Tampere University

Sami Mustalahti

IMPROVING CONTINUOUS PRACTICES IN SYSTEM-ON-CHIP DEVELOPMENT

Faculty of Engineering and Natural Sciences
Master of Science Thesis
March 2019

ABSTRACT

Sami Mustalahti: Improving Continuous Practices in System-on-Chip Development
Master of Science Thesis
Tampere University
Master's Degree Programme in Automation Engineering
March 2019

This work is about continuous practices in embedded System-on-Chip development. Continuous practices include continuous integration, continuous delivery, and continuous development. These practices mean committing small code changes often to the repository's main branch. Then the changes are automatically tested and integrated with the rest of the system. In the case of continuous deployment, all the changes are automatically deployed to production without any human interaction. Continuous practices are meant to make development faster and more effective, give feedback faster and improve quality by reducing bugs. These tasks are important in today's industry which is continuously changing, and customer satisfaction is as important as ever. This creates the demand to deliver new products and updates rapidly along with high quality.

In the Nokia Networks' System-on-Chip department there was a need to increase the level of automated processes by improving the continuous practices. This work studies continuous practices based on literature and identifies ways to improve continuous practice processes used at System-on-Chip development. The implementation of this work was done at the System-on-Chip departments' software unit where the current state was analysed. The main improvement points found in the analysis were related to automated function, investment and working habits. Based on the analysis the implementation plan was formed. The implementation included adding more functions to the continuous integration server, improving feedback and making the results more visible. These were done by creating more Jenkins jobs and integrating Robot Framework to the testing. All the improvements were not possible to do within the time scope or without the support of the whole department. Therefore, those problem points were analysed, and detailed plans were formed to solve them in the near future.

Keywords: continuous integration, continuous delivery, continuous deployment, automated testing, system-on-chip, Jenkins

TIIVISTELMÄ

Sami Mustalahti: Jatkuvien menetelmien parantaminen järjestelmäpiiri kehityksessä.
Diplomityö
Tampereen yliopisto
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Maaliskuu 2019

Tämä työ keskittyy jatkuviin menetelmiin sulautettujen järjestelmäpiirien kehityksessä. Jatkuvat menetelmät sisältävät jatkuvan integraation, jatkuvan toimituksen ja jatkuvan tuotantoonviennin. Nämä menetelmät tarkoittavat pienten koodi muutosten tallentamista useasti versionhallinnan säilytyspaikan päähaaralle. Tämän jälkeen muutokset testataan automaattisesti ja integroidaan muun järjestelmän kanssa. Jatkuvan tuotantoonviennin tapauksessa kaikki muutokset viedään tuotantoon saakka automaattisesti. Jatkuvien menetelmien tarkoitus on tehdä kehitys nopeammaksi ja tehokkaammaksi, nopeuttaa palauteaikaa, ja parantaa ohjelmiston laatua vähentämällä ohjelmistovirheitä. Tämä on tärkeää nykypäivän teollisuudessa, joka on jatkuvan muutoksen alla ja asiakastyytyväisyys on tärkeämpää kuin koskaan.

Nokia Networksin järjestelmäpiirien kehitysosastolla oli tarve lisätä testiautomaation tasoa jatkuvia menetelmiä parantamalla. Tämä työ tutkii jatkuvia menetelmiä kirjallisuuden pohjalta ja selvittää mahdollisia tapoja parantaa jatkuvien menetelmien käyttöä järjestelmäpiirien kehitysosastolla. Työn käytännön toteutus tehtiin järjestelmäpiirien kehitysosaston ohjelmistoyksikössä. Yksikön alkutila tutkittiin, ja tämän tutkimuksen perusteella löydetyt tärkeimmät parannuskohteet liittyivät automatisoituihin toimintoihin, investointiin ja työskentelytapoihin. Lähtökohta-analyysin perusteella muodostettiin toimintasuunnitelma. Toimeenpano sisältää toimintojen lisäämistä jatkuvan integroinnin palvelimelle, palautteen/palaute-ajan parantamista ja tulosten tekemistä näkyvämmiksi. Nämä toteutettiin lisäämällä Jenkinsiin enemmän tehtäviä ja sisällyttämällä Robot Framework testiautomaatioon. Kaikkia kehitysideoita ei ollut mahdollista toteuttaa työn aikarajoissa ja ilman koko osaston tukea. Tästä syystä nämä ongelmat analysoitiin ja niistä muodostettiin kehityssuunnitelmat, jotta ne voidaan ratkaista lähitulevaisuudessa.

Avainsanat: jatkuva integraatio, jatkuva toimitus, jatkuva tuotantoonvienti, testiautomaatio, järjestelmäpiiri, Jenkins

PREFACE

I'm grateful to Kari Seppänen and Nokia Networks for the opportunity to finally finish my thesis. I would also like to thank my examiners Professor Hannu-Matti Järvinen and Professor Matti Vilkko for the time they have taken from their busy schedule. Special thanks goes to the University's academic officer Johanna Heinola-Lepistö who has been patiently guiding me through the bureaucracy.

Of course, I want to thank my lovely wife who has been supporting me through the whole process. Thanks also to my family and friends.

Tampere, 20.03.2019

Sami Mustalahti

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

5G	Fifth-generation wireless technology
ABIL	Airscale Baseband extension modules Indoor, version L
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASSP	Application-Specific Standard Part
BIP	BTS Intranet Protocol
BSP	Binary Space Partitioning
BTS	Base Transceiver Station
CMOS	Complementary Metal–Oxide–Semiconductor
CPRI	Common Public Radio Interface
CPU	Central Processing Unit
CVS	Concurrent Versions System
DSP	Digital Signal Processor
DUT	Device Under Test
EEPROM	Electronically Erasable Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory
EthSS	Ethernet Sub System
FPGA	Field-Programmable Gate Array
HAPS	High-performance ASIC Prototyping Systems
HDL	Hardware Description Language
HW	Hardware
I/O	Input/ Output
IP	Intellectual Property
ISTQB	International Software Testing Qualifications Board
L1	Layer-1, Physical layer on OSI model
MD5	Message-Digest algorithm
MIT	Massachusetts Institute of Technology
OSI	Open Systems Interconnection model
PC	Personal Computer
PLD	Programmable Logic Devices
SCM	Source Code Management
SHA-256	Secure Hash Algorithm 2 with 256-bits
SoC	System-on-Chip
SRAM	Static Random-Access Memory
SW	Software
URL	Uniform Resource Locator

1. INTRODUCTION

In the intensive, technology-driven industrial environment it is important to get results fast. The industry, and the technology with it, is changing rapidly. For this reason, the first company on the new wave with a new product usually has the advantage. The goal is to release good quality products fast while still meeting the customer specifications. To achieve this, automation technology comes to aid. Automation reduces human intervention and relieves them from monotonous repetitive work, which gives workers more time to focus on innovating and other more important tasks. Automation also improves throughput, quality, and consistency of the product. Therefore, it is important to harness automation for our service in all sectors of industry.

The business needs to be developed, which entails that the tasks and the processes within the company need to be developed. In software development, this can be accomplished by testing frequently, reducing feedback time and integrating often. This is where continuous practices are utilized. Continuous practices refer to a software development practice, where the software code in development is continuously put through automated tests and integration to detect bugs and malfunctions in an earlier state. This makes the feedback loop faster and allows the software always to be in a working state. Continuous practices include continuous integration, continuous delivery, and continuous deployment.

This thesis is made in a company, Nokia Networks. It focuses on continuous practices in SoC (System-on-Chip) development. The company wants to improve the usage of continuous practices in SoC to create reliable code faster and with greater quality to their customers.

1.1 Problem statement and objective

Companies, for instance Nokia Networks, are interested in finding solutions to release high-quality software faster into the market. The companies need to release software fast, in order to meet the demands of the customers and to gain an advantage over competitors. With the help of continuous practices, this goal can be reached. The continuous practices can be very helpful, but still the multiple benefits of these practices are often forgotten. Additionally, even if the benefits are clear, the implementation can be seen complex or out of reach.

This thesis discusses the continuous practices and how to improve them in SoC development in Nokia Networks. The objectives of this thesis are to:

- Analyse the state of the continuous practices and find ways to improve them.
- Clarify the process of the continuous practices to see the benefits of it.
- Increase the use of continuous practices and improve their automated functions.

The approach to reach these objectives is to review the literature for the principles and successful implementations of continuous practices. Thereby providing the base for analysing the current system in SoC development and offering concrete evidence of the benefits achieved by other companies who have implemented continuous practices. Then with the help of the analysis of the system, this thesis will identify potential improvement areas to improve and generate plans to improve them. The research questions are defined to support these objectives.

The research questions for this thesis are:

- Based on the literature, how have the continuous practices been implemented and what kind of benefits have been achieved?
- How can SoC development improve its continuous practices?

Due to the close relation to practical problems, the research method chosen for this thesis is constructive research. It is defined in [1] as: *"Constructive research is used to define and solve problems, as well as to improve an existing system or performance, with the overall implication of adding to the existing body of knowledge."* As seen in the definition this research method complements the objectives of this thesis well. Since the goals of the objectives could also be formed as *to define and solve problems* and then *to improve an existing system or performance*. Overall, these are the main objectives of this work.

There are three main limitations to this thesis. Firstly, the work is limited only to the SoC software unit. Secondly, since there are no measurement tools for the efficiency of the current system, the data comparison between before and after the work cannot be measured. Lastly, since the implementation of the continuous practices takes time and effort from the organization, all the results cannot be seen within the time scope of this thesis.

1.2 Structure

This thesis entails six chapters: introduction, background, methods and tools, implementation, results and analysis, conclusion and future work. In the introduction, the focus is on describing the motivation for the thesis and the problem statement. In addition to these, the objectives and the structure are also being defined in the introduction. In the second chapter, background, the theoretical foundation for continuous practices is established based on literature review. The goal is also to define continuous practices and their benefits. The third chapter, methods and tools, concentrates on selecting the approach for

implementation and presenting the tools that are used to implement continuous practices in SoC SW (software) unit. The fourth chapter, implementation, presents the state of continuous practices in SoC SW unit and the implementing of new automated steps to the process. The fifth chapter, results gathers the achieved results and improvement ideas and analyses them. The sixth chapter, conclusion and future work, summaries the work and presents possible aspects for further development.

2. BACKGROUND

This chapter reviews the literature not only to present a theoretical background to the continuous practices and automated software testing but also to enlighten their key concepts. This chapter also gives insight on how to implement continuous practices and what are the principles in theory. At the end of the chapter, the achieved benefits and faced challenges that have been found in the literature are compiled.

Continuous practices are used in the software development industry. They are practices that facilitate organizations to release new products and features reliably and frequently. [2]. They include continuous integration, continuous delivery, and continuous deployment. These basically cover the same principles but are wider implementations of each other.

Continuous practices have been around in software engineering for many years and they have influenced the industry and invoked interest in the research community [3]. Recently, a growing number of software companies have been adopting continuous practices, and the research on the topic has also gathered more effort since many companies are considering continuous practices [4].

2.1 Continuous integration

Continuous integration is a software engineering practice in which developers share the main branch, where they merge their code frequently [5]. The original idea of continuous integration came from a part of Extreme Programming which is a software development methodology. Its foundation lies on the day-to-day principles that are adopted by the employees to their daily working habits [6]. The goal of these principles is to speed up the development process and eventually improve software quality by reducing integration problems when developing software collectively [5].

Martin Fowler, one of the earliest adopters of continuous integration, describes it the following way: *"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible."* [7].

Fundamentally, it is meant to decrease risks by producing a shorter feedback cycle. The goal of continuous integration is to catch bugs and identify malfunction issues in an earlier state which will consequently result in faster delivery with better quality code. The con-

tinuous integration practices are also intended to improve the communication by improving visibility within and between teams in order to increase the cross-functional activities and problem-solving [8].

2.2 Continuous delivery

Continuous delivery is a software engineering practice in which developers build their software in such a way that it is always in a state from which it can be deployed to the production or to a production-like environment [9].

With continuous delivery the software products that are produced by a continuous integration server are deployed to the production server with a single click of a button. Continuous integration is a prerequisite for successful continuous delivery [10]. In continuous delivery, the frequency of the deployment is not the determining factor. It is the ability to release software at will [11].

2.3 Continuous development

Continuous deployment is a software engineering practice in which every software change goes through automated steps and at the end of this the software is automatically released to production. This results in many deployments to production each day. Continuous delivery is a prerequisite for continuous development [9].

It is not uncommon to use these continuous terms wrongfully or interchangeably since there is no clear industry-wide definition for these [3][9]. Figure 1 is made to clarify the difference between the terms and to exemplify how the terms are used in this paper.

