Development and maintenance of an open-source project

Thursday 21st March, 2024 - 19:26

Fedor Chikhachev
University of Luxembourg
Email: fedor.chikhachev.001@student.uni.lu

This report has been produced under the supervision of:
Alfredo Capozucca
University of Luxembourg
Email: alfredo.capozucca@uni.lu

Abstract

This Bachelor Semester Project was realized during the second semester of the Bachelor in Computer Science. This paper contains an explanation of what has been done during the semester of work; the objectives of this BSP are to learn about open-source development process and implement the studied material on a real, existing project.

The targets are going to be described in more detail in the following dedicated sections.

1. Introduction

In today's software industry, open-source projects have become a popular way of developing software. The open-source movement has gained significant momentum in recent years, and it is now a widely accepted practice for developing software, both in the corporate world and among individual developers. Open-source software is characterised by the freedom to use, modify, and distribute the software's source code, making it a valuable resource for developers and users worldwide.

This Bachelor Semester Project focuses on the development and maintenance of an open-source project, especially converting the proprietary project to the open-source. This work consists of two primary parts: scientific and technical. The scientific part delves into the criteria required for a project to become open-source and different approaches to meet these criteria. The technical part focuses on developing a solution to created a partly automated workflow to accept contributions for an existing project.

The scientific part of the project aims to establish the standards and principles that open-source projects should follow. There are several criteria that a project needs to satisfy to be considered open-source, such as the availability of source code, distribution rights, licensing and contributing guidelines. By exploring these criteria, the project seeks to describe all the requirements the project should met to be called "open-source" and compare different approaches, which are used in modern open-source development. Moreover, it presents a concrete workflow for researchers, who would like to continue developing their project with the help of community. As an example for assessment purposes of completed work, an existing project is used – e4l [4], in order to show, how to complete the presented guidelines.

The technical deliverable of the project is about the creating an environment for maintaining an open-source project. It is an automation workflow, consisting of several scripts, which can be helpful for maintainer while reviewing and managing income contribution for the project.

In summary, the semester project aims to explore the development and maintenance of an open-source project. It will examine the criteria required for a project to become open-source and the different approaches used to meet these criteria. Additionally, the project proposes a model for accepting contributions to an existing open-source project using public and private repositories synchronised with a CI script. By conducting this research, the project seeks to contribute to the growing body of knowledge on open-source software development and promote its adoption in various domains.

2. Project description

2.1. Domains

2.1.1. Scientific. The scientific domain of this project is **Software Development**

Software development is the process of designing, building, testing, and maintaining software applications. It involves creating software solutions to meet specific business or user requirements using a variety of programming languages, tools, and technologies. The development process typically involves multiple stages, such as planning, analysis, design, implementation, testing, and deployment. The aim of software development is to create reliable, scalable, and efficient software

solutions that meet the needs of users and businesses, while adhering to standards and best practices in the industry.

2.1.2. Technical. The domain of the technical part is the work involved in creating a demonstration of repositories synchronisation workflow. To achieve this goal, several tool are used:

- **Version Control System (Git)**. VCS provides a way to track the evolution of source code by storing and organising different versions of the code. With a VCS, developers can collaborate on a project by working on the same codebase without interfering with each other's work. VCS records changes made to the code in a central repository, along with the author, time, and a brief description of the changes. Additionally, VCS provides features such as branching and merging, which enable developers to create alternate versions of the code and merge them back into the main branch when necessary. One of widely-used VCS in the world is Git. Thus, the technical deliverable relies on it.
- **Scripting programming languages (Bash)**. Bash provides a command-line interface for interacting with the operating system. It allows users to execute commands, manage files and directories, and automate tasks through shell scripts. Bash scripts are written in a text file with a .sh extension, and can be executed by running the script through the command line. This language is an essential tool while setting up the testing environment or running continuous integration scripts.
- **CI Tools (Github Actions)**. Continuous Integration (CI) tools are software tools that automate the process of building, testing, and deploying code changes. The main goal of CI tools is to help development teams catch bugs and issues early in the development process, before they become bigger problems. Github Actions is a CI tool inside Github cloud code storage, which provides several additional functionalities to interact with hosted repositories.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. The scientific deliverable of this BSP includes a study of the development process of open-source artifacts and their possible usage scenarios. It answers the research questions posed in the project objectives: what is an open-source project (artifact) and which criteria should it follow, and how to turn into open-source projects that have been done inside the University of Luxembourg?

The project investigates the standards of open-source development. This involves an analysis of the different tools, platforms, and technologies that are used in open-source projects, as well as an investigation of the different governance models and licensing frameworks that are used to manage and distribute open-source software.

