University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Bachelor Thesis

Continuous Integration System Implementation

Pilsen, 2020

Auchynnikau Ihar

Místo této strany bude zadání práce.

Declaration

I hereby declare that this bachelor's thesis is completely my own work and that I used only the cited sources.

Pilsen, 21st July 2020

Auchynnikau Ihar

Abstract

The main goal of this work is to design and implement the Continuous Integration (CI) system in the BootLoader department of ZF Engineering Plzeň. The transition to continuous integration reduces the complexity of code integration and makes the development cycle more effective by early detection and elimination of errors and contradictions.

Bachelors thesis describes the process of development and implementation of Continuous Integration system, which includes analysis and description of the build and test tool chain used in the Bootloader department, architectural design, choosing appropriate technologies and their implementation. The thesis also describes possible extension of the CI system.

Abstrakt

Cílem této práce je navrhnout a implementovat systém Continuous Integration v oddělení BootLoader společnosti ZF Engineering Plzeň. Přechod na CI snižuje složitost při integrace kódu a umožňuje jej efektivněji vyvíjet včasným odhalením a odstraněním chyb.

Bakalářská práce popisuje proces vývoje a implementace CI systému, který zahrnuje analýzu a popis toolchainu používaného v Bootloader oddělení. Součástí prace je návrh architektury, výběr vhodných technologií, implementaci. Práce také popisuje možné rozšíření CI systému.

Contents

1 Introduction

Nowadays, issues related to continuous integration are very relevant. This practice of software development allows to consolidate working copies into a common main development branch several times a day and perform frequent automatic builds of the project to identify potential defects quickly and solve integration problems.The transition to continuous integration reduces the complexity of code integration and makes it more predictable by early detection and elimination of errors and contradictions.

The main goal of this work is to design and implement the Continuous Integration (CI) system in the BootLoader department of ZF Engineering Plzeň. It will allow to reduce the cost by fixing the code defect due to its early detection, speed up code development by means of building and testing on the servers and also increase the quality of the code.

Bachelor's work will be performed on the basis of an analysis of the accepted Continuous Integration and Continuous Delivery approaches in ZF Engineering and the literature on this issue.

This work consists of five chapters. The first chapter describes the principles, concepts, and tools used in continuous integration and continuous delivery. The second chapter describes and analyses the build and test tool chain used in the bootloader team. The third chapter is dedicated to designing a CI system. The fourth chapter describes the creation of a CI system, as well as tools and scripts, created during the development of this system. The final chapter discusses future extensions for this system.

2 Principles, concepts and SW tools

This chapter describes the principles, concepts, and tools used in continuous integration and continuous delivery. Understanding the principles of CI / CD is necessary for further realization of bachelor's thesis.

2.1 Continuous Integration and Continuous Delivery

Continuous integration (Figure 2.1) a practice in which software developers continuously or frequently integrate their work with that of other members of the development team [11].

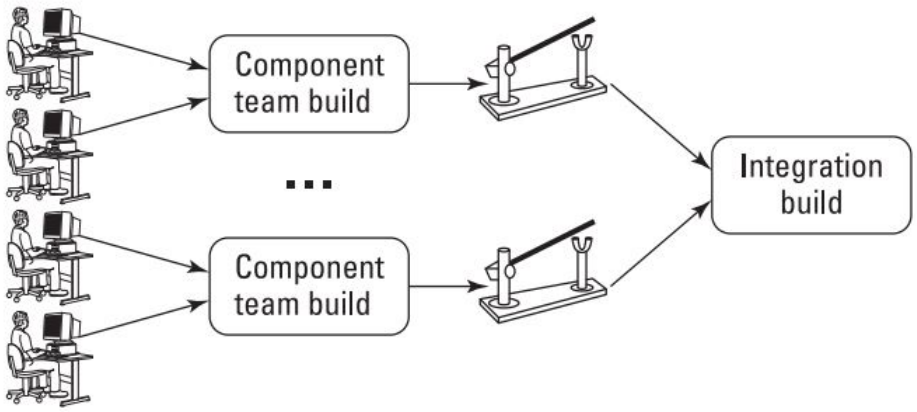

Figure 2.1: Collaboration via continuous integration

Continuous integration is a software development practice in which developers routinely combine software code changes in a central repository, after which build, testing, and launch are automatically performed. The concept of continuous integration is most often applied to the build or integration stage of the software release process and includes both the automation component (for example, the continuous integration or build service) and the development culture component (for example, learning to integrate frequently). The main objective of continuous integration is to quickly find and fix errors, improve software quality and reduce the time spent on checking and releasing new software updates [12].

To speed up CI, it's convenient to parallelize tests on some powerful platform.

Business value of CI:

- Accelerate Delivery - achieved by the fact that we immediately find out about the build error and, accordingly, we can begin to fix it faster.
- Repeatability - the whole process is repeatable, that is, if no changes have occurred, then the assembly will also be successful (or not successful).
- Automation - there is no need to manually run the assembly, on a person's computer or build server, there is no need to prepare the assembly - pump out the sources from source control, etc.

Continuous integration naturally leads to the practice of **continuous delivery**: the process of automating the deployment of the software to the testing, system testing, staging, and production environments (Figure 2.2). [11]

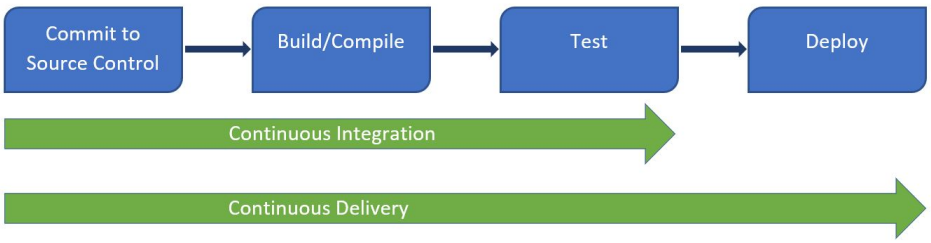

Figure 2.2: Continuous integration and continuous delivery

Continuous delivery is one of the fundamental principles of the development of modern applications as it extends the practice of continuous integration due to the fact that all code changes after the build stage are deployed in a test and/or working environment. With proper implementation, developers will always have a ready-to-deploy built software instance that has passed the standardized testing procedure.

Continuous delivery allows developers not only to automate testing at the module level, but also to perform diverse checks of application updates before deploying them to end users. Such testing may include testing the user interface, loading, integration, API reliability, etc. All this allows developers

to check for updates more thoroughly and identify potential problems in advance [12].

Basically, for a continuous delivery process you need to perform manually at least one step: approve the deployment in a production environment and run it. The continuous delivery pipeline can include additional steps, either manually or automatically In complex systems with many dependencies[2].

2.2 Tools for CI/CD

This sub-chapter will describe and analyze the tools that will be used in the process of continuous integration and continuous delivery. In this chapter there will be no comparison and selection of tools for the created project, since the tools, that are the standard for the tooling and bootloader teams will be used. Tooling team is engaged in continuous integration in ZF Engineering Plzeň.