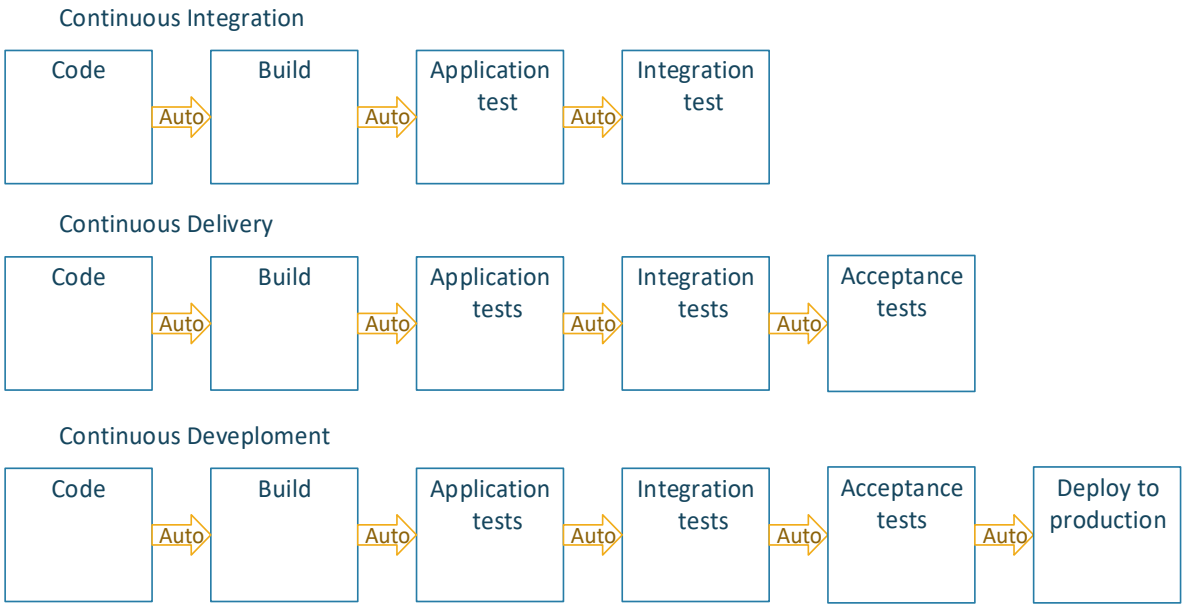

Figure 1 Continuous practices

2.4 Tools of continuous practices

Continuous practices are very tool orientated. Some tools are necessary, others are good to use, and many are available. Figure 2 presents continuous practice tools found in an extensive literature study. These tools are sorted into seven categories:

1. Version control system
2. Code management and analysis
3. Build system
4. Continuous integration server
5. Testing
6. Configuration and provisioning
7. Continuous development server

All tool categories are not compulsory in continuous practices [2]. These have been combined from multiple studies and function as a guideline on how the process in continuous practices could be from a tooling point of view.

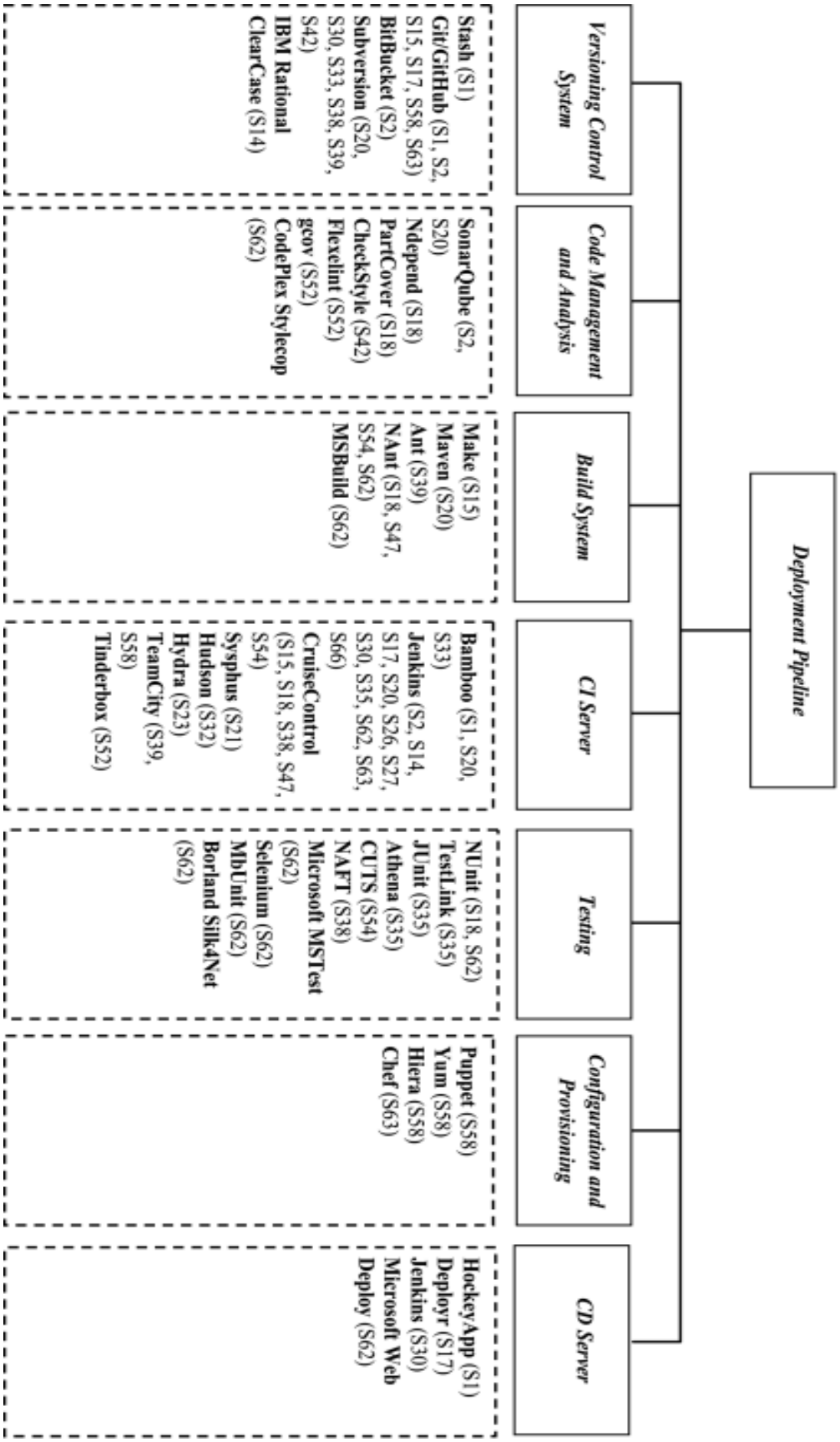

Figure 2 Different tools used in continuous practices [2]

Tools mentioned in Figure 2 are open source and commercial tools which are intended for implementing continuous practices. Other types of tools are also available that aim to assist the progress of continuous practices. In the paper [2], these tools have been sorted into six categories by their properties, which include: “

1. Reduce the build and test time in continuous integration.
2. Increase visibility and awareness of build and test results in continuous integration.
3. Support automated continuous testing.
4. Detect violations, faults, and flaws in continuous integration.
5. Address security and scalability issues in the deployment pipeline.
6. Improve the dependability and reliability of the deployment process.”

This study will not inspect individually all of these facilitating tools, but the tools used in the SoC unit will be opened in Chapter 3 Methods and tools. The general categories described in Figure 2 will be explained in more detail in the following chapters.

2.4.1 Version control system

Version control system, also known as source control system, tracks modifications to a file or files and saves them so one can review every individual change afterwards. It allows reverting files or projects back to a previous state, compare changes over time and see who modified the file [12]. The core of a source control system is its repository. It is the central storage of the systems data. Any number of developers connected to the repository can read or write to these files. The special feature of a repository is that it records the file changes in the repository in such a way that every version of the files that is retained in the repository [13]. Source control system makes it possible for developers to work parallel on the same project without waiting for each other’s tasks to finish [10].

2.4.2 Code management and analysis

Code management and analysis are intended to reinforce the build process. It scans a code and makes quality checks to the code while collecting metric data such as test code coverage and coding standard violations. Depending on the rules given to the analysis tool, the tool reports and visualizes them to the developers [2][10]. With code bases getting more complex, code management and analysis help to catch bugs early and keep the code within the company’s coding guidelines.

2.4.3 Build system

Build systems are responsible for automatically transforming the source code of a software project into a collection of artefacts and deliverables such as executables and development libraries [14]. Build systems are an important part of the software development process. Developers frequently interact with the build system since they need to rebuild new testable artefacts after making a code modification [15]. The specific requirements and constraints of a build tool are written in a configuration language. A single build can include hundreds of command calls which need to be executed in a specified order to produce the required artefacts and deliverables fast and correctly [14][15].

2.4.4 Continuous integration server

Continuous integration server monitors the repository and triggers a build system (see 2.4.3) when a code change is detected. The integration server can be configured to compile code, run tests, check code coverage and style. The setup and conditions of the continuous integration servers are varied, and the criteria are up to the team. Continuous integration server can and should inform the developers about failures and successes. It is possible to do by email or updating the status of the build. The purpose of the continuous integration server is to validate changes made to the project. The end-product of a continuous integration server is an executable of the software that is available to the team. [10][16]

2.4.5 Testing

Testing, in this case, means the automation of software testing activities with the use of automated test tools. These activities consist of the development and the execution of test scripts and the verification of testing requirements [17]. Most of the test cases are repeated many times during a project. The testing tools play a big role in software development since doing this manually would be troublesome, prone to human errors and time-consuming.

2.4.6 Configuration and provisioning

The configuration is used to automatically select which features should be compiled and included in the final deliverables. Along with build tools, compilers and libraries necessary to compile those features. The provisioning will automatically install all these software and libraries and make them available for your application [18].

2.4.7 Continuous development server

Continuous deployment server automatically deploys software to production that has passed all the tests. Some servers can differentiate according to which users the product is released [2] since some companies are using beta testers or have a preview group. Continuous development server tools also offer a rollback mechanism if the product has unwanted properties in a production environment. Then the company can swiftly switch back to a working release and work on the issues [11].

2.5 Practices

A core practice of continuous integration is that all developers commit to the main branch at least once a day. Beyond version control, a continuous integration server is one of the most important tools of the development team which they can take advantage of. Its purpose is to track the code repositories for changes, check out the code if there is a change and run the predefined commands to trigger the build [6].

Martin Fowler [7], summaries: "*The nine key points of continuous integration practices are:*

1. *Maintain a single source repository*
2. *Automate the build*
3. *Make your build self-testing*
4. *Every commit should build on an integration machine*
5. *Keep the build fast*
6. *Test in a clone of the production environment*
7. *Make it easy for anyone to get the latest executable version*
8. *Everyone can see what is happening*
9. *Automate deployment*".

It is very important that the whole development team takes the automated build environment seriously and keeps the build times short to get feedback quickly on the build status. Continuous practice should not be maintained by a single person and if the build is to fail it should be the whole teams' top priority to fix it [10]. The investment in the test environment may be expensive, but the benefits in the long run are vast and cannot be disregarded [11]. Ultimately, everything one can run from one's command line should be integrated into the build [19]. The continuous practices are not just for individual developers they are a team effort [10].

2.6 Automated Testing

Homés [20], defines testing as "*A set of activities with the objective of identifying failures in a software or system and to evaluate its level of quality, to obtain user satisfaction*." It

is a set of tasks with clearly defined goals. Testing software is vital to avoid faults and failures that might affect the user experience. Such failures usually have a negative impact on the company image and in the worst-case scenario cause fatalities. Testing software systems is a complicated task, because of the wide range of the possible faults in the systems and therefore the testing should not be disregarded [21].

2.6.1 Software testing levels

Software testing is divided into four different levels as shown in Figure 3. Where the lowest is the first test to be performed and a prerequisite for the next levels. Bottom-up the levels are unit testing, integration testing, system testing, and acceptance testing. All tests are required before the release of working software. The levels of testing will be defined in more detail in the following paragraphs.

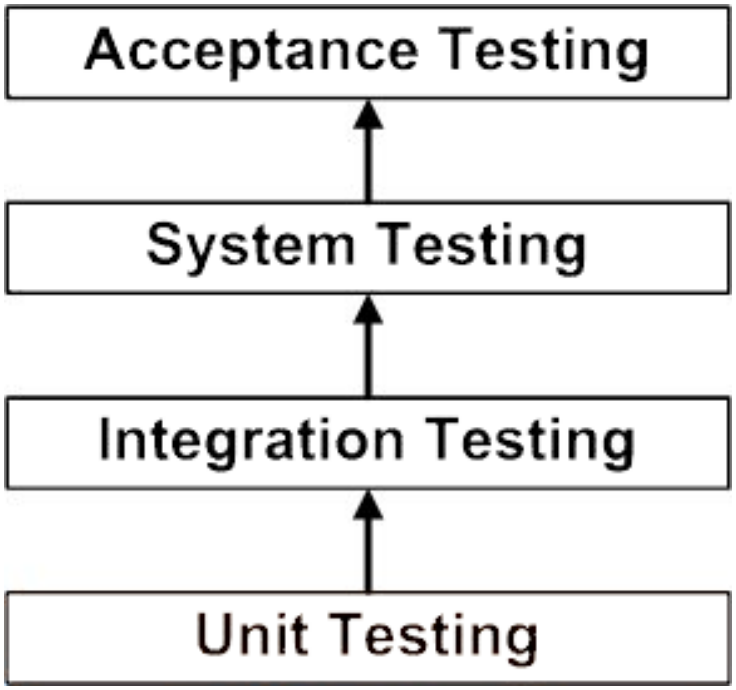

Figure 3 Different levels of testing [22]

Unit testing is a level of software testing in which individual units of software are tested. The goal is to verify that each software unit independently operates as designed in the specification. Unit tests are executed on the smallest parts of the software that can be tested separately. Since testing is done separately from the rest of the system, sometimes simulators, drivers, and stubs are needed to assist in testing. Unit tests focus on functionality and performance. Unit testing increases the confidence in code since well-made unit tests catch the faults early. Faults are easier and cheaper to find and fix in unit testing. [20][22]

Integration testing is defined by ISTQB (International Software Testing Qualifications Board) as *"A testing performed to expose defects in the interfaces and in the interactions between integrated components or systems."* In this case, integrated components refer to software units and hardware components that have been separately unit tested. A system includes an operating system, file systems, database systems, etc. Interfaces and interactions are tested inside the system and between systems. There are many ways to perform integration testing such as big bang, top-down, bottom-up and hybrid testing. Each has its advantages and drawbacks. It is important to understand the architecture of the software and the influence of different components before deciding on the integration testing method [20][22].