Overall, the scientific project provides a comprehensive overview of the open-source development process, standards, and best practices. It helps to inform the development of new open-source projects and provides insights into how to continue the development of existing project with the help of open-source community.

2.2.2. Technical deliverables. The technical deliverable of the project is an environment consisting of several automation CI scripts. The required goal is to make the contribution's review process more convenient for the worker who will be in charge of this.

The produced software is primarily targeted to help an existing project – Energy4Life [4], which was completed inside the University of Luxembourg and has an opportunity to continue its journey as Open Source software. Thus, the criteria for the technical deliverable were produced with an assistance of project's team, represented by Dr. Alfredo Capozucca.

The software should provide an environment for maintainer to access and control the contributions, which would be send to this project. The main requirements are *maximum automation of this workflow* and *contribution's reliability checks via private computational facilities*.

In brief, an additional CI environment was created, which supports the semi-automated synchronisation between public version of source code and private version, which is considered as Source of Truth (SoT) and used as the source for deployment and testing.

To sum up, the necessary steps have been taken to ensure the open-source availability and reliable deployment of the e4l software for potential users and contributors, as well as the comfortable and convenient workflow for project's maintainers.

3. Pre-requisites

3.1. Scientific pre-requisites

Before addressing the subject of this report's scientific matter, no specific scientific prerequisites are required.

3.2. Technical pre-requisites

For a technical part, it is required to demonstrate basic competences in using Git Version Control System, understand, what is commit, branch, remote repository, pull requests.

4. A Scientific Deliverable 1

4.1. Requirements

4.1.1. Functional requirements. After reviewing the scientific result of this work, the reader will get acquainted with the information in several sections:

1) Definition of open-source software. All required criteria, aspects to verify that the software is actually open-source.

2) Delving further into each of the essential components of open-source software and/or its source code, namely contribution guidelines, licensing, and code of conduct.
3) Required steps to convert an existing project to the open-source.

4.1.2. Non-functional requirements. The non-functional requirements are the methods and criteria to make the deliverable more reader-friendly. The main requirements for this part are the clarity of structure and readability of text. Moreover, for a better understanding of definitions and terms, it is necessary to utilise the citations and references properly.

4.2. Design

The main goal of this deliverable is to explain, what is an open-source project and which guidelines should be followed to create one or turn existing project into open-source.

The first step in producing the scientific deliverable was to find and analyse relevant sources of scientific knowledge on the topic.

- Github, which serves as a host for majority of open-source projects, has published a guide "Starting an Open Source Project" [3]. This page provides a comprehensive guide for starting an open-source project. It covers the benefits of open-source software, the importance of having a clear project vision and goals, and the necessary steps for creating and maintaining an open-source project. The guide also discusses various topics such as project licenses, code of conduct, documentation, and community management. Additionally, it includes several case studies and examples of successful open-source projects.
- Formally and officially, open-source criteria listed on the website of Open Source Initiative [8], with the link to originally posted Debian Free Software Guidelines [11] [9]. This document is well structured so it will be an ideal polygon to start with.

The text begins with the definition of open-source software and listing and explanations of the criteria it should follow. These can be done by citing the Open Source Initiative website, as this is the approved source of all information about open source development.

Having dealt with the definition, the subsequent section of the study explores the process of transitioning from proprietary to open-source development. It describes the steps which should be followed, as well as the roles involved on each step.

With the procedure turning into open-source revealed, it should be prescribed more consciously, what each point in this guideline means and how it can be then completed.

4.3. Production

4.3.1. Definition of Open-Source. Open-source software term refers to software that is made available to the public with its source code, allowing users to view, modify, and distribute the code freely. The Open Source Initiative (OSI) provides a widely accepted definition of open-source software [8], which includes 10 criteria that a software must meet in order to be considered open source.

1) **Free Redistribution.** "The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale".
2) **Source Code.** "The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicised means of obtaining the source code for no more than a reasonable reproduction cost, preferably downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed".
3) **Derived Works.** "The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software".
4) **Integrity of The Author's Source Code.** "The license may restrict source code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software".
5) **No Discrimination Against Persons or Groups.** "The license must not discriminate against any person or group of persons".
6) **No Discrimination Against Fields of Endeavour.** "The license must not restrict anyone from making use of the program in a specific field of endeavour. For example, it may not restrict the program from being used in a business, or from being used for genetic research".
7) **Distribution of License.** "The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties".
8) **License Must Not Be Specific to a Product.** "The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution".

9) **License Must Not Restrict Other Software.** "The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software".
10) **License Must Be Technology-Neutral.** "No provision of the license may be predicated on any individual technology or style of interface".

