system component
W_e	execution window
W_f	failure window
W_p	prioritization window
x	sample
\subseteq	is a subset of
\forall	for all
α	significance level
\in	is member of
\mathbb{R}	real numbers

1. INTRODUCTION

In continuous integration environment new changes in code are integrated to existing codebase continuously [1, 2]. The changes may occur frequently which causes challenges for testing. Time between consecutive execution of tests may be short, which raises need for test optimization techniques, since traditional retest-all approach is too time-consuming [3]. Consequently, the longer the testing takes, the slower is the feedback for developers.

This thesis is conducted as a part of continuous integration (CI) team in a large telecommunications company, called as *the company* in this thesis. The team is responsible for integrating changes to existing hardware-dependent code. The purpose of this thesis is to analyze existing test selection methods and to apply their principles in order to implement an initial method for test selection, which reduces duration of integration testing stage in continuous integration environment. Currently, a complete set of tests are performed for each build. Since the integration tests take long time to run, not all the builds can be integration tested separately. The goal is to reduce the duration of integration testing significantly so that it can be performed for every build. Then the feedback is provided faster, and tracking faults will be simpler. That is accomplished by focusing testing on *changed parts* of the code. Subset selection of tests requires finding dependency between changes in a commit and the tests. Reducing the number of executed tests has always risk of reduced fault detection capability, which should not be deteriorated. This must be monitored when selective testing is utilized.

The reason behind urgent need for the focused testing is limited number of hardware targets, on which the integration testing is conducted. Their testing capacity is not sufficient to separately test all the commits submitted to the codebase. Investing in more hardware targets would resolve the problem, but at the time of writing, it is not an option. Therefore, the solution for the problem is to improve integration testing to be smarter, more focused on changes. The scope of this thesis is to create a proposal of continuous test selection method, which will be implemented and tested in a real test environment.

Currently test coverage data is not directly available for a single test, which causes challenges for this thesis. Finding dependencies directly between tests and code changes is not possible. However, there is a possibility to reduce the number of executed tests by excluding products that are not dependent on source code changes. The products are devices, of which each has different hardware and source code. The codebase is common for all the products, and therefore to be able to find dependencies between changes and products, their file coverage in the codebase must be investigated. All submitted changes are not related to the source code, and other possible types of dependencies also need to be examined.

The next goal after finding dependent products for each change, is to expand the test selection to reduce the number of executed tests per product. For that the test coverage must be known, and an existing test coverage analyzer can be partially utilized for that purpose. However, it cannot provide coverage data for all the tests, thus using this tool alone is not a comprehensive solution for the test selection. At the time of writing, the tool is not used to analyze coverage of integration tests, but it can possibly be used to support the implementation later.

2. SOFTWARE TESTING AND DEVELOPMENT

In traditional software development process software testing and development are conducted in separated phases with slow development cycle, while in modern agile methodologies the loop is much shorter [4]. That means that software testing is done more frequently when utilizing agile methodologies. Continuous software development practices push agile methodologies even further, enabling integration of development work and its deployment to production even more frequently, continuously. That is accomplished by automating software testing and deployment.

Under this chapter the fundamental concepts and principles of software testing and agile development are addressed. Continuous software development practices, especially continuous integration, which is essential in this thesis, is covered in general here. It is described in more detail in Chapter 4 along with other features specific to this thesis. Continuous practices can be improved in many ways, for instance, shortening build and test time, increasing visibility, and improving fault detection [5]. Test optimization techniques aim to reduce testing cost. They are an important part of this thesis, and therefore, also covered in general under this chapter.

2.1. Software testing

The purpose of software testing is to ensure good quality of developed software. Unlike commonly thought, the objective is not to prove that software functions correctly – it is to find as many faults and errors as possible. In other words, software testing attempts to prove that the tested software does not work rather than it does. Thus, the definition of software testing can be summarized to “*Testing is the process of executing a program with the intent of finding errors*” [6 p. 10-11]. However, in practice the objective of testing is to prove that the software being tested meets the functional and non-functional requirements set to it. Testing is often time-constrained and finding all the faults is not possible. Software testing can be conducted on multiple software levels with different methods utilizing various types, techniques and tactics.

2.1.1. Levels of software testing

Generally, the software testing can be divided into four levels (from lowest to highest): unit testing, integration testing, system testing and acceptance testing [7 p. 132]. While unit testing focuses on a very small section of code e.g. function, acceptance testing tests entire system for acceptability. These testing levels and their objectives are illustrated in Figure 1. Sometimes regression testing is included in the list of software levels, which is incorrect, since regression test can be conducted at any software level, it is a testing type, not a level. [8]

Unit testing is the lowest level of software testing. There independent units or components of software are tested. The purpose is to ensure that the units function individually as designed. At this level, the testing is executed most frequently since it is lightweight and enables rapid detection of the issues. White-box testing method (see

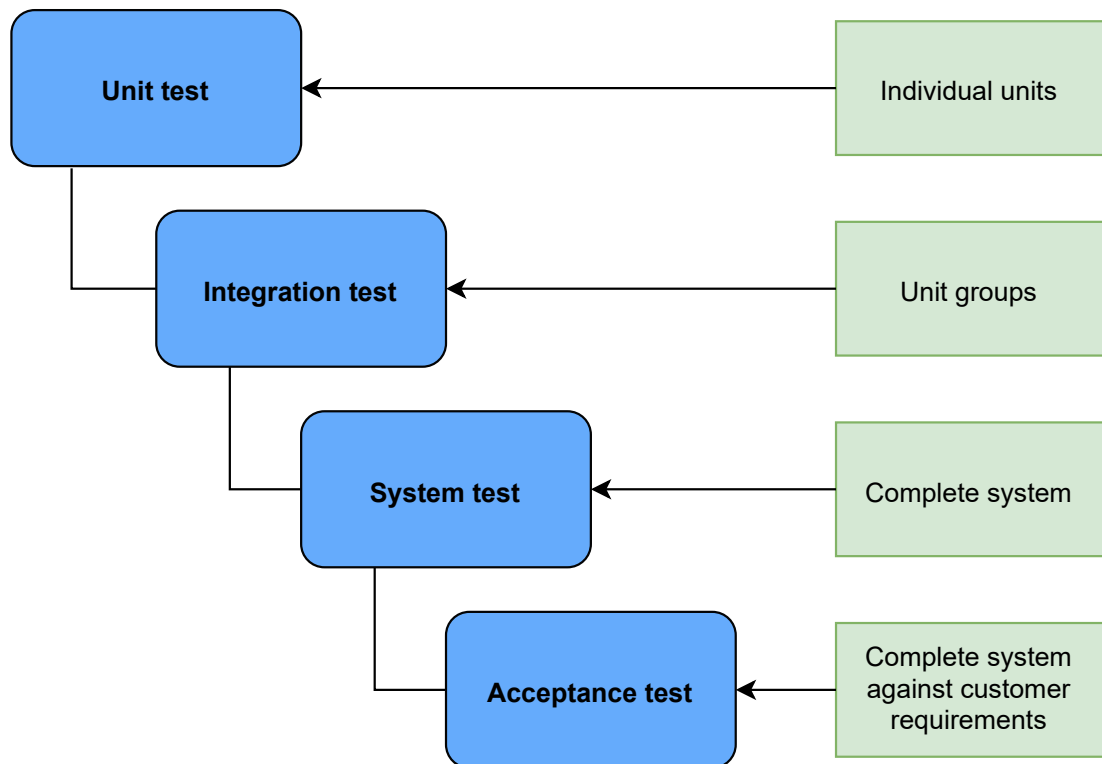


Figure 1. Hierarchical levels of software testing.

Chapter 2.1.2) is often utilized in unit testing. Generally, unit testing is performed by software developers. [7 p. 133][8]

In integration testing the units are tested as a group [8]. Successful unit tests do not guarantee successful integration testing. Interfaces of the units will be tested to verify proper communication between the units. Therefore, the purpose of integration testing is to reveal faults related to interaction between integrated units [7 p. 134]. It can utilize Big Bang approach, in which most of the units are tested together at one time. Or alternatively, top-down, or bottom-up approach, or combination of those two. In those approaches integration is started from either the top or the bottom level units or both simultaneously. Integration testing is often performed by testers of a separate integration team, not the developer.

In system testing an application is tested in its entirety for the first time. The aim is to find out if the software reaches the specified customer requirements. System tests are conducted in an environment which is close to a real production environment by testers independent from developers [8]. Hence, the operation of the application is verified against requirements to ensure its eligibility for the customer. Gray-box testing method is often used at this level. [7 p. 134]

The highest level of software testing is (user) acceptance testing. The system is tested to find if it is ready to be released. In other words, does it meet the business' needs of the user [7 p. 135]. This can be conducted in a company who developed it (internally) or by the user (externally). If the software passes testing, it is deployed to production. [8]

2.1.2. Testing methods

Testing methods can be either dynamic or static. Generally, if the code being tested is executed, it is dynamic testing, otherwise it is static testing. Static code analysis means analyzing code without executing it. Static methods can be used to find syntax errors as well as structural errors in code, such as uninitialized variables [7 p. 41][9]. Static methods verify the code. Nowadays, text editors automatically suggest possible syntax errors in code, but in need of deeper analysis, there is a long list of static code analysis tools available. Dynamic testing, in turn, validates the code. There the code is executed, and functional behavior of the software is tested. Now it must meet the business requirements, which static methods do not consider.

Black-box testing is a software testing method, in which the tester does not know nor is concerned of the structure of tested system or component. The tester can set the inputs and observe the outputs, but the tested program is seen as an unknown “black box” for them. The focus of black-box testing is to validate that the program fulfills the functional requirements – or more precisely, to prove it does not behave correctly. Thus, this testing emphasizes testing against program’s functional requirements [10]. One advantage of black-box testing is that the tester does not need to have knowledge of programming language used in development. As a disadvantage, it is hard to know, how many program paths are left untested because of the unknown nature of program structure. [6 p. 13]

In white-box testing the tester can see the internal structure of the program. Hence, test data is derived based on the logic and the paths of the source code. The tester should try to reach as high coverage of statements, paths or branches as possible, but in large programs this is difficult [6 p. 14]. Furthermore, if the program changes frequently, keeping the test cases up to date may be challenging. Here the tester, unlike in black-box testing, needs knowledge of the implementation and the structure of tested program and needs to understand the programming language to be able to create test cases. Anyhow, reaching high coverage in testing will result in discovering errors and vulnerabilities [11].

Gray-box testing is a combination of white-box and black-box testing methods. There the tester has partial knowledge of the internal structure of tested program. The tested modules/components are studied to acquire a better understanding of the program as in white-box testing, and then test cases are executed with straightforward way of black-box testing. [12]

Agile testing obeys the guidelines of agile software development (described in Chapter 2.2) [7 p. 371]. In an agile software development environment development and testing are integrated to enable smooth product release flow. When attending to a traditional project, everyone is responsible of quality, but testing is done by independent people. In agile testing, however, everyone can also write test cases. This leads to continuous develop-test iteration, where the workload of independent testers is reduced due to the help of developers. [13]

Random testing is a set of unscripted software testing methods without a formal test plan. Because of their undocumented nature, they are not repeatable [7 p. 71]. It is common for random testing that it is done with minimal or nonexistent information about the software. This testing is comparable to real-world usage: anything can happen to the software in a way that could not be caught by structured tests. Random

testing is often thought as the worst case of program testing [14]. Therefore, random testing can find defects that structured tests cannot, such as software/system crashes. It can be effective in finding errors with small effort, but it cannot merely be relied on as a testing method [14].

2.1.3. Testing types, techniques and tactics

Regression testing is about finding defects in software after a change in code. New version of the code is tested to function same way as previous version with purpose to give confidence that changes in code had no effect on operation of unchanged parts of the code [3 p. 1][7 p. 329][15]. Using same test cases, or subset of them, to a code after changes, is known as *regression testing*. It is experienced that changes to the original code are more error prone than the original program code [6 p. 19, 106]. Testing in continuous integration environment (see Chapter 2.3.1) can be considered to be, almost entirely, regression testing.

Smoke testing is set of most critical test cases to verify operation of basic functionality of software. In other words, it is minimal effort to find if the software is broken or not [16]. Results provided by smoke testing determine is there point to continue testing any further. If they failed, it indicates that testing should be stopped in that phase, because the software does not meet requirements for minimal robustness. Smoke testing can be performed quickly, but entire test set may take long time to execute. Therefore, when smoke testing is involved, it is possible to quickly find instabilities, and stop testing without using more time. When complete test set grows larger and more time-consuming, performing smoke testing becomes more and more beneficial.

Testing can be split to functional and non-functional testing types. In functional testing software is tested against specified functional requirements without referring to its internal structure (see black-box testing method described in Chapter 2.1.2). On the contrary, in non-functional testing, attributes of the software are tested that are not functional, for instance, performance, scalability, security, efficiency and usability aspects. [7 p. 343]

Almost all software has some performance criteria. To verify and validate that software reaches the criteria, *performance testing* is needed [7 p. 343]. It tests features as response times, throughput rates, reliability and resource usage under certain load. In this testing, the environment is needed to be similar to production environment for accurate results. The goal of tester is, again, to prove that the software does not meet specified performance criteria. [6 p. 101]

Testing can be categorized to manual and automated testing. As their name indicate, manual testing is done manually by human, while automated testing is performed by computer. In continuous integration, see Chapter 2.3.1, the testing is highly automated. [7 p. 41]

2.2. Agile software development

Agile software development is set of principles in software development that are utilized by different agile methodologies such as Adaptive Software Development (ASD),

Crystal methods, Extreme Programming (XP), Scrum, Lean Development and many others [17, 18]. For all these methodologies, there are common core values defined by Agile Manifesto. The core values listed in Agile Manifesto are illustrated in Figure 2 [19].

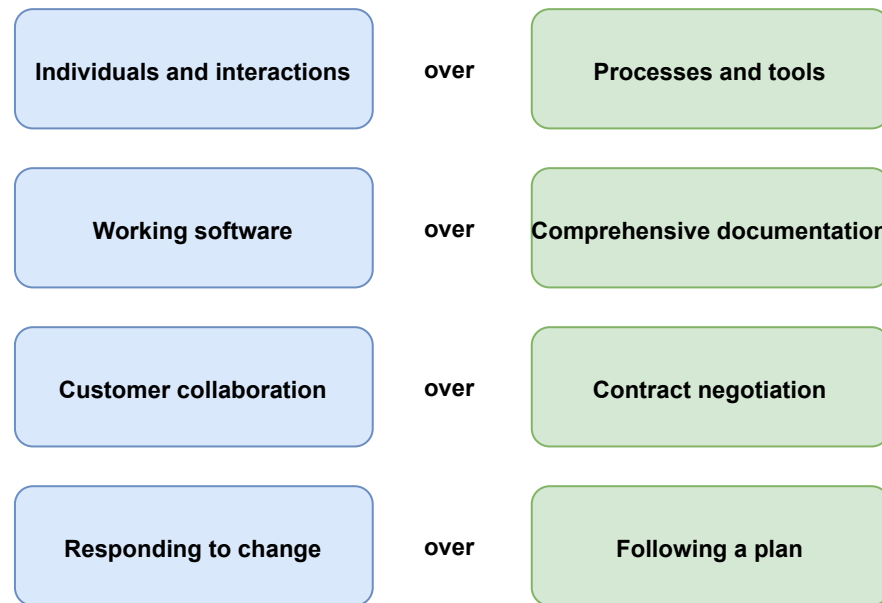


Figure 2. The values of Agile Manifesto.

In traditional plan-driven software development, customer may change the requirements during development, because they rarely know at the beginning, what do they exactly want [20]. This is problematic, since software requirements are defined at the beginning of the project. If there are changes in the requirements later, all passed phases need to be partially gone through again due to their sequential nature (e.g. waterfall model). This will lead to increased costs and additional time needed to finish the project. Or on the other hand, if redesigning, redevelopment and retesting is not performed, or communication is inadequate, the resulting software/system is not what customer has requested. Overall, the long development process has increased risk that the resulting software does not match with the customer needs, since there may be customer interaction only at the beginning and end of the process [7, 21].

Agile methodologies do not move from one software development phase to another, but they continuously iterate through short agile software development cycle seen in Figure 3. Agile manifesto states that delivering working software should occur frequently, from a couple of weeks to a couple of months, shorter being preferred [19]. In general, length of the cycle is from one to four weeks depending on the methodology. Agile methodologies emphasize commitment to rapid feedback cycles, collaboration and communication, and that way, continuous improvement. *Working software* is used as the primary measurement of progress in agile methods, and documentation other than code is discouraged [19, 21]. To be able to deploy working software, continuous software engineering practices (see Chapter 2.3) can be utilized to support agile methodologies and to improve software quality and development agility [4, 22].

Agile software development methodologies are solution for frequently changing requirements. While traditional methods are inflexible and cannot adjust to changing needs, agile methods are able to address those problems [17, 21]. Nevertheless, due to lesser planning at the beginning, agile methods have higher risk of major architectural errors because of lack of knowledge. On the contrary, traditional plan-driven methods consume more effort on initial planning, which certainly reduces the ability react to changes. If requirement changes are foreseen in planning, however, these methods are capable to stay within budget and schedule even in big projects [17]. In summary, agile methods adapt well to changing requirements, while traditional plan-driven methods are inflexible and thus future changes in requirements should be taken account in planning phase.