In system testing, complete and integrated software is tested in a production-like environment to verify that specific requirements are met. Tests at system level focus on the overall functionality of the system. Tests are performed from the user point of view, they do not focus on the structure of the code itself. Tests include the functional and non-functional aspects of the software [20].

Acceptance testing is done in a production like environment and aimed at reaching a level of confidence in the systems' functional and non-functional aspects. Acceptance testing ensures that the acceptance criteria for software are met. Acceptance tests should be done in coordination with the customer before the development of the software starts. Tests can test all kind of attributes of the system. Acceptance testing is important because it can be used as an indicator for the developer and the customer that the software or a feature is complete when it passes all the tests [23][20].

2.6.2 Software testing methods

There are different testing methods for catching different kinds of defects. All methods have their own strengths and weaknesses. Most methods can be used in different levels of testing. One of these methods is called a black-box testing method which is a specification-based technique, also known as input/output-driven testing [24]. In black-box testing, the tests are developed around the system's functionality and tested against the specification. Since the system is considered as a black-box, the tester does not need to concern the internal implementation of the system. The tester only needs the information on the input data and received output data [25]. Advantages of black-box testing are that it tests the actual functionalities of the system and the tests are done from the end user's point of view. It is also easily understandable even by a non-technical staff. Therefore, testing can be conducted without developer involvement, allowing an objective perspective. Drawback of this method is that it requires a well detailed specification. Errors that are hiding in the internal logic are not always detected if these do not affect the outputs directly. Also, thorough testing is not feasible since it would require that every input condition would be tested [22][25].

Another testing method is white-box testing. It is a structure-based testing technique that focuses tests on an analysis of the internal structure of the system. White-box testing examines the systems paths of logic regardless of the functionality of the systems [24]. In white-box testing, the knowledge of the implementation of the system is required from the tester since testing might need to test cases that may test only one logic component of the system [25]. White-box testing can be conducted on all the test levels, but it is mostly used in lower level testing since it can be started before the whole system is finished. [24][22]. Advantages of the white-box testing are that it can be started early, and it is thorough since it focuses on the code that is produced. It can cover many different logical paths and detect internal errors easily. Disadvantages are that testing can be very tedious because tests might be complex and ready-made tools are not usually available. This requires highly skilled testers and a lot of maintenance since tests are mostly tied to the system that is being tested. Also, passing white-box testing does not mean that the functionality of the system is correct [22][25].

A grey-box testing method is a combination of specification-based and structure-based testing techniques. Grey-box testing focuses on the functional and logical aspects of the system. In grey-box testing, a tester should know about the specification and internal structure of the system. Another option is to create the tests together with the developers to simplify and reduce the tests that are needed. Since some of the functionalities can use the same function that is reused in the system and to test the one test would be sufficient rather than ten. This is the major advantage of grey-box testing. It reduces the number of tests and clears up specification requirements. Grey-box testing is mostly used in integration testing level but can be used in other levels where it is needed to test the logical and functional aspects of the system at the same time [25][22].

2.7 Benefits of continuous practices

In the software development industry, the development process is mostly fixed. It is not feasible to repeat the same process manually every time to when you want a new release out. The continuous practices take this process out of your hands and hands it over to computers which are well suited for the repetitive process [19]. According to Rossel, [10] this is the biggest benefit of continuous practices. The software is compiled and fully tested automatically reducing the chance of human errors and making it considerably easier to get a working executable [10].

According to Martin Fowler [9], "*The principal benefits of continuous delivery are:*

- *Reduced deployment risk: since you are deploying smaller changes, there is less to go wrong, and it is easier to fix should a problem appear.*
- *Believable progress: many folks track progress by tracking work done. If 'done' means 'developers declare it to be done' that's much less believable than if it's deployed into a production (or production-like) environment.*

- *User feedback: the biggest risk to any software effort is that you end up building something that is not useful. The earlier and more frequently you get working software in front of real users, the quicker you get feedback to find out how valuable it really is.*"

In a study made by a Finnish research group, the benefits that the companies mentioned the most often were the ability to get faster feedback and with that the ability to deploy more often in order to keep customers satisfied. This also improved quality and productivity [26]. A company called Paddy Power also found similar benefits after implementing continuous practices to the company. Paddy Power described the achieved benefits to be significant in comparison to the previous way of working. The six main benefits are presented in Figure 4. The number of open bugs in applications was reported to have decreased by more than 90 percent [27].

Figure 4 Paddy Power benefits [27]

The engineering department at Rally Software discovered that after the company had transitioned to continuous practices it was easier and faster to get the potential out of new employees. When the tests and test coverage is good, new employees can be more fearless and productive since the automated tests act like a safety net [11].

Manual vs. automated

Releasing big software releases manually at scheduled time periods requires a lot of careful planning. These preparations are time-consuming and derail developers from other work. On the day of the release, there are a lot of changes happening at once, which makes tracking bugs and defects challenging. Clearing all the bugs in a short time is a laborious process, which makes it expensive for the company and tedious for the developers [11]. Therefore, it is much easier for developers and the company to use continuous practices. The developers and the whole company feel less stress during releases when implementing continuous practices [4][11][27].

In addition to the stress levels being decreased, Chen [27] also reported that after adopting continuous practices, productivity and efficiency had improved significantly. This might be because of the fact that with the old ways of working, developers and testers used to spend 20 percent of their time setting up and fixing environments, and operations engineers used to spend days releasing new software. With continuous practices, environments are set up automatically and tested and working software is released basically with a click of a button [27]. After the change to continuous practices the developer throughput increases [11].

Miller found out in his study that the actual cost of using continuous practices in his project was at least 40 percent less than the hypothetical cost of a process that does not use continuous practices while maintaining the same level of code and product quality. In his opinion, the 40 percent reduction of costs is the smallest savings that one can expect from continuous practices. Since his product was relatively small and the biggest benefits are in bigger projects with lots of repetitive work [28]. Other companies have also found that continuous practices helped facilitate the release process and reduce manual work. The amount of saved time and effort depended on the number of manual tasks which were done before automating these tasks [26].

2.8 Challenges in continuous practices

The main challenges in adopting continuous practices fully include domain restrictions, resistance to change, technological problems, customer desires, and developer skill and confidence. Most companies have a higher continuous practice potential than what they practice. In many cases, companies consciously do not choose to fully utilize automatic continuous practices even though they understand the benefits of it [26]. This might be because some parts of the organization or people feel unincluded. The key to adopting

continuous practices is to have support and interest from the top management in the organization. After this, the process of adopting and utilizing continuous practices must be transparent and clearly communicated through the whole organization not just to engineers but also to sales, marketing, and other departments, since everyone will be affected [11].

Research interviews have uncovered that even if the company is capable of continuous practices, the customer was not able or willing to handle a shorter release rate. The company in question had to tone down the fully continuous practices back to scheduled releases. Some companies are also struggling with the diversity of its clients. In the telecom sector, releasing software directly to network devices is challenging because network configurations vary among clients [26]. This causes them to manually test different customer configurations. Some of the interviewees were overwhelmed by the amount of work and time automating the acceptance tests and performance tests take, even though they had already successfully automated the unit and integration tests [4].

There is no ready-made solution available for continuous practices that includes every aspect and is customizable [27]. Hence, setting up the infrastructure required for continuous practices is time-consuming and complicated. The organization, and more importantly the developers, must have the knowledge of fully automated continuous practices. Unfortunately, companies do not necessarily have these kinds of resources at their disposal [26].

2.9 System-on-Chip

System-on-Chip, also known as SoC, is an integrated circuit which contains all the required electronic components for a system in a single chip. SoC consists of functional blocks which usually include processors, memory units, system timing generators, DSP (Digital Signal Processor), communication circuits, data converters and voltage regulators [29]. Typical function block design of SoC can be seen in Figure 5. SoC designs involve hardware and software components with increasing amounts of embedded software. Modern embedded systems are developed to be more and more sophisticated with the integration of multiple functionalities in the same system, often implemented on a single chip [30].

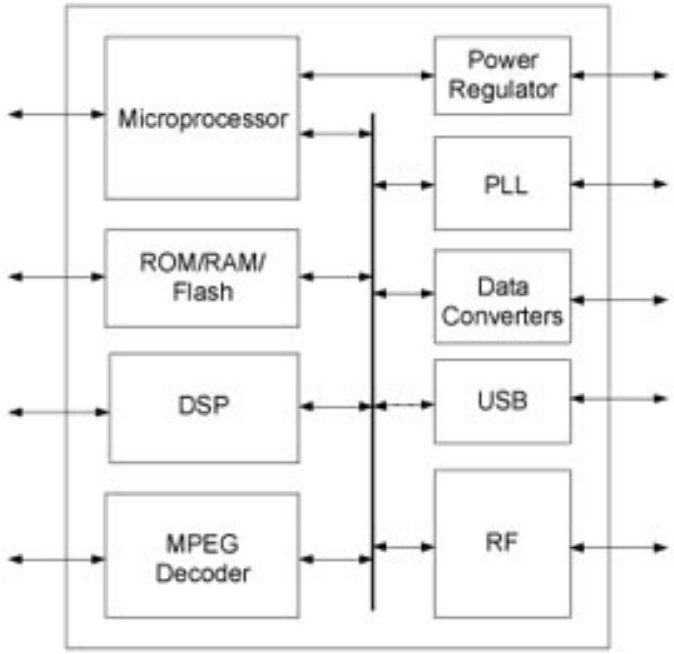

Figure 5 Typical function blocks of SoC [29]

Some of the advantages of SoC are its size, design flexibility, low power consumption, and manufacturing costs in high volume and the disadvantages are limited processing capabilities, long design time, expensive manufacturing costs in low volume and difficult to service compared with other systems. Many of the modern devices utilize SoCs in their designs, phones, smart watches, and most of the small battery powered devices. There are different types of design styles in System-on-Chip devices, few of them are PLD (Programmable Logic Device), ASIC (Application-Specific Integrated Circuit), ASSP (Application-Specific Standard Part), and FPGA (Field-Programmable Gate Array) [31]. Next, we will focus more on FPGAs because it is the design which is used in SoC Loner project.

2.9.1 FPGA

Field-Programmable Gate Array (FPGA) devices were introduced by Xilinx in 1985. They were designed to fill the gap between the current technologies by providing better I/O (Input/Output) capabilities, design flexibility, programmability, and fast system developing in a cheaper logic platform [30][32]. A common definition of an FPGA found in the literature is to define it as a matrix of configurable logic blocks, linked to each other by an interconnection network, which is entirely reprogrammable [30][33][31].

In the 1990s, the sophistication, the size, and the data processing power of FPGAs began to increase, which made them to be considered as an appropriate solution in telecommunication, networking, signal and image processing fields. All of which involved an ever-increasing demand for fast data throughput and processing of large data blocks [33]. In

the change of the millennia, FPGAs were already adapted to automotive, industrial and consumer use. This is also when high-performance FPGAs became available, which meant that today's FPGAs could be used to implement almost anything [31].

FPGAs are manufactured by multiple companies from which the three largest are Xilinx with 50% market share, Altera-Intel with 37% market share and Lattice Semiconductor with 10% market share [34]. Manufacturers resort to many different technologies and offer different device families. Each device family differs in the details of device architecture, device programming technology, internal signal routing, power, capacity, voltage, I/O support, and packaging. This is because manufacturers want to separate themselves from others and gain a competitive advantage on some specific field or applications which require certain architectures or features. Most manufacturers offer at least cost-optimized and performance-optimized FPGA families [32].

FPGA devices' distinguishing factor is its programmability and in the design process, they need to be programmed to define their functional operation. Based on FPGA devices' programmability they can be sorted into two categories, reprogrammable and one-time programmable devices. There are different programming technologies for FPGAs and each of them has a different approach to programmable logic architecture. These technologies are called [32][35]:

- SRAM (Static Random-Access Memory)-based
- Anti-Fuse-based
- EPROM (Erasable Programmable Read-Only Memory)-based
- EEPROM (Electronically Erasable Programmable Read-Only Memory)-based

The table in Figure 6 from Cofer's and Harding's book "Rapid System Prototyping with FPGAs" describes the differences between technologies.

Configuration Technology	***Technology Overview and Features***
SRAM-based	An external device (nonvolatile memory or µP) programs the device on power up. Allows fast reconfiguration. Configuration is volatile. Device can be reconfigured in-circuit.
Anti-Fuse-based	Configuration is set by "burning" internal fuses to implement the desired functionality. Configuration is nonvolatile and cannot be changed.
EPROM-based	Configuration is similar to EPROM devices. Configuration is nonvolatile. Device must be configured out of circuit (off-board).
EEPROM-based	Configuration is similar to EEPROM devices. Configuration is nonvolatile. Device must be configured and reconfigured out of circuit (off-board).

Figure 6 FPGA configuration technologies [32]

Volatile configuration is retained only if it is receiving power. Non-volatile configuration is retained even without power and it is stored in memory [36]. From these technologies, the SRAM has become the most widely used technology in modern FPGA boards due to its re-programmability and use of CMOS (Complementary Metal–Oxide–Semiconductor) technology, which is used for constructing SRAM and integrated circuits [35].