4.3.2. Steps to follow. After studying the materials and guides about creating open source projects, an instruction to follow has been produced. It is shown as a diagram on picture 1. Each task is a separate rectangular block; A stands for *Actor*, a person or a group of people, who performs this task; S stands for *stakeholders*, a person or a group of people, who is involved into this task; D stands for *Deliverable*, final output after finishing the task.

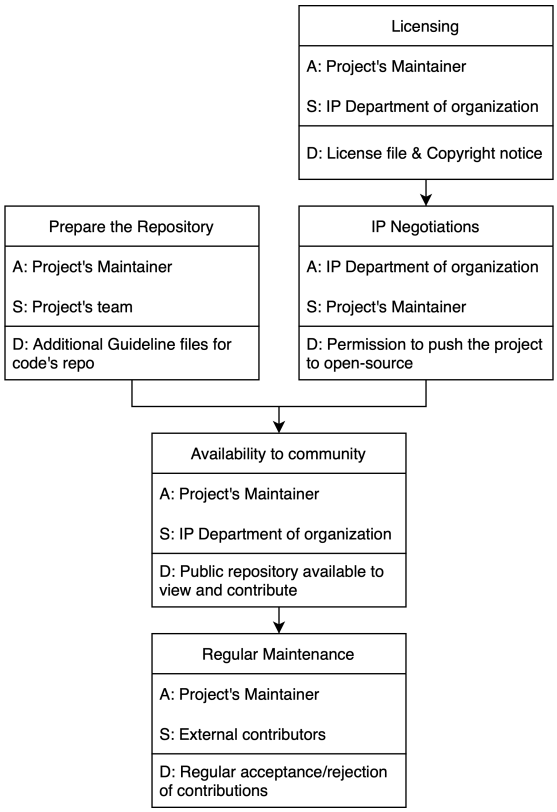


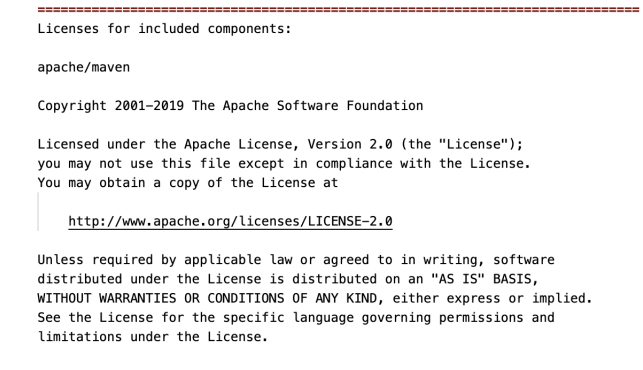
Fig. 1. Step-by-step diagram representing the procedure of turning software into open source. A – Actor, S – Stakeholder, D – Deliverable.

There are five main steps to complete:

1) **Prepare the repository.** There exists some special recommendations for how should the repositories of open-source projects be look like; which additional files should they have inside.
2) **Choose the license.** There exists a long list of licenses, approved by OSI [7]. Every license has its features and disadvantages, so it is an important decision for project's maintainer to define, which part of his work will be protected with special terms, and how it will be protected.
3) **Intellectual Property (IP) negotiations.** This step refers to the employees of company/institution, who would like to turn the project made up inside their working team into open source.
4) **Make the code available to public.** After receiving confirmation from your employer to publish the project in an open-source form, the code now should be available for the public. The implementation of this point can be discussed with the IP department as well. The most common and widely used solutions for these purposes are cloud code storages, such as Github or Gitlab. Each of platforms has its advantages, so it is up to maintainer to choose the most convenient environment for work.
5) **Regular maintenance.** After the project was openly published, it is supposed that the software will now receive contributions to it source code. The maintainer should review them and accept significant changes and/or improvements.

4.3.3. Contents of repository. Firstly, let us define the attributes, which should be available in each open source project repository. It is highly recommended to create all these files for the project, as it will help maintainers and contributors in working together.

1) **Licensing & Copyright.** Every open-source project should determine legal aspects of working on it. File, which is usually called *LICENSE* or *LICENSE.MD*, contains the terms and conditions under which the software can be used, modified, and distributed. It specifies the legal rights and obligations of the software's users and contributors.
   The LICENSE file usually contains License type the software released under (MIT, Apache, GNU GPL etc.), Copyright Notice, which specifies the authors or the organisation initially worked on this project. After this, there is usually provided either the full text of license, which describes the permissions and the limitations for the contributors, the warranty terms (or lack of warranty) and any other useful information.
   Another important aspect, which should be displayed in the license file of software, is the mention of all used licensed third-party content. This can be done by providing copies of their copyright notices, as well as the types of licenses they were originally distributed. In the picture 2 it can be seen how it is implemented for a package maven within the license file of a project, which uses this tool.
   As it comes for the third-party materials, it is eminently to mention about the compatibility of different licenses. It is not a rare case to face a situation, when the license chosen for licensing the project violates the licensing terms of one of the components of your



```
========================================================================
Licenses for included components:

apache/maven

Copyright 2001-2019 The Apache Software Foundation

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

------------------------------------------------------------------------
```