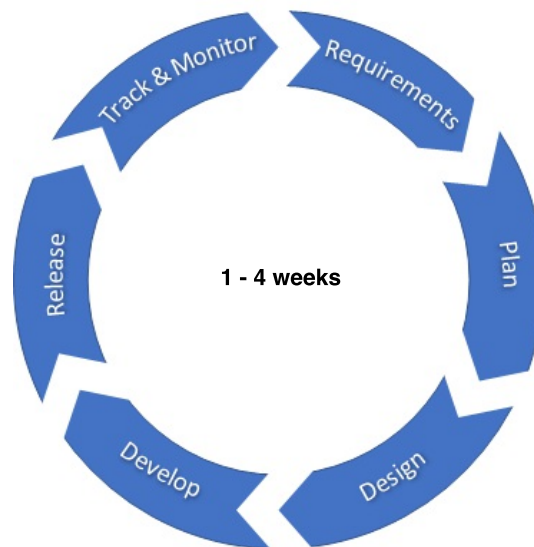


Figure 3. The agile development cycle.¹

Adoption of agile methodologies is not trouble-free. There are issues related to management and organization, people, processes and technology. Agile methods require tools which support it, and people need to be trained to use those tools. Development process – which traditionally is process-centric – must be changed to people-centric. Team size is changed to small, in which collaboration in team and with customer is always present. Organization culture has to be changed to support agile methodologies, since now the process is people-centric. Manager does not anymore encompass command-and-control style managing, but rather leadership and collaboration. [21]

2.2.1. Scrum

Scrum is not a strictly defined method, process or technique. Rather, it is a framework under which it is possible to utilize different techniques in agile software development. It defines the way how to manage system development process, not the techniques used

¹<https://number8.com/kanban-versus-scrum/>

for implementation. It is focused on actions of team members to make development process flexible so that it is capable of reacting to changes in environment [18]. With scrum it is possible to develop products which require encountering complex problems and adaption. Scrum framework contains following things: scrum teams with roles, scrum events, and scrum artifacts. Scrum events and artifacts are illustrated in Figure 4. [23]

Scrum team is cross-functional team which organizes itself to reach the goal of each sprint. It contains following members: product owner, development team, and scrum master. *Product owner* is responsible for the project; they keep managing product backlog list to make sure that everyone in development team knows what to do next. *Scrum master* is a management person, who makes sure that scrum theory and principles are followed during development. They interact with both customer and scrum team. Scrum master tries to keep scrum team productive all the time by removing any obstacles it could encounter. *Development team* is doing the programming work. Job of team members is to pick items from the product backlog and turn them into releasable increments in functionality. [18, 23]

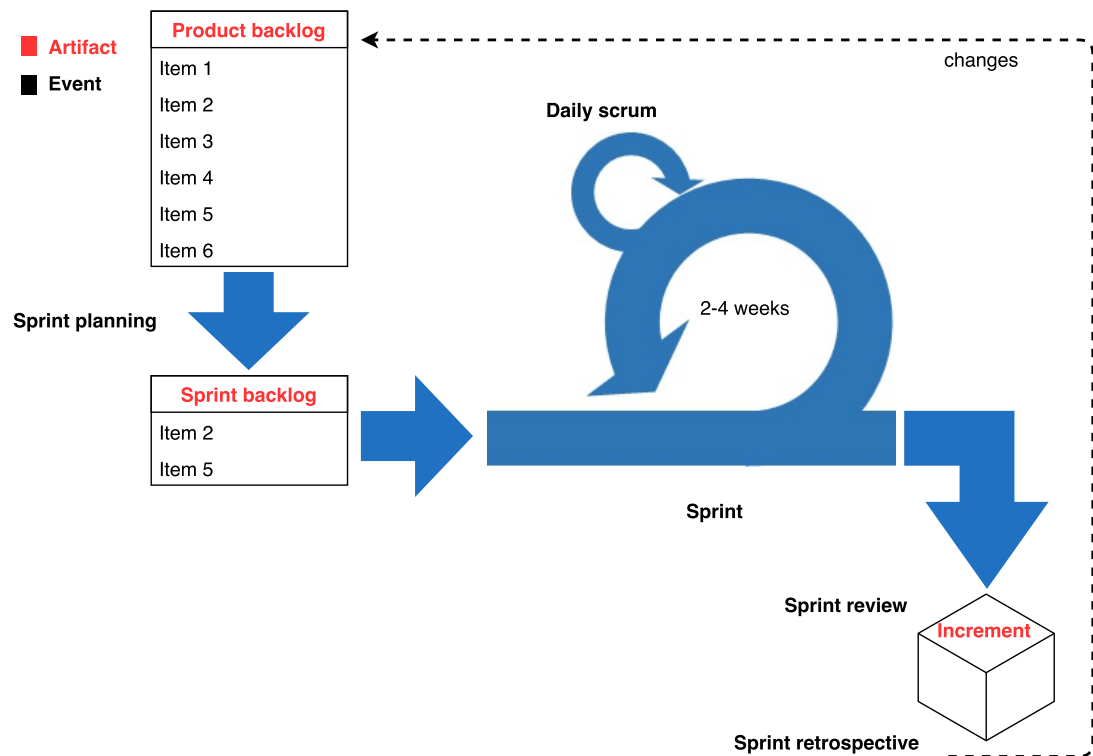


Figure 4. Scrum workflow with event and artifacts.

Following *Scrum events* are all included in workflow of scrum framework: sprint, sprint planning, daily scrum, sprint review, and sprint retrospective. All the events are limited by time, so they do have a pre-defined maximum duration. *Sprint* is the core event of scrum, it is an iterative cycle whose purpose is to create or to enhance functionality. Each sprint contains traditional phases of software development: requirements, analysis, design, evolution and delivery [18]. Duration of the sprint is fixed beforehand, but for other events, see Figure 4, the duration can be shortened if they complete

earlier and the goal is achieved. When a sprint concludes, the next sprint is started immediately [23]. The length of a sprint is one month in maximum [18, 23]. *Sprint planning* is a meeting conducted at the beginning of a sprint, in which it is decided what will be done next sprint and how. *Daily scrum* is short, daily meeting for development team. In that, future work done before the next meeting is planned for each member and work done since the last meeting is covered [18, 23]. In *sprint review*, which takes places at the end of the sprint, the results are presented. There stakeholders and the scrum team review the work accomplished during the sprint. If need for any changes to product backlog arises, they will be made. *Sprint retrospective* is held after sprint review but before the planning of the next sprint. Its purpose is to find things, that could be done better in the next sprint. Therefore, these meetings support continuous improvement [23].

Product backlog, sprint backlog and increment are *scrum artifacts*. *Product backlog* is the list of items that are required by final product. In other words, it lists everything that is needed to do in the entire project. It is not fixed and never complete, since changes will be made during iterations. Product owner is responsible of maintaining it. *Sprint backlog* is starting point for each sprint. It contains set of product backlog items, which are going to be implemented during the sprint. Items to sprint backlog are selected in sprint planning. Only development team is allowed to add items into the sprint backlog during the sprint if needed. It gives good vision of state of the sprint, since remaining work is seen as well as finished tasks. *Increment* is all the product backlog items completed during the sprint summed up and integrated to previous work. Increment must be in condition that it can be released. One increment is always a step closer to scrum goal. [23]

In the company development work done by CI team obeys principles of scrum framework. The sprint length is usually set to four weeks, occasionally set lower if necessary. Software developers committing their changes as input to CI system, see Figure 5, also use scrum, and the work is managed by using a digital workflow tool.

2.3. Continuous software engineering practices

Continuous integration (CI) is a practice under continuous software engineering, in which work of developers is integrated continuously to existing code. In traditional software development, the project is integrated at the very end, which causes lots of integration problems [2, 24]. Those cause delays and lost productivity. The main purpose of CI is to minimize integration errors and find causes for them as quickly as possible, and therefore speedup the development process as well as provide rapid *feedback* for the developers [1]. In other words, it removes or reduces the gap between software development and deployment. To be efficient, CI requires fully automated building and testing machine.

Continuous software engineering contains many other practices such as continuous planning, continuous delivery (CD), continuous deployment, continuous verification, continuous testing, and so on [25]. The usage of abbreviation CD is not consistent in literature. Sometimes CD is used to mean continuous deployment instead of continuous delivery or both. In this paper CD stands for continuous delivery, and continuous

deployment has no abbreviation. In this chapter CI, CD and continuous deployment practices are covered.

2.3.1. Continuous integration

Because in CI development work is integrated to existing code frequently, the project is ready to be released immediately or after a short time. The work should be integrated at minimum once a day [1, 24]. It is experienced that integrating an entire project at the end of it will lead to serious problems, which can cause delay and increase the cost of the project [2]. Therefore, functional CI system is necessary to maintain successful software development. The longer the integration is delayed, the higher is the possibility for integration problems [26].

The goal of continuous integration CI is to build every change. That means that when developer makes changes in source code, only that change is built. In this situation, finding cause for failed build is easier, since there is only one commit involved. Build is the unit in which the change moves through CI pipeline. Building does not only mean compiling, it may also contain inspection, testing, and deployment. In other words, building is verifying that new version of source code works as intended as a cohesive unit. A typical cycle of operation in CI system is listed below and also seen in Figure 5: [2]

- Developer **commits** changes to version control system
- CI server polls changes in source code, change triggers a **build**
- Change is built, **integrated**, and tested
- Results are provided for developer as **feedback**

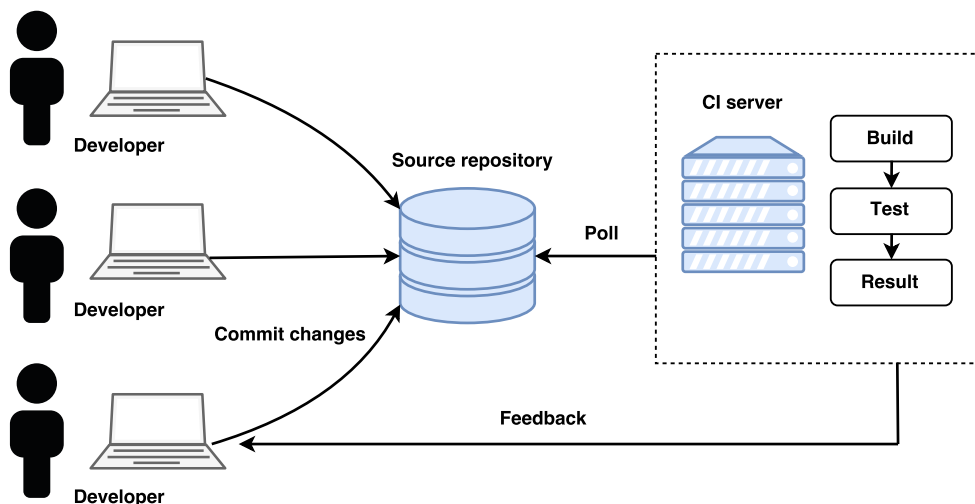


Figure 5. Components and operation of typical CI system.

There are many benefits in the use of CI. One is *reduced risk*. When new code is integrated to the existing one bit by bit, the risk of facing major integration problems is reduced. Moreover, integration is ongoing all the time, so it is not separate event anymore [24]. Other great benefit is *rapid feedback*. Developers know after a short time if their changes were good or not. CI also enables better project *visibility*, so that state of the project is easily seen. As mentioned previously, software is quickly *releasable at any point in time*. And in addition, with help of *automation* there is less manual processes, which ensures that they run same way for every build. [1, 2]

2.3.2. Continuous delivery

Continuous delivery (CD) is an approach in software engineering in which software is developed in short cycles with further automation. It is an extension of continuous integration and it automates building and testing processes [27]. Often CI alone is not enough. Having manual further testing and releasing process causes long feedback cycle and slows down the development. For example, based on reported bugs, developers need to make fixes on functionalities they developed many weeks ago [28 p. 105-106]. That is why CD is needed, to keep *feedback cycle short* and the software *deployable all the time*.

As mentioned previously, CD ensures that the software is ready to be released at any point of time, always when needed. That is achieved by having *automated acceptance testing* [28 p. 109][29, 30]. When release candidates are provided continuously, the risk of integration problems for a single release is reduced. Moreover, since every change is versioned, finding causes for errors is faster [30]. In CD, however, the decision to deploy software to production is manual unlike in continuous deployment (see Chapter 2.3.3) [24, 31]. The difference between continuous delivery and continuous deployment pipelines are seen in Figure 6.

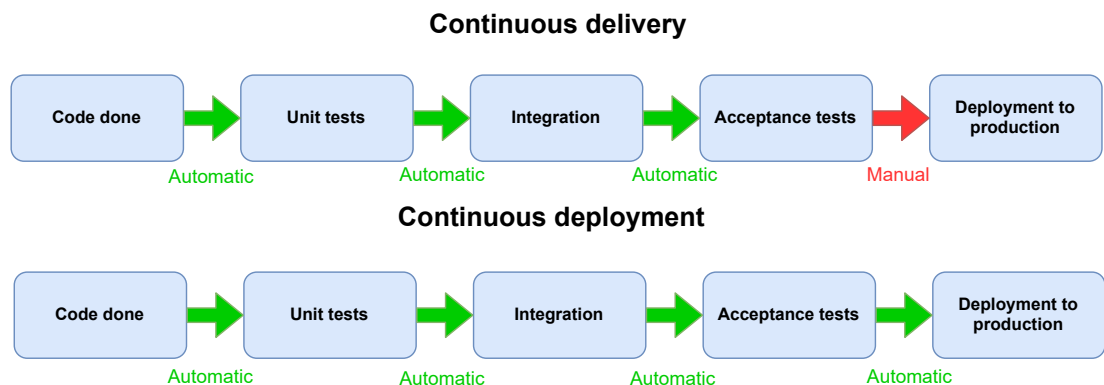


Figure 6. Continuous delivery and deployment pipelines from version control to ready product.³

CI has been in use for some time, but CD has not yet been widely established. The reason for that is the higher complexity of CD, and there are plenty of challenges to

³<http://blog.crisp.se/2013/02/05/yassalsundman/continuous-delivery-vs-continuous-deployment>

overcome to create successful CD concept [25]. Even though the benefits of CD are obvious in release frequency, fast feedback and reduced risk, the implementation of CD may require lots of effort. Implementing CD in large, existing environments may be challenging due to changes needed in organization in addition to challenges in processes and technology [22, 31]. More accurately, issues in build design, system design, integration testing in addition to organization, human and resource related issues are present when adopting CD [26]. Therefore, when adopting CD or CI, detecting possible problems in early stages will make the implementation easier.

2.3.3. *Continuous deployment*

Continuous deployment automates CD even further (CD is a prerequisite). It is added at the end of CD pipeline to automate releasing and making them available for customer [24, 29]. In other words, it means that when developer's change passes CD pipeline, it is deployed to production *automatically*. Thus, the difference between CD and continuous deployment is that while CD focuses on keeping software ready to be released state, in continuous deployment the software is also released continuously without human interaction (see Figure 6).

Continuous deployment may sound risky, since there is no human decision involved in releasing, but because every change is deployed and they are smaller, the overall risk is reduced [28 p. 109]. In continuous deployment as well as in CD, the feedback is provided continuously from every phase of testing. Central role of feedback in continuous delivery or deployment pipeline is illustrated in Figure 7, in which an example of changes moving through the pipeline is presented [28 p. 109]. Continuous deployment has the fastest feedback cycle and it is able to bring new features to production quickly, and gain feedback from the customers. For continuous deployment and CD, the largest disadvantage is painful implementation. In continuous deployment, the cycle must also be fastest to react customer feedback and problems e.g. when erroneous releases that are deployed in production. Close collaboration is needed with customer continuously [4]. [30]

Continuous deployment is utilized by some large companies to answer frequently changing market needs. Often these changes are unpredictable, and that is why software developing companies need flexibility. That explains the increasing attractiveness of these agile practices. A company has to be reactive for changes, and this is possible due to agile software development methods and continuous deployment [32]. The main benefits from continuous deployment in addition to CI's benefits are increased release frequency, improved customer satisfaction and bringing development and operations closer to each other [33]. Overall, continuous deployment is the most fast-paced of these continuous software engineering practices due to lesser human intervention.

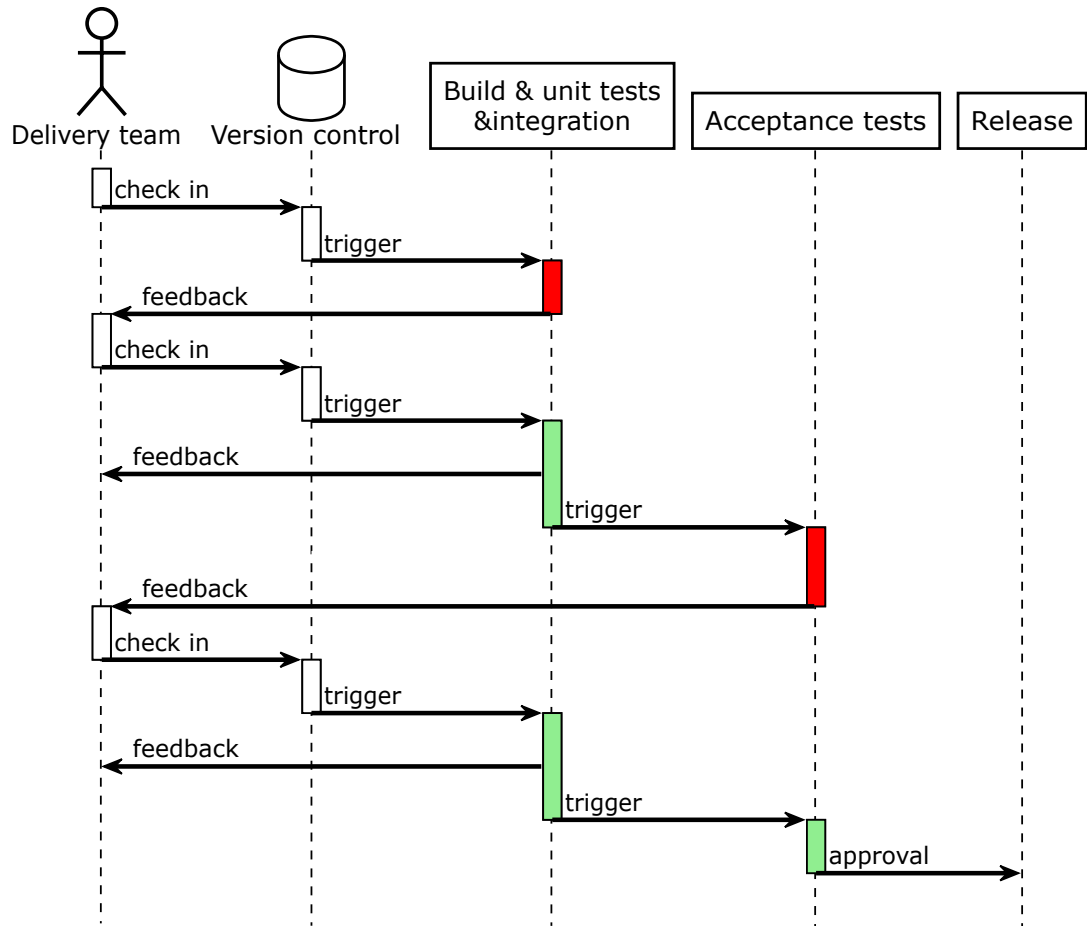


Figure 7. An example of code changes moving through continuous delivery/deployment pipeline. Red color denotes failed and green color passed tests.