2.9.2 IP block

IP in this paper is an intellectual property, and an intellectual property block is a block of logical data. These IP blocks are used by integrated circuit industry to design and manufacture FPGA and ASIC into a specific product. One FPGA or ASIC product contains multiple IP blocks. Ideally, the IP blocks are designed so that they are mostly independent and modular in order to make them reusable in multiple different products and designs by different vendors and manufacturers [37]. For example, CPU (Central processing unit) is an IP block [38].

IP blocks can be sorted to three categories: soft blocks, hard blocks and firm blocks. Soft blocks are delivered using a register-transfer level or higher-level description languages such as HDL (hardware description language) or Verilog. With soft blocks it is the chip manufacturers responsibility to implement and synthesize the logic to the gate level in ASIC's or FPGA's. Soft blocks have the benefit of flexibility and reusability, but their performance varies depending on the implementation process [37][39].

Hard blocks are ready-to-use IP blocks. They have a fixed layout and are optimized for a certain application. These blocks cannot be modified by the chip manufacturers. They are meant to be used in ASIC's and FPGA'S for plug-and-play solutions. Hard blocks benefits are high and predictable performance, but this reduces greatly their flexibility and increases the design costs [37][39].

Firm blocks are between the soft and hard blocks. They are delivered with some fixed placement data and with parametrized circuit description which is configurable to make optimization for specific designs and applications possible. Configurable parameters allow the chip manufacturers more flexible implementation in ASIC's or FPGA's. Firm IP block offers better performance than soft and better flexibility than hard, yet it does not excel either one completely [37][39].

3. METHODS AND TOOLS

This chapter focuses on the methods and tools that are used to improve continuous practices in SoC SW. As the literature review indicated, there are a great number of benefits on implementing continuous practices, which is why those principles are implemented in SoC SW. It was also stated that moving to a fully automated continuous deployment system is not always easy and it takes time. Therefore, this work will focus more closely on a few improvement aspects.

The key aspects to focus on are:

- Increasing the level of automation
- Increasing visibility
- Improving feedback.

The approach to achieve this is to identify manual steps in the process and automating the steps by adding them to the continuous integration server. Taking advantage of available tools and technology to increase visibility and to make feedback faster. While implementing these changes the goal is to identify other factors that are hindering the full utilization of continuous practices.

Since the tools are an important part of continuous practices, a short introduction of the tools used in this project to improve continuous practices in SoC SW will be presented in the following sections.

3.1 Git

Git is a distributed version control system. It is open-source and free, and it is designed to manage all projects regardless of their size with speed and efficiency [40]. Git is a creation of Linus Torvalds. Torvalds created Git in 2005 when he needed a new version control system to keep track of his ever-growing amount of Linux kernel code. From there on, Git has grown to be the most used source code management system; even Windows is using it [41].

One benefit of Git is that it does not have a single failure point since it is a distributed source code management. Meaning that one does not do a checkout of the current source code but a clone of the whole repository to one's local machine. Thus, every user has a backup of the main server on their machines which can be used to restore the server in a case of a crash. This also makes it possible to work offline and push the changes later, since all the changes are saved locally [40][41]. Another great feature of Git is its branching model. Git makes it possible to have different branches for development. This allows

users to try new things more freely without worrying about how it might affect the system or others. It is also possible to select which branches you push when pushing to the main repository [40].

3.2 GitLab

GitLab is a Git repository management tool. GitLab was written by Dmitriy Zaporozhets and Valery Sizov in Ruby and was launched in 2011 as an open-source software under MIT licence. GitLab has multiple features which include permission management, issue tracking, documentation pages, and task lists. All of them are operated from a clear web interface [42][43].

GitLab makes the Git repository user-friendly and intuitive to use from its web user interface. This makes it easier to adapt employees to work with Git since people are nowadays more accustomed to graphical interfaces than command lines. GitLab also provides good documentation platform for teams and projects to keep track of the progress and issues. Another benefit is that GitLab's permission management is organized and versatile [43].

3.3 Jenkins

In this project Jenkins has been chosen for the continuous integration server. Jenkins is also used for other functionalities through its various plugins. Jenkins is an open-source automation server that can be used to automate many tasks associated with software development such as building, testing, delivering and deploying. Jenkins was created by Kohsuke Kawaguchi. Jenkins started its life as a Hudson in 2004 at Sun Microsystems but in 2011 it was released under MIT (Massachusetts Institute of Technology) open-source license and renamed as Jenkins [44].

The Jenkins pipeline consists of plugins that support implementing and integrating continuous pipelines. This pipeline provides everything that is needed to get the code from source control through all the necessary steps to be deployed to the customer. Jenkins and its plugins are written in Java [44][45].

3.4 CMake

CMake is a cross-platform open-source build tool that is designed to build, package and test software. It was first released in 2000. Since CMake is an open-source, new features can be added to it. CMake is created to use simple CMakelists.txt files instead of having different build files for each platform. This lowers the risk that software is not building correctly on different platforms and reduces duplication. CMakelists.txt files are then used to generate build files, also known as makefiles [46][47].

With CMake, it is possible to generate a native build environment that will compile source code, build executables, generate wrappers and create libraries. CMake supports in-place and out-of-place builds. Therefore, with CMake, it is possible to support various builds from a single SCM repository branch. It has support also for static and dynamic library builds [46]. It is possible to use CMake from a graphical user interface or from the command line [47].

3.5 Yocto

The Yocto Project is an open-source collaboration project including many companies, people and communities. It provides templates, tools and methods to create custom Linux-based systems for embedded products regardless of the hardware architecture [48][49]. With Yocto's task executor and scheduler, Bitbake, one can create working Linux operating system automatically by defining four key areas, user configuration, metadata, machine BSP (binary space partitioning) configuration and policy configuration. Once these are configured Bitbake can create the project [48].

Benefits of Yocto are its flexible framework that allows one to re-use software for future devices and only rebuild those parts that need changing. Yocto also provides one common Linux for all major architectures. These save time and money when developing embedded software. Yocto also provides many development tools to improve, i.e., debugging and performance, one of which is called devtool. Devtool provides functions that make testing, building and packing software easier [48][50].

3.6 Robot Framework

Robot Framework is a multi-purpose open-source test automation framework for acceptance testing and for test-driven development. The basic idea of Robot Framework was created by Pekka Klärck in his master's thesis in 2005. In the same year, the first release was developed in Nokia Networks. The second version was open-source and it was released in 2008 under Apache Licence 2.0. Now Robot Framework is sponsored by Robot Framework Foundation [51][52].

Robot Framework is made to be user-friendly, highly flexible and extendable. It uses a keyword-driven testing approach and it is an operating system and application independent. Robot Framework can be extended by creating new test libraries and keywords with Python or Java [51]. This makes Robot Framework very flexible and able to work with many different systems and test environments.

3.7 Coverity

Coverity is a code analysis tool provided by Synopsys. It is designed to find vulnerabilities and deflects, i.e., resource leaks, buffer overruns and use of uninitialized data from the source code. Coverity's main method of finding defects is a static analysis [53].

Coverity started as a Stanford University research project in 2002. In 2006, Coverity started a three-year-long project with the United States Department of Homeland Security to improve the quality and security of the open-source software. In the first year of the project, over 6000 defects were fixed. In 2014, Synopsys acquired Coverity [53].

4. IMPLEMENTATION

This chapter describes the implementation of continuous practices to SoC development project Loner. Loner is a 5G L1 (Layer-1) ABIL (Airscale Baseband extension modules Indoor, version L) FPGA project at Nokia Networks which is developed in three different units: FPGA HW (Hardware) design, FPGA SW and application SW. At the SoC development FPGA SW unit, the work in the Loner project consists of the development of the Loner IP blocks (Intellectual Property):

- CPRi (Common Public Radio Interface)
- EthSS (Ethernet Sub System)
- BIP (BTS (Base Transceiver Station) Intranet Protocol).

The work includes device drivers, software APIs (Application Programming Interface), user guides and testing. The FPGA HW unit designs hardware IP blocks. The Application SW unit is the customer of our FPGA SW unit. Since after the work is completed at our SoC development FPGA SW unit, the Application SW unit continues with the testing and development of the application layer. In this thesis, we will focus on the process of implementing continuous practices to the Loner project in FPGA SW unit.

4.1 Testing in Loner project starting point

There are not many continuous practices in use in the Loner project, although part of the testing has automatic features. Currently, there is not any automation between hardware and software team. When the hardware team makes a new update to the hardware, i.e., bitstream they inform the software team by email or verbally in the break room. The software team does not use the new bitstream immediately. Usually, the usage of the update depends on when the customer adopts the current version of the FPGA bitstream.

4.1.1 Continuous integration server

Some of the continuous integration server functionalities have been set up. The continuous integration server used is Jenkins which has been set up to monitor the repository and start a build when a commit is made. Currently, this is used only in a few IP blocks in development. Integration testing is conducted manually with some automatic features. There is a python script, that needs to be started manually. The script starts a test case which automatically runs the tests that are relevant to the case. Deployment to the customer is done by committing to a Git repository. The repository contains all APIs from which the customer can continue development work.

Next, the automated parts are explained in more detail. Figure 7 shows the current flow of the development. It includes all the parts of the development with markings which are automated, and which are conducted manually.

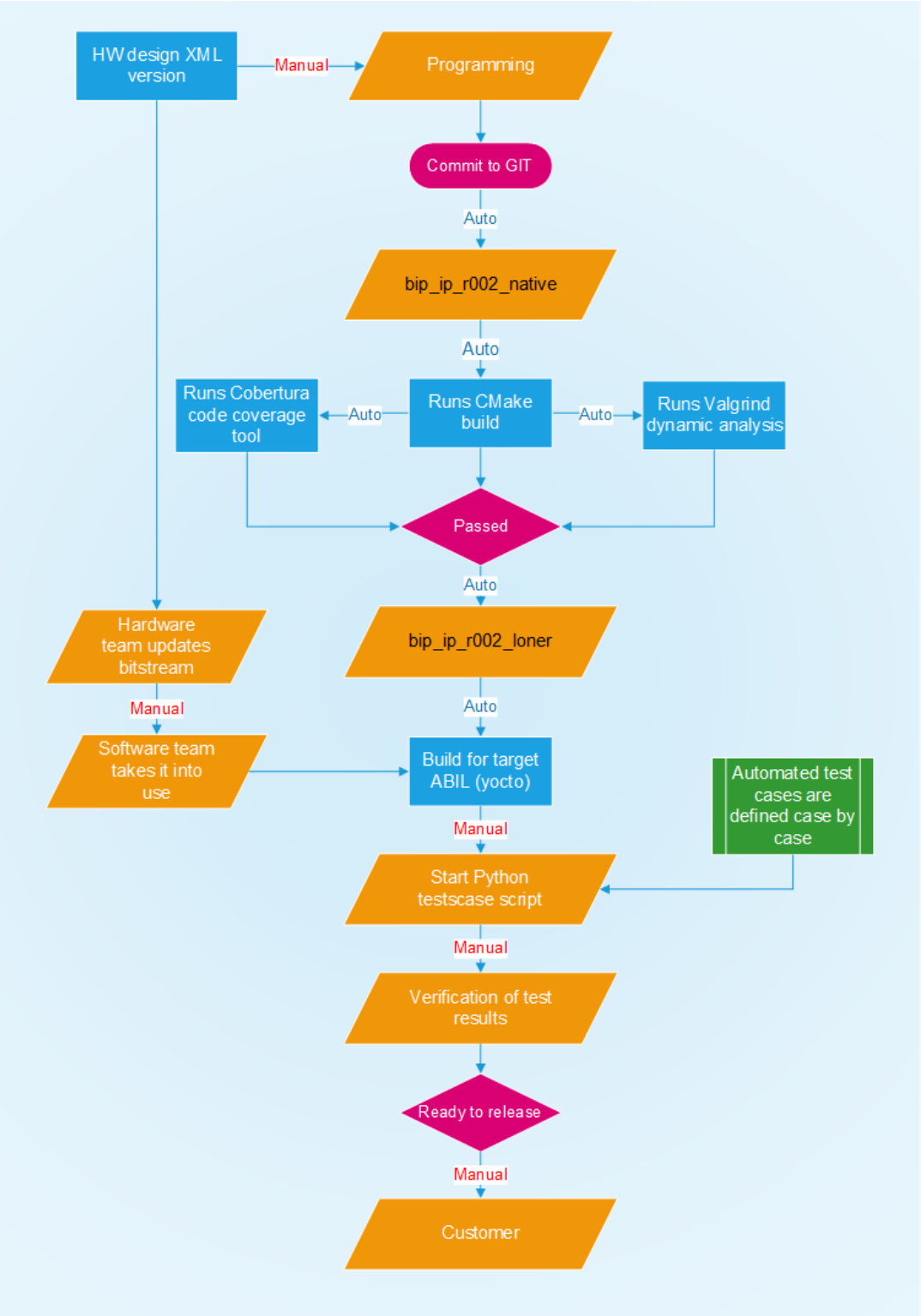

Figure 7 Workflow

When a new code commit is made e.g. to BIP Git repository Jenkins starts a job named bip_ip_r002_native that runs the native build. The build runs unit tests, reports the code coverage and analyses the code. After the job bip_ip_r002_native has been completed successfully, Jenkins starts another job named bip_ip_r002_loner which builds for target system ABIL with Yocto Devtool.