Fig. 2. An example of copyright notice of third-party package inside license file

project. A common example – compatibility with GPL-like licenses. One of the term of this family of licenses requires to distribute all derivative artifacts of GPL-licensed software under the same GPL license. It means that if you have ever used any resource within your project, which has a GPL license coverage, then your project should be also GPL-licensed. Otherwise it can be considered as licensing violation and can possibly lead to any sort of sanctions.

2) **Contributing.** To simplify the work of project's maintainer, the project's team provide special guidelines on how the contributions to the software should be formatted and sent. Usually these requirements can be found in *CONTRIBUTING* or *CONTRIBUTING.md* files.
   It includes the information about some technical aspects of contributing to the project. For example, there can be found some information about setting up the environment on your personal computer for convenient work on the project, e.g. how to perform tests or which coding style guideline is used in the project. Moreover, contributing guideline provides information on how contributions are reviewed and approved, including the criteria for acceptance and the roles and responsibilities of the maintainers.
3) **Code of Conduct.** To establish a set of guidelines and expectations for behaviour within a community of contributors, there exists a Code of Conduct, which is usually written in *CODE_OF_CONDUCT* or *CODE_OF_CONDUCT.md* file. By encouraging constructive communication and collaboration, a code of conduct promotes a productive and positive environment for all contributors.
   There exists a common template for code of conduct file, Contributor Covenant, which can be freely copied and added to your project. This code of conduct is used by many projects and companies, including Google, Microsoft and Linux.
4) **Contributor License Agreement.** A Contributor License Agreement (CLA) is a legal document between a contributor to an open-source software project and the project's maintainers, which defines the terms under which the contributor submits their work to the project. The agreement outlines the rights and responsibilities of the contributor, such as granting a license to use their contributions, ensuring that the contributions are original, and agreeing to indemnify the project for any legal issues that may arise from the contribution.
   A CLA typically requires contributors to sign it before their contributions are accepted into the project. It helps to ensure that the project has the necessary legal permissions to use the contributions and that the project and its users are protected from legal liability.
   Some open-source projects require a CLA, while others do not. It's up to the maintainers of the project to determine if it is necessary for their particular project.

4.3.4. Legal advice. Defining the concepts leads us to the procedure of turning your project into open source. Two cases can be distinguished: starting a project "from scratch" (a new one), or turning an existing project into an open-source. For the second moment, there exists an important aspect concerning revealing the project to open source community.

If the project is not started as open-source, then it considers *proprietary* till that moment. It is a type of software that is owned by a company or an individual and is licensed for use under specific terms and conditions, also known as closed-source software because the source code is not made available to the public or users.

As it can be seen from the definition, the rights to this software belongs to particular subject, which can be either a person/group of people or an institution (university, research facility, company etc.)

If a software creator has an "employee IP agreement" with their organisation or another institution, the organisation will exert some influence over their projects, particularly if they are relevant to the organisation's business or if the person utilises any resources provided by the institution during development. The organisation should readily grant permission, possibly through an employee-friendly IP agreement or an internal policy that has already been established. If not, the individual can engage in negotiations (e.g., by explaining how their project aligns with the organisation's objectives for professional learning and development) or consider finding a more accommodating institution before proceeding with the project.

In the case of open-sourcing a project on behalf of their company, it is crucial to inform the organisation. The legal team likely has specific policies in place regarding the appropriate open-source license to use (potentially with an additional contributor agreement) based on the company's business requirements and their expertise in ensuring compliance with the licenses of the project's dependencies. The legal team should be willing to collaborate with the creator to determine the best course of action in this regard. [2]

In all cases, an IP department of the institution or company,

within the project has been started or is creating at the moment, should be warned about all the plans of turning software into open-source.

4.3.5. After release. After performing every step from the instruction provided, the project's team has now revealed their project for the open-source community and should be waiting for the contribution. Although the project is now considered "open-source", it is required that special actor(s) called "project's maintainer" should exists.

This role involves reviewing changes proposed by contributors. The maintainer should check, that:

1) changes provided by user are significant;
2) a contributor has signed Contributor License Agreement (CLA, if required);
3) a contribution follow the contributing guidelines provided inside the repository;
4) the behaviour of contributor as well as the proposed changes do not violate code of conduct;

Ensured that all requirements are completed, the contribution can be accepted.