2.2.1 Jenkins

Jenkins is an open source automation tool. Plugins to enable continuous integration are part of Jenkins. Jenkins is mainly used to implement continuous build and testing of projects, this makes it easier for developers to integrate changes into the project. This approach also makes it easier for users to get a new build. Jenkins enables continuous software delivery for integration with a wide range of testing and deployment technologies. This product is written in Java.

With Jenkins it's possible to speed up software development through automation. Using Jenkins it is possible to combine all kinds of software development life cycle processes, including build, documentation development, testing, packaging, deployment, static analysis, and much more.

Continuous integration in Jenkins is achieved through plugins. There are plugins for integrating specific tools. For example, Git Client, Python, HTML Publisher, Dashboard View, etc. If the required plugin does not exist, the programmer can create it by himself, which is supported by the Jenkins community. There are many tutorials and videos for creating plugins for Jenkins [8].

Features:

- Jenkins is a stand-alone Java application that can run on Windows, Mac OS X, and other unix like operating systems.

- Hundreds of plugins can be found in the Update Center, so Jenkins integrates with almost any tool related to continuous integration and continuous delivery.
- The possibilities of Jenkins can be almost unlimited expanded thanks to the plug-in connection system.
- Various modes are available. They are Freestyle project, Pipeline, External Job, Multi-configuration project, Folder, GitHub Organization, Extensive Pipeline.
- The next feature is Jenkins Pipeline. According to the official Jenkins web page, Jenkins Pipeline is a set of plugins that supports the implementation and integration of continuous delivery pipelines in Jenkins. A continuous delivery pipeline is an automated expression of the software transfer process from version control to users and customers.
- Jenkins allows to run builds with various conditions.
- Jenkins can work with Libvirt, Kubernetes, Docker, etc.
- Using the REST API, developer can control the amount of data received, update config.xml, delete jobs, receive all builds, receive/update job description, perform builds, enable/disable tasks [7].

Advantages:

- It is an open source tool with community support.
- It has over 1000 plugins to facilitate the work. If the plugin does not exist, then it can be easily created.
- It is free.
- Jenkins is built on Java and therefore works on all major platforms.

Disadvantages:

- Jenkins is a dedicated server (or several servers) which is required. It entails additional costs for the server itself, etc.
- It takes time to set up Jenkins.
- Installing a large number of plugins can lead to maintenance problems and sometimes Jenkins does not even restart successfully after updating the plugin.
- There are many limitations on the Jenkins API.

2.2.2 Bitbucket

Bitbucket is a web service for hosting projects and their joint development, based on the Mercurial and Git version control system. Bitbucket has an improved code preview system, which requires less time for a complete exchange of information (time to reverse the direction of data transfer), as well as for a request to include a code.

Bitbucket integrates with Jira and creates a system for bug tracking.

It allows users not to leave the current tool in order to receive a report on the status of the ticket or to correct an error. Bitbucket has built-in chats in which users can exchange messages and leave comments [3].

The ZF is currently transitioning from ClearCase to Git, so new projects like CI are using Git. Older projects use ClearCase, but they are gradually moving to Git.

2.2.3 Jira

Agile development methods are used in the tooling and bootloader teams. Such methods are a series of approaches to software development focused on the use of interactive development and dynamic formation of requirements to ensure their implementation as a result of constant interaction within self-organizing working groups consisting of specialists in various fields.

Jira Software is an agile project management tool that supports any agile methodology such as Scrum, Kanban and so on.

Scrum is one of the agile methodologies in which a product is created as a series of iterations of a fixed duration. The structure of this platform consists of four components. They are sprint planning meetings, stand-ups (daily Scrum meetings), sprints, and retrospectives [9].

This methodology will be used In the developing of the CI system. All tasks are necessary to create a continuous integration system and they will be recorded in the Jira. Sprint planning and evaluation of the past sprint will be held every two weeks. Comments will be written to all tasks as they are completed.

2.2.4 Groovy

Groovy is an object-oriented programming language developed for the Java platform as an alternative one to the Java language with the capabilities of Python, Ruby and Smalltalk.

Groovy uses Java-like syntax with dynamic compilation of bytecode in the JVM and works directly with other Java codes and libraries. The lan-

guage can be used in any Java projects or as a scripting language [6].

Groovy features (distinguishing it from Java):

- Is a scripting language.
- Has a static and dynamic typing.
- Contains built-in syntax for lists, associative arrays, arrays and regular expressions.
- Short circuiting mechanism exists in Groovy.
- Contains overload operations.

The main advantage of this language is its deep integration with Jenkins.

2.2.5 Library lib_groovy.jar

This library was developed by the tooling team and it is a set of scripts to facilitate the development of CI projects.

This library makes it easy to work with ClearCase, configuration files, emails, executable files, logs, and so on. ClearCase will be described below.

The library is written in Groovy, supplied as a jar file and is periodically updated. When developing a CI system, functions and modules will be added to this library.

3 Analysis and description of the build-and-test tool chain used in the Bootloader department

Since the CI system for such team as a bootloader will be created for the first time, it was necessary to gain knowledge about how the development cycle in this team occurs, what structure the software has, what it is used for, analyze the team wishes of the automation and their expectations from creating the CI system.

For this, before the start of development, several meetings were held with the bootloader team. At the meetings, wishes were heard for a system of continuous integration, options were proposed for how it would look. In addition to the meetings, an introduction was made to the structure of the team, to development approaches in this team, to the developed software, to the programs that are used by this team.

This chapter will describe and analyze the tool chain used in the bootloader team to develop software.

3.1 Basic information about the bootloader team

The bootloader team develops and tests a boot software for the AURIX microcontrollers. This software allows customers to upload and run their applications on microcontrollers received from ZF Engineering Plzeň. There are some programming languages, for instance, C, C++, Python,which are used for software development in the team.

Bootloader team use hardware from companies Vector and Lauterbach.Such programs as Lauterbach Trace32, Vector CANape, Vector CDM Studio are used For build-and-test in the Bootloader team. Programs,developed in the tooling team are used to sign software. ClearCase is used for the storage of versions .

3.2 Description of the current build-and-test tool chain

Further, the described tools are used by developers from the bootloader team and will be presented on the VDI or VOBs. The basics of working with these tools are key prerequisite for the correct implementation of the CI system.

3.2.1 AURIX microcontrollers

Microcontrollers are an integral part of any modern machine. For example, 10-year-old BMW 7 has from 60 to 65 microcontrollers. Microcontrollers control ECU functions such as braking, steering, power for windows and seats, headlights and taillights as well as safety check to monitor other active microcontrollers. These data show the importance and necessity of developing microcontrollers and software for them.

AURIX (Automotive Realtime Integrated NeXt Generation Architecture) is the 32-bit Infineon family of microcontrollers designed for the automotive industry. AURIX was designed to meet the highest safety standards. Its multi-core architecture is based on the use of up to three independent 32-bit TriCore processors [1].