4.1.2 Testing

The test case follows an object-oriented model. When starting Python script test case, classes in Figure 8 will be instantiated and test case class will utilize variables and functions defined by other classes.

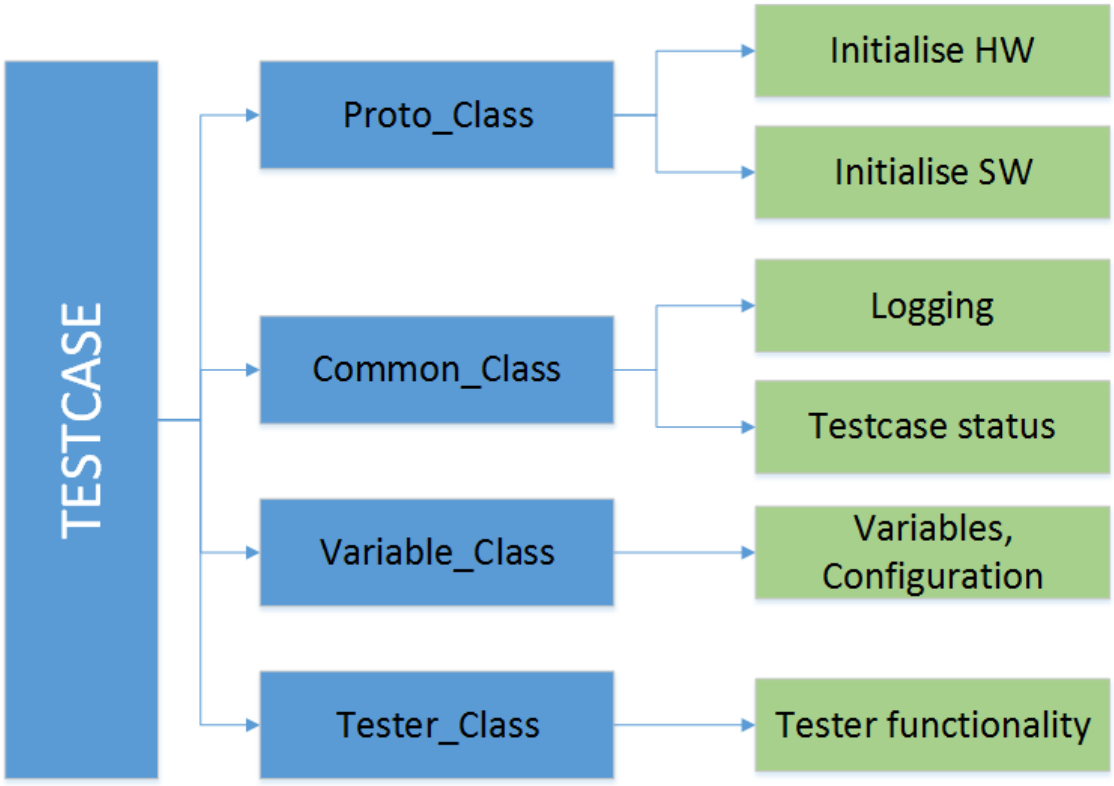

Figure 8 Testcase

First, it will automatically reserve a production test environment from the reservation system to prevent other users from interfering with the environment. After the test, the environment will automatically be released for other users. The environment and the hardware are decided beforehand based on the needs of the project because the software is dependent on specific hardware setup.

The next script will configure the DUT (Device Under Test) hardware i.e. program bitstream to HAPS (High-performance ASIC Prototyping Systems). The bitstream comes from the previous Jenkins build. After this, it will configure the DUT software including

Linux kernel, hardware drivers and basic API software to provide userspace interface for application software.

Now DUT is up and running. The next test case specific configuration will be made. This includes, for example, IP configuration of DUT and configuration of the tester object. In typical cases, tester object sends test data to DUT. The test data can be looped back to the tester. It is also possible that DUT sends test data and it is looped back to it. This can be done for example with X-STEP, which is a specific test system made by Sarokal Test Systems company. Then the received data will be analysed either in the tester or in DUT.

Log file and test status script are running during testing and the results are saved into a file. The script is logging intermediate results; if one of the intermediate results fail, the whole test fails even though the test case will complete. Serious faults can stop the testing immediately when the fault occurs.

4.1.3 Environment

SoC SW has a laboratory space that has been set up for testing software and hardware functionalities. The testing has been conducted with the development environment, HAPS. The environment has racks with multiple FPGA boards that can be configured to resemble the device under test. This development device setup is slower than the end-product would be and therefore causes problems in testing.

With some Loner IPs, for example with CPRi drivers, the hardware unit testing is conducted by simulating FPGA registries with PC (Personal Computer). The laboratory has received an actual ABIL FPGA board that is the same than the end-product to test with. Currently, it has only been in manual use to test that all the connections work and the setup is suitable for testing.

4.2 Next steps in adopting continuous practices

The first step is to add the two other IPs, EthSS and CPRi from the Loner project to the Jenkins pipeline. This means that committing to those repositories would start unit tests and builds automatically, similarly than in BIP. After completing these, the second step is to automate the integration testing and to implement it into the Jenkins pipeline in such a manner that the test cases would start after Yocto builds are completed.

SoC SW team has also found that there is a need for more formal interaction between the hardware and the software team. The plan is to improve and clarify the communication between teams and add hardware artifacts to the automation pipeline. The goal is to get the information when there is an update in HW design, so the lowest software layer code can be generated automatically. The new bitstream file should also be automatically added to the project. This would be a big step towards the fully automated system. Even

so, this is not so straightforward since different customers have different hardware setups and the team need to coordinate with the customer which version is in use.

After the SoC test laboratory is finished, it is planned to be embedded in the continuous pipeline. There would be a remote connection to the testing setup, e.g., FPGA boards. The boards would start, install needed software and run necessary tests automatically after the previous condition is filled.

To maintain a unified path SoC SW team should set basic guidelines to follow when implementing continuous practices. Such as each project should have at least one main job that builds the master branch of the project. This build should fulfil the following points:

- Code compiles without fatal warnings.
- Code is analysed with a static analyser tool without fatal issues. The analysis and the results are published for each build.
- Code is analysed with a dynamic analyser without fatal issues. The analysis and the results are published for each build.
- Unit test cases for released code are executed and passed. Pass ratio is published for each build.
- Unit test code coverage for released code is over 85%. Coverage reports are published for each build.
- Latest documentation is published from source code.

4.3 Implementing new features

The basis for implementing continuous practice is the continuous integration server, in SoC SW that is Jenkins. All the automated tasks are started through it. The status and results are displayed in the Jenkins web-user interface. Therefore, tools and techniques to be implemented should be well suited for Jenkins.

4.3.1 IPs added to Jenkins pipeline

The two other IPs are included in the Jenkins pipeline in the same way that the BIP IP. New Jenkins jobs are created for both EthSS and CPRi. After creating these jobs, they are configured to work as part of the automated process. In configure settings, general information is given to the job, e.g., accurate name and a short description as shown in Figure 9. In Loner project, Jenkins jobs are connected with a GitLab project and with Redmine, which is a project management tool, to get notifications from Jenkins and to track commits and tasks.

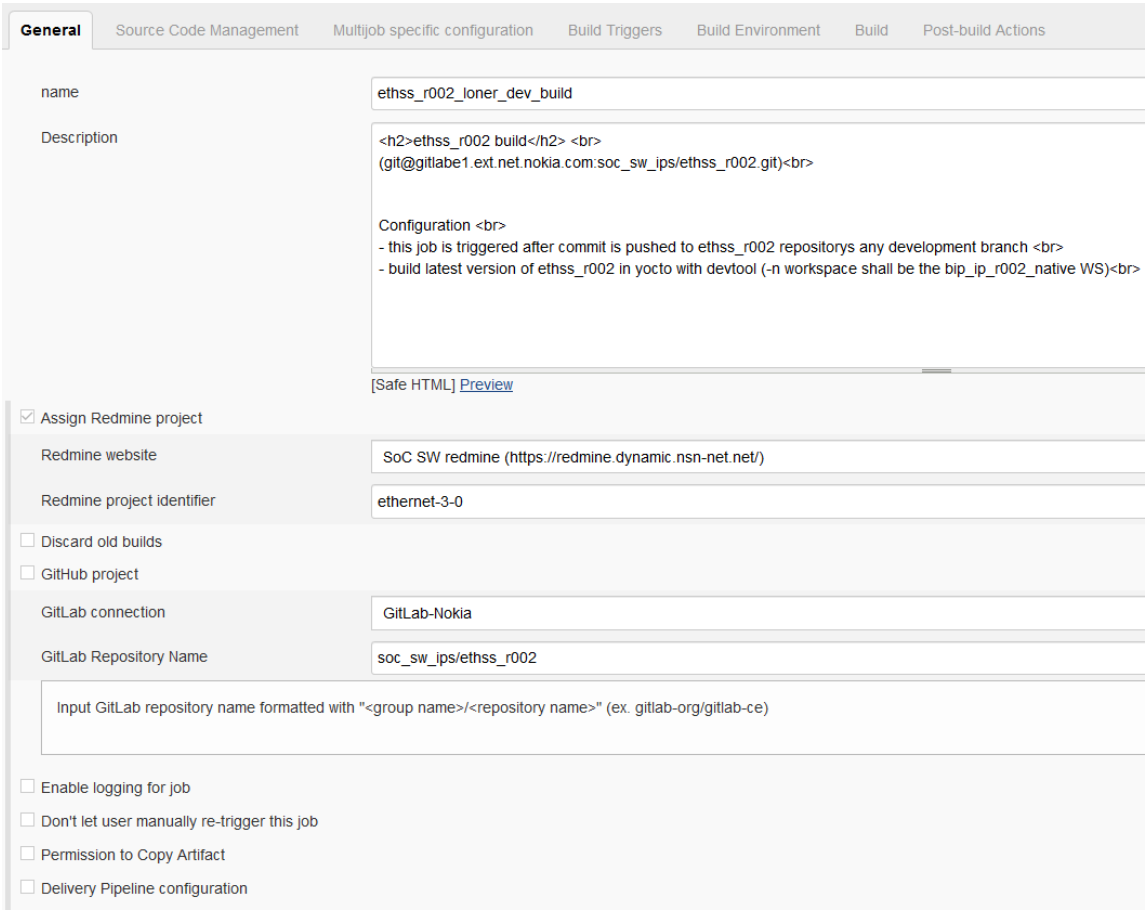

Figure 9 Jenkins general info

Jenkins offers most of the common SCM (Source Code Management) systems e.g. CVS (Concurrent Versions System), Git and Subversion as shown in Figure 10. The source code management system is selected based on the team's preferences. SoC SW is using Git for SCM, so that is chosen in this case. The repository link to the correct work repository is established by writing its URL (Uniform Resource Locator) to the configurations. From the SCM configuration, it is possible to choose which branch of the repository is tracked for changes. SoC SW is using development branches for individual programming and merging the changes to the master branch after a task is completed. It is debatable, whether only the master branch should be tracked, or should all the branches be tracked. Currently, both methods are in use.

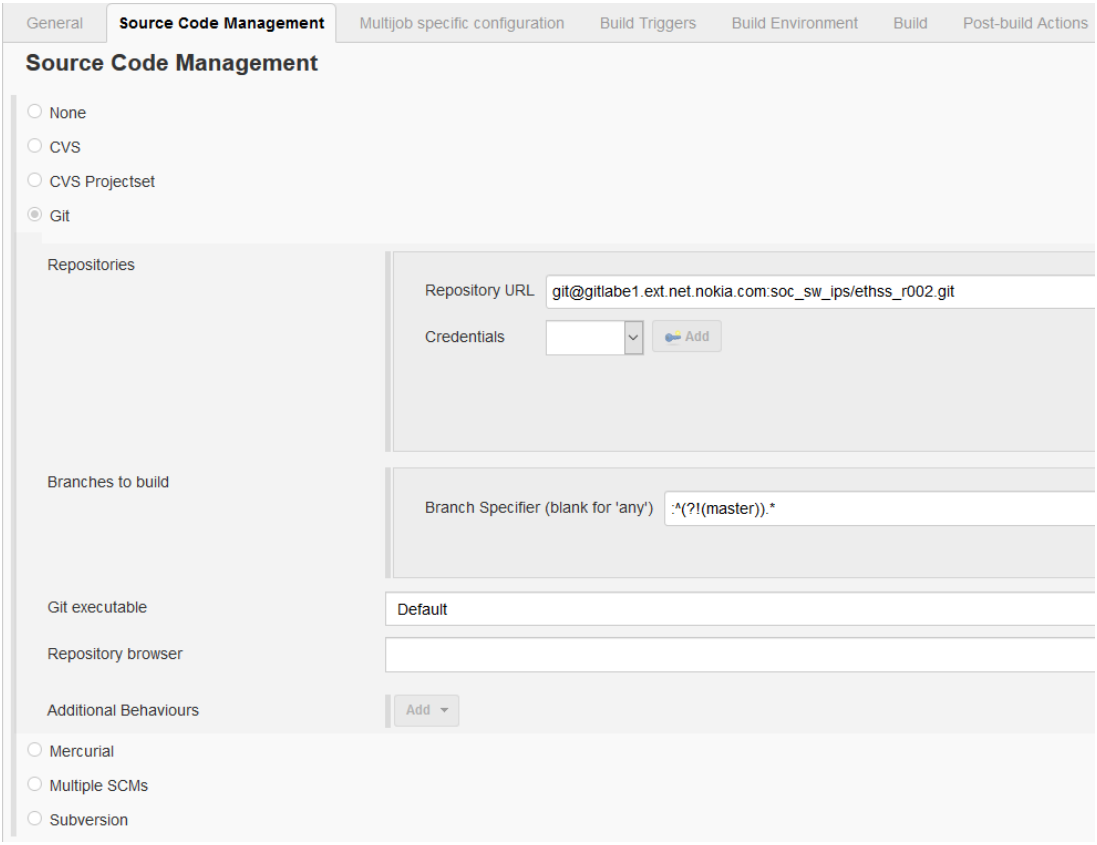

Figure 10 Jenkins SCM

There are different options for the build triggers from which to choose, as shown in Figure 11. For example, it is possible to build periodically or by polling the SCM repository at certain time intervals. In Loner project, the build triggers have been configured to start the build when a new commit is pushed to Git, known as GitLab webhook, or after the completion of another Jenkins job. Only Loners Yocto build is configured to be built periodically. Yocto build is set up to do repository synchronization nightly.