2.4. Techniques for test optimization

Continuous integration can be considered, almost entirely, to be regression testing. These changes in software are continuously tested to work as previous version of the software (see regression testing in Chapter 2.1.3). Generally, there are three main ways to optimize testing: *test selection*, *test minimization* and *test prioritization* [3, 34]. The goal of these optimizations is to reduce testing cost by reducing the number of executed tests, or to enable providing feedback faster or more accurately [3, 15, 34]. The main question to be answered is that which tests to select to execute on what code and when [3]. Test suites tend to grow, and at some point, running them all becomes infeasible. That is the reason why test optimization techniques are needed. Overall, testing is balancing between benefit and cost [35, 36]. Therefore, the cost of high execution time may often overwhelm the benefit of good fault finding capability [37]. When the advantages of test optimization are greater than the disadvantages, it can be considered beneficial.

The most straightforward, and the most time-consuming, approach for regression testing is *retest-all* approach. In that the new version of software is tested with the same complete test set than previous version [35, 36, 38]. Nonetheless, because complete test set is always executed, it finds all possible faults the test set can find. Often running entire test set is not possible, because of limited time between the test executions. This may be faced in constrained environments, such as CI environment [35, 36].

There are research papers available of test optimization, but many of them do not consider continuous integration. The time between executing tests for different builds might be short in CI environment [35]. Nevertheless, many of those techniques seem to be applicable to CI environment as well. Many of these test optimization techniques rely on test suites' code coverage. Code instrumentation is often used to monitor code coverage of test suites. This information can be used to match them with changes in the code. It needs to be noticed that code coverage information needs to be updated frequently to stay accurate [35]. Certainly, the need of this operation is dependent on how frequent are the changes affecting the test suite code coverage. Anyhow, approaches that perform test selection using test execution history do not require maintaining code coverage information. In addition to coverage-based and history-based techniques, there are approaches utilizing mathematical models and probabilities [3 p. 14-24]. Different implementations utilizing the techniques introduced in this chapter are addressed in Chapter 3.

Broadly speaking, a test suite can be considered as larger entity than a test case and a test suite contains many test cases. The usage of these terms is not clear in literature and they are sometimes used interchangeably. These techniques, however, are applicable in both levels since their principle is similar.

2.4.1. Test selection

Test selection is a technique, which attempts to reduce the total execution time of tests. Test suites to be executed are selected based on *changes in code*. The difference to test minimization (see Chapter 2.4.2) is that it attempts to select all the tests that are required for the current version of program [3 p. 14]. Test selection is performed by selecting an adequate subset from a complete test set based on changes in code, and use that for testing.

Test selection, however, is not trouble-free. First, selecting the subset of the tests might be challenging. Secondly, sometimes it is difficult to know if more testing is needed in addition to the selected subset [35]. Most likely, even most sophisticated test selection techniques cannot find all dependencies between the changed source code and the test suites. If test suites are selected too strictly, the risk of larger number of faults passing the testing undetected increases.

Following is an example of selective retesting. Let P to be program and P' be modified version of program P . Let T be all the test sets for P . Typical procedure for selecting regression tests goes as following [36, 38]:

- Select $T' \subseteq T$, a set of tests execute on P'
- Test P' with T' . Establish P' 's correctness with respect to T'

- When needed, create T'' , a set of new test cases for P'
- Test P' with T'' , establishing P' 's correctness with respect to T''
- Create T''' a new test suite and test history for P' , from T , T' , and T''

Of which first step involves regression *test selection problem*: correct T' is needed to test changed program P' [3 p. 4][38]. In this paper, test selection problem is addressed.

2.4.2. Test suite minimization

The objective of test suite minimization is to find minimal sized set of test cases that fulfills the requirements set by test criterion [37, 39]. Initially, identifying test cases in a test suite that are redundant or obsolete is needed, since they do not bring any value towards selected test criterion [3]. Test criterion are defined as “*rule or collection of rules that imposes requirements on a set of test cases*” [37]. Thus, the idea of minimization is to find minimum amount of cases that are dependent on current version of the program by identifying obsolete or redundant cases and removing them from the test suite. So, there can be test cases in complete set, which do not increase the coverage of test suite and are useless at that moment. Those are treated as redundant and removed. Test suite minimization can also be called “*test suite reduction*“, with the difference that reduction means removing the test cases permanently [3 p. 4]. Test suite minimization problem is stated as follows [3 p. 4][15][40]:

Given: A test suite T , a set of test requirements r_1, \dots, r_n , that all must be satisfied to provide sufficient testing of the program, and subsets of T , T_1, \dots, T_n , of which one is associated with each of the r_i s such that any one of these test cases t_j belonging to T_i can be used to test r_i .

Problem: Find a representative set, T' , of test cases from T that satisfies all of the r_i s.

In optimal case, T' should be minimal set of cases which satisfies all requirements (at least one from each T_i), and hence, it satisfies testing criterion as well. Minimization effect is maximized with minimal set that covers the requirements [3 p. 4]. In general, defining the effect of minimization to fault detection capability is not straightforward [3].

Typically, the minimization relies on coverage based measures. Finding if the test case is truly redundant might be difficult, thus there is risk of reduced capability to find faults [15]. The benefit of test suite minimization is that under certain conditions it reduces the cost of regression testing without reducing fault finding capability due to executing the minimal set of tests that satisfies the test criterion [37]. Previous statement is denied by the argument that methods which claim to minimize test suites without reducing fault detection capability are lacking in generalizability and are suitable for particular use cases only [40].

2.4.3. Test prioritization

Test prioritization does not shorten total test execution time, but with that it is possible for developer to receive feedback from their changes faster [3, 22, 35]. If in initial situation, tests are executed in random order without any prioritization, the faults can be found continuously during the testing phase and in the end, all are found and feedback from them is received. With test prioritization, however, it is possible to reorder tests in a way that the most critical tests, which detect *most of faults*, are executed *early*. Other advantage of prioritization is that it is always safe to do. It has no effect on fault finding capability of test suite, since there is no subset selected from initial suite [3 p. 40]. If fast feedback time from testing is crucial, prioritization is beneficial, but as mentioned, the total execution time remains the same.

Test case prioritization problem is stated as follows [3 p. 5][39]:

Given: Let test suite be T , set of permutations of T , PT , and a function f is relation between permutations PT and real numbers \mathbb{R} , thus $f : PT \rightarrow \mathbb{R}$.

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

Output of f is called award value, and it defines the order in which test cases are ordered [39]. Set of permutations PT are all possible different orders of combinations the test cases can take in suite.

To be able to perform prioritization, some criterion is needed for that, i.e. what is the metric used to prioritize, to have an effect on the award value. There are multiple approaches addressing this issue. One commonly used metric is structural coverage. The idea behind coverage based approaches is that the higher is the achieved coverage, the higher is the fault detection rate. Therefore, tests are ordered to maximize coverage as early as possible. Coverage metric itself can be, for example, decision, branch or function based [3 p. 26]. It is also possible to find dependencies between code modules and tests, called *dependency coverage*, and use that for prioritization [34]. One approach is to prioritize tests to find high-risk faults as early as possible, or prioritize tests to faster reveal faults in code changes made in certain parts of code [39].

Overall, finding sufficient coverage data might be cumbersome. Fortunately, not all the prioritization techniques rely on coverage. One simple and inexpensive approach is to use test suite execution and result history for test prioritization [35, 36]. The tests are prioritized based on their effectiveness in historical view, that how well they did previously find failures. More effective tests are certainly prioritized higher. Some approaches utilize both history and coverage [34].

3. APPROACHES FOR TEST SELECTION IN CI

In the literature there is plenty of research conducted about topics in test optimization. However, because continuous integration is a relatively new practice, the amount of research conducted in CI environments is more limited. The main difference is that in traditional regression testing, the test optimization techniques do not have strict time constraints. On the contrary, in CI environment the time between executing tests is limited and optimization has to be done *continuously*.

The methodologies introduced under this chapter are relatively recent, and they are implemented in CI environment. They all rely on either code coverage measures between tests and source code, or utilize historical test execution data, or both. The environment, in which this thesis is conducted is introduced in Chapter 4. The approaches here are selected to have something beneficial regarding to the goals of this thesis. Thus, test selection has the primary focus, but all the approaches here also incorporate test prioritization.

3.1. Test execution history based selection and prioritization

Elbaum et al. introduce method for increasing cost-effectiveness of continuous integration processes by utilizing test selection and prioritization techniques. Testing composes of pre-submit and post-submit phases. In pre-submit phase it is pointed out by developers which modules need to be tested, and then the most cost-effective subset of tests is executed before submitting code to codebase. In post-submit phase all test suites are executed, but first prioritized in an order to provide faster feedback (suites to fail most likely are executed first). As result, faults are found faster, and test execution times are reduced, which leads to improved cost-efficiency. The research is conducted on test suite level instead of test case level, but it is claimed to work on either level. [35]

Test suite execution history data is used for test selection and prioritization. Hence, the approach does not rely on code coverage information. General problem in CI environments is that code coverage data gets out of date quickly because of frequent changes in source code. Since the test execution history does not utilize any coverage information, it is well applicable in CI environments. [35]

Idea of pre-submit testing is to quickly find as many integration problems as possible. Changes in code can proceed to heavy post-submit testing only if pre-submit testing is passed. This makes obtaining feedback faster and reduces unnecessary test executions. Currently pre-submit testing suites are selected by code module dependencies based on the changed modules the developer has reported. The problem here is that if there are too many modules reported, the post-submit testing may become a bottleneck. On the other hand, if too few modules are reported, it will cause excessive execution of post-submit tests since faults are not caught earlier. [35]

Selecting pre-submit subset of tests is based on last execution and last failure windows, W_e and W_f respectively, which are applied for each test suite. Those windows can be either time or execution count based, now they are based on time. The size of execution window defines maximum time or execution count before test suite is re-executed. Failure window ensures that a failed test suite is executed in future until it is

out of the failure window. W_e makes sure that not a test suite is totally ignored, which could happen with W_f only. These windows are illustrated in Figure 8. The point of

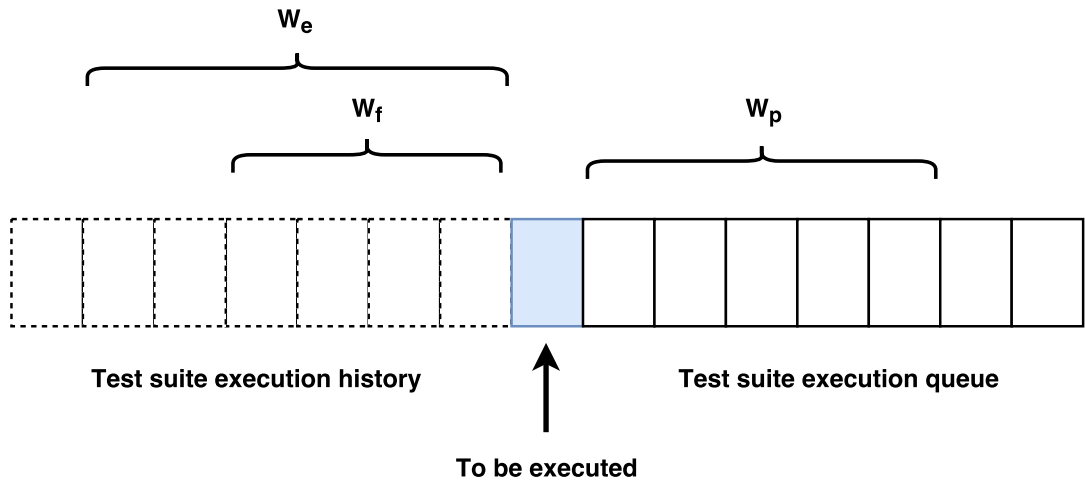


Figure 8. Test suite execution windows. W_p is utilized only in post-submit phase.

this new way of selection is to reduce needed effort but still preserve high effectiveness in fault finding. Algorithm used to select pre-submit tests is presented in Listing 1. [35]

Pre-submit test selection algorithm

Parameters:

Test suites 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'

Listing 1. Pre-submit test selection algorithm.

In post-submit phase tests suites are continuously prioritized with frequency defined by prioritization window W_p , which is also seen in Figure 8. It can also be either time or execution count based window, now it is based on time. After the prioritization window is exceeded, test suites in queue are re-prioritized (reordered) with the help of W_e and W_f . If W_p value is set to 1 in the case of execution count window or to very small number as a time window, it equals to no prioritization. The point of this post-submit test prioritization is needed to get feedback earlier regarding the faults that managed to get this far undetected. Post-submit test prioritization algorithm is presented in Listing 2. [35]

Post-submit test prioritization algorithm

```

Parameters:
  POSTQueue
  Failure window  $W_f$ ,
  Execution window  $W_e$ 
  Starting point  $P_p$  in POSTQueue
for all  $T_i \in$  POSTQueue after  $P_p$  to
  lastEntry.POSTQueue do
  if TimeSinceLastFailure( $T_i$ )  $\leq W_f$  or
    TimeSinceLastExecution( $T_i$ )  $> W_e$  or
     $T_i$  is new then
     $T_i$ .Priority  $\leftarrow 1$ 
  else
     $T_i$ .Priority  $\leftarrow 2$ 
  end if
end for
SortByPriority(POSTQueue,  $P_p$ , lastEntry.POSTQueue)
 $P_p =$  lastEntry.POSTQueue

```

Listing 2. Post-submit test prioritization algorithm.

The selection of sizes of windows W_e , W_f and W_p certainly depends on case. They can even be adjusted dynamically. Anyway, pre-submit test selection provides savings in execution time which overwhelms the reduction in finding faults in pre-submit phase. Flaky test suites, which fail because of environmental factors, are one noticed problem here. For handling those, rerunning is suggested before judging them to be failed. [35]

3.2. Faster fault finding by test optimization

Yoo et al. address problem of shortening feedback time in environment with short development cycle. Continuous integration is not clearly mentioned to be part of this research, but it seems to be present since there is mention about short-cycled regression testing. The issue here is that despite heavy parallelism on testing, the time between submitting changes and receiving feedback is too long. The goal is not to reduce number executed tests, but to select and prioritize subset of them for early execution. [34]

This approach claims to be combination of introduced test optimization techniques introduced in Chapter 2.4. Thus, it contains elements of test minimization, selection and prioritization. In this approach testing is split to pre-submit and post-submit phases also seen in other approach (see Chapter 3.1). Both dependency coverage between source code modules and tests, as well as fault history of tests are utilized. False positives, tests that fail without cause in code, can be called as environmental failures and they are filtered out by heuristic decision based on fault history. The approach

aims to select adequate subset of tests to pre-submit phase and prioritize them to enable earlier feedback. [34]

It is experienced that fault history is useful addition when selecting subset of tests for early fault detection. Coverage dependency alone does not succeed well in this task. With both of those combined, faults are found earlier. Due to that, the test execution time is reduced 33% - 82% using this approach. As future challenges, better detection of environmental factors and improved dependency precision are listed. The principle of dependency coverage is illustrated in Figure 9. [34]

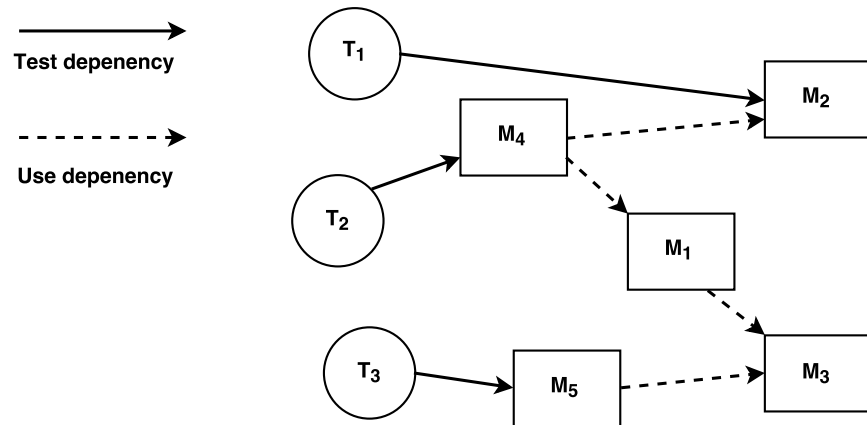


Figure 9. Example of dependency coverage. Changes in module M_3 has impact on modules M_1 , M_4 and M_5 . Based on dependency coverage, it can be seen that test T_1 covers only module M_2 , T_2 covers M_4 , M_2 , M_1 and M_3 , while T_3 covers M_5 and M_3 .

In this approach dependency coverage between modules and tests is mentioned to be already available. Techniques used here may be useful in this thesis, but it is not explained how the initial dependencies between tests and code changes are obtained. Those are used to calculate dependency coverage values. The key issue is to first find feasible way to determine dependency between code changes and test sets.

3.3. Code-churn based test selection

Knauss et al. provide solution for a problem of selecting suitable tests and prioritizing them in CI environment in system level. It is implemented based on changes in code churns. They are defined as added, deleted or modified lines between different versions of a code [41]. The implementation relies on a statistical model, which is created based on correlations between test-case failures and changes in source code. In addition, some qualitative research methods, semi-structured interviews and workshops are utilized. The goal of this approach is to support CI by providing as short feedback loop as possible. [22]

The test selection technique used in this approach is Code-Churn Based Test Selection (CCTS). It comprises two phases: historical analysis of test execution results and code churns and finding an optimal subset of tests using created statistical model. The first phase, historical analysis, requires two inputs: a list of source code changes and

list of executed tests cases and their results. As an output it has *contingency table*, see example Table 1, which lists executed test cases and changed modules. The table shows number of failed test cases per day for each module-case pair. From this table it is possible to observe, which test cases are sensitive to fail for certain code changes. The table is used as statistical model in next phase. [22]

Table 1. Example contingency table.