4.4. Assessment

The presented deliverable answers the scientific question and describes the open source definition and procedure of making the project open source.

Moreover, the studied material about open source development and community has been applied in real environment. With the support of Dr. Capozucca, it became possible to test the presented knowledge in a real environment, on the example of an existing project Energy4Life. This project has been created within Department of Computer Science in University of Luxembourg, and the goal was to complete the procedure of turning this project into open source.

Firstly, the required repositories attributes has been prepared: a template provided by Code Covenant was used as a Code of Conduct; Contributing guidelines has been discussed and then described in special file; License file has been composed with the copy of AGPLv3.0 license as well as with copyright notices of all used in the project third-party resources (packages, libraries, frameworks etc.). For licensing part it was also required to check the usage of 3rd-party materials while developing the software, so this check was done semi-automatically with the usage of Scancode [6] tool, recommended by IP Officer of the University.

Then, the negotiations with PaKTTO department has started. A special form of type B2 was provided, and it was required to fill it in with the essential information about the project and the team which worked on it. For example, it was necessary to list all used third-party packages, to provide information about project's funding from external sources, as well as the description of functionality and goals of the project.

The form was successfully fulfilled with all requested information and sent to IP department. However, the process of reviewing is not yet finished, as it takes a lot of time to check the correctness of all the information provided. Thus, the project has not been yet publish for the open-source community and does not accept external contributions at the moment.

Overall, the insights acquired through the research conducted on open-source development proved to be highly valuable during the practical implementation of the project at hand.

5. A Technical Deliverable 1

5.1. Requirements

5.1.1. Functional requirements. The software has a main goal: provide a convenient environment to manage the contribution for an open-source project, in particularly, E4L website. To accomplish this goal, the resulting software should have several functionalities:

- Fetch actual (open) pull requests from the public code repository
- Transfer these proposed changes to the private, maintainer's repository
- Maintainer should be able to review changes, run some tests and accept or reject these changes
- Synchronise the status of pull request between public and private repositories: if it was closed in private version, then it should be closed in public version too.

5.1.2. Non-functional requirements. The code, written to satisfy this requirements, should be opened for modifications and well-commented, as it can be changed in the future for specific reasons by the actual maintainers of the project.

5.2. Design

It was supposed by the current maintainers of E4L project, that the project's actual version after its transition to open-source, will be still hosted within University of Luxembourg Cloud code storage, based on Gitlab. this is due to the convenience of using the internal resources of the university, the absence of any resource restrictions and the already configured environment for development. However, there is no direct access to this repository for students, who are not the employers of University. Thus, for creating this software, Github facilities are used initially with the simple testing environment set up.

The required functionality is closely connected with the working on the code inside the repository, so the most convenient solution for solving this problem – the usage of Github Actions tools. It is an implementation of Continuous Integration / Continuous Development tools with several features added, such as Repository Secrets and Actions Marketplace, providing ready-to-use solutions for specific tasks.

The most complex task to complete is to synchronise the code between two different remote repositories. This task is

further complicated by the fact that some of the files inside a private repository must remain exclusively in it. Nevertheless, there exists an open-source solution for this task – *copybara* by Google [5]. It is a tool used internally at Google. It transforms and moves code between repositories. As in our requirements, source code needs to exist in multiple repositories, and Copybara allows to transform and move source code between these repositories.

However, this tool has not covering all specific workflows we need to implement. Thus, to complete the missing functionality, Github API is used. It helps to manage the details of the repository's environment, manage branches, pull requests and maintainers comments. To gain access to API of private repositories, a user-agent should acquire a secret key with read and write access to target repository.

GitHub Secrets are a way to securely store and manage sensitive information used in GitHub Actions workflows, such as API keys, tokens, and other credentials. Secrets are encrypted and can only be used by authorised users and actions. They are not visible in the GitHub UI or in logs, and are only accessible to the workflow where they are defined. Secrets can be used in your workflows by referencing them with the $\${\{secrets.SECRET\_NAME\}\}$ syntax.