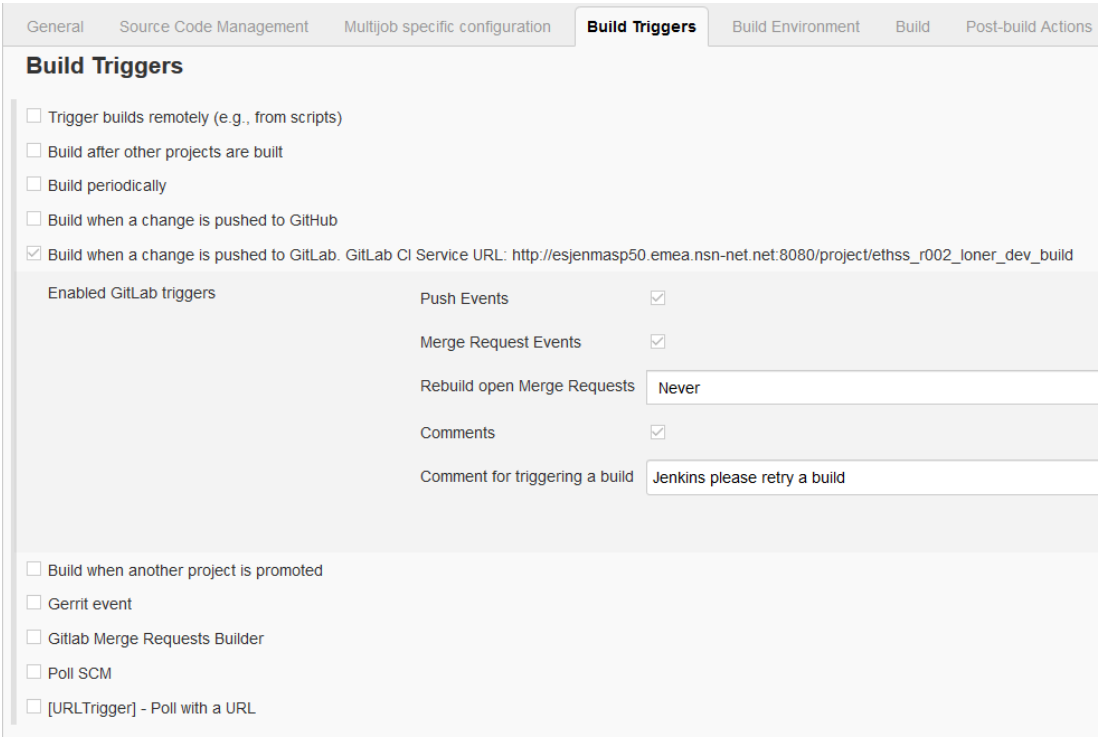

Figure 11 Jenkins build trigger

Selecting build steps and configuring build functions are the most important part of a Jenkins job. SoC SW is using shell scripts to run the builds and tests as seen in Figure 12. It is possible to make conditional build steps so that the second build only starts when the prerequisite is met.

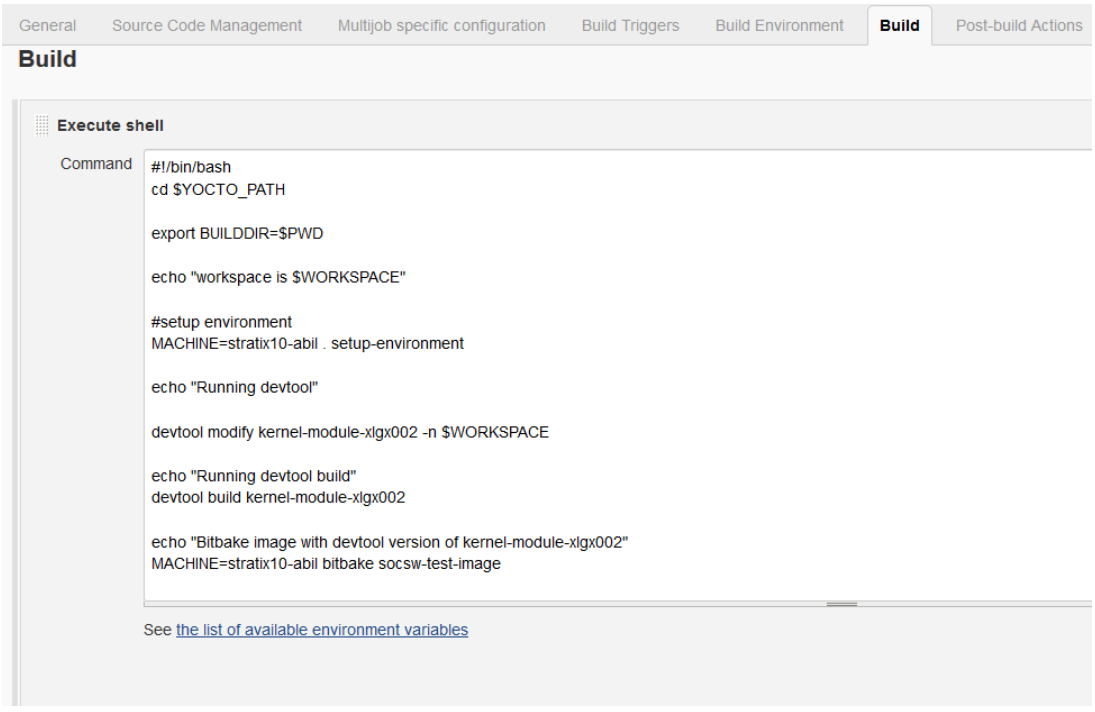

Figure 12 Jenkins build

It is possible to select post-build actions, such as sending an email to a team's mail group to inform that build has failed. It is also possible to trigger another Jenkins job after a successful completion as illustrated in Figure 13, where Jenkins starts a job that runs integration tests for EthSS. This is the method how SoC SW starts downstream Jenkins jobs automatically. This feature is now implemented into all the IPs. In this configuration, it is possible to select which artefacts to save and to record fingerprint to it. This is important, since these images are used in the downstream jobs and with the fingerprint it is possible to track from which build the image came from.

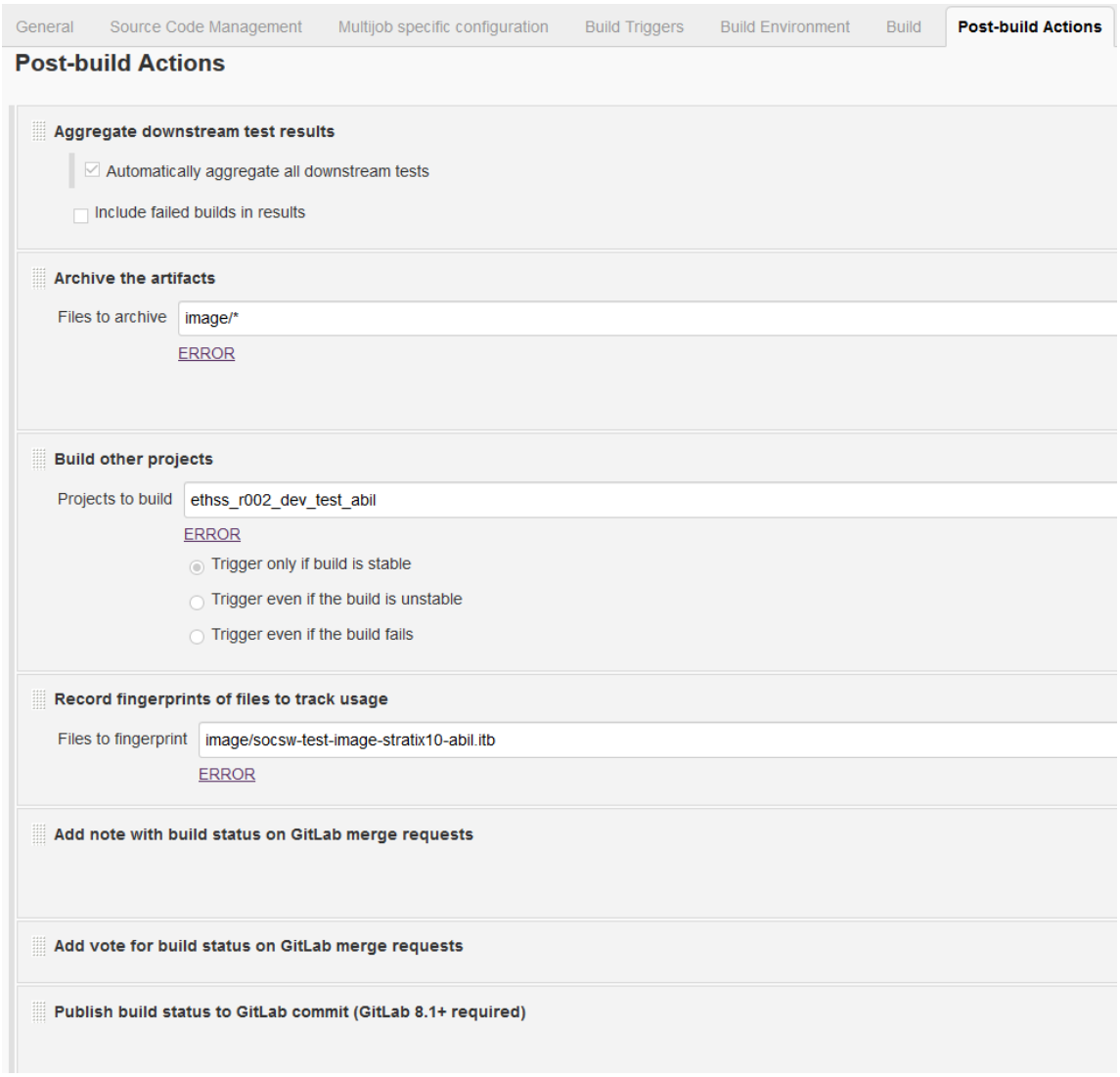

Figure 13 Jenkins post-build

4.3.2 Automatic integration tests

Integration test cases are started automatically via Jenkins. The logic of the integration testing is conducted as previously mentioned with one exception. The script that starts the other scripts as shown in Figure 8 (page 26) is now done with Robot Framework.

Figure 14 is an example of a Robot Framework script. Essentially, the script starts the tests that are written in Python. These tests are made into a Robot Framework test cases. The test cases are comparing the return value of Python tests. According to this value the test case is set to pass or fail.

The move to Robot Framework was quite simple, but not without some drawbacks. It was found that the buffer in Robot Framework did get full in a few test cases that had a lot of output. This caused the test program to be unresponsive, which prevented the Jenkins from completing the job. Since the testing is conducted with only one FPGA board, this buffer overflow would also halt all the other test jobs as well.

At first, the issue was solved by opening a separate shell within the Robot Framework and executing the tests with a lot of output in there and saving the output to a file. The use of the shell command is not recommended by Robot Framework since it can interfere with the communication of the Robot Framework process and make it an operating system dependent. For these reasons, this solution was not acceptable. To overcome the logging issue, changes were needed in the Python test scripts. This was good since it meant a thorough inspection into the logging was needed. It became evident that it was producing too much data and the commonly needed information was scattered amidst it. The old logging system was made clearer and humanly readable. The new logging system only prints out the most necessary things and makes a separate file for the more comprehensive analysis needs.

```
*** Settings ***
Library           OperatingSystem
Library           Process

*** Variables ***
${ROOT}=    /home/soc_sw_ou_ci/jenkins/workspace/ethss_r002_dev_test_abil/ethss/testcase


*** Test Cases ***
abil_update
    [Documentation]    Runs basic_update.py
    ${result}=    Run Process    python    ${ROOT}/basic_update/basic_update.py    1   cwd=${ROOT}/basic_update/

    Should Be Equal As Integers    ${result.rc}    0


bridged_ping_vlan_ctag
    [Documentation]    Runs bridged_ping_vlan_ctag.py
    ${result}=    Run Process    python    ${ROOT}/bridged_ping_vlan_ctag/bridged_ping_vlan_ctag.py    1

    Should Be Equal As Integers    ${result.rc}    0


basic_ping_test
    [Documentation]    Runs basic_ping_test.py
    ${result}=    Run Process    python    ${ROOT}/basic_ping_test/basic_ping_test.py    1   cwd=${ROOT}/basic_ping_test/

    Should Be Equal As Integers    ${result.rc}    0
```

Figure 14 Robot Framework

The decision of changing from Python to Robot Framework was made because Robot Framework provides better support, information, and visibility in Jenkins. With Robot Framework the results from the tests can be displayed in Jenkins jobs' main view (Figure 15). This makes it easier to see which and how many tests have passed/failed. With Robot Framework Jenkins can also track the trend between the builds and visualize it in a graph as shown in Figure 16Figure 15. In addition, there are links to the Robot Frameworks' test report and log, which can be used to view more detailed information about the tests that Robot Framework has run.