	Test case 1	Test case 2	Test case 3
Module 1	1	0	2
Module 2	2	5	1
Module 3	0	6	2
Module 4	0	4	3

The second phase, finding an optimal test suite, takes as input earlier created contingency table and list of recently changed modules. Now the contingency table is used to recommend test cases for source code changes with the help of precision, recall and f-measure information retrieval methods. Both recall and precision should be high, hence, f-measure is calculated, which is combination (geometric mean) of those two. High recall or precision alone does not guarantee sufficient results. Therefore, f-measure is used to select an optimal subset of tests. The recommendations of system are compared to what the tester would truly select. The information retrieval methods are based on four categories of errors: [22]

- **True positives:** The set of tests that are recommended and failed
- **False positives:** The set of tests that are recommended and did not fail
- **True negatives:** The set of tests that are not recommended and did not fail
- **False negatives:** The set of tests that are not recommended and failed

CCTS was tested in two companies as a case study [22]. In the contingency table it can be seen, how many times change on module caused failure for each test. Now tests were prioritized so that test with highest value is first on a list. Subset of highly prioritized list of tests was selected with two strategies, which both provided similar results. The simpler strategy was to select the first $n\%$ of the tests. Use scenarios for CCTS are reprioritizing tests (earlier fault finding) and re-structuring tests to run in different time scopes, e.g. hourly, daily, weekly (reduced execution time). [22]

CCTS resulted in a speedup of up to 3.7 [22]. That is substantial, approximately 73% reduction in test execution time. There is no clear mention what effect utilizing this approach has on fault detection capability. It is probably assumed that heavier tests are performed regularly to catch faults that passed e.g. daily testing undetected.

3.4. Discussion

Common for all the approaches here is that test selection is used to select subset of an entire set of tests for early execution. In the approaches introduced in Chapters

3.1 and 3.2, the entire test set is executed after pre-submit phase is finished. That means all the tests are executed at some point. In CCTS approach described in Chapter 3.3, restructuring tests for example for daily runs is mentioned as one use scenario. Especially at the beginning, regular running of all available tests ensures that the testing is not merely dependent on the test selection. Hence performance of selective testing can be tracked in terms of fault detection capability.

The history-based approach seen in Chapter 3.1 seems to be lightweight to implement and technically possible in the CI environment, see Chapter 4. The challenge is that there are some tests in the environment, which are unstable or “flaky”, and they may fail without actual errors in the code. Those are also called environmental failures [34]. In addition, in the environment of this thesis the software being tested is hardware-dependent platform software, and every fault passing the testing undetected is critical. Therefore, relying merely on test execution history does not seem to be reasonable solution here.

Dependency between changed files and tests at some level is needed for reliable test selection. The CCTS approach tracks changes on code churns level. In this thesis it does not seem reasonable to go deeper than file level, since that information is easily available. Developers’ recommendations could be used to support the test selection, but alone it would make it heavily developer dependent. It is clear that dependency between changes in code and tests need to be found. The first goal is to find dependency between different products and code changes to select tests only for a specific product. The next goal is to find dependency between test sets and code changes and later expand that to test suite and case level. The environment with all necessary details is explained in Chapter 4.

4. ENVIRONMENT & TOOLS

In this chapter components, tools and principles of CI environment are described in the scope of the thesis. An overview of the environment is given with more details when necessary. The environment described here is part of bigger continuous integration system, and it is a part of which the CI team is responsible for. The function of the system is to support the development of platform software in this business line. It is a notable detail, that here the software is heavily hardware-dependent, and integration testing is conducted on hardware targets. All necessary information about the environment that is needed in understanding this thesis is covered under this chapter.

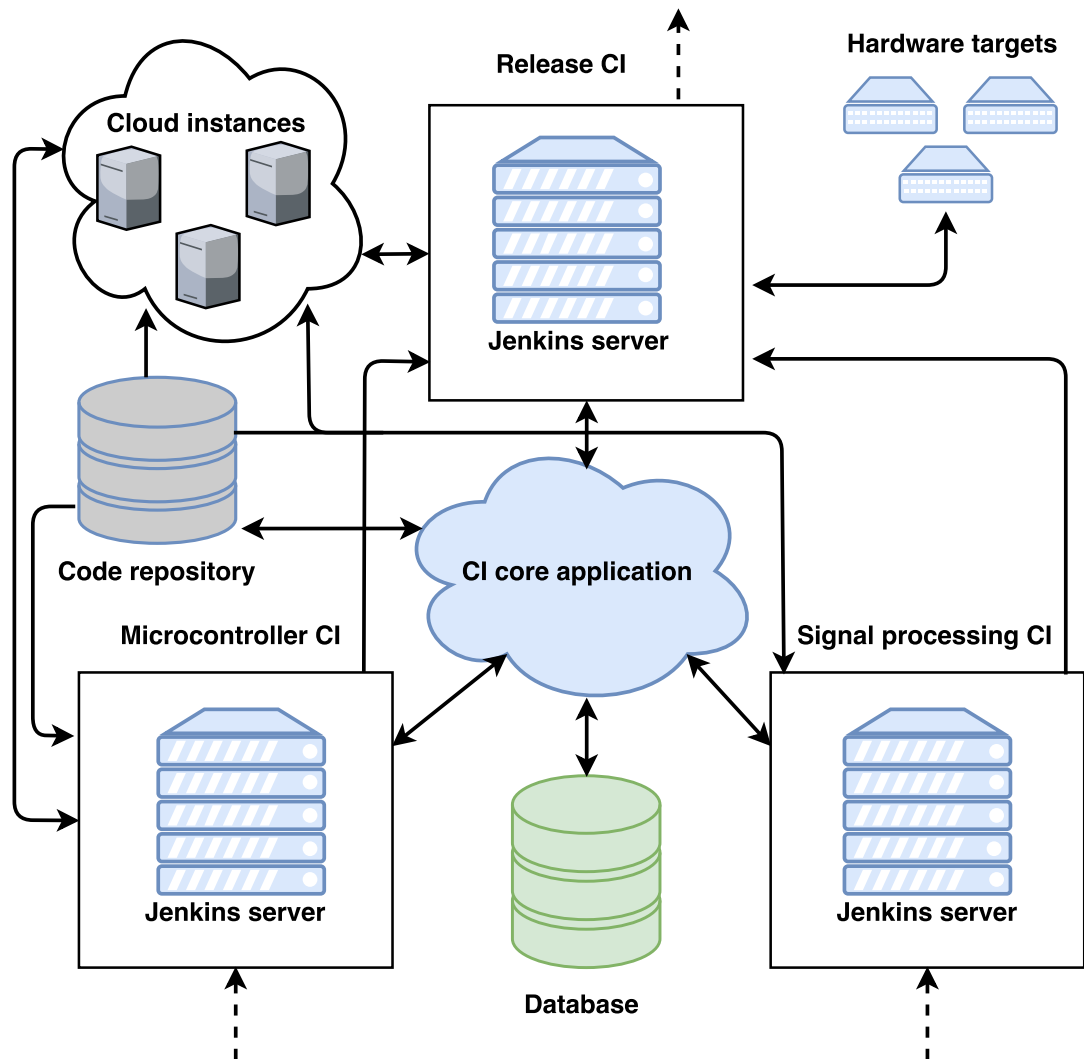


Figure 10. Main components of the CI environment in this thesis.

4.1. Continuous integration environment overview

The business line overall consists of four *system components*, which all have their own CI machine. In addition, *release CI* is a CI component, but not a system component. Therefore, there are four system components and five CI components. All the CI components together are called *CI system*. In this thesis, handling entire CI system is not reasonable, but it is focused on three of the CI components, which are microcontroller CI (*M CI*), signal processing CI (*U CI*), and release CI. These three CI components together with *common components* in the system are called *CI environment*, in this thesis. Overall view of the CI environment is illustrated in Figure 10.

The remaining two system components, named *L* and *C*, are low level system components, and they are not directly in the scope of this thesis. However, they are part of integration and their change has impact on test selection even though their code changes are not considered. Other essential components in the environment are CI core application, database, version control system (code repository) and Jenkins automation servers. These are introduced under this chapter in more detail. Hardware targets are covered under Release CI in Chapter 4.2. All the servers and cloud instances in the environment are running on Linux operating systems.

To be exact, there is a duplicate of each system component as well as database and CI core application (development environment i.e. Sandbox CI, and production environment i.e. Trunk CI). They form two similar but separate CI environments. The point of having these is to be able to make configuration changes and new features to environment safely. The idea of separate database instances for development provides flexibility for testing [2]. It is possible to test new scripts and configurations in development environment safely without affecting production side. If the changes work as intended in development environment, they can be moved to production environment with minimal or nonexistent changes.

4.2. Release CI

Release CI is the last component of the CI system, and it is the most relevant for this thesis. Here builds of *M* and *U* as well as builds of previous system components are integrated together. Builds of one product *p* are combined under objects called frame *F* (see Figures 12 and 14). Hence, a frame contains builds of all preceding system components for same product and it is the object flowing through pipeline of release CI, illustrated in Figure 11.

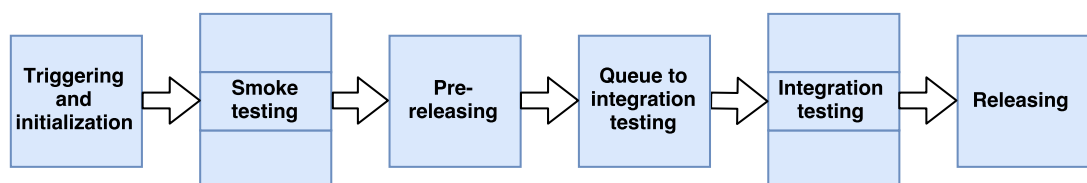


Figure 11. Simplified pipeline of release CI.

When builds for each p are ready in M and U CIs, the release CI is triggered. The frames are created of these builds. All the frames are combined to form a release matrix R , seen in Figure 12. Even though testing stages in pipeline are conducted for each frame separately, releasing for example, is made for an entire matrix. After their initialization operations, the frames proceed to smoke testing stage of integration. This testing is conducted in parallel to minimize time used. The purpose of smoke testing here, as covered in Chapter 2.1.3, is to find possible instabilities which prevent it from working as soon as possible. The idea is rather similar to approaches in Chapters 3.1 and 3.2, where relatively fast pre-submit testing is used to find if build is good or not. If those occur, the testing is stopped without spending more time. This stage is relatively fast, consuming around 20 minutes of time. Therefore, it is effective and worthwhile to do before proceeding to long-lasting integration testing.

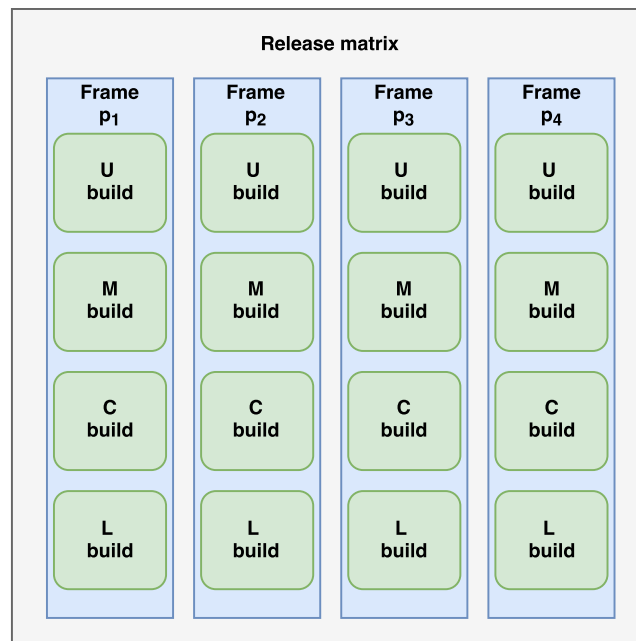


Figure 12. Anatomy of release matrix and frame. Matrix contains frames of all products supported by branch. Each frame contains builds of all system components for one product.

If smoke testing succeeds for all frames, they continue to pre-releasing stage. As mentioned, the frames are not released or pre-released one at a time. To make release, a full set of successfully tested frames, the matrix, is needed. Number of frames in matrix depends on software development branch B , and not all branches support all products.

After pre-releasing, the matrix is moved to integration testing queue, which is the next stage. The integration tests are similar to post-submit phase in Chapters 3.1 and 3.2, where heavier testing is performed. Because integration tests take long time to run, all matrixes cannot be tested, and they are discarded periodically in the queue. Purpose is to test the most recent matrix available to test most recent frames. Leaving some of the matrixes untested is a conscious risk, which has effect on traceability of the results of changes. A matrix is picked to integration testing when there is a free target,

and frames are tested in parallel again. Integration testing stage itself, for a single frame, can take up to several hours in addition to time spent in queue. This depends on the product being tested. Smoke and integration testing stages are conducted on real hardware targets using Robot Framework. If all the frames pass integration testing, a release is made.

Because integration tests take a considerably long time to execute, pre-release provides application layers faster access to latest changes. If faults arise in integration testing later, their criticality is analyzed and pre-release is set to *not released* or *released with restrictions* state accordingly. The advantage of pre-releasing exceeds its disadvantages, and therefore, it is done in every releasing process.

Integration testing and its queue have crucial role in this thesis. Integration testing procedure is illustrated in Figure 13, which contains "Queue to integration testing" and "Integration testing" stages originally seen in Figure 11, now with more detail. In Chapter 5.1 challenges in integration testing are addressed. The relations of the objects are seen in Figure 14.

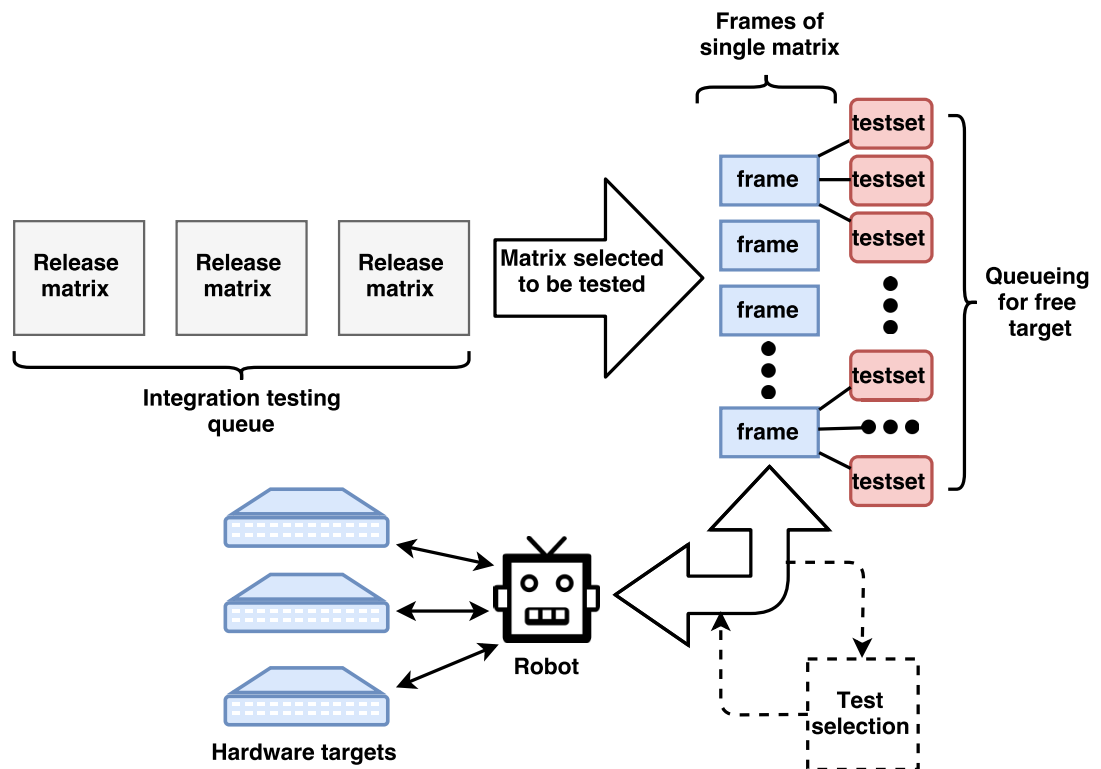


Figure 13. Principle of queuing and integration testing in release CI. Test selection is going to be implemented in this thesis.

In integration testing queue, release matrixes waiting to be tested are prioritized periodically, and the oldest matrixes are discarded since the queue may grow long. When a matrix is selected to be tested, all the test sets for all the frames it contains are executed. A frame contains different builds of a single product, as seen in Figure 12, which means that source code test set dependency can be found via product. More queuing can occur after being selected to integration testing, because even though some

test sets can be executed on single target simultaneously, some of them need to queue for a free target. Hence, the total time consumed for a frame here is measured from entering the integration testing queue until the execution all the test sets is finished.

Robot framework is generic open-source test automation framework, which can be used for acceptance testing and acceptance test-driven development [42]. Jenkins automation server (see Chapter 4.3.6) supports Robot Framework via plugin. In this environment, testing stages in release CI are conducted using Robot Framework on hardware targets. Main branch (see Chapter 4.3.5) has its own targets, while for integration branches the targets are common. Hence, usually shorter time is spent in queue in main branch than in integration branches. After testing has finished, and there are no more frames in queue, the target is set free by Robot. As input, it takes a list of test sets (see Chapter 4.3.3) to be executed. Here it is possible to reduce the number of executed test sets by dropping them off from the list. As mentioned in Chapter 4.3.6, one test set matches one job in Jenkins.

4.3. Common tools, items, and principles

Under this chapter, tools and items which are common in all the CIs are covered. The CI system obeys object-oriented programming (OOP) paradigm. Summary of objects and their relations needed in this thesis are illustrated in Figure 14. Build object b is the object moving in CI pipeline in M and U CIs. Commit objects and metric objects m are linked to build objects. Frame object F , which contains build of each system component, is the object moving in release CI pipeline. In release CI, metrics are linked to frames instead of builds. Release matrix R is formed of a group of frames, which are supported by that software development branch B . Product object p defines for which types of hardware variants the builds and the frames are created. Thus, the hardware and source code combination is different for every type of product.

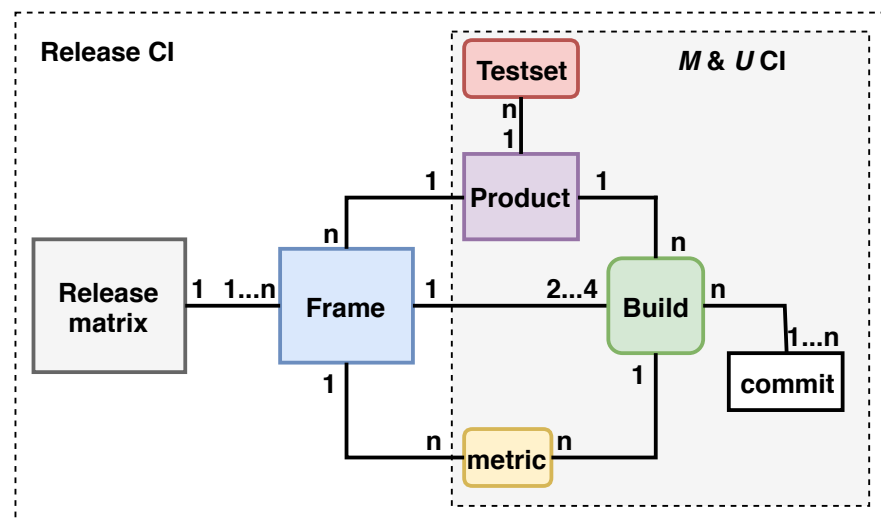


Figure 14. Relations of objects in database. For example, metric is related to one frame but the frame can be related to many metrics.