5.2.1. Functional decomposition. The program is divided into several general parts:

- **Check contributor's eligibility.** In order to protect the project from possible legal issues, there exists a CLA: Contributor License Agreement. Its purpose described in the scientific section of this paper.
  As the goal is to minimise the human participation on all stages of workflow, this part is automatised. There exists an open source solution for this task – Contributor License Agreement (CLA) assistant [10]. It provides two views: for maintainers and for contributors. The maintainer can set up the exact text of the CLA and attach it to repository's facilities in order to provide an automation. The contributor then can authorise with his credentials used for contributing (Github account), sign the CLA by ticking the corresponding box, with this approving his acceptance of CLA. From now on, all the contributions made by this user are successfully automatically verified and eligible to be reviewed.
  If the user have not signed the CLA, he receives a warning while submitting his contribution via Pull Request. Such contributions are not considered.
- **Fetch eligible pull requests from public repository to private.** A script should periodically (e.g. using *cron*) scan the public repository for the availability of new pull requests, which has not been yet fetched to the private repository. This is done using Github API calls. Then, for every pull request from PUBLIC repository Copybara package is used to create a new pull request in PRIVATE repository, which is actually a clone of public pull request.
- **Run CI for all incoming changes using private repository facilities.** To check that pull requests remain the existing functionality untouched, it is required to run the tests workflows for all incoming changes. Thus, the special script is triggered inside private repository when a new pull requested was fetched to it from public repository. This script runs the existing testing pipeline for the project. Moreover, the fail of this workflow blocks the possibility to merge the pull request to the main branch in order to prevent undefined behaviour of software.
- **Synchronise the merge or the rejection for the pull request from private repository to public.** After the automated test has been passed, the maintainer can perform some manual checks and after that decide, whenever the contribution is valuable or not. The maintainer has a choice of accepting or rejecting the pull request. In that case, his decision should be equally synchronised with the public repository, and the initial contributor should be warned, whenever his contribution was good or not.
- **Deploy/release code from private repository to public.** After collecting several accepted pull requested in private repository of the project, the code then releases to the public version of repository. Copybara package synchronises the code in main branches of private and public repositories, using private one as a source of truth. This action is done manually, after the maintainer has been insured, that this would be a new stable version of software.

5.3. Production

The resulting piece of software consists of several files. The source code can be found in this BSP Github repository available at the link [1].

Majority of files are Github Actions YAML files, which share the same structure, described below:

- $name$: The name of the workflow.
- on: The events that trigger the workflow.
- $jobs$: The jobs that the workflow performs.
- $runs-on$: The operating system that the job runs on. Usually, for convenience, Ubuntu is the choice here.
- $steps$: Array of consecutive instructions, which are needed to run to complete the workflow.

Snippet 1 is an example of initialised environment for *public_to_private.yml* file:

```yaml
name: Public to Private Pull Requests import

on:
  workflow_dispatch:

jobs:
  copybara:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
```

```
11        uses: actions/checkout@v2
```

Listing 1. Example of environment initialization in YAML file

Firstly, the name of the workflow is defined. It is then visible on the "Workflows" tab at Github repository. Then, we set a trigger for a workflow: in this case, *workflow_dispatch* means that this script can be run manually from "Workflows" tab by pressing a special button. After, the jobs are defined, with the first one titled *copybara*. For each job it is defined on which system/environment it will run; *ubuntu-latest* means that the latest stable release of Ubuntu Linux will be an environment. Each job has some steps. The first step in this job is commonly used between all the scripts – Checkout code. It uses a ready-to-use Github Actions workflow called *actions/checkout@v2*, which clones the active repository (the one inside which the script is running) to the environment home folder. It simplifies the interactions with the code stored inside the repository, as its actual version is always cloned to the machine before running the main parts of scripts.

Below are brief descriptions of all created YAML files with the essential code snippets.

5.3.1. public_to_private.yml. This file is a GitHub Actions workflow definition file called *public_to_private.yml* that automates the process of syncing code changes from a private repository to a public repository using the Copybara tool. The workflow is triggered manually using the *workflow_dispatch* event. Here is a breakdown of the different sections of the file:

- Checkout code: Checks out the code from the private repository, main branch.
- setup-git-credentials: Sets up Git credentials using a secret called GIT_CREDENTIALS. These credentials are required to authenticate to Github while using Copybara tool.
- Install Java: Installs Java version 17 from Oracle using Ubuntu packet manager.
- Install Copybara: Installs and configures the Copybara tool by cloning the repository, installing dependencies and building the Copybara jar file. Here, the script clones the actual version of Copybara software and using Bazel building tool for Java source code creates an execution file *copybara_deploy.jar*. This file then will be an executor for code synchronisation and migration workflow.
- Run Copybara: Configures Git, sets environment variables, and runs the Copybara tool to sync code changes from a private repository to a public repository. The point of this step is to fetch opened pull requests from the public version of the repository and clone them into private version. To acquire all opened pull requests an API call to public repository is used (line 3 in snippet 2). After that, opened pull requests are filtered with *jq* BASH command, which reads the JSON response from API and checks *state* field in it (line 4). Then, using again *jq* command, the script gets all URLs of opened pull requests (line 5) and in a loop passes links to this pull requests to Copybara executor, which clones this pull request to private repository (line 6).

```
1 git config --global user.name "Copybara"
2 git config --global user.email
      copybara@copybara.com
3 response=$(curl -H "Accept: application/vnd.
      github+json" -H "Authorization: Bearer
      $GITHUB_TOKEN" -H "X-GitHub-Api-Version:
      2022-11-28" https://api.github.com/repos/
      FChikh/sync_repos_public/pulls)
4 pull_requests=($(echo "$response" | jq -r '
      [.[] | select(.state == "open") | .
      html_url] | .[]'))
5 echo "$response" | jq -c '.[] | select(.state
      == "open") | .html_url' | while read
      object; do
6 java -jar copybara/bazel-bin/java/com/google/
      copybara/copybara_deploy.jar .github/
      workflows/copy.bara.sky pr_to_private "
      $object" --force
7 done
```