Figure 15 Jenkins Robot Framework

Figure 16 Robot Framework results graph

The laboratory setup is ready for the actual Loner ABIL FPGA board testing. All the test cases are now tested on real end-product hardware. The board has a reservation system and queue, which are used automatically by the test cases. There are not any more development environments, boards or slow register simulation testing with PC. The production like environment gives the best feedback for real use-case testing.

4.3.3 Hardware-software communication

Communication with the hardware team has been improved and automated. There is a continuous pipeline running for testing hardware changes.

When the hardware team updates their bitstream file, FPGA configuration data, they also update a file in GitLab which has the release information, version, source file locations

etc. Continuous integration server, Jenkins, tracks this repository and gets a notification of the changes via GitLab webhook. After this, Jenkins starts a Python script that parses the hardware teams file to find the version number and URL address. It also fetches MD5 (Message-Digest algorithm) and SHA-256 (Secure Hash Algorithm 2 with 256-bits) checksums for the bitstream from a different repository.

Then the script finds the previous version of the Yocto BitBake recipe and updates to its metadata the new bitstream version number, source file locations, and checksums. Then according to this BitBake recipe, new Yocto is built. Jenkins saves the images and passes them on to another Jenkins job, which tests software's combability with the new bitstream.

The test job configures the ABIL FPGA test board in the laboratory with the new images and runs a small test set to see if there are any changes needed in the current software because of the bitstream update. This is not the official way to release the new bitstream. This is made to give the software team a head start on the official release if there are any changes needed.

4.4 Tool linking and status visibility

Since there are many tools in use, it is important to link them together. Otherwise, they may become a burden. Figure 17 provides an integration plan with the tools, which will create more visibility for the developers and managers.

SoC SW is using Jira for releasing and creating issues. These same issues are linked to GitLab via GitLab Jira integration plugin. This is done because developers feel more familiar working with GitLab and managers with Jira. This forms a bridge between developers and managers and frees both from learning yet another tool. From GitLab, developer can select an issue to work on. When the work is finished, the developer commits the code to Git with the issue tracking identification tag to specify which issue developer has solved. The commit starts Jenkins which runs automated tests and publishes the build status to GitLab and Jira. In GitLab the build status is linked to the relevant commit and in Jira the build status is linked to a correct project and issue. This is done to make the issue progress and status more visible to developers and managers. This tool linking is made possible by all the integration plugins that Git, GitLab, Jenkins, and Jira offer to make the programs work seamlessly with each other.

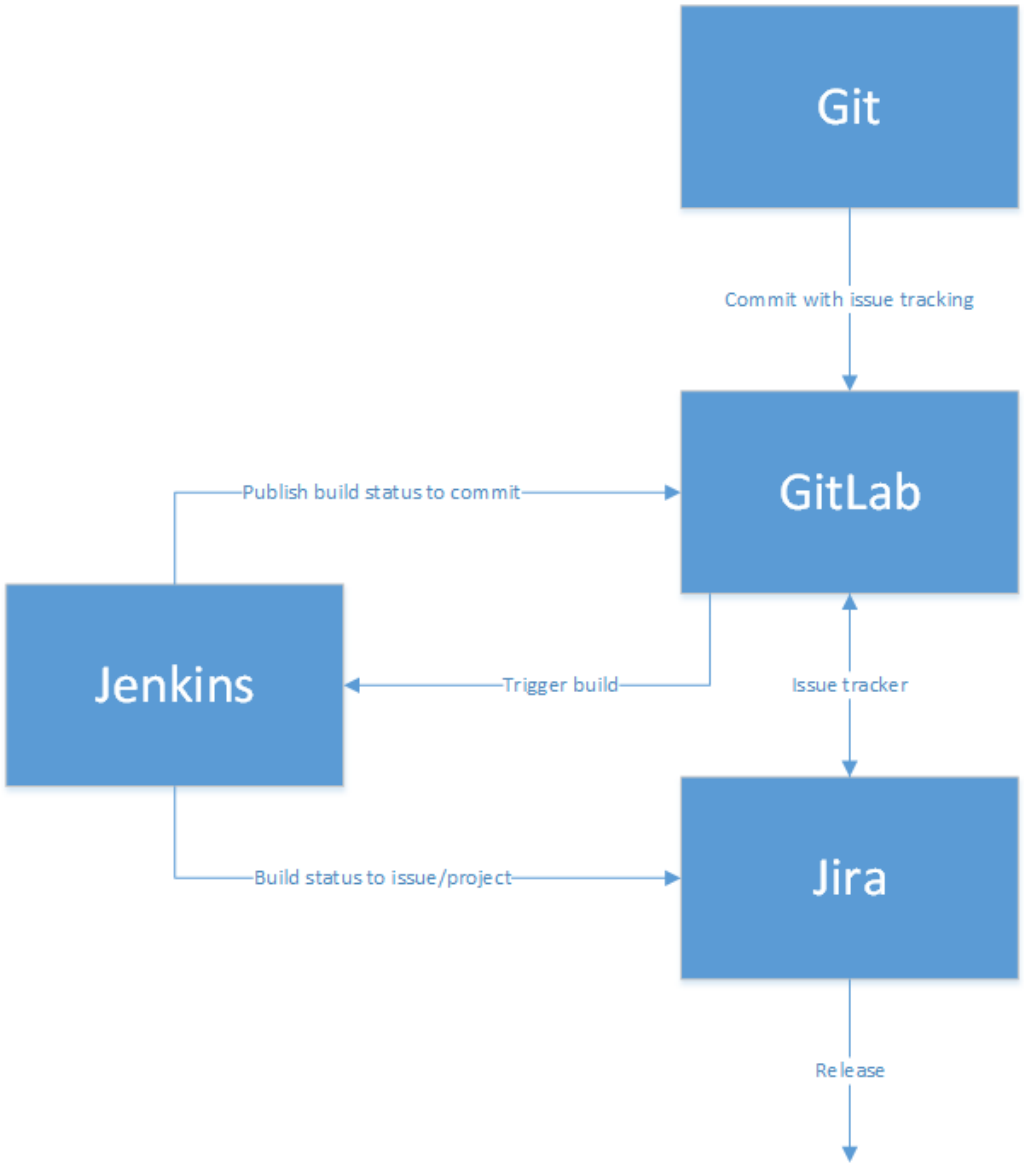

Figure 17 Tool linking

A dedicate screen is a great way to improve visibility within the team when it is reserved to show the status of the builds and the task list. When the screen is positioned so that everyone in the team can see it with one glance, it makes it easy to track the status of the builds and it saves time since people do not need to go searching for Jenkins from their bookmarks.

5. RESULTS AND ANALYSIS

This section presents the improvement results of the implementation and the improvement ideas that were found during the process. In general, the use of continuous practices in SoC software team has improved. All the IPs have been implemented into the continuous pipeline. This has improved the testing frequency, reliability and increased the level of automation. Some improvements to the common practices were also implemented but these will take time to sink into the everyday habits.

All the Loner IPs have been added to the continuous integration server and are building and testing automatically when a commit is pushed to the Git which then activates the GitLab webhook that has been set to start the appropriate Jenkins job. Jenkins job builds the artefacts and runs the analysis tools automatically. Then Jenkins starts the integration tests that are made by Robot Framework, which has improved the visibility of the testing. After the testing is completed the artefacts are deployed to the SoC SW repository. The deployment to the customer is still done manually because SoC SW did not want to send every commit to the SoC Application unit. Changes are committed to the Application units' repository when a sufficient number of commits have been made.

Figure 18 is the same figure (Figure 7) that was previously shown except now it has more automated steps. These new automated steps have now been implemented into all Loner IP blocks.

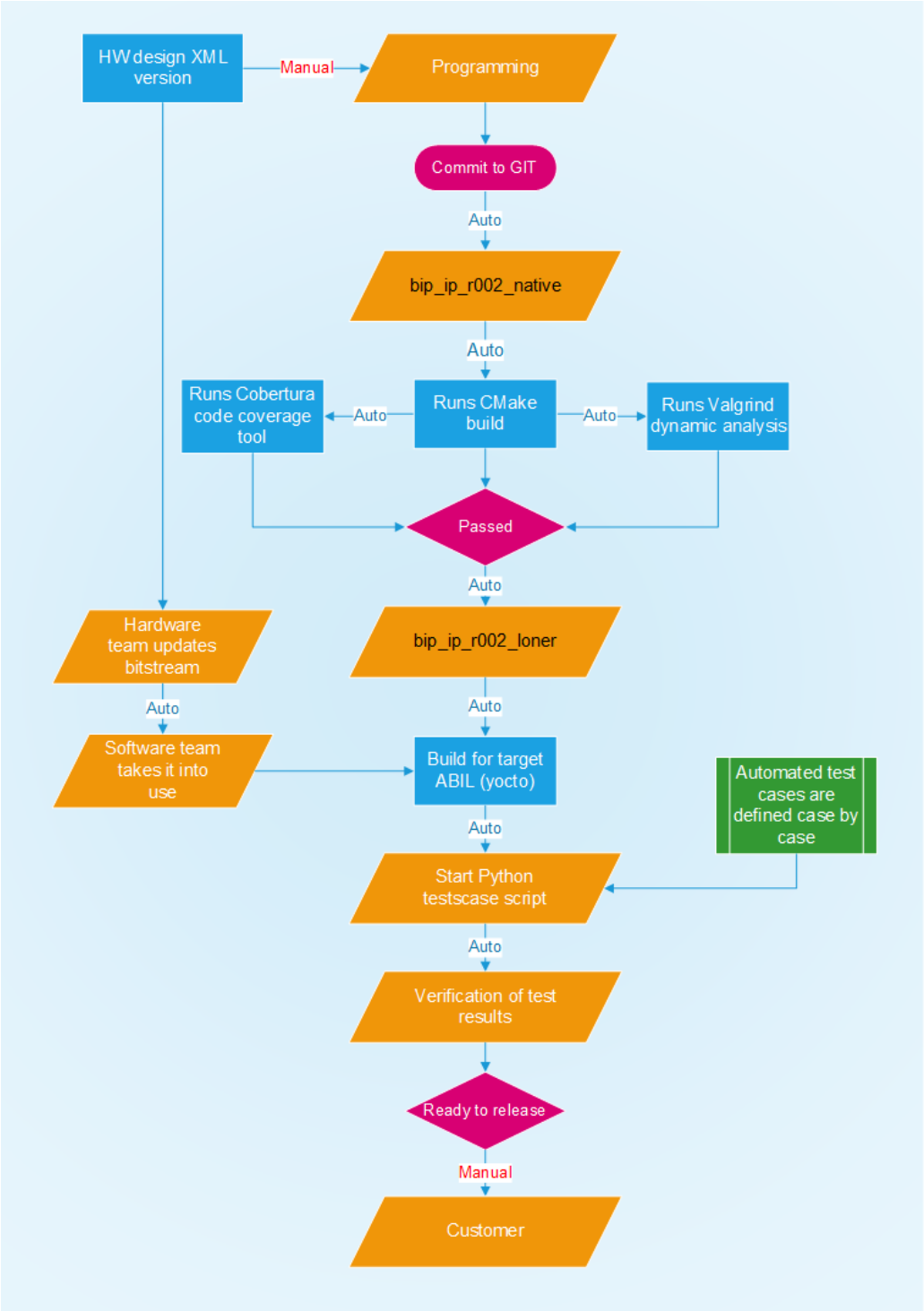

Figure 18 Workflow

During the current state analysis and implementation phase, the goal was to find factors that are hindering the full utilization of continuous practices. As a result, five main reasons were found that were preventing the full potential of continuous practice:

1. Lack of resources
2. Lack of investment
3. Amount of automated tests
4. Old working habits
5. Information loss between teams and team members.

5.1 Resources

One big bottleneck in testing is the needed test hardware, the FPGA boards. Testing on hardware with limited processing capability itself is slow and tests might take up to an hour. This combined with the fact the test laboratory is currently having only one FPGA board to test on causes long queuing times for testing. This is, in the sense of continuous practices, unacceptable. This could be solved by investing more in test hardware so that there would be multiple FPGA boards to test on. Three boards would be sufficient to start with. One for Jenkins master jobs, one for developer test jobs and one for manual testing and development.

Although this solution would be valid now, problems might arise again in the future when there is a need for a new type of FPGA boards. This would create the need to order and buy new FPGA boards and the old ones would become redundant. This costs money and creates waste. There is a possibility to solve this hardware limitation by virtualizing FPGAs in the cloud.

5.2 Investment

In the nowadays hectic software industry and in SoC SW, too, the developers have their hands full all the time with the endless flow of multiple projects and supporting new employees. On top of that, they would need to plan the use of continuous practices. Implementing a continuous pipeline is not something that the developers are used to or have much experience with. This causes extra stress, since implementation requires a lot of studying and the limited time they have is hardly enough for learning about new procedures. This results in fragmented pipelines and quick fixes. This implies that more resources and investment are needed to adopt a fully automatic and continuous pipeline within a sensible time frame. In the company size of Nokia Networks, it might be even feasible to have a few people dedicated to planning and implementing continuous practices for different projects and organizational units to speed up the process. They would set up the continuous pipeline, so the developers and testers could focus more on programming and making tests.