4.3.1. CI core application and database

CI core application has a central role in the environment (see Figure 10). It can be described as a single program, but rather it is a group of scripts making database operations and communicating with Jenkins servers. The CI core runs on separate server. It has two important tasks: manage and maintain CI pipeline, and provide views for anyone who needs them. The purpose of different views is to provide visibility of the status of the CI. The CI pipeline exists in database, not in the Jenkins servers, and therefore the CI core and Jenkins servers communicate continuously. When a script in Jenkins job needs database operation, it can access the database through remote procedure call (RPC).

The database is based on MySQL, which is relational database management system. It is used with Django web framework, which provides object-relational mapping (ORM) to use rows of MySQL database as objects of Python. It is a programming language, which supports many programming paradigms including object-oriented programming [43]. The database can be accessed using Python scripts, which is a convenient way to make write and read operations. Django includes a wide variety of components as default, for example, web templating system and Uniform Resource Locator (URL) dispatcher to create web applications out of the box [44].

Objects are defined by *models.py* file in Django. Every class defined in this file can be thought of as a database table, in which field and behavior is set by attributes. They are defined under each class. Objects and attributes determined in the file, can be accessed by queries. An example of a query, retrieving all objects of Person model is shown in Listing 3. [44]

```

from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name  = models.CharField(max_lenght=30)
    age       = models.IntegerField()

all_persons = Person.objects.all()

```

Listing 3. Django model and query example.

Object location in the CI pipeline is defined by certain attributes, which can be called coordinates. When Jenkins job finishes for a build for example, it makes call for the CI core to move the build object to next location. As mentioned in Chapter 4.3.6, Jenkins servers are not aware of the structure of the pipeline. Jobs in Jenkins may trigger other jobs, but those do not cause any object movement in the pipeline. Therefore, all transitions from one stage to another, seen in Figures 16 and 11, are managed by the CI core.

Views are created by querying wanted objects and attributes in database and rendering them in Hypertext Markup Language (HTML) templates to provide visual representations. These views can be displayed using web browser. The number of different views, as expected, is indefinite. It heavily depends on use case what kind of view is

useful. It is convenient for the developers to have a view showing whether or not the changes they recently committed passed all the testing. For the CI team, overall picture of CI pipeline is more important, not just focusing on status of a single commit. For statistical purposes the views can provide information of long-term measures of data. For example, a statistical view could show average throughput time of commits for each month or week. All in all, a view can be anything as long as the contents are retrievable from database. *Metric* objects are often in central role when creating views. They are covered in separate chapter, see 4.3.2.

4.3.2. Metrics

Metrics are the most numerous object type in the database. They are linked to either build or frame objects, which flow through CI pipelines of the system components covered in Chapters 4.2 and 4.4. Single metric object stores information of one job completed in CI pipeline for one build/frame object. E.g. when compile job is completed for build object b in M CI, a metric m is linked to b with time stamps of start and end of compilation, and result of the job. This is done for every job that the object b encounters in the pipeline. In release CI, metrics are linked to frames, but else their operation is similar (see Figure 14).

Metrics are important since they provide visibility and enable collecting detailed statistics in the CI system. Views, which are described in Chapter 4.3.1, are often based on data achieved by querying metrics. Typical view showing all jobs executed for build object with results is based on metrics, and it gives visibility of the status of the CI system. In the implementation and testing parts of this thesis, metric data is utilized multiple times.

4.3.3. Test sets, test suites and test cases

There are three levels of tests in this environment. Test set is the highest level entity and it defines a type of the testing. Under a test set there is a collection of test suites. They test if tested program code can perform required functionalities. Test cases are the lowest level of tests. One test suite contains collection of test cases. One test case tests specific functionality of the tested program, e.g. turning device power on/off.

A test set matches one job under Jenkins in testing stages of CI pipelines. For example, in release CI pipeline, seen in Figure 11, there is multiple test sets per product under smoke testing and integration testing stages. When frame of product p reaches integration testing stage, all the test sets are executed for that product. In integration testing stage the number of test sets depends on product and branch configuration. Integration testing procedure is illustrated in more detail in Figure 13.

4.3.4. Version control system

Version Control System (VCS) stores all the code that is used in the CI environment, for example, source code and scripts. Even though different VCSs may operate different way, basic functionality needed in this thesis obeys similar principle.

In this thesis it is remarkable, that in VCS, there is common repository for all products in each branch. That means that different products have common source code and they are separated in run-time of compilation by compiler flags. Output of compiler is used to find dependencies between the products and changes submitted to VCS in this thesis. Main branch, for example, supports 6 different products named $p_1 \dots p_6$. Consequently, that means that there is source code of all 6 products under the same repository. Repositories for different system components M and U , however, are separated. In this thesis, changes submitted in both M 's and U 's repositories needs to be investigated.

4.3.5. Branching

The CI system obeys principles of Trunk-Based Development (TBD) branching model. It is well suitable with CI, and for business companies, TBD requires CI due to high volume of commits. Ensuring goodness of commits is simply not possible without CI. [45]

In TBD the mainline, the primary software development branch, is called *trunk*. In this CI system it is named *main branch*. In addition to it, there are several types of other branches. Feature branches aim for customer releases and they are made based on customer release schedule planning. Integration branches integrate software of this business line to other business lines. Lifetime of integration branch is few days and during that time the integration is completed and then integrated platform is updated to other business lines' trunks. There are two integration branches alive simultaneously. When one is merged, the other is created as seen in Figure 15.

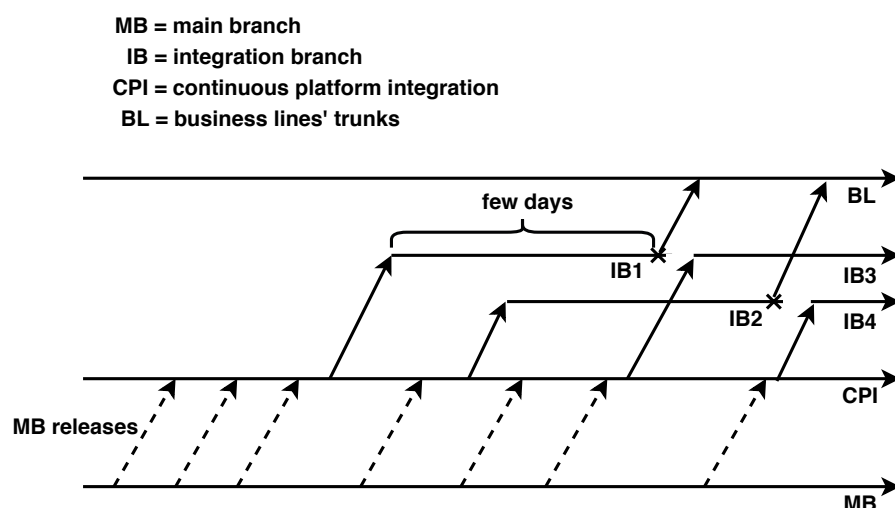


Figure 15. Principle of branching in the environment.

Because creating integration branch directly from main branch could have lot of issues, new releases in main branch are continuously picked to continuous platform integration (CPI). Its purpose is to pre-integrate releases of main branch and after certain time, integration branch is created from pre-integrated releases in CPI.

Overview seen in Figure 10 is valid for all branches. Nevertheless, different branches may have different configurations in the CI system. For example, number of products can vary, and job configurations may differ between the branches. This thesis focuses on main branch at the beginning, since it has the most activity of all the branches, and hence testing and validation is the most convenient there.

4.3.6. *Jenkins automation server*

Jenkins is an open source automation server which can be installed on Windows, Mac OS X or Unix-like operating systems or even run as standalone. It can be used to automate many kinds of tasks in building, testing, delivering and deploying software. Jenkins' functionality is highly extensible thanks to comprehensive plugin support. [46]

Jenkins is distributable, it can operate on multiple machines and spread the workload (see cloud instances in Figure 10). Jenkins plugins enable extending Jenkins functionality in various ways. For example, it is possible to add a plugin which sends e-mail feedback to developer automatically when build finishes. Jenkins can be extended to support different version control systems by plugins too. If one wishes to manually adjust certain parameters in builds, it is possible with extension too. Many other attributes in monitoring, logging, warning reporting and managing pipeline can be enabled or enhanced with plugins. [46]

Jenkins automation server does not know the existence of CI core application or CI pipeline. Even though Jenkins has support of maintaining the CI pipeline, here it is managed by the CI core. Roughly speaking, Jenkins is used to configure and manage CI jobs to enable the integration of new code automatically and continuously. It polls source code repository and automatically triggers a build and tests changes in code and provides feedback of test results. For example, if a build with recent change fails, the developer must be notified. Even though the build passes, the feedback is required. The CI core is capable of showing which job failed and where, and it points to corresponding Jenkins job. From Jenkins' user interface it is possible to see the job configuration and obtain its logs and build artifacts.

Build artifact is anything created during Jenkins job. They are pieces of data, which can be used for tracking causes of failed jobs. For example, saving launch parameters into a file to be stored as artifact will provide information on whether the job ever received correct parameters. In this environment artifacts are stored in Jenkins servers, in which they are stored under folder named according to metric. Each build/frame has its own folder containing all its artifacts. If a job is executed in cloud instance, the artifacts are transferred to corresponding Jenkins server via Secure Shell (SSH). Stored artifacts are removed periodically, but regardless they provide backward visibility.

The stages seen in CI pipelines (see Figures 11 and 16), are executed in Jenkins as jobs. Mostly one stage has a single Jenkins job. Nevertheless, parallel testing stages (e.g smoke testing and integration testing) have several concurrent jobs. In any testing

stage in microcontroller, signal processing or release CI, one Jenkins job matches one test set, which are described in chapter 4.3.3. In this thesis, test selection could be done by selecting relevant test sets, meaning that number of Jenkins jobs launched in testing stage is reduced.

4.4. Microcontroller and signal processing CIs

These two system components are separate entities, but they operate similar way. As seen in Figure 10, they provide their output to release CI. As an input they use output of previous system components along with source code or script changes in code repository. Even though these components are different in terms of code and scripts, their CI pipeline is similar.

Pipeline of these system components, M and U , is illustrated in Figure 16. Here the

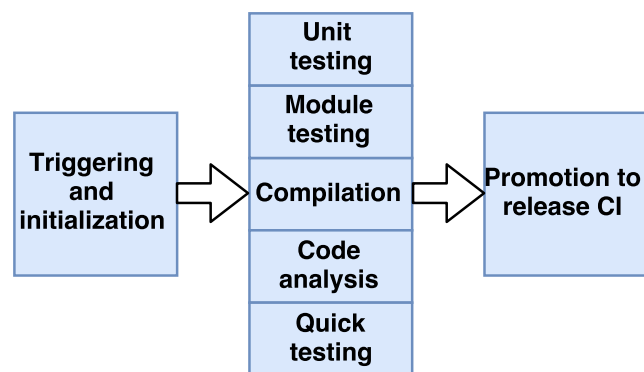


Figure 16. Simplified pipeline of microcontroller and signal processing CIs.

object flowing through the CI pipeline is build object b , see Figure 14. It contains all required attributes and relations to other objects that are necessary to move and track it in the pipeline. The first stage in the pipeline is *triggering and initialization*. Jenkins server continuously polls code repository for changes, and when they are found, a build is triggered. A build object is created in initialization stage. Now the build object can proceed to next stage of multiple parallel CI jobs.

Unit testing and *module testing* jobs are the first level of software testing, see Figure 1 and Chapter 2.1.1. Even though modules and units are often treated as same, there is slight difference between them. While unit may cover only few functions of code, module can be thought as a small number of units grouped together [47]. *Quick testing* contains a collection of critical integration tests conducted on hardware targets. Its purpose is to ensure that the build will have at least working basic functionality in terms of integration before forwarding it to release CI. Thus, quick testing can also be considered as initial part of smoke testing, which truly takes place later in release CI, introduced in Chapter 4.2. For smoke testing in general, see Chapter 2.1.3.

In *compilation* job, the source code, written in C/C++ language, is compiled so that it can be executed in embedded base transceiver station devices. In general, compiler is a tool, which translates program from a programming language to another [48]. The compilation job has essential part in the implementation chapter of this thesis.

After the build has successfully passed these parallel jobs, it is ready to be *promoted to Release CI*, see Figure 10. Promotion occurs if every product in that branch has passed parallel stage. All these builds are then used to create frame objects in release CI, as described in Chapter 4.2.

4.4.1. Test coverage analyzer

This instrumentation-based tool is in use in module tests of *M CI* (see Figure 16). It can analyze coverage of individual test cases on a statement level. In other words, it is able to list covered statements of all files a test case has visited. In this thesis, file level information would be sufficient at the beginning. The output report is obtainable as Extensible Markup Language (XML) file. With these reports it is possible to link code changes to certain test sets, test suites and test cases. Similar dependencies were used as a part of test selection method introduced in Chapter 3.2.

Utilizing this tool would considerably increase the accuracy of test selection and reduce time needed for integration testing stage. However, there are couple of challenges, which are discussed in Chapter 6.3.1.

5. IMPLEMENTATION

The steps of implementing initial selective testing method is presented in this chapter. The implementation incorporates collection of file dependencies in compilation jobs, interaction with VCS, and querying the database. As mentioned in Chapter 4.3.4, all the products have common codebase, and this fact has a central part in the implementation. Approaching the problem on file-level seems to be reasonable choice, since file-level information, at least, is available everywhere. Function or module level approach would increase complexity significantly.

The first goal in the implementation is to only test products that have dependency with changes in code. The next goal is to only execute change dependent test sets under single product, and then extend it to test suite and test case levels if possible in limited time frame of the thesis. Reaching that goal would substantially reduce time consumed in integration testing stage. One remarkable restriction is that there is no direct linkage available between individual test sets and source code.

At the beginning it was decided that when attempting to reach the second goal, a daily run to execute all available tests is needed. That way, detecting potential problems in test selection is possible. When comparing faults found by daily retest-all run and recent run by selective testing, the difference tells about the magnitude of reduction in fault detection capability. In the approaches introduced in Chapter 3, when utilizing test selection, more comprehensive testing is supposed to take place regularly. Running all the tests daily is a logical step not to merely rely on the selective testing method.

5.1. The proposal

The capacity of integration testing stage is not high enough to perform integration testing for all the changes that have been built and tested in earlier stages. This is due to limited number of hardware targets. Investing in more hardware targets would diminish this issue, but now it is not a viable option.

Test selection cannot absolutely reduce the duration of integration test execution, but it can make the testing *smarter* so that the testing is more relevant. In this thesis, the test selection is performed mainly based on changes in source code. In other words, the point is to test only the products which are affected by source code changes, and skip testing of other products. Then the testing is more focused on the changes, which increases the effectiveness of the testing because irrelevant changes are ignored. Other types of dependencies must also be investigated, because changes submitted to VCS do not only contain source code changes.

Steps to create continuous test selection mechanism are illustrated in Figure 17. The basic idea is to find file coverage for each product, and use those to find dependencies with files changed in VCS between the revisions (delta). When there is no dependency found for a frame of the product, testing that frame can be skipped. The steps are explained in detail under this chapter.

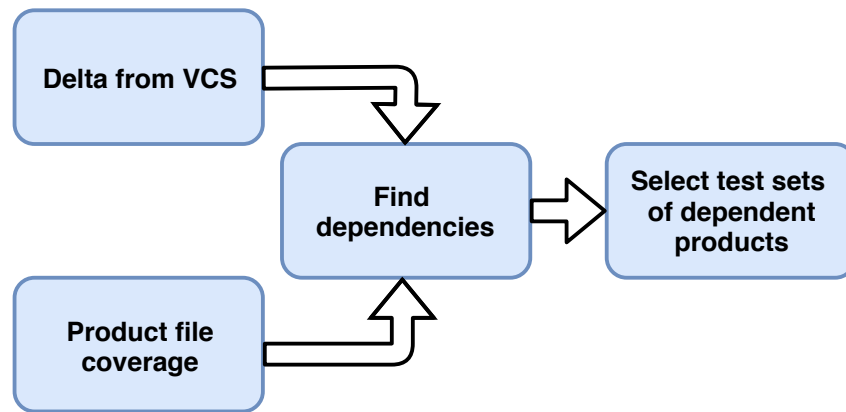


Figure 17. Steps of proposal for continuous test selection.

5.2. Product file coverage

To be able to separate products from each other, one must know source code files contained in each of them. As covered in Chapter 4.4 and illustrated in Figure 16, every build is compiled in the parallel stages of *M* and *U* CIs. Compilers are able to list files and file dependencies created during compilation, and the compilers had already been configured to create dependency files.

Dependency collection function was created to support to all the products in both system components. Dependency file extension is defined in compilers' settings, and there were many types of them. The idea was to read contents of all dependency files and list them in convenient output format. In the environment XML format was used for many other reports and it was selected to be the format of these product file coverage reports too. Call for dependency collection in script of compile job can look as follows:

```
collectDependency *folderName/*.d > dep.xml
```

The principle of the function is to first search all files found with given parameter with `find` command. Then it uses `sed` (stream editor) command to parse file paths in desirable form, and finally it uses `echo` command to make path to be printed in the XML file. The call for the function is made in compilation script in a directory where the source code for the product has been compiled. In summary, that call would search all files with extension `.d` under folder named `folderName`, and save output to file named `dep.xml`. In Linux systems '*' stands for a wild card character.

The compilation dependency report is created every time a compile job is executed. That means that the product's file coverage stays up to date as long as the most recent dependency report is accessed. The dependency collection takes in total few seconds, so the time needed for the operation is negligible. The report file was configured in compilation jobs of the Jenkins servers to be stored in server among the other build artifacts. Saving the artifacts is covered in Chapter 4.3.6.

5.3. Changed files between revisions (delta)

Source code and test codes of products are stored in VCS. It supports delta command, which can be used to display difference between two revisions or path depending on parameters. The output contains all modified, added, and deleted files between the revisions. In addition, the output can be configured to be exported as XML file.