Minimum knowledge of microcontrollers' origin and purpose allows us to understand the essence of this project. These microcontrollers with the software developed by the bootloader team are used in a large number of cars of well-known brands.

3.2.2 Lauterbach Trace32

The Trace32 debugger makes it possible to test embedded hardware and software by using the on-chip debug interface. A single on-chip debug interface can be used to debug all cores of a multi-core chip such as AURIX microcontroller. Debugging an Infineon TriCore device requires a Lauterbach Debug Cable together with a Lauterbach Debug Module.

This program is applied in conjunction with with CANape for testing software on a physical unit.

3.2.3 Vector CANape

CANape is a software tool from Vector Informatik. This developing software is widely used by OEMs (original equipment manufacturer) and ECU

(electronic control unit) suppliers of automotive industries. The main aim of the usage is to calibrate algorithms in ECUs at runtime.

The parameters of a control algorithm can only be determined in a limited extent by using a laboratory model. The algorithms of the functions are a permanent fixture in the ECU program. Parameter values such as characteristic maps and curves can only be determined and optimized by measurements at the test bench and in driving trials. Solving these challenging ECU development tasks is possible with a CANape [4].

The bootloader team uses the CANape while testing the software, sending commands to the microcontroller with preinstalled developed software and analyzing the output data. Also, bootloader team uses the CANape for loading the software to the memory of the microcontrollers.

A CANape license is required to use CANape. The number of licenses is limited, which will affect the design of the CI system.

3.2.4 Vector CDM studio

Vector CDM studio is an efficient tool for editing parameters and setting files. It is used to display, compare and edit parameters created in ECU calibration. Filters are used to reduce the number of parameters shown on the screen in the process of solving complex tasks. In addition to calibrating parameter values, it can take values from different files and merge them to create new version levels [10]. Vector CDM studio is installed with Vector CANape, as these programs are the part of the same package.

This program allows the Bootloader team to change parameters in an already built project without rebuilding it again. That in turn allows to build the project once, and later on when testing or providing ready software to the customer it allows to customize it quickly . This approach reduces the time between the customer's request to provide the software with the new parameters and to provide this software directly . It also helps to avoid errors that may occur at the stage of build of the project under certain circumstances.

3.2.5 Software signature tool and Gate Keeper

The entire ZF group uses SW tools developed by ZF Engineering Plzeň to sign the software. This tools let to count cyclic redundancy check (CRC) and to sign software developed in ZF. This in turn guarantees the authenticity of this software. The programmer can configure the rights to the end user of the software by combining software signature tool and CDM studio.

Gate Keeper is used to connect and maintain a connection with a key server. In the case of using the Software signature tool the user must be connected to the key server with the help of Gate Keeper. This server allows you to generate new keys, certificates, and also allows you to determine variety of rights the user has.

3.2.6 ClearCase

ClearCase is a version control system developed by IBM's Rational Software division.

ClearCase is a tool for managing the process of creating a software product. Management is carried out by controlling versions of all files included in the project, providing administrators, developers and managers with full information about the current status of the project. Simply put, the program creates a special database (VOB) in which it stores all the accompanying information about all files with their versions that were put under control. The accompanying information also indicates the person and the time of changes which were made to a specific version of the project file or directory. The entire history of changes is displayed in graphical form, in the form of a version tree [5].

Views are used to work with files in ClearCase. View is the workspace in which changes are made to files. In Windows Explorer the view looks like any other folder. Configspec is used for creating a view.

The configspec is a text file that contains rules for selecting versions of elements whish should appear in a view. ClearCase was one of the first SCM (Source Code Management) products that included not only version control capabilities, but also workspace management with the usage of an management using an invariant approach with the help of which developers can elaborate high-quality software products in short terms and support existing products without getting complexity in their versions. The main difference from other version control systems is not only the fact that ClearCase has a more global approach to solving the task of version control, but also that this program use special utility "oMake" to assemble the project into an executable file.

This tool is used at all stages of development since it allows all developers to have access to the latest version of the software and make changes to it. Also, the use of this tool saves the developer from losing the already written code in the event of a software error or hardware breakdown. It allows you to go back to the previous version, for example, if a critical error is detected in the current one.

3.3 Analysis of the current build-and-test tool-chain

Currently, the first (1G) and second (2G) AURIX microcontrollers of two generations are used. These two generations are different in structure, what leads to differences in the software developed for them. A VOB has been created for each AURIX generation. These VOBs contain all versions of the boot software from bootloader team. These two VOBs are identical in structure. The software for the first and second generation has the same structure, despite being different. These factors allow a person working with the first generation to switch to work with the second one quickly and vice versa. These factors will also create one CI system for two generations. The differences will only be in the configuration files.

Software development in the Bootloader team consists of three main steps: build, testing and release. Deliverables are created from source codes. Deliverables are executable files, configs, SQL scripts, etc. Build is work products received from source codes. It is created both manually on demand and by automated assembly systems on a schedule.

A release is a build that a development team provides to a consumer. Both, a team of testers and users can be in the role of a consumer of the release. An internal release is what is given to the consumer within the company or team accordingly. An external release is accordingly given out. Each subsequent step includes some parts of the previous one and is impossible if any problems were found in the previous step. Build is carried out every time when something is changed and merged. Testing is done after each build. The release is carried out once in a certain period.

Build, testing and release are currently performed manually. It brings the disadvantages and advantages, which will be described below.

3.3.1 Build

Build is the first step in the build and testing chain. This step allows to create a new version of the software, the reliability and quality of which will be tested in the next step.

In build case, the programmer needs to carry out the following steps:

1. Create a view using the latest version of configspec. This view contains the files necessary for the build as well as the build results.

2. Run batch file that will build the project. This batch file contains calls to other batch files and make commands.

3. Run batch file three more times with the different parameters for generating statistics, lint logs and do make check.

4. Copy all the necessary generated files to local computer.

5. Start with the second step for the next variant. Under the option we can see the build project for another customer. Builds for different customers differ from each other only in their parameters.

The programmer must have ClearCase installed on his local computer to build the project successfuly.

The constant participation of the programmer is necessary according to this approach to the build of the project. The project is built directly on the programmer's computer, which can take all the resources of this machine. That in turn does not let the programmer to do anything else during the project build. The build speed depends on the characteristics of the computer.

3.3.2 Testing

At this step, the software is checked for errors. For example, checking for the user's presence of access rights to parts of the software that he should not have, or vice versa, the absence of any rights. Regression tests are used.

In case of testing, the programmer must complete the first two steps from the build as well as:

1. Launch signature tools for signing the generated versions of the program.

2. Reserve a license for CANape.

3. Run the batch file, which will create different test copies of the software using CDM studio.

4. Run Trace32 and CANape.

5. Use CANape, upload the test version of the software on the AURIX unit.

6. Use CANape run test scripts. Track script execution in Trace32 and control the results in CANape.

7. Follow all the previous steps for the next variant.

A programmer must have the following programs installed on his local computer. they are ClearCase, Vector CANape, Vector CDM studio.