5.3 Automated tests

There is a need for more tests to gain full confidence in automated tests. This is an issue with many continuous practices projects since they need a great number of tests, to release reliable software automatically and confidently. This issue will most likely be solved with time since developers make more tests all the time. It is also very important that the tests are reliable and stable, so they only fail if they are meant to fail and not randomly, for example when test environment is running slowly, and timeout is set too short. Otherwise, the mentality towards failed builds becomes too loose.

5.4 Working habits

During the planning and implementation of the new automated steps, it was noticeable that there was still some doubt towards a fully automated system and all the possible benefits achieved by it. This might be due to the lack of knowledge about continuous practices since many of the developers have not used continuous practices before. It is common not to believe the benefits before experiencing them personally.

There are also still noticeable remnants of the old traditional delivery process in the work culture and habits. One example is that code freezes are still sometimes used. Since there are multiple projects using the same IP, the one that gets frozen gets left behind in the development and creates its own branch which must be supported after the freeze is lifted. Although the freeze is not the only reason why there are currently many versions of the same IP being used within the company. There are also other teams that use the different version because that works for them. Even so, since the development is done in IP blocks the best practice is to support only one version of certain IP. This version should be in all the applications without exceptions.

Trying to support multiple projects or releases that use the same IP but different versions of it, cause confusion, stress and extra work to the developers and the managers. This drives them away from their original work of developing new features. The idea of continuous practices is to have one clear path of development and always releasing the latest versions. The current way is going against this and hindering the implementation of the continuous practices.

5.5 Communication

The communication between different teams needs to be improved. Currently, it seems that teams are developing only parts assigned to them and not the whole product. The work is considered done when it passes teams' own tests, without thinking about the next steps. The information and the knowledge are lost between the teams. There was a case in which a vital part of the software had been changed, but the information was never given to the SoC SW team. After a month SoC SW noticed that they had been using the

old system, which of course caused huge work overload. In smaller cases, information might take days to arrive, which is still days too many.

There is a lot of knowledge among the teams, but somehow it stays inside the team and does not spread in other parts of the organization. This causes that the same thing is invented, made and studied multiple times in multiple places. It was not only once that SoC software team found out that their new idea had been already implemented in another team.

The whole team mentality should be changed when it comes to a continuous pipeline because it encourages silo thinking. There could be only one continuous pipeline for the project where all the teams would contribute their input collectively to support cross-functional aspects. Teams should design the automated pipeline together, possibly with the help of continuous implementation team who would take care of the basic setup of the pipeline. Each team could have designated members to meet and further develop the pipeline based on the needs of their teams. This would bring more visibility to the developers and to managers to see the steps and status of the project more clearly. It would also remove overhead and repetition. If the tests would be designed and planned together in the same way, it could make the testing faster and more robust.

6. CONCLUSIONS AND FUTURE WORK

It has become evident that implementing continuous practices into a domain with embedded systems is not so straightforward as to a web-service application domain. There are hardware and mechanical aspects that need to be taken into consideration. There are also constraints in computing capacity and hardware accessibility. These make the testing slower than in other applications, which makes continuous practices harder to implement in the way it they were originally intended to. This means that the working methods need to change more than in the domains without hardware and the principles of continuous practices needs to bend a little.

6.1 Research process

When starting this thesis, three objectives were defined by Nokia networks. The first objective was to "Analyse the state of the continuous practices and find ways to improve them." The second objective was to "Clarify the process of the continuous practice to see the benefits of it". The third one was to "Increase the use of continuous practices and improve their automated functions". Then from these three objectives, two research questions were formed.

The first research question, "Based on the literature, how have the continuous practices been implemented and what kind of benefits have been achieved?" was answered by a literature review. In this review, different researches on the best practices and implementations of continuous practices were discovered. Based on the research found in literature the theory behind continuous practices was clarified, and the achievable benefits linked to studies where companies had implemented continuous practices.

The second research question, "How can SoC development improve its continuous practices?" was answered by current state analysis. The analysis was conducted with three selected key focus points in mind: increasing the level of automation, increasing visibility and improving feedback. During this analysis, manual steps were discovered and improvement ideas compiled. In the implementation phase some of these ideas were applied and from others, implementation plans were formed. The visibility was improved by implementing more graphical results and clearer logging. The level of automation was increased by adding integration testing and more IPs to the continuous integration server.

There were challenges that were faced during this thesis. Some were expected; like time restriction to detect sustainable results and the small focus group. Others, like finding information and speed of change, were little unexpected but apparently quite common in a large multinational company. The telecommunication domain also had its own challenges. Nonetheless, the thesis project went smoothly, and all the research questions were

answered. Therefore, the selected research method, constructive research, was appropriately chosen for this thesis.

6.2 Future work

During the process, few possible new subjects arose. These subjects were not thoroughly discussed in the scope of this thesis but could be focused more on the future work.

One would be the implementation of a bigger Jenkins pipeline that would involve all the dispersed organizational units. This could have the potential to improve visibility and feedback time at the whole organizational level. However, this is an ambitious plan and would require the backing from the top level of the organization.

Another one would be finding a modular and reusable way of creating new Jenkins pipelines. This would at least require changing Jenkins pipeline web-interface configurations to a newer “Pipeline as Code”-format and storing them in a repository in source code management. Then the configuration settings would be safely backed up in the repository and possibly there would be one less interface to learn since the pipelines could be configured from Git.

One topic that came up often during the process was the use of containers such as Docker or Kubernetes in continuous practices. The potential in using one of these technologies could also be a good subject for future work.

REFERENCES

[1] A. Oyegoke, The constructive research approach in project management research. Published: International Journal of Managing Projects in Business Volume: 4, Issue: 4, pages 573-595, 2011.

[2] M. Shahin, M. A. Babar, L. Zhu, Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. Published: IEEE Access Volume: 5, 2017, Date of Publication: March 22, 2017.

[3] D. Stahl, T. Martensson, J. Bosch, Continuous practices and devops: beyond the buzz, what does it all mean? Published: Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on 30 Aug.-1 Sept. 2017.

[4] M. Shahin, M. A. Babar, M. Zahedi, L. Zhu, Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges. Published: Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium. Date of Publication: 9-10 Nov. 2017.

[5] P. M. Duvall, S. Matyas, A. Glover, Continuous integration: improving software quality and reducing risk. Published: Pearson Education, 2007.

[6] M. Meyer, Continuous Integration and Its Tools. Published: IEEE Software Volume: 31, Issue: 3, May-June 2014, Date of Publication: 21 April 2014.

[7] M. Fowler, Continuous Integration. Available: https://www.martinfowler.com/articles/continuousIntegration.html. Accessed: 21.03.2018.

[8] J. F. Smart, Jenkins: The Definitive Guide: Continuous Integration for the Masses. Published: O'Reilly Media, Inc., 2011.

[9] M. Fowler, Continuous Delivery. Available: https://martinfowler.com/bliki/ContinuousDelivery.html. Accessed: 21.03.2018.

[10] S. Rossel, Continuous Integration, Delivery, and Deployment. Published: Packt Publishing, 2017.

[11] S. Neely, S. Stolt, Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). Published: Agile Conference (AGILE), 2013, Issue Date: 5-9 August 2013.

[12] S. Chacon, B. Straub, Pro Git. Published: Apress, 2014. Available: https://git-scm.com/book/en/v2. Accessed: 21.03.2018

[13] B. Collins-Sussman, B.W. Fitzpatrick, C.M. Pilato, Version Control with Subversion. Published: 2011. Available: http://svnbook.red-bean.com/en/1.7/. Accessed: 23.03.2018

[14] S. McIntosh, Build System Maintenance. Published: Software Engineering (ICSE), 2011 33rd International Conference, Date of Conference: 21-28 May 2011.

[15] S. McIntosh, B. Adams, A.E. Hassan, The Evolution of Java Build Systems. Published: Empirical Software Engineering, (Volume 17, Issue 4–5, pp 578–608), Springer, August 2012.

[16] O. Kupka, F. Zavoral, Cider: An Event-Driven Continuous Integration Server. Published: Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual, Date of Conference: 21-25 July 2014.

[17] O. Taipale, J. Kasurinen, K. Karhu, Trade-Off Between Automated and Manual Software Testing. Published: International Journal of System Assurance Engineering and Management, (Volume 2, Issue 2, pp 114–125), Springer, June 2011.

[18] B. Adams, S. McIntosh, Modern Release Engineering in a Nutshell - Why Researchers Should Care. Published: Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference, Date of Conference: 14-18 March 2016.

[19] G. Versluis, Xamarin Continuous Integration and Delivery: Team Services, Test Cloud, and HockeyApp. Published: Apress L. P., 2017.

[20] B. Homès, Fundamentals of Software Testing. Published: John Wiley & Sons, Incorporated, 2011.

[21] M-C. Gaudel, Formal methods for software testing. Published: Theoretical Aspects of Software Engineering (TASE), 2017 International Symposium, Date of Conference: 13-15 Sept. 2017.

[22] http://softwaretestingfundamentals.com/. Accessed: 04.04.2018.

[23] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Published: Addison-Wesley, 2010.

[24] D. Graham, Foundations of Software Testing: ISTQB Certification, Chapter 4 - Test Design Techniques. Published: Cengage Learning, 2008.

[25] W. E. Lewis, Software Testing Techniques: Software Testing and Continuous Quality Improvement. Published: Auerbach Publications, Third Edition, 2009.

[26] M. Leppänen, S. Mäkinen, M. Pagels, V-P. Eloranta, J. Itkonen, M. V. Mäntylä, T. Männistö, The highways and country roads to continuous deployment. Published: IEEE Software (Volume: 32, Issue: 2, Mar. - Apr. 2015).

[27] L. Chen, Continuous Delivery: Huge Benefits, but Challenges Too. Published: IEEE Software (Volume: 32, Issue: 2, Mar. - Apr. 2015).

[28] A. Miller, A Hundred Days of Continuous Integration. Published: Agile, 2008. AGILE '08. Conference, Date of Conference: 4-8 Aug. 2008.

[29] S. Hung-Lung Tu, D-L. Shen, R-J. Yang, Analog Circuit Design for Communication SOC. Published: Bentham SciencePublishers, 2012.

[30] M. Bakloutiab, Ph. Marquetc, J.L. Dekeyserc, M.Abidab, FPGA-based many-core System-on-Chip design. Published: Microprocessors and Microsystems Volume 39, Issues 4–5, June–July 2015, Pages 302-312.

[31] C. Maxfield, The Design Warrior's Guide to FPGAs: Devices, Tools and Flows. Published: Elsevier Science & Technology, 2004.

[32] R.C. Cofer, B.F. Harding, Rapid System Prototyping with FPGAs: Accelerating the Design Process. Published: Elsevier Science & Technology, 2011.

[33] E. Monmasson, M.N. Cirstea, FPGA Design Methodology for Industrial Control Systems. Published: IEEE Transactions on Industrial Electronics Volume: 54, Issue: 4, August 2007. Date of Publication: 09 July 2007.

[34] FPGA manufacturers, http://hardwarebee.com/list-fpga-companies/. Accessed: 14.08.2018.

[35] I. Kuon, R. Tessier, J. Rose, FPGA Architecture. Published: Now Publishers, 2008.

[36] D.A. Patterson, J.L. Hennessy, Computer Organization and Design: The Hardware/Software Interface. Published: Elsevier Science & Technology, 2007.

[37] R. Saleh, S. Wilton S. Mirabbasi, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, A. Ivanov, System-on-Chip: Reuse and Integration Pre-designed. Published: Proceedings of the IEEE Volume: 94 Issue: 6, June 2006.

[38] Intellectual property, https://whatis.techtarget.com/definition/IP-core-intellectual-property-core. Accessed: 20.11.2018.

[39] L.E. Horta, J.W. Lockwood, Automated Method to Generate Bitstream Intellectual Property Cores for Virtex FPGAs. Published: Field Programmable Logic and Application, Springer 2004.

[40] Git, https://git-scm.com/. Accessed: 01.05.2018.

[41] F. Santacroce, Git Essentials - Second Edition. Published: Packt Publishing, 2017.

[42] GitLab, https://gitlab.com/help. Accessed: 03.05.2018.

[43] J. Hethey, GitLab Repository Management. Published: Packt Publishing, 2013.

[44] Jenkins, https://jenkins.io/doc/. Accessed: 03.05.2018.

[45] N. Pathania, Learning Continuous Integration with Jenkins. Published: Packt Publishing, 2016.

[46] CMake, https://cmake.org/overview/. Accessed: 08.05.2018.

[47] B. Hoffman, D. Cole, J. Vines, Software Process for Rapid Development of HPC Software Using CMake. Published: DoD High Performance Computing Modernization Program Users Group Conference 2009 (HPCMP-UGC), Date of Conference: 15-18 June 2009.

[48] Yocto Project, https://www.yoctoproject.org/about/. Accessed: 08.05.2018.

[49] O. Salvador, D. Angolini, Embedded Linux Development with Yocto Project. Published: Packt Publishing, 2014.

[50] A.P. Navik, D. Muthuswamy, Dual band WLAN gateway solutions in Yocto Linux for IoT platforms. Published: Internet of Things for the Global Community (IoTGC), 2017. Date of Conference: 10-13 July 2017.

[51] Robot Framework, http://robotframework.org/. Accessed: 09.05.2018.

[52] S. Bisht, Robot Framework Test Automation. Published: Packt Publishing, 2013.

[53] Coverity, https://scan.coverity.com/. Accessed: 09.05.2018.