These file differences between two revisions r_n and r_m are called delta or Δf . Generally, for r_n and r_m in certain path, the command to get Δf resembles following:

```
<vcs> <delta command> r_n r_m <repository path> > output.xml
```

The command gets delta for files in given repository path between the two revisions and stores it in XML file named `output.xml`. It is notable that path to repository is different for every branch, and it needs to be automatically switched when required.

A commit is linked to build objects at the very beginning of the pipelines of M and U CIs. The builds receive the revision number of the commit. Due to the common codebase of the products, the commit is also common for all the products. Getting delta between the revisions is simple, as explained, but finding the correct revisions to create the delta is more complex. The delta should be created for revision to be tested and the revision that has previously been tested. More precisely, the delta should be created for revisions of current and previously tested product in same branch and same system component. It should also be defined in a way that it can be automatically created between the right revisions without interaction. Successful solution is presented in Chapter 5.3.2.

5.3.1. Defining revisions for delta: first attempt

The initial idea was to define product dependency for a commit at the time it triggers in M or U CI. The dependency was saved to commit object, and it was available for every build that commit was linked to. An existing database object, baseline, was utilized in this attempt. Baseline is an object which maintains a list of all the source code files for the system component in a branch, and it is able to output changes of latest update. For main branch there are two baseline objects, one for M and other for U . The baseline object is updated every time a trigger occurs in corresponding system component, and therefore it stays up to date. Baseline also saves revision which it had been last time updated to. Trigger is the very first stage in the pipeline, seen in Figure 16.

The idea of this attempt is illustrated in Figure 18. Even though it first appeared to be rational, it was quickly proven not to work. Regardless of commit's product dependency would have been set correctly, the information about the revisions of delta is easily lost. For example, the delta is created to a new revision r_n triggering builds $b_1 \dots b_n$ from the revision r_p to which baseline has been previously updated to. If builds previously triggered with revision r_p fail in any point of the CI pipeline, the commit's dependency set in builds $b_1 \dots b_n$ is incorrect. It points to previous build, which never reached integration testing. That means that product dependency set in commit of builds $b_1 \dots b_n$ is not guaranteed to point to previously tested build, rather to previously triggered build, and thus is incorrect. This problem would be faced in integration testing queue (see Figure 13) at the latest, since not every release matrix is

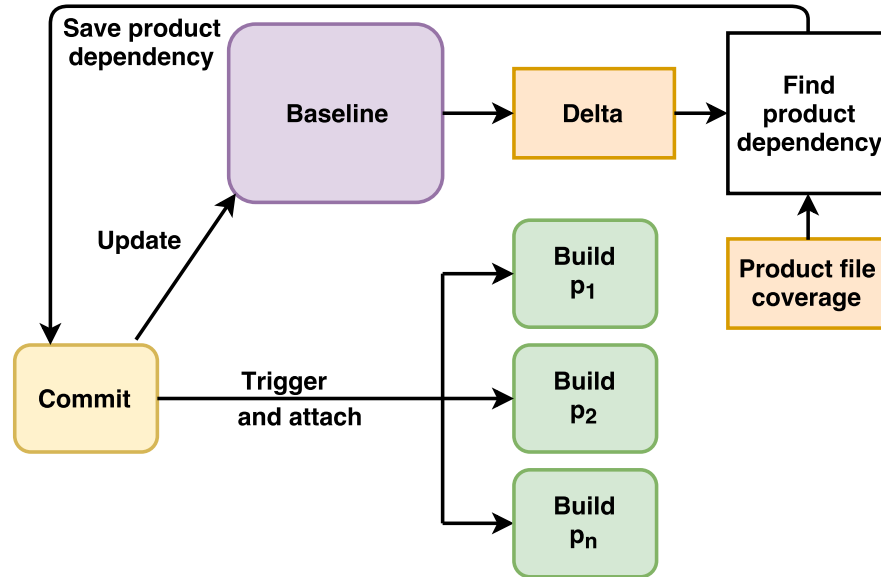


Figure 18. Delta creation and usage in first attempt. This took place in triggering stage of M and U CIs.

selected for testing. Selecting matrixes is not linear, so delta would not be generated for correct revisions.

The product dependency set in commit would be later used in integration testing stage for test selection. Only the test sets of products that had dependency with commit would have been selected. But due to the earlier problem, this attempt would not succeed, and other approach was needed.

5.3.2. Defining revisions for delta: second attempt

Defining the revisions for delta was moved to integration testing stage to avoid problem in the first attempt described in Chapter 5.3.1. Now the revisions for delta were searched after a release matrix was selected for integration testing (see Figure 13). One frame in matrix represents one product. The idea now was to search the previous frame that has reached integration testing, for same product, and get revisions from U and M builds from there. That was done for every frame separately. Delta was now formed between previously tested revision and current revision without being dependent on the order in integration test queue.

Simplified steps of delta creation are shown in Figure 19. That procedure is explained in more detail here. The most recent frame, F_c , for product p_n , is selected to be integration tested among other frames in the matrix. F_c contains U build b_U with revision r_{U_c} , and M build b_M with revision r_{M_c} . Previous frame for same product p_n in same branch that reached integration testing is F_p , which is found using database queries. F_p contains U and M builds as well. The builds have revisions r_{U_p} and r_{M_p} , respectively. Now the delta is created between frames F_c and F_p . To be exact, from revision r_{U_p} to r_{U_c} for b_U , and from r_{M_p} to r_{M_c} for b_M . If it is detected that $r_{U_p} \geq r_{U_c}$, and $r_{M_p} \geq r_{M_c}$, previously tested frame has higher revisions that the frame about to

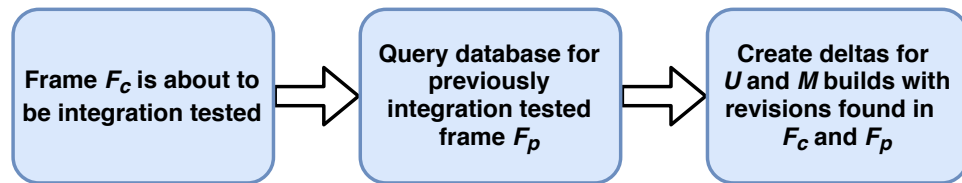


Figure 19. Principle of delta creation in second attempt, taking place in integration testing stage of release CI.

be tested, and the testing can be skipped. Otherwise the deltas are created using the revisions.

Now the delta was created between the frame about to tested and the frame that had previously been tested. Because the delta creation occurred after the release matrix is selected for integration testing, prioritizing or discarding matrixes in integration test queue, see Figure 13, did not have an effect on revision numbers used for delta creation. Hence, the delta was always created correctly between the latest and previously tested revision, and it is continuously obtainable.

5.4. Finding dependencies

Files in delta, obtained in Chapter 5.3, could roughly be divided into three categories:

- source code files
- test code files
- metafiles

All these three categories were handled in a different way. Briefly speaking, the source code files are all the files that are compiled during compilation stage in M or U CI. Test code files, again, are the files that are used for testing purposes in those CIs, or in release CI. Metafiles contain important metadata such as interface versions for products, and they needed to be taken into account too. Thus, in total there are three types of dependencies:

- source code dependencies
- test code dependencies
- metafile dependencies

Dependency search was called for one frame, which is one product, at a time. In the frame, the dependencies were searched separately for both U and M builds (see Figure 12). The revision of U or M build of a frame might have already been tested, since frame created for new M revision does not automatically have new U revision or vice versa. In approaches introduced in Chapters 3.2 and 3.3 the dependencies were searched between modules in source codes and tests. Here the dependencies are searched between changed files and product, which is linked to test sets.

Finding source code dependency was straightforward. Changes in delta were compared with latest corresponding product file coverage reports, which were created in compilation jobs in M and U CIs as described Chapter 5.2. Files in delta XML and product file coverage XML reports were parsed so that they had the same format and can be compared to each other. If matches were found in the files of delta and the files of product coverage report, the build was marked to have source code dependency.

Metafile dependency could exist only if metafiles were changed in delta. If changed metafiles were detected in delta, they needed to be parsed separately. The metafiles in delta had to be downloaded from VCS to access their contents. To get changes in the metafiles between revisions, two versions of them were retrieved using the same revision numbers which were used to create the delta. Now the files could be compared to find differences between them and to parse products that they affect. If product of a build in a frame was found in products affected by metafile changes, the build was marked to have metafile dependency. For example, if either of builds of low level system components, L or C was updated, it was detected in metafiles and resulted as dependent.

The third and last type of dependency is test code dependency. A frame could have test code dependency if test code files were found in delta. It was detected for both U and M build separately. Changes in test codes should always be tested even though they are not actual source code. They can help to catch new faults, because updated test codes can find faults which previously were undetected.

For U 's code repository, test codes have been mapped in a way that it is known which source code module is affected of certain test code change. When test code change was found in delta, the name of corresponding affected module was known, because it is included in the path of the changed file. Next it was searched that if the module existed in product's file coverage report that previously was collected in a way described in Chapter 5.2. If the module was found in the report, build was marked to have test code dependency. The modules are not product dependent, so multiple products can have same module. For M 's code repository, there was not similar mapping of test codes and modules. Therefore, for now, if any integration stage test code changes were detected for M using keywords, the build was marked to have test code dependency without knowing the product it really affected.

Final dependency of a frame was defined using all the previous three types of dependencies of both builds. It is shown in Table 2, how different permutations of code

Table 2. Build's dependency determined by code dependencies.

Source code dependency	Metafile dependency	Test code dependency	Build's dependency
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

dependencies result in dependency of a single build of U or M . Only if no dependencies were found, the build was marked to have no dependency. The final dependency was determined by using logical OR operator for build dependencies, seen in Table 3. It was used to decide when testing a frame is required. If final dependency was 1, test sets of the frame had to be executed. Otherwise, the testing stage could be skipped.

Table 3. Final dependency of frame determined by build dependencies.

U build's dependency	M build's dependency	Final dependency
0	0	0
0	1	1
1	0	1
1	1	1

5.5. Putting the steps together

Most of the steps were combined to create one script located in a server of the CI core. The number of database operations is high, so it was reasonable to have the script running close to the database. The script was called from Release CI server for each frame when a release matrix is selected to be tested (see Figure 13). The collection of product file coverage was not included in this script since it must be done during compilation in M and U CIs. Output reports from it, however, were continuously available.

The script was written in Python language, and therefore database queries were easy to make since it is used in Django thoroughly. The script was implemented to have pessimistic approach, so that if anything went wrong during the execution of the script, the return value was set to 1 (dependent) or remained empty. The call of the script was used to see if the frame had dependency with the changes in code repository or not. As input parameters the script needed unique identifier of the frame being tested and name of the branch, which both were already available in release CI's script the call was made from. As output, the script returned value 0 or 1 to indicate if the frame was dependent on the code changes. If output value of the script was 0, the frame was not dependent and the testing could be skipped, else the frame needed to be tested. Thus, the script is named as *frame dependency script*.

Simplified control flow diagram of the script is presented in Figure 20. The execution of the script started when a release matrix was selected for integration testing, seen in Figure 13, before activating Robot Framework. The script was called for a frame once, and the flow was repeated for both M and U builds in the frame to get final dependency. With this implementation, the first goal of excluding testing of change independent products was reached.

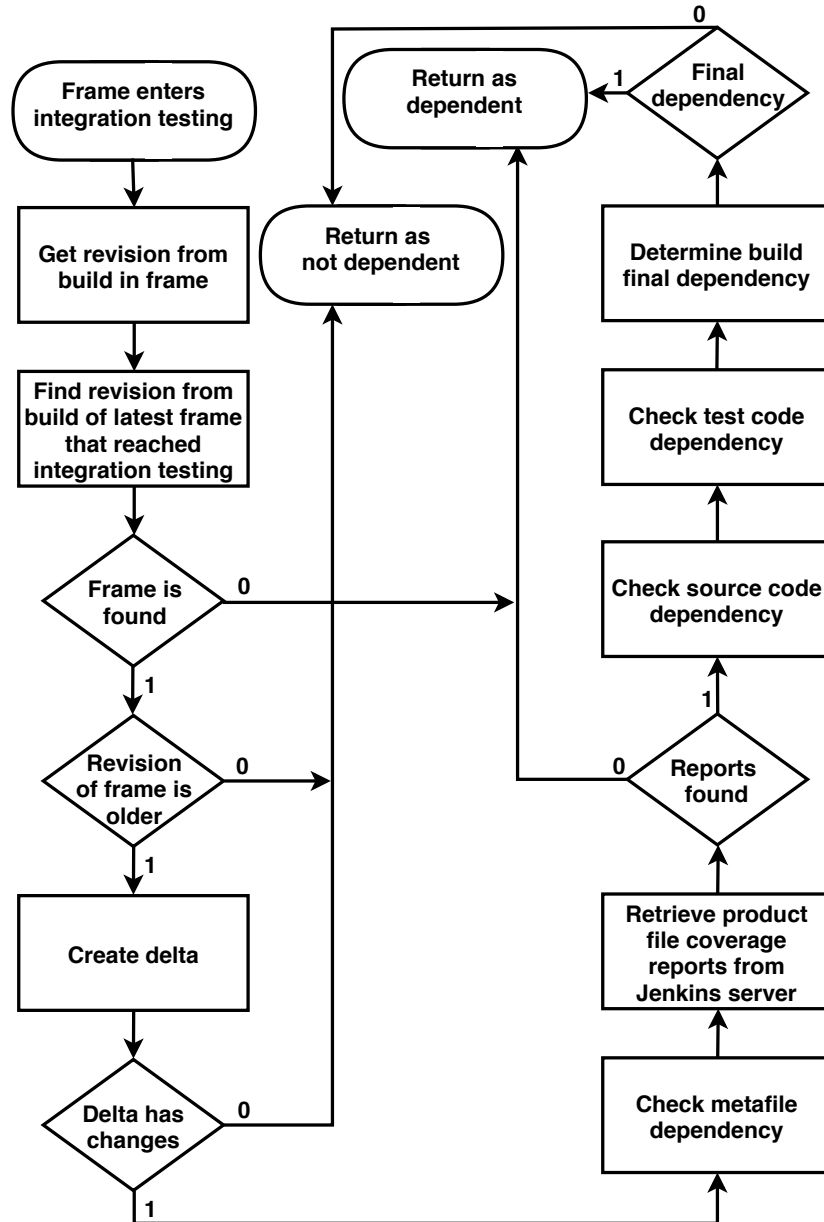


Figure 20. Flowchart of frame dependency script.

6. RESULTS AND VALIDATION

The test selection method was taken in use in the CI environment. At first, it was utilized only in main branch. That was because it had the highest activity, and improvements provided by test selection method were the most beneficial there. Moreover, collecting enough data was possible in the branch within limited time frame of the thesis.

An interesting quantity here was the time it took for a frame to finish integration testing from the moment it was pre-released (see Figure 11). In other words, the time spent from the moment a frame entered integration testing queue until all test sets were completed for it (see Figure 13). If testing a frame was skipped, it certainly reduced the time the frame spent in testing stage, but also probably shortened queuing time for frames of next release matrix in integration testing queue.

Two sets of data were collected: one from time before utilizing test selection and the other after it was taken in use. Now there were two independent sets of data to be compared. Statistical tests were used to evaluate statistical significance of the results. For all calculations and visualization, RStudio was used [49]. *Independent two-sample t-test with unequal variances* and *Mann-Whitney U test* were used to determine the statistical significance of the results. Because all the tests were not always executed anymore, it has to be monitored if fault detection capability of the integration tests remained on the same level as before. This was done by examining the number faults reported from integration testing stage during both of these periods.

6.1. Data collection

The data was collected from the database by querying frame objects and their metrics. Only frames of main branch were included now. The total time for a frame spent in integration testing stage was found using timestamps saved in metric objects in Unix epoch format.

Let timestamp t_q be the time when the frame entered integration testing queue, and t_f be timestamp of last metric that finished integration testing. The time for a frame spent in integration testing stage was calculated simply by subtracting t_q from t_f . Because of epoch format, the subtraction results directly in seconds spent in integration testing stage. As mentioned earlier, one test set matches one Jenkins job and each of those are marked as metric for a frame.

The first data set, D_1 , was collected from frames created within the time range of six weeks before toggling selective testing on. The data is illustrated in Figure 21, where one black circle symbol represents the total time spent in integration testing for a single frame. The collection was repeated to get another data set after selective testing was toggled on. This data set is named D_2 and it was collected during three weeks, and it is also seen in Figure 21, marked with red plus symbol. The total time spent in integration testing stage (queue and testing) is seen in y-axis as normalized. Every value related to duration is always **normalized by dividing with maximum value** found in data sets, and as a result the highest value is 1.0 and others below that. Hence, they do not represent actual durations.

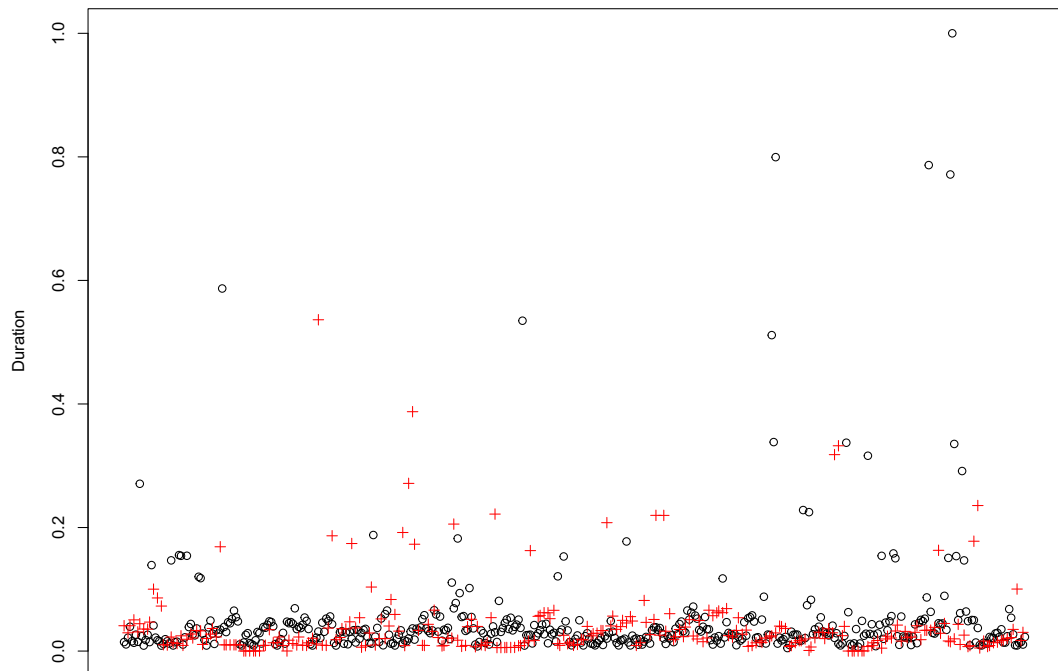


Figure 21. Data sets D_1 and D_2 marked with black and red colors, respectively. D_1 was collected before and D_2 after utilizing the test selection method.