During testing as well as during the build, the constant participation of the programmer is necessary. Also, the programmer must run all the scripts for testing manually and also control their results manually. It can lead to additional errors.

3.3.3 Release

The release must contain the project build, test results and documentation. The release must also be registered in ClearQuest. IBM Rational ClearQuest is a centralized database that records information about detected defects and required changes.

The release is the final and last frequent step in the development of software, but it is also the most time-consuming and responsible part. Software is provided to the customer after release. The programmer must follow all instructions strictly and monitor all steps at the release stage. A mistake at this stage can bring reputational and financial losses to the company.

A release is possible only if there were no errors detected during the build and testing.

3.3.4 Conclusions of the manual approach

Manual approach allows the programmer to create the project as well as generate the minimum number of logs and build statistics.

Advantages of manual approach:

1. The programmer always monitors all stages of the build and testing the project.

2. The programmer responds to the founded error immediately .

Disadvantages of manual approach:

1. The programmer must constantly be near the computer and monitor the progress of the build and testing. It takes a large amount of working time. When the programmer does not need to monitor the progress of the build he nevertheless cannot work on his computer as the build uses all the resources of this machine. Limiting the resource consumption of the build will stretch the time required for the build.

2. Lack of logs and statistics complicates the detection of errors or may prevent from identifying an error.

3. The programmer who makes the release becomes responsible for the entire project.

This approach takes a lot of time that programmers could spend on the development and reduces the quality of progress. All the advantages of a manual approach are leveled out by the correct approach to creating a continuous integration system. By the way, a correctly designed CI system will allow to reduce programmer's responsibility for the performance of the software.

4 Design CI system

The created CI system must meet the following requirements:

- Implement a tool for replacing the manual evaluation in the toolchain by an automated evaluation Wherever it is possible,.
- Evaluate the possibility of parallelization of the builds over multiple Jenkins agents. If the parallelization speed-up the builds, you should implement it.
- Groovy or Java can be used for the implementation of CI . The tools can be standalone command-line executables or a library.

Design is an essential part of any small or large project. For the implementation of a continuous integration system designing is as important step as developing this system. Errors, which were made at the design stage, can fatally affect the submission of the entire system.

This chapter will describe crucial steps required to implement a CI system. An analysis will be made over the possibility of parallelizing parts of the build.

4.1 Replacing the manual evaluation

Jenkins Master must be installed, configured and run on the pre-prepared VDI (Virtual desktop infrastructure) to replace the manual launch of the build and testing. Jenkins Master will launch jobs on other Jenkinses named Agents. Agents will installed on other VDIs. All necessary programs described earlier must be installed and configured On VDIs with Jenkins Agents.

A job (Check Job) will be created in Jenkins that will run every five minutes and monitor the appearance of new versions of the software. Configspec is a trigger that indicates the emergence of a new version of the software. Check Job which is using groovy script will check for a new version of configspec. It is possible to perform job triggering from ClearCase, but it requires a plugin and it is not reliable. When a new version of configspec is detected, another job (Build/Test Job) will be launched. It in turn will build or test the project. It depends on the parameters of the job. The division into two jobs is dictated by the fact that the Build/Test Job can be launched not only by the trigger, but also by the schedule or manually.

Jobs at startup will clone from git configuration files, libraries and code files to perform the required tasks. Job in Jenkins will have a small declarative pipeline that will describe how to clone another pipeline from git. This pipeline will run groovy scripts which in turn will do the main part of the work.

This structure was created because it is necessary to have as little code as possible in Jenkins. That will allow, if necessary, switch from Jenkins to any server in the shortest possible time or in case of any problems with the current Jenkins master will allow to create a new master in the shortest possible time and not lose information from the old one.

4.1.1 Build

A set of groovy scripts and libraries for the Build Job will allow to:

- Create view, mount VOBs, prepare work environment.
- Build the project.
- Generate build log.
- Programmatically analyze the build results and logs, generate statistics.
- Copy generated files to the share.
- Send an email about a successful or unsuccessful build. Mail will contain results of build, statistics and links to result files and folders.
- Remove view and unmount VOBs.
- Delete old results if it is necessary.

Build automation will be the first part implemented. All solutions undertaken at this stage will be subsequently used at the testing and release stage. Since the CI for the bootloader team will be the first CI of this type, there may occur some situations when it is necessary to remodel some parts due to some problems at the implementation stage.

4.1.2 Testing

Testing will consist of two parts. the first is testing on the simulation of a microcontroller and the second is based on a real physical unit. First, the

software will be tested on a simulation. If the tests pass successfully, it will be tested on a real unit.

Testing will consist of two parts because of several reasons. Firstly, the use of simulation will save the resource of the microcontroller, since the memory of the microcontroller has the final possible overwrites. This number is estimated in thousands. Secondly, the number of microcontrollers is limited and they are necessary for developers in their work. Testing will occupy them thereby slowing down the development. The need to use testing on a physical unit exists due to the fact that the simulation may also contain errors. What is more, not all parts of the microcontroller can be made the part of the simulation.

The simulation will be provided by the bootloader team and does not require any additional modifications. The simulation will run on VDI. Simulation program will be located on the bootloader VOB.

A computer with pre-installed programs and an attached microcontroller will be provided for testing on a physical unit.

Test Job has the same steps as Build Job but some of the groovy scripts and libraries will be different.

Jobs created for testing on a real unit and on a simulation will have the same code. The differences will be in the configuration files. This solution will simplify and accelerate the development of CI system.

4.1.3 Release

Although the release is a mandatory part of software development, it will not be implemented as the part of the bachelor's work but it will be implemented later. Before this step is implemented in the development of the CI system, all parts of the build and testing should be fully carried out.

4.2 Evaluating possibility of parallelization

Because when test and build different components the same files are used. Since parallelization was not supposed when developing in the bootloader team, there is no protection in the build code against simultaneous access to files of several variants. To avoid these errors the following solution was proposed. Namely, builds will be parallel but a separate view will be created for each parallel build. This solution ensures that one build will not affect the results of another one (Figure 4.1).