Listing 2. Usage of Copybara within Bash and CI for cloning pull requests

5.3.2. private_to_public.yml. This file completes almost symmetrical operation comparing to the previous one. It sets up a workflow to clone the actual state of master branch from private repository (Source of Truth) to public. The workflow is triggered manually using the *workflow_dispatch* event.

These two consequent files partly shares the structure; they work on the same environment. The only different part is *Run Copybara* step: this time, Copybara do not need any additional information and arguments to run except the configuration file *copy.bara.sky* and workflow name (line 3 in Listing 3).

```
1 git config --global user.name "Copybara"
2 git config --global user.email
      copybara@copybara.com
3 java -jar copybara/bazel-bin/java/com/google/
      copybara/copybara_deploy.jar .github/
      workflows/copy.bara.sky
      push_to_public_master --force
```

Listing 3. Usage of Copybara within Bash and CI for pushing to public repo

The environment variable GITHUB_TOKEN is used to authenticate with the GitHub API and PRIVATE_REPO_TOKEN is used as a token for accessing the private repository, so they should have corresponding access rights.

5.3.3. tests.yml. This file is used to simulate the existing testing environment of the project. As a Proof of Concept, the payload of software's concept is a C++ library file with the tests provided for it.

This workflow triggers when a new pull request targeted to the master branch is opened or updated. Here is a breakdown of the different sections of the file:

- Checkout code: This step checks out the code of the pull request head branch so that the tests can be run

against it. Here we are working not with master branch, but with newly created branch referring to newly cloned pull request, as it is required to check, that contributions works correctly.

- Install dependencies: This step installs the necessary dependencies required to build and run the tests, specifically *cmake* and *libgtest-dev* – Google Test package environment, which is required to run the tests.
- Build and run tests: This step moves into the project directory and creates a build directory. It then runs *cmake* to configure the project for building and runs make to build the project. Finally, it runs *ctest* to run the tests (Snippet 4)

```
1       cd biginteger
2       mkdir build
3       cd build
4       cmake ..
5       make
6       ctest
```

Listing 4. Test environment simulating role of CI for the project

5.3.4. close_pr.yml. This file describes a workflow which synchronises the closure of pull request in private repository and the closure of corresponding pull request in public repository, which was initially cloned into private.

This workflow is triggered when the pull request has been close, but has not been merged into master branch of private repository. It updates the pull request in public repository with the corresponding comment.

Here is a breakdown of the different sections of the file:

- Checkout code: This step checks out the source code from the repository so that the subsequent steps can work with the code.
- Send API request to sync-repos-public: This step sends two API requests to the sync_repos_public repository. The first API request gets the pull request number from the title of the pull request using string manipulation in Bash (lines 1-3 in snippet 5). Then, it sends a comment on the pull request indicating that it has been declined using HTTP POST request to Github API (lines 5-9 reveals the *curl* command as well as all payload data). The second API request changes the state of the pull request to closed using HTTP PATCH request to Github API (lines 10-14).

Note that the API requests in this workflow use the PUBLIC_REPO_TOKEN Github Secret (line 7, 12) to authenticate with the GitHub API, so this token should have corresponding access rights.

```
1  my_string="${{ github.event.pull_request.title
       }}"
2  pr_number=${my_string#*refs/pull/} # remove
       everything before "refs/pull/"
3  pr_number=${pr_number%%/*} # remove everything
       after the first "/"
4  echo $pr_number
5  curl --request POST \
6     --url https://api.github.com/repos/FChikh/
          sync_repos_public/issues/$pr_number/
          comments \
7     --header "Authorization: Bearer ${{ secrets
          .PUBLIC_REPO_TOKEN }}" \
8     --header 'Content-Type: application/json' \
9     --data '{"body": "Your pull request has
          been declined."}'
10 curl --request PATCH \
11    --url https://api.github.com/repos/FChikh/
          sync_repos_public/pulls/$pr_number \
12    --header "Authorization: Bearer ${{ secrets
          .PUBLIC_REPO_TOKEN }}" \
13    --header 'Content-Type: application/json' \
14    --data '{"state": "closed"}'
```

Listing 5. Script for closing pull request simultaneously in private and in public repositories

5.3.5. close_pr_merge.yml. This workflow is similar to the previous one, with the exception that in this case the private pull request was merged into master branch of private repository, and the corresponding comment appears in public pull request.