6.1.1. Threats to validity

Obviously, there were outliers in the collected data, since the highest values are known to be too high to be valid durations in this branch. As seen in Figure 21, there were several of those values that deviate significantly from “normal”. The duration of testing itself in general, is known accurately, and even summed up with queuing time it should stay much lower than what the magnitude of the highest values represent. Queuing time can vary a lot depending on the level of activity in the system, and it has no strict maximum. For example, a single data point that is hundreds of times higher than expected testing duration, cannot be valid under normal circumstances. They, however, can be explained by problems faced in testing. If some of the test sets were manually re-executed for a frame due to suspected environmental failure, it can distort the duration collected from metrics. These outliers needed to be handled not to disperse results because they do not represent the duration of actual testing.

Even though both data sets are collected in the same branch, the period of time is different. Throughput speed of the CI system may vary under different periods because of multiple factors. For example, there may have been more frequent commits in a certain time period than in another, or there could have been problems with some hardware targets during a certain period. The effects of all such environmental factors are difficult to determine, even though they could have an effect on the results. The

reason for utilizing test selection only in one branch is that it was easier to track possible misbehavior of it. In a single branch the causes and effects of selective testing were controllable.

6.2. Results

Data sets D_1 and D_2 are visualized as a box plot in Figure 22. It is clearly seen from the data that their distribution is not normal. It was expected, because sample points must have positive value, but they did not have any upper limit. Due to the outliers in data, the median and mean deviated from each other significantly. The mean calculated for D_1 was almost double of its median. Same was discovered for D_2 . In Figure 22, a slight reduction in the median can be observed, but the box plots are not very descriptive due to outliers with high values.

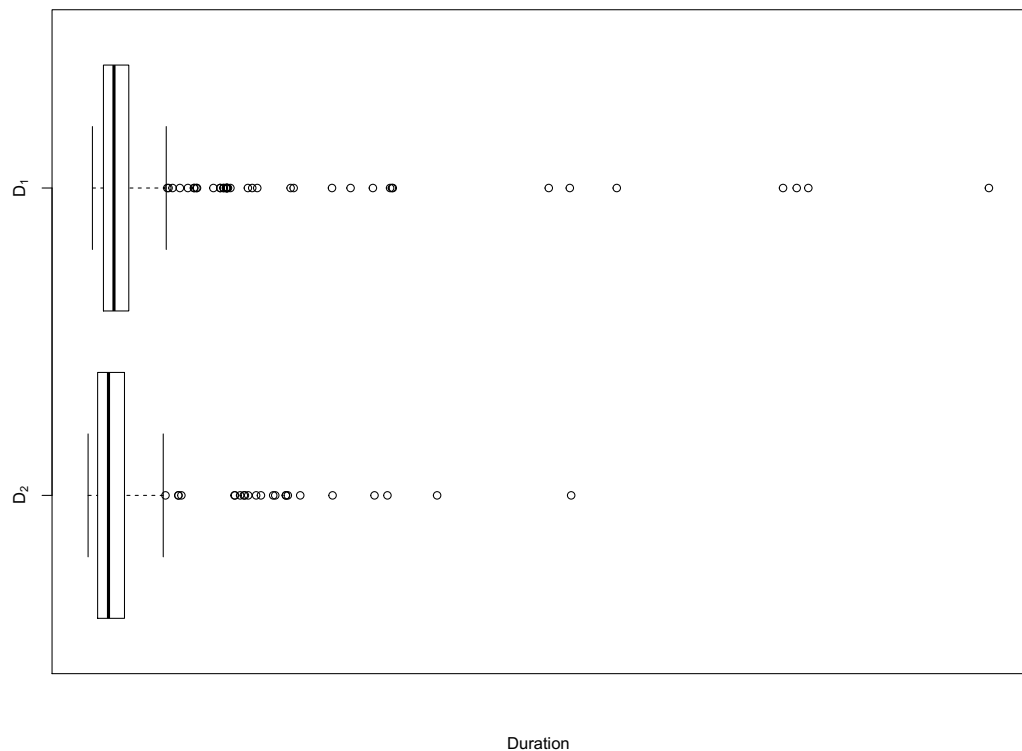


Figure 22. Box plots of data sets D_1 and D_2 .

Since the median was significantly lower for both data sets than the mean, it suggested that the distribution of the data is right skewed (positive skew) with long right tail and short left tail. Normal Q-Q (quantile-quantile) plots illustrated in Figure 23 supported this assumption. It was also interpreted that data is distributed rather equally in the data sets, but the largest differences between them were located at the end of the right tail.

R implementation of Hartigan's dip test for unimodality, `dip.test`, strongly suggested that the distribution of both of the data sets was unimodal, resulting in $p \geq 0.5$. That strengthened the assumption of right skewed distribution of the data. At this point, however, histogram of the data would not be descriptive due to the outliers which are discussed in Chapter 6.1.1.

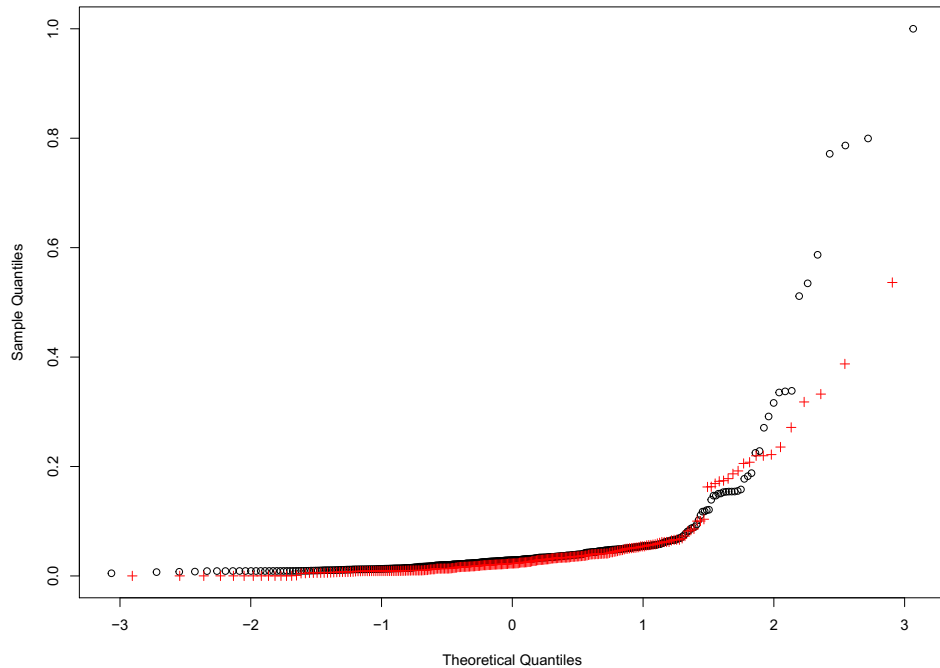


Figure 23. Normal Q-Q plot of D_1 and D_2 .

6.2.1. Detecting outliers

Rather frequent and high value outliers indicated that mean based outlier detection would not be a rational solution. For median, moderate number of occurrences of outliers have much less impact than for mean [50, 51]. The high value outliers will increase the mean while for the median their effect is irrelevant as long they do not occur too frequently. Here the mean and the median deviated much from each other because of the outliers with high values.

Common approach for outlier detection is to count values outside mean \pm standard deviation (SD) multiplied by 3 as outliers, see Equation 1. However, this approach has multiple limitations. It assumes that data is normally distributed even with outliers. They can have a heavy impact on the SD and the mean. Lastly, the outlier detection does not work properly when sample is small. [50]

Median absolute deviation (MAD) is claimed to be a more robust indicator of central tendency, which means that it is more resilient to the presence of outliers than SD. It has breakdown point of 50%, which means it can handle up to 50% of observations

being bad and still giving correct results [52]. On the contrary, the breakdown point of SD is 0% meaning that even a single bad observation has an impact on it. MAD is calculated by taking the median of all values' deviation from median. It can be used the same way as median than SD is used with mean. For example, $\text{median} \pm 3 \cdot \text{MAD}$, seen in Equation 2, can be used as a rule for outlier detection [50, 51]. Generally, the multiplier of SD or MAD is either 2, 2.5, or 3.

Interquartile range (IQR) can be used for outlier detection. It is defined as middle 50% of data without symmetry assumption. Hence, it is defined as $IQR = Q_3 - Q_1$, where Q_3 is the third and Q_1 the first quartile of data set. Values exceeding $\min(\max(x), Q_3 + 1,5 \cdot IQR)$ or falling below $\max(\min(x), Q_1 - 1,5 \cdot IQR)$, where the x is the sample, are treated as outliers. In this distribution, only the upper threshold was relevant, more closely $Q_3 + 1,5 \cdot IQR$ since the maximum value is always greater than that. IQR's breakdown point is 25%, and it should work well with asymmetric distributions, because it does not assume symmetry [52].

Table 4. Mathematical measures of the data sets.

	D_1	D_2
Mean	0.0501	0.0410
Median	0.0288	0.0227
SD	0.0961	0.0726
MAD	0.0138	0.0130
Q_3	0.0452	0.0405
IQR	0.0281	0.0298
T_e	0.3386	0.2289
T_d	0.0707	0.0627
T_q	0.0873	0.0852

SD, MAD, IQR, and Q_3 were calculated using commands `sd`, `mad`, `IQR`, and `quantile`. MAD was calculated without assuming normality of the data and setting constant to 1, whereas in RStudio it is set to 1.4826 to assume normality by default. With these measures thresholds for outlier detection were calculated. Threshold T_e is based on mean and SD, whereas T_d is based on median and MAD. T_q is calculated with the help of IQR and the third quartile. Their formulas are presented in Equations 1, 2, and 3. Only positive thresholds were calculated because the sample values cannot fall below 0. All these measures are presented in Table 4.

$$T_e = \text{mean}(D_i) + 3 \cdot \text{SD}(D_i) \quad (1)$$

$$T_d = \text{median}(D_i) + 3 \cdot \text{MAD}(D_i) \quad (2)$$

$$T_q = Q_3(D_i) + 1.5 \cdot \text{IQR}(D_i) \quad (3)$$

The looser thresholds of D_1 are illustrated with data in Figure 24. T_d and T_q are relatively close to each other, and T_e is located much higher than the previous two. The impact of outliers raised T_e , while more robust thresholds T_d and T_q stay on a

reasonable level. T_d was slightly stricter for both data sets than T_q , as seen in Table 4. After visual approximation T_q was selected to be the most rational threshold since it matched the estimated possible maximum total duration best. T_q was also almost equal for both data sets, and the less strict threshold of D_1 was selected to be used.

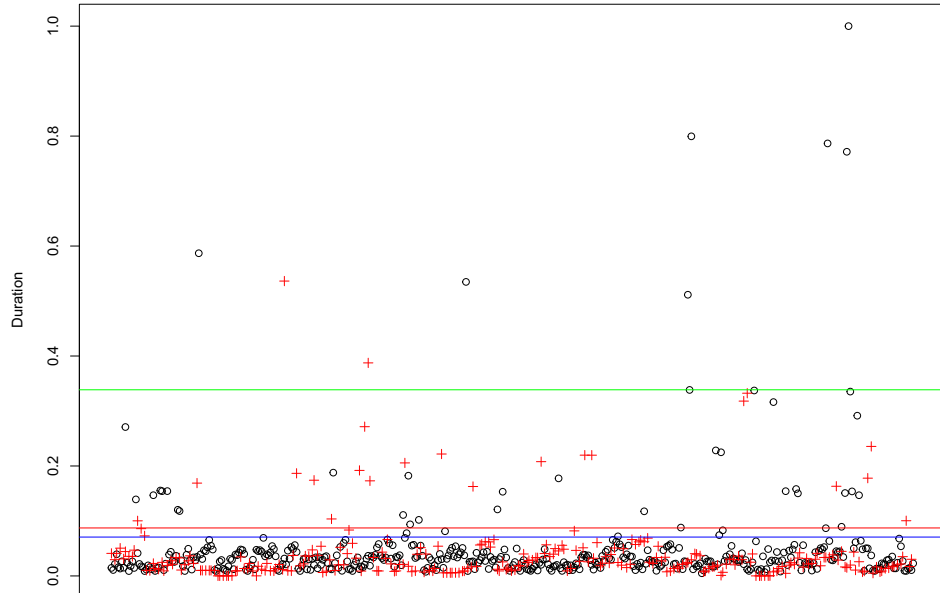


Figure 24. Outlier detection thresholds plotted in D_1 and D_2 . Green colored line denotes T_e , red T_q , and blue T_d , found in D_1 column in Table 4.

The outliers were detected using T_q . Observations exceeding that threshold were removed, resulting in total 8.4% of data points being discarded from both of the data sets. These cleansed data sets are now called D'_1 and D'_2 . In Figure 25 box plot of the data is presented after outlier removal. Now histograms of data were also descriptive, and they are seen in Figure 26.

Summary of mean and median of the data sets are seen in Table 5. After outlier removal it was notable that mean and median are much closer to each other than previously with outliers. That is, because some data were removed from the right tail of distribution, which led to the distribution of the data to be more normal. In Figure 25 it is seen that IQR of D'_2 , which covers middle 50% of the data, has been shifted towards zero when compared with IQR of D'_1 . Same was observed for median, which was also seen from the numbers in Table 5.

Table 5. Summary of reduction in mean and median.

	D_1	D_2	difference:	D'_1	D'_2	difference:
Mean	0.05011	0.04097	18.2%	0.02943	0.02497	15.2%
Median	0.02881	0.02273	21.1%	0.02715	0.02112	22.2%

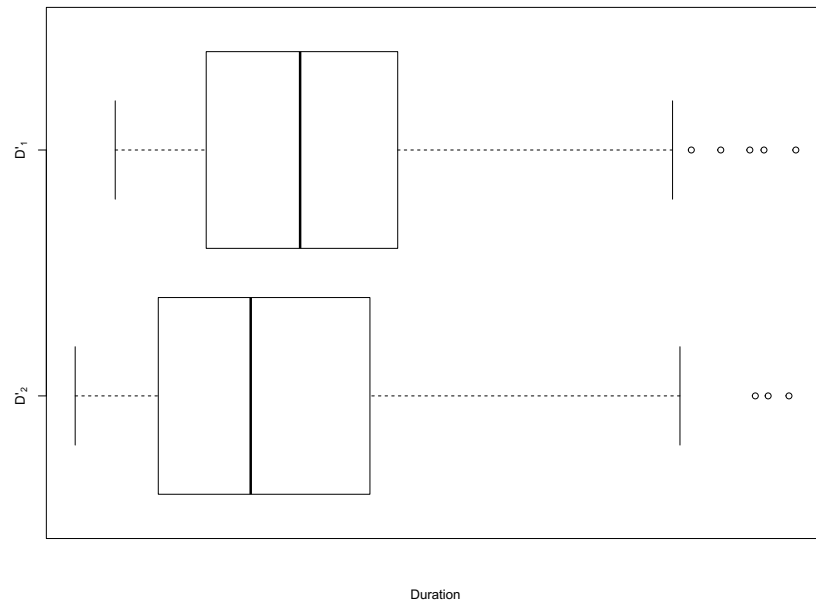


Figure 25. Box plots of data sets D'_1 and D'_2 .

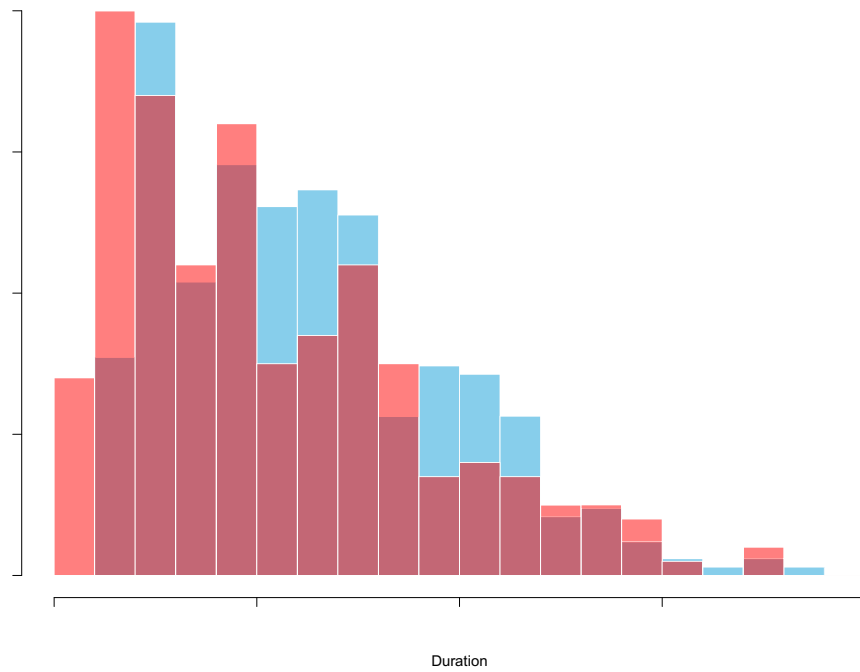


Figure 26. Histograms of D'_1 (blue) and D'_2 (red). Dark red is the color of intersection of the two sets.

6.2.2. Tests of statistical significance

Outlier removal always has the risk of removing valid data too. Statistical tests are nowadays easy to conduct, and they were done before and after removing the outliers. *Independent two-sample t-test with unequal variances*, also called *Welch's t-test*, and *Mann-Whitney U test* were used to assess statistical significance. It is claimed that there is no real reason to use equal variance t-test over Welch's t-test, since the latter can well handle samples with equal variances. The main difference between the U test and t-test is that latter assumes normality of the data while U test is nonparametric test and does not make that assumption. However, it is claimed that t-test is valid for even non-normal distributions when sample size is sufficiently large [53].