Although each component consists of variants that can also be built in parallel, the variants will be built in series. The reason for this is that the

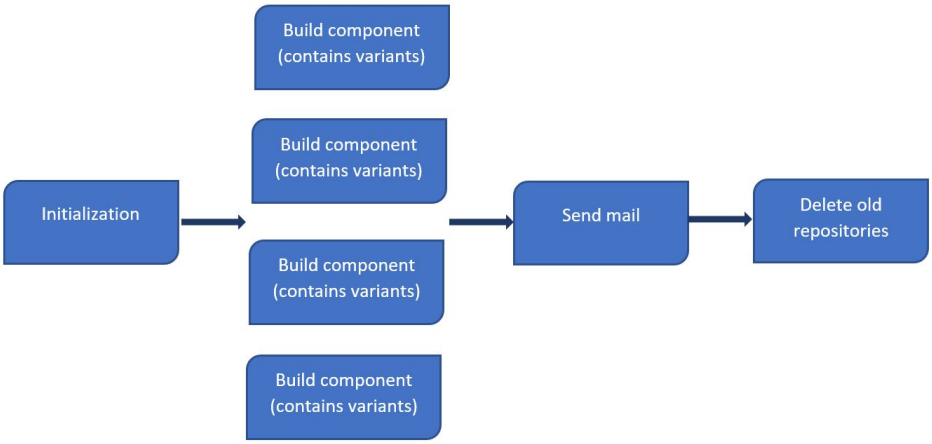

Figure 4.1: Build steps

project is fully built only with the first variant. Next variants will use already created binary files in which only the parameters are changed. The creation of a huge number of views (one view is for each variant) will not lead to significant acceleration, if possible, to start the required number of parallel processes at the same time but will lead to a significant increase in the loading on resources and an increase in the total build time. In the absence of the required number of parallel processes the build time will increase, the total build time will also increase due to switching processors between processes.

Parallelization will be implemented by using pipeline and only with the build. Testing will be carried out step by step because at the moment the software created for testing does not allow parallelization.

5 CI system implementation

To establish the continuous integration system it was necessary to configure the obtained VDI, create the jobs in Jenkins, generate pipelines, write groovy scripts and set up additional tools for processing the results. This chapter will define all of the above.

5.1 VDI configuration

Since the standard set of programs was preinstalled on the obtained VDI, which did not include programs specific for the bootloader, it has been necessary to install them. Moreover, all Jenkins JOBs on VDI run under the pooluser account. Pooluser account is an account that has many different access rights but is not a real user. So that the pooluser can log in to VDI, it should be configured in a certain way.

5.1.1 Accesses

It was necessary For the pooluser to configure an access to the VOBs which contained merged projects and some programs necessary for work. This part of the work was done by the IT department.

It was also necessary to enable RDP access for the pooluser to VDI because Jenkins will be installed under the pooluser account. For this reason the pooluser was added to Remote Desktop Users and Direct Access Users groups.

5.1.2 Programs

The following programs have been installed. they are Vector CANape, Vector CDM studio. Installing these programs was not a trivial task as two versions of CANape had to be installed. This happened because testing requires version 16 of CANape but for editing binary files it was necessary to use CDM studio from CANape version 13. CANape versions are higher than 13. it doesn't provide api for access to CDM studio. ClearCase has been preinstalled. Lauterbach, Trace32, Gate Keeper, Signature tool do not require installation and are located on the bootloader VOBs. However, while developing a continuous integration system the Gate Keeper and Signature tool programs were updated to the latest versions.

5.2 Jenkins installation and configuration

Two VDIs and two instances of Jenkins were used for this project. The first instance is the master which contains jobs and launches them. The second instance is an agent. Only one agent was created, since at this stage the amount of its resources was sufficient. If necessary it is possible to increase the number of agents. Agent is managed by a master. The agent is located on the VDI where the build will run.

Using Master together with agents allow builds on these agents, thereby reducing the load on the master server, perform builds on various software/-operating systems, and simultaneously run different steps of the same build on different Jenkins agents. For example, running parallel builds. Job execution logs from agents are visible on the master. The master also stores the build history.

After installing Jenkins, it was necessary to configure its auto start in the case of rebooting VDI. A reboot may occur, for example, if a software updates on VDI.

5.2.1 Master

A pre-installed Jenkins master was obtained for this project. It was necessary to configure security credentials in pre-installed Jenkins master to connect to the git, to configure the path to the agent as well as to create job for building, testing and release.

The following steps were done to give Jenkins an access to the git repository in which the project was located:

- A personal access token has been added to the Bitbucket (Figure 5.1), which allows Jenkins to pull and clone project from the repository. Personal access token is a substitute for login and password.
- An item containing this token in encrypted form was added to the Jenkins credentials.

Further, the user can use the ID of this token to clone the git repository by using Jenkins. If necessary, the token can be configured in such a way that Jenkins will be able to do the commit.

Then in Jenkins pipeline type jobs were created allowing to build and test the project. Four jobs were created. Two were made for different generations of build and two for different generations of testing. Each job is run according to the schedule via cron. A scheduled launch is used to make sure that the

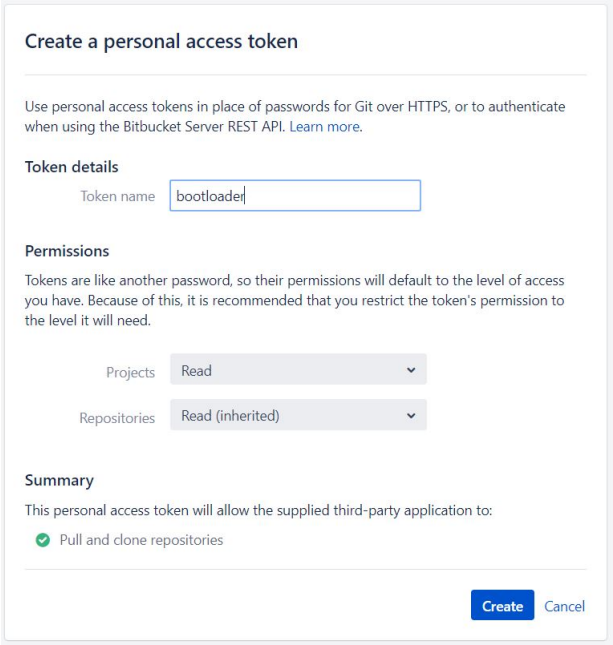

Figure 5.1: Personal token

project will operate in full working order at the beginning of the next week. For example, the H 20 * * 5 rule says that the job is launched every Friday after 8 o'clock in the evening. Symbol H says that the job should not be launched exactly at the time indicated below but approximately at this time. This property allows Jenkins to decide whether he has the resources to complete the build now or it should wait a while. 20 is the time. 5 is the day of the week. Also each job can be run by another job which in turn checks for a new version of the software. Two additional jobs were created for this reason. the Job which checks the new version of the software, runs every five minutes. During the run this job blocks the Build job. In turn the Build job also blocks the Check job during its run. A trigger which says that a build or test job should be run is the modification of the version of the configspec file in ClearCase. Configspec file must have a label. Label is a user-defined name attached to a version. Labeling is the last step in merging project changes.

5.2.2 Agent

Standard installation package of the Tooling team was used for the installation of Jenkins agent. This installation package includes all necessary plugins and is deployed on a properly configured VDI in minutes.

The agent must be started with the path parameters to start and connect

to the master. Path parameters contain a link to the agent node in the master and a secret, which contains the secret for connecting to the master and identifying itself as a node.

Secret and link are generated on the master (Figure 5.2).

Figure 5.2: Connect agent to Jenkins

To do this it is necessary to go to the master's web interface and choose Jenkins / Nodes / New Node, then enter the name, number of executors, the remote root directory and select launch method "Launch agent via Java Web Start". The path to the node and the secret will be generated with the help of Jenkins.