Histograms of the data sets are illustrated in Figure 26. It was discovered that the distribution of data sets is almost similar, but there is slight location shift towards zero in D'_2 compared with D'_1 . The distribution of the data was not normal. Even though t-test can handle violation to normality assumption with large sample sizes, the data with outliers is not expected to be optimal for it due to severe violation to normality assumption and high standard deviation.

The U test compares medians of similarly shaped distributions, and t-test makes comparison between means. In RStudio t-test can be conducted by command `t.test`, and given with parameter `var.equal = FALSE`, the unequal variances t-test, Welch's t-test, is performed. Command `wilcox.test` given with two samples as parameter, performs the Mann-Whitney U test. Because now it was tested that if the median or the mean is lower compared with the previous data set, even one-tailed tests should be sufficient. Nevertheless, two-tailed tests were used for reliability.

Now the hypotheses for Welch's t-test were:

$$\begin{aligned} H_0: & \text{Means of both populations are equal} \\ H_1: & \text{Means of both populations are not equal} \end{aligned}$$

and for U test:

$$\begin{aligned} H_0: & \text{Medians of both populations are equal} \\ H_1: & \text{Medians of both populations are not equal} \end{aligned}$$

where H_0 is the null hypothesis and H_1 is the alternative hypothesis. Significance level α was defined to be 0.05. Results of statistical tests are reported in Tables 6 and 7.

Table 6. Values of Welch's t-test.

	Data samples	
	D_1, D_2	D'_1, D'_2
t-value	1.5564	3.2863
Degrees of freedom	724.73	484.39
p-value	0.12	0.001089

Table 7. Values of Mann-Whitney U test.

	Data samples	
	D_1, D_2	D'_1, D'_2
U value	72698	62532
p-value	0.0003503	0.00004498

6.2.3. Analyzing the results

When analyzing Tables 5, 6, and 7, it was seen that with the original data sets, D_1 and D_2 , mean testing duration dropped by approximately 18.2% after utilizing the test selection. Nonetheless, Welch's t-test resulted in $p = 0.12$. Now $p > \alpha$, which means that the null hypothesis cannot be rejected and therefore the change in mean is not statistically significant. Median, on the other hand, dropped by 21.1% and its significance was tested using Mann-Whitney U test. It resulted $p = 0.0003503$, and now $p < \alpha$, which indicated that the null hypothesis can be rejected and the difference in medians is statistically significant.

Because of the outliers in data, the same tests were conducted on data sets D'_1 and D'_2 from which the outliers had been excluded. Again from Tables 5, 6, and 7, it was obtained that now mean was approximately reduced by 15.2% and median by 22.2% after utilizing the test selection. Both statistical tests resulted in $p < \alpha$, and hence, the changes are statistically significant. Ultimately these results should be more relevant giving a better picture of the impact of the test selection, since clearly the original data sets included some bad observations. The reduction in mean was smaller between cleansed data sets than with the original data sets. The drop in median remained almost the same because it is more resilient to outliers.

6.3. Discussion

Selective testing is not a new test optimization technique, it has been used for decades, but utilizing it in this CI environment is more recent and less researched topic. However, the basic idea of finding dependency between tests and code modules has been used by many approaches also in CI. The general challenge in CI environments is that the time between consecutive test executions is short, setting strict time constraints [35, 36]. In this thesis an initial code change based test selection method was implemented in CI environment to cover integration tests. The implementation overall is not very generalizable, having features specific to this environment.

The main challenge here was that code coverage information of test sets, test suites, or test cases was not directly available at the time of writing. Moreover, different products have common code repository, in which they have a lot of common code. A partial solution was found for these problems. Test sets are product-specific, so if products affected by code changes were found, the test sets linked to them could be selected. Now dependencies had to be determined between the products and code changes, which required knowing the file coverage of a product. Compilers are able to list source code files they compile as so-called dependency files, and those were combined and used to form product file coverages. Other types of files, such as test

codes and metafiles, are not source code and they needed to be handled differently. In general, file-level approach was reasonable choice since information of changed files is easily obtainable from VCS. Compilers also list dependencies on file-level. Thus, this approach minimizes complexity. Test results and execution history were not utilized in the implementation, because relying merely on them would be risky without test coverage information.

The first goal of finding dependencies between code changes and products was accomplished. As a result, slight but statistically significant reduction in integration testing duration was achieved (see Table 5). There are many environmental factors, which have an effect on the duration of the testing, such as frequency of commits and potential problems with hardware targets. It can be observed from collected data though, that during collection of the first data set there was relatively bit less tested frames compared with the three weeks period of other data set. However, some stability improvements were made for CI system and target testing connection protocols during these months. They indicate that true improvement is likely bit lower than what is seen from the numbers.

One major restriction is that the low-level components, L and C , are updated relatively often. That requires executing all the tests, and test selection cannot take place. There are also inaccuracies in detecting some dependencies. For some of the test code changes the detection is rather vague with pessimistic attitude. That is, if certain test codes change, every product must be tested. In the future these issues should be improved, as discussed in Chapter 6.3.1.

Accomplished slight time reduction is a moderate start, but overall the numbers are not completely satisfying. The testing time needs to be reduced more to be able to test every change. Even though the performance of this implementation could be higher, it has several advantages. First, it is *automatic* and *continuous* from the viewpoint of CI. It should, in theory, work without any human interaction. However, the method is still in an early stage of development, and it intrinsically has the possibility of having bugs and those should be monitored. Secondly, it is lightweight. The duration of the call of the frame dependency script, introduced in Chapter 5.5, is in magnitude of few seconds per frame. Time consumed by the other part of the implementation, the product file coverage report generation, see Chapter 5.2, is also negligible.

Thirdly, the test selection method is developer independent. No actions are needed from developer because changes are found in VCS. And lastly, fault detection capability of integration testing was not detected to deteriorate during the short period due to the cautious nature of the implementation. For example, ignoring test code changes would certainly reduce the number of test executions, but it would raise the risk of letting faults pass the testing undetected. The number of faults reported by integration testing followed the same trend as in time before selective testing. In the future, this must be monitored with concern, as discussed in Chapter 6.3.1.

6.3.1. Future work

As mentioned at the beginning, the implemented test selection method is an initial solution addressing the problem of time consumed in integration testing. More steps forward should be taken to make the test selection method more effective since the

current implementation is rather rough. Even though the first goal of testing change-dependent products was accomplished, the second goal of expanding test selection to deeper level was not fulfilled.

Test code analyzer (see Chapter 4.4.1) could be used in the future to improve the test selection method. At time of writing, it does not support the integration tests in the environment, but it could be used to support the implementation later. Expanding the test selection to test set, test suite, and test case levels requires utilizing this tool. Having up-to-date test coverage information would lead to achieving significant reduction of testing duration within a frame, since the changes can now be linked to tests. In optimal situation, only small subset of available tests would be executed for a small change. When the number of executed test cases is small, the probability of having unstable test cases is certainly smaller, and this reduces delay caused by automatic re-execution of these unstable test cases in targets. Furthermore, the dependency of changes in test codes can be addressed with greater accuracy because with the analyzer the coverage of each test is known.

Getting the accuracy of test selection to this point, however, has its challenges. Discovering all so-called transitive dependencies may be difficult and that requires continuous analysis of the code changes and test results. There is a risk of increase in undetected faults if these dependencies are missed. A daily retest-all run is unquestionably needed to monitor the accuracy of the test selection. In long-term analysis the test results should reveal dependencies which are not directly observable. An index of dependency data could be created and kept up to date in the database to facilitate access to the data.

One considerable step forward would be finding the effects of changes of low level system components (L and C) in integration testing and select tests accordingly. This would resolve the need of costly execution of all the tests when low level system component change occurs, and further reduce the testing duration. Finding the dependencies may be complex, thus the tests should, at least, be prioritized to detect potential problems caused by low level system component changes as early as possible.

The test selection would be beneficial also in other CI pipelines' testing stages: unit testing, module testing, quick testing, and smoke testing. Even though the run time of unit testing and module testing stages is short, they provide the first feedback for developers, and shorter execution time is advantageous. Quick testing is important because there hardware targets are used for the first time for testing a build, providing important feedback. Again, less test cases would need less targets, speeding up the testing. Likewise in smoke testing, benefit would be reduced testing time. If the number of executed cases is high in any stage, test prioritization technique should be considered.

A sophisticated approach would be to utilize machine learning techniques to make the test selection understand surrounding environment and its changes to minimize human intervention needed. All in all, any improvements in the accuracy to detect dependencies would result in a better performance of the test selection method, and that would reduce the time needed for the testing. That is, because now the method is overcautious due to the lack of information regarding e.g. test code dependencies.

7. SUMMARY

In this thesis, an initial test selection method was implemented to reduce testing duration of integration testing stage of an agile CI environment. The software developed in the environment is hardware-dependent platform software so even after utilizing selective testing, a good level of fault detection capability should be maintained. Any faults remaining undetected in testing can be critical. There was not former test selection solution in use, and existing approaches in literature were analyzed. As a result, a test selection method was implemented and taken in use a in single software development branch to test its effectiveness and accuracy.

The implementation is based on changes submitted to the code. A list of changed files between revisions was obtained directly from VCS. Code repository in the VCS is common for all products, and they also have a lot of common code. For each change submitted to the codebase, dependent products were searched by utilizing lists of source code files created by compiler. In addition, test code files and metafiles needed to be handled to have comprehensive test selection method. Now only test sets of dependent products needed to be executed. The operation of created test selection method is automated and continuous with insignificant performance penalty.

Results indicated slight but statistically significant reduction in total time consumed in integration testing stage. Data sets were collected before and after utilizing test selection method. The data has some bad observations and those were removed. Mean of the duration of integration testing reduced by 15.2% while median decreased by 22.2% for cleansed data sets. Statistical significance was assessed with Welch's t-test and Mann-Whitney U test, with significance level $\alpha = 0.05$. The results were found to be statistically significant. Fault detection capability of the integration tests was discovered to stay on the same level during the time frame of this thesis. Nevertheless, it is notable that certain environmental factors could have an effect on the results in different periods of time even though there were not any major changes in the environment.

8. REFERENCES

- [1] Fowler M. & Foemmel M. (2006) Continuous integration. Thought-Works 122.
- [2] Duvall P.M., Matyas S. & Glover A. (2007) Continuous integration: improving software quality and reducing risk. Pearson Education, chapt. 1-2, 4. ISBN: 9780321336385.
- [3] Yoo S. & Harman M. (2007) Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification and Reliability* 22(2), 60 pages.
- [4] Olsson H.H., Alahyari H. & Bosch J. (2012) Climbing the "stairway to heaven"—a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: *Software Engineering and Advanced Applications (SEAA)*, 2012 38th EUROMICRO Conference on, IEEE, pp. 392–399.
- [5] Shahin M., Babar M.A. & Zhu L. (2017) Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access* 5, pp. 3909–3943.
- [6] Myers G.J., Sandler C. & Badgett T. (2011) *The art of software testing*. John Wiley & Sons, 2nd ed., 256 pages. ISBN: 0471469122.
- [7] Lewis W.E. (2016) *Software testing and continuous quality improvement*. CRC press, 3rd ed., 688 pages. ISBN: 9781439834367.
- [8] Pearson L. (2015), The four levels of software testing. URL: <https://www.seguetech.com/the-four-levels-of-software-testing/> Accessed: 10.1.2018.
- [9] Hailpern B. & Santhanam P. (2002) Software debugging, testing, and verification. *IBM Systems Journal* 41, pp. 4–12.
- [10] Murnane T. & Reed K. (2001) On the effectiveness of mutation analysis as a black box testing technique. In: *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, IEEE, pp. 12–20.
- [11] Rongala A. (2015), What is white box software testing: Advantages and disadvantages. URL: <https://www.invensis.net/blog/it/white-box-software-testing-advantages-disadvantages/> Accessed: 11.1.2018.
- [12] Khan M.E. & Khan F. (2012) A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Sciences and Applications* 3, pp. 12–15.
- [13] Talby D., Keren A., Hazzan O. & Dubinsky Y. (2006) Agile software testing in a large-scale project. *IEEE software* 23, pp. 30–37.

- [14] Duran J.W. & Ntafos S.C. (1984) An evaluation of random testing. *IEEE transactions on Software Engineering* , pp. 438–444.
- [15] Harrold M.J., Gupta R. & Soffa M.L. (1993) A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, pp. 270–285.
- [16] Kaner C., Bach J. & Pettichord B. (2001) *Lessons learned in software testing: A Context-Driven Approach*. John Wiley & Sons, chapt. 8. ISBN:9780471081128.
- [17] Boehm B. (2002) Get ready for agile methods, with care. *Computer* 35, pp. 64–69.
- [18] Abrahamsson P., Salo O., Ronkainen J. & Warsta J. (2002) Agile software development methods: Review and analysis. *VTT publication* 478, pp. 20, 29–36.
- [19] Beck K., Beedle M., van Bennekum A., Cockburn A., Cunningham W., Fowler M., Grenning J., Highsmith J., Hunt A., Jeffries R., Kern J., Marick B., Martin R.C., Mellor S., Shwaber K., Sutherland J. & Thomas D. (2001), *Manifesto for Agile Software Development*. URL: <http://agilemanifesto.org/> Accessed: 5.1.2018.
- [20] Parnas D.L. & Clements P.C. (1986) A rational design process: How and why to fake it. *IEEE transactions on software engineering* 12, pp. 251–257.
- [21] Nerur S., Mahapatra R. & Mangalaraj G. (2005) Challenges of migrating to agile methodologies. *Communications of the ACM* 48, pp. 72–78.
- [22] Knauss E., Staron M., Meding W., Söder O., Nilsson A. & Castell M. (2015) Supporting continuous integration by code-churn based test selection. In: *Proceedings of the Second International Workshop on Rapid Continuous Software Engineering*, IEEE Press, pp. 19–25.
- [23] Schwaber K. & Sutherland J. (2017), *The Scrum Guide™*. URL: <http://www.scrumguides.org/scrum-guide.html> Accessed: 26.1.2018.
- [24] Prince S. (2016), *The product managers’ guide to continuous delivery and devops*. URL: <https://www.mindtheproduct.com/2016/02/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/> Accessed: 15.1.2018.
- [25] Fitzgerald B. & Stol K.J. (2014) Continuous software engineering and beyond: trends and challenges. In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, ACM, pp. 1–9.
- [26] Laukkanen E., Ikonen J. & Lassenius C. (2017) Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Information and Software Technology* 82, pp. 55–79.
- [27] Azeri I. (2017), *What is CI/CD?* URL: <https://www.mabl.com/blog/what-is-cicd> Accessed: 5.1.2018.

- [28] Humble J. & Farley D. (2010) *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 463 pages. ISBN: 0321601912.
- [29] Laster B. (2017) *Continuous Integration vs. Continuous Delivery vs. Continuous Deployment*. O'Reilly Media, Inc. ISBN: 9781492028901.
- [30] Wolff E. (2017) *A Practical Guide to Continuous Delivery*. Addison-Wesley Professional, chapt. 1, 7, 12. ISBN: 9780134691626.
- [31] Chen L. (2015) Continuous delivery: Huge benefits, but challenges too. *IEEE Software* 32, pp. 50–54.
- [32] Claps G.G., Svensson R.B. & Aurum A. (2015) On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software technology* 57, pp. 21–31.
- [33] Leppänen M., Mäkinen S., Pagels M., Eloranta V.P., Itkonen J., Mäntylä M.V. & Männistö T. (2015) The highways and country roads to continuous deployment. *IEEE Software* 32, pp. 64–72.
- [34] Yoo S., Nilsson R. & Harman M. (2011) Faster fault finding at google using multi objective regression test optimisation. In: *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*, Szeged, Hungary.
- [35] Elbaum S., Rothermel G. & Penix J. (2014) Techniques for improving regression testing in continuous integration development environments. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, pp. 235–245.
- [36] Kim J.M. & Porter A. (2002) A history-based test prioritization technique for regression testing in resource constrained environments. In: *Proceedings of the 24th international conference on software engineering*, ACM, pp. 119–129.
- [37] Pan J. & Center L.T. (1995) Procedures for reducing the size of coverage-based test sets. In: *Proceedings of International Conference on Testing Computer Software*, pp. 111–123.
- [38] Rothermel G. & Harrold M.J. (1997) A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6, pp. 173–210.
- [39] Rothermel G., Untch R.H., Chu C. & Harrold M.J. (2001) Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, pp. 929–948.
- [40] Rothermel G., Harrold M.J., Von Ronne J. & Hong C. (2002) Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability* 12, pp. 219–249.

- [41] Kim M., Zimmermann T. & Nagappan N. (2014) An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering* 40, pp. 633–649.
- [42] Robot Framework Foundation (2018), Robot framework: Generic test automation framework for acceptance testing and ATDD. URL: <http://robotframework.org/> Accessed: 8.3.2018.
- [43] Python documentation. URL: <https://www.python.org/doc/> Accessed: 8.3.2018.
- [44] Django software Foundation, Django documentation. URL: <https://docs.djangoproject.com/en/> Accessed: 13.3.2018.
- [45] Hammant P. (2013), What is trunk-based development? URL: <https://paulhammant.com/2013/04/05/what-is-trunk-based-development/> Accessed: 9.3.2018.
- [46] Jenkins. URL: <https://jenkins.io/> Accessed: 30.1.2018.
- [47] Morell L. & Marciniak J. (2002) Unit/Module Testing. *Encyclopedia of Software Engineering* 2, p. 1806. ISBN: 9780471377375.
- [48] Farrell J.A. (1995), Anatomy of a Compiler. URL: <http://www.cs.man.ac.uk/~pjj/farrell/comp3.html> Accessed: 12.3.2018.
- [49] RStudio. URL: <https://github.com/rstudio/rstudio> Accessed: 21.4.2018.
- [50] Leys C., Ley C., Klein O., Bernard P. & Licata L. (2013) Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology* 49, pp. 764–766.
- [51] Davenport K. (2013), Absolute deviation around the median. URL: <http://kldavenport.com/absolute-deviation-around-the-median/> Accessed: 23.4.2018.
- [52] Rousseeuw P.J. & Croux C. (1993) Alternatives to the median absolute deviation. *Journal of the American Statistical association* 88, pp. 1273–1283.
- [53] Lumley T., Diehr P., Emerson S. & Chen L. (2002) The importance of the normality assumption in large public health data sets. *Annual Review of Public Health* 23, pp. 151–169.