5.3 Project structure

Since the project consists of many scripts and configuration files, all files were grouped into directories. The main folder of the project contains scripts and the following directories:

- pipelines contain pipelines
- config contains configuration file for each component
- configSpec contains Configspecs for access to bootloader VOBs. These Configspecs contain rules for getting access to the latest version of the Configspec which is necessary for the build.
- project_lib contains tools that are not part of scripts and can be used separately.
- rules contain rules and files for the BuildLogParser

- tmpl contains HTML template for email and CMTChecker output file.
- variants_cfg contain configuration files for the CMTChecker

5.4 Pipelines

Each step whether it is a build, testing or release consists of some steps.Each step is represented by one or more scripts. Pipelines describe the sequence of execution of these scripts.

The build pipeline contains five consecutive stages. The first stage is 'Git clone'. This stage copies all the necessary files from git to agent.The 'Git clone' step was separated into a stage because it uses only Jenkins' capabilities for cloning and does not use groovy scripts. The second stage is 'Initialise'. This stage runs the initialization script which is described below. The 'Initialise' step was separated into a stage because it is common to all parallel steps in the subsequent stage. The next stage is 'Builds'. This stage contains four parallel stages that build different components. The fourth stage is 'Send Mail'. This stage contains a script to analyze the results and send the email. 'Send Mail' is common to all parallel steps in the previous stage. The last stage is 'Delete'. This stage step was separated into a stage because it runs a script that removes old builds and is not a part of build.

Each Job contains a small pipeline that lets clone a project from git and run it on Agent.

Template for Job's pipeline:

```
node ('master'){
    checkout([$class: 'GitSCM',
        branches: [[name: "version"]],
        userRemoteCongigs: [[
            credentialsId: "credentials id",
            url: "path to git folder"
         ]]
    ])
    load "path to pipeline in checkout project"
}
```

5.4.1 Build

The pipeline that will be launched on the Agent consists of four parts. They involve initialization, build, processing results and sending emails, deleting

directories.

The first step is initialization. At the initialization stage a directory is created on the share. Build results, logs and statistics files will be copied to the share. Also configspec is created.

The next step is build. At this stage the project is build. The part of statistics is created, files are copied to the share. The build consists of four components that can be processed in parallel. Each component has its own view. Since part of the files that use the components are the same. If several views are not created, then errors may occur. it is possible that several components try to change something in the same file at the same time.

In the third step an analysis of the build results is performed and an email with the build results is sent to the developers.

The last step is to delete the results of old builds from the share.

5.4.2 Testing

The pipeline for testing has almost the same structure but instead of the build, it has such stage as testing and does not have parallelization. During testing, parallelization is not used for several reasons. Firstly, testing does not take as much time as the build takes. Secondly, the simulation requires software licenses which are in limited quantities. Variants are built at the testing stage.Tests are generated and run. Test results are analyzed, logs are copied to the shares and the email is sent.

5.5 Groovy scripts

All the scripts described below in addition to their described actions also send an email to their developer in exception in these scripts.

5.5.1 initialise.groovy

This script creates a directory for storing the results of a build or testing, creates a configspec with timestamps as well as a file with links to the Tasking licenses servers.

The name of the created directory is in the format dd-MM-YYYY.HH-mm-ss. For example, 22-04-2020.11-21-15.

Time stamps are made in the configspec for the possibility to repeat the project build without changing the file versions. Timestamps are added only to links containing the word LATEST. This word means that the latest

versions of files that are located at the link of the previous LATEST should be taken from the view. Timestamps allow to use the latest version of files up to this time inclusive.

5.5.2 bootloader.groovy

This is the main script that forms a view with the names created by the template based on the configuration file and the parameters thtough which the script is run. Use the configspec in the previous step to create a view. Then this script mounts the VOBs. VOBs are different for different generations of AURIX. The next step is building variants. All variants which are based on configuration files are built sequentially. The results of the builds are copied to the share. A directory with logs is being prepared. At the stage of sending an email this directory will be turned into a zip archive. A CMT checker HTML file for each variant is created and copies the results to the share. CMT checker HTML file contains information about code complexity measurement tools (CMT) execution. This script creates a file build_result.xml file that contains information about the results of CMT Checker and make checking for each variant.

The bootloader.groovy script contains the following additional functions:

def openBatches() - create file object for each batch file.

def copyFiles() - copy files from the view to the share and create CMT checker HTML file.

void addToBuildResultXML() - creates result xml or appends to result xml. This file contains the results of the CMT checker and make checking.

5.5.3 sendMail.groovy

This script processes the results of the build and sends an email with statistics and links (Figure 5.3). It also creates an archive with the most important logs. This archive is one for all components.

The script analyzes the build results of each variant sequentially. The analysis is as follows. The script receives information from the configuration file. This information reflects files which should be present in the folder with the results. The script looks at these files and building upon their existence

Hello,
This email contains informations about BUILD

Status of build **COMPONENT 1** is: **OK**
See the results here
See the logs here
See the logs in HTML format here

COMPONENT 1					
Variant	**LINT warnings pct COUNT/MISRA**	**LINT errors pct COUNT/MISRA**	**LINT overall pct COUNT/MISRA**	**CMT checker**	**Compiler combined**
VARIANT 1	0	0	0	ref	ok
VARIANT 2	0	0	0	ref	ok

Figure 5.3: Email example

or non-existence and their content decides what status the build of this variant has. Possible statuses are ok, warning, file. OK status means that this build is successful. Warning claims of some minor errors that do not affect the overall performance of the project but they need to be addressed and corrected. The failure is a critical error that needs immediate correction. A project with this error is not working. The email title indicates the status of the worst-case scenario after sending an email with the results of the build.

The lint log is parsed during the analysis of the logs. Lint or a linter is a small program that checks source code to flag stylistic and programming errors, bugs and suspicious constructs. It also analyzes the results of CMT Checker and makes check from the file build_result.xml.

The sendMail.groovy script contains the following additional functions:

String createMailBody() is the main function that creates email body.

def zipDir() creates *.zip archive that contains main log files.

String getbuildBodyHeader() creates header before table with information about status of the build. It can be a SUCCESS if there are no errors found. it can be WARNING if not critical errors are found and it can be FAILED if there are some critical errors. Errors mean the absence of any file, errors in the logs or complete build failure.

void checkFiles() this function checks for the existence files in the output directory. The list of files is specified in the configuration file.

String createResultTable() creates build result table.

String getNewResultTableRow() creates header of the result table.

boolean checkBuildLog() checks the build log by using the build log parser.

5.5.4 deleteResultFolder.groovy

This script is executed after the build and sending the email. The script checks the number of directories on the share. If the number of directories is greater than a number from configuration file, it deletes the oldest.

5.5.5 bootloaderTest.groovy

This script is the main one in the testing pipeline. The general structure of this script is similar to bootloader.groovy. The only difference is that batch files run only once without additional parameters, for example, lint and CMT.

5.5.6 sendMailTest.groovy

The general structure of this script is similar to sendMail.groovy. Only the analysis of logs differs. It happens because the test results are a build log and one log file is for each test. These log files contain test results. At the moment the script only checks for the presence of log files and analyzes the build log file using Build log parser.

5.5.7 versionChecker.groovy

This script is very important for the implementation of the continuous integration system as it checks for a new version of the software that needs to be built and tested. If this version is available, it signals to the job that started it and that the build needs to be launched. This allows to automate builds.

First of all, versionChecker.groovy script starts the view and mounts the drives. Then it checks for a new version of the configspec. If there is a new version of the configspec, the script checks the presence of a label on it. The availability of the label indicates that the programmer merged the project and the last action assigned the label to this configspec. Next, the script copies the new version of the configspec to the share and signals this action to Jenkins. Next, Jenkins will process the script signal and run the build, if possible.

5.6 Tools

Tools were created in the process of developing the system of continuous integration. These tools can and will be used in other projects. Such tools are: a script for comparing CMT results and a program for parsing logs.

5.6.1 Build log parser

Build log parser was developed as a part of the lib_groovy.jar library.

Build log parser works with the build log. This log contains information about the build run, namely, warnings, errors, some statistics, info messages. The size of this log does not exceed 5 megabytes. This fact facilitates the development, since it is not required to monitor the possible lack of RAM.

Build log parser reads the buildLog file line by line or sector by sector and creates HTML page with the link to each significant message. For example, error, warning, info and so on. Types of messages are defined by the user in the configuration file or default values can be used. Messages can be defined as lines or as sectors. Significant regular expression for messages must be defined in the XML file.

Configuration xml file should be used for declaring regex rules. this file contains the list of rules, delimiters and types. Rules are used in queue form.

The First rule from the file is checked first, the second is verified secondly and so on.

there are four types of items for the part depicting characteristics. they are name, color, priority and errorFlag. Name is a message name. Color is a link color in the created HTML file. Priority is a link order in the reference part. ErrorFlag accepts values true if the message of the user - specified type was found. on the other hand, ErrorFlag will accept values false. ErrorFlag is used to indicate critical errors.

This parser can be used not only for parsing buildLog but for parsing other text documents.

This tool contains the following classes:

- BuildLogParser.groovy is the main class that controls the rest. It contains two methods. Boolean run(String pathToRules, String pathToBuildLog, String outputFile) method starts this tool and contains default actions defined in the application's settings. Boolean process(String pathToRules, String pathToBuildLog, String outputFile) method starts parsing and decides which class will be used, namely, SectorParser.groovy or LineParser.groovy.

```xml
<Configuration>
    <Rules>
        <Delimiters>
            <Delimiter value="^Start:\\((I|W|D|E|F)\\) \\d{2}.\\d{2}.\\d{2}-\\d{2}:\\d{2}:\\d{2} ---\$">
            </Delimiter>
            <Delimiter value=".{0,}\[.py\].{0,}">
                <!-- # match line contains 'error ', case-insensitive -->
                <Rule type="error">(?i).{0,}error.{0,}</Rule>
            </Delimiter>
            <Delimiter value=".{0,}\[.com\].{0,}">
                <!-- # match line contains 'error ', case-insensitive -->
                <Rule type="error">(?i).{0,}error.{0,}</Rule>
                <Rule type="warning">(?i).{0,}warning.{0,}</Rule>
            </Delimiter>
            <Commons>
                <Rule type="error">(?i).{0,}exit status [4-9].{0,}</Rule>
                <Rule type="warning">(?i).{0,}exit status [1-3].{0,}</Rule>
                <Rule type="info">(?i).{0,}exit line exit status 0.{0,}</Rule>
            </Commons>
        </Delimiters>
        <Lines>
            <Rule type="info">(?i).{0,}test string.{0,}</Rule>
            <Rule type="info">(?i).{0,}build failed.{0,}</Rule>
        </Lines>
    </Rules>
    <Types>
        <!-- -->
       <Type name="ERROR"           color="red"         priority="1"        errorFlag="true"/>
       <!-- -->
       <Type name="WARNING"         color="blue"        priority="2"/>
       <!-- -->
        <Type name="INFO"           color="silver"      priority="3"/>
    </Types>
</Configuration>
```

Figure 5.4: Example of XML configuration file

- RuleConfigReader.groovy reads rule configuration file and creates RuleTypes object.It contains one method RuleTypes readFromXml(String configFileText).

- RuleTypes.groovy is a factory that creates rule type objects. It includes three methods. Void addType(String name, String color, int priority, boolean errorFlag = false) method adds new type to ArrayList if this type does not exist. RuleType getRuleTypeByName(String name) returns rule type by type name. ArrayList<RuleType> getSortedTypes() sorts types by priority and returns.

- RuleType.groovy creates rule type objects. Contains getters, setters and compareTo methods.

- RulesParser.groovy reads delimiters, sector and line rules from configuration file. it involves two methods and getters. Void parse(String path, RuleTypes types) method reads rule files and creates rules and delimiter Queues. Rule createRule(String name, String regex, RuleTypes types) method creates the rule.

- BLParser.groovy is an abstract class for parsing log files by lines or by sectors.
- SectorParser.groovy parses log by sector and lines together.
- LineParser.groovy parses log by lines.
- Delimiter.groovy is a class for creating object with one delimiter from rule files.
- Lines.groovy is an object which is created on the basis of this class. It will contain all lines from buildLog in two structures including Queue with all lines and Map with founded items.
- Line.groovy is a class for one line from the log.
- Rule.groovy is a class for creating object with one rule from *.rule files.
- Sectors.groovy is an object which is created under this class. It will contain all sectors from buildLog in two structures including Queue with all sectors and Map with founded items.
- Sector.groovy is a class for one sector from the log.

Using this tool allows to navigate in the build log easily and get the most significant information about the build very quickly.

5.6.2 CMT checker

This tool is designed to compare two excel tables that contain CMT statistics for each function in each project code file and create html results page. CMTs are code complexity measurement tools. This tools help to write the code with a good complexity. The Code with a good complexity contains less errors. What is more, it is easier and faster to test, to understand and to maintain. One of the excel tables is generated during the build, the other one is located on the VOB. It was generated earlier and it is a reference for comparison. Tables contain information about the build functions.

The CMT parser is a tool that was designed to replace an existing one because the existing tool required a Microsoft Office license. The previous version of this tool was launched manually by the programmer and was a macro in excel. The new version of the tool is a groovy class. Replacing this tool avoids using of Microsoft Office license and the installation of the Microsoft office instruments on VDI. It in turn saves money for the company.

This tool contains the following methods:

- def parse(String path) is the function which reads the excel file and returns it as a map.
- void checkFile(String path) checks if there is any file and it is not the directory.
- String compareNewAndOld(def cmtMapNew, def cmtMapOld, def configMap, String pathToTmpl) compares two maps created from excel files and identifies new modules. It checks each module and compares it with the default value from the configuration file.
- String countMetrixes(def cmtMapNew, def configMap, String pathToTmpl) checks each module from Excel and compares it with the default value from the configuration file.
- String allMapToHtmlTable(def map, String pathToTmpl) creates html table from the map.

5.6.3 Library lib_groovy.jar

Several changes have been made to this library. First, Build log parser was added as a separate extension. Secondly, the function for working with ClearCase labels was added. This function allows you to get all the labels belonging to the file specified as a parameter. If there are no labels, the function returns an empty array. This function is used to verify the existence of a new version of the software.

6 Possible extensions

Only the main part of the CI system is implemented at the moment. The CI system will expand over time. It depends on the needs of the bootloader team. CI will also be adapted for other teams, the structure of which is similar to the bootloader team.

Some future extensions are already known. They are also in the process of development or described as assignments. These extensions will be described in the current chapter.

6.1 Build

One extension is currently scheduled for builds. This extension is related to branches in ClearCase. Continuous integration works only on the main branch of the development at the moment. Programmers merge their own branches into this branch. The new extension will allow the programmer from the bootloader team to launch a build on his own branch before merging the branches together. Which will lead to earlier error detection.

6.2 Testing

Two extensions are currently planned for testing. The first extension is going to run testing on a physical unit.There are prepared scripts At the moment. These scripts are used in testing on simulation of a microcontroller. This extension is necessary because the simulation cannot cover all parts of this microcontroller and may also contain errors. The second extension is the addition and automation of unit tests. Mutex will also be added to regulate the number of licenses used.

6.3 Release

At the moment, the CI system does not cover the release. To add this step to the CI system and expand it to a CD system will be required to:

- Exploring the complete release creation cycle.
- Obtaining additional rights for the pooluser, for access to the VOBs and to various systems necessary for creating a release.

- Mastering work with new programs by means of their API. For example, ClearQuest.
- Studying the structure of the documentation that should be generated for each release.
- Many more smaller subtasks are possible.

6.4 Tools

To expand the CI system, it will also be necessary to improve existing tools and create new ones.

Build log parser

At the moment, Build log parser is designed to process logs less than 5 megabytes in size. In the case of a log larger than 5 megabytes, problems with displaying in the browser are possible. Since the log is displayed in one file and the number of tags used increases. This may cause the browser to freeze. Also, when parsing, the entire document structure is held in memory in the form of objects. Which can lead to errors when out of memory. To avoid these problems, the Build log parser will be finalized as follows. First, the log will be processed in parts, the sizes of the parts will be determined later. This will avoid a lack of memory when processing logs and will allow you to process documents of any size. The HTML generated by the parser will also consist of subpages, which will reduce the load on the browser.

Calibration tool

The next extension will be the calibration tool. This tool will be used to replace parameters in an already built project. This will allow us to refuse to install CANape 13 version and leave only 16 version. This should also reduce the number of CANape licenses used. Which in turn carries financial benefits for the company.

This tool will replace parameters based on configuration files. Names of parameters, starting addresses, size, new data will be indicated in the configuration files. After replacing the parameters, the script should recalculate the checksum.

7 Conclusion

Continuous integration in the modern world has become a necessary part of software development. This practice makes it easier and faster to develop, as well as improve the quality of the code.

At the beginning of the bachelor's work, the principles of CI/CD and the programs necessary for their implementation were described. Next, the toolchain that the bootloader team uses was described and analyzed. A comparison was made of the approach to software development without and with a continuous integration system. Then a continuous integration system was implemented for build and testing. During the creation of the continuous integration system, a release was not implemented.

The release will be the last step in the implementation of CI/CD system. An automatic release will allow to close the chain build-testing-release and say that for this project all the conditions of Continuous Delivery are met. This step will be created in the future. Also, the ability to programmers to build on their own branch will be added. Testing on a physical machine will be added. For this type of testing, all the necessary scripts are implemented from the tooling team side, but the implementation from the team bootloader side is in progress.

During the implementation of the CI system the CMT parser and Build log parser tools were developed to process the build results. Build log parser is already in use in another project.

An important part of the bachelor's work was the communication with the bootloader team and with colleagues who have already created continuous integration systems. So it was necessary to study the development approaches in the bootloader team and learn from the experience of the tooling team.

The given project allowed me to improve my programming knowledge, communication abilities and also allowed me to master the creation of CI systems.

Glossary

API - Application Programming Interface

CAN - Controller Area Network

CD - Continuous Delivery

CI – Continuous Integration

ECU - Electronic Control Unit

JVM - Java Virtual Machine

OEM - Original Equipment Manufacturer

RDP - Remote Desktop Protocol

SCM - Source Code Management

VDI - Virtual Desktop Infrastructure

VOB - Versioned Object Base

Bibliography

[1] *32-bit TriCore^{TM} AURIX^{TM}– TC2xx* [online]. Infineon Technologies AG, 2020. [cit. 2019/30/06]. Available from: https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc2xx/.

[2] Anastasov, M. *Continuous integration, continuous delivery, continuous deployment* [online]. 2020. [cit. 2020/12/07]. Available from: https://sudonull.com/post/64771-Continuous-integration-continuous-delivery-continuous-deployment-just-a-nes

[3] *Bitbucket* [online]. 2020. [cit. 2020/12/07]. Available from: https://bitbucket.org/product/.

[4] *CANape product information* [online]. Vector Informatik GmbH, 2019. [cit. 2019/12/09]. CANape product information. Available from: https://assets.vector.com/cms/content/products/canape/Docs/CANap/_ProductInformation/_EN.pdf.

[5] *ClearCase* [online]. 2020. [cit. 2020/12/07]. Available from: https://www.ibm.com/support/knowledgecenter/en/SSSH27_8.0.0/com.ibm.rational.clearcase.help.ic.doc/helpindex_clearcase.html.

[6] *Groovy* [online]. 2020. [cit. 2020/12/07]. Available from: https://groovy-lang.org/.

[7] *Jenkins* [online]. 2020. [cit. 2020/12/07]. Available from: https://www.jenkins.io/.

[8] *What is Jenkins?* [online]. Amazon Web Services, 2020. [cit. 2019/30/06]. Available from: https://www.edureka.co/blog/what-is-jenkins/.

[9] *Jira* [online]. 2020. [cit. 2020/12/07]. Available from: https://www.atlassian.com/software/jira/agile.

[10] *Managing Parameter Sets Easily and Traceably with vCDMstudio* [online]. Vector Informatik GmbH, 2020. [cit. 2019/30/06]. Available from: https://www.vector.com/int/en/products/products-a-z/software/vcdmstudio/.

[11] Sharma, S. *DevOps For Dummies*. John Wiley Sons, Inc., 2014. ISBN 978-1-118-73378-3.

[12] *What is Continuous Integration* [online]. Amazon Web Services, 2020. [cit. 2019/30/06]. Available from: https://aws.amazon.com/devops/continuous-integration/?nc1=h_ls.

CD content

The work is accompanied by a CD with the following structure:

- directory: scripts/ - this directory contains source codes and configuration files.
- directory: thesis/ - this directory contains the text of bachelor thesis.