Note that the API requests in this workflow use the PUBLIC_REPO_TOKEN Github Secret to authenticate with the GitHub API, so this token should have corresponding access rights.

5.3.6. copy.bara.sky. This is a configuration file for Copybara, a tool used for migrating code between repositories. The file specifies three workflows: "pr_to_private", "push_to_public", and "push_to_public_master". Each workflow shares the same structure which consists of several parameters:

- **name:** The name of the workflow to identify, its ID. This name is passed as an argument to Copybara execution file.
- **origin:** The source of the changes to be copied; URL to the repository, from where the code will be "copied".
- **destination:** The destination of the changes; URL to the repository, where the code will be "pasted".
  Moreover, both *origin* and *destination* can specify additionally the branch of the repository used. Also, a pull request can be both origin or destination.
- **origin_files:** A list of files to be copied from the origin repository. The "**" is a wildcard character that matches all files and directories.
- **destination_files:** A list of files to be copied to the destination repository.
  Moreover, both *origin_files* and *destination_files* can be extended with the field *exclude*. It specifies which files to exclude. In our case, in *push_to_public* workflow, a folder *.github* is excluded from the synchronisation in each of presented workflows, as it contains private files which should not be exposed to the public.
- **authoring:** Specifies the author of the changes in the destination repository. For providing the Proof of Concept, it is enough to fit it with a default username and email.
- **transformations:** A list of transformations to be applied

to the changes before they are committed to the destination repository.

Additionally two variables are defined within this file: *sourceUrl* and *destinationUrl*. These variables store the addresses to distinct remote repositories. *destinationUrl* provides an address to the Source of Truth repository (Private repository), while *sourceUrl* holds the address of public repository, which accepts the contributions.

Below are the descriptions of each workflow described in this file:

- The "pr_to_private" workflow copies all files from the incoming pull requests to the Public repository at *sourceUrl* to the clones of destination repository at *destinationUrl*. In details, as an input it gets the reference to the pull request, opened in public repository; this reference is stored in *CONTEXT_REFERENCE* variable. Then, in private repository, it creates a new branch with the name specified in *pr_branch* field and creates a pull request from newly created branch to the one, defined in *destination_ref* field (in our case it is *master*). Lines 20-24 in snippet 6 signs that this synchronisation is supposed to preserve an authorship information and expose a COPYBARA_INTEGRATE_REVIEW label. This workflow is triggered by a Pull Request being created or updated on the source repository.

```
1  core.workflow(
2      name = "pr_to_private",
3      origin = git.github_pr_origin(
4        url = sourceUrl, #public
5      ),
6
7      destination = git.github_pr_destination(
8        url = destinationUrl, #private
9        destination_ref = "master",
10       pr_branch = "from_external_${
             CONTEXT_REFERENCE}",
11       title = "pr from external ${
             CONTEXT_REFERENCE} ",
12       body = "this is a sample pull
             request",
13       integrates = [],
14     ),
15
16     origin_files = glob(["**"]),
17
18     destination_files = glob(["**"], exclude = [".
         github/**"]),
19
20     authoring = authoring.pass_thru("Copybara <
         copybara@example.com>"),
21     transformations = [
22         metadata.restore_author("
             ORIGINAL_AUTHOR",
             search_all_changes = True),
23         metadata.expose_label("
               COPYBARA_INTEGRATE_REVIEW"),
24     ],
25 )
```

Listing 6. Example of Copybara workflow

Other two workflows are structured similar, but have some changes in details.

- The "push_to_public" workflow fetches changes from the destination repository at *destinationUrl* and pushes them to the source repository at *sourceUrl* in the form of a Pull Request with the name "pr from private repo".
  Overall, this workflow runs inside *private_to_public.yml*, which is triggered manually, and copies all files from the destination repository to the source repository, excluding the .github directory.
- The "push_to_public_master" workflow fetches changes from the destination repository at *destinationUrl* and pushes them directly to the master branch of the source repository at *sourceUrl*. runs inside *private_to_public.yml*, which is triggered manually, excluding the *.github* directory, which contains some private files. It is quite similar to the previous workflow with the exception that the changes from private repository are pushed directly to the master branch of public repository, without any intermediate steps.

5.4. Assessment

The requirements was satisfied: a created piece of software provides a convenient environment for project's maintainer and demonstrates it work as a Proof of Concept, implemented with an artificial test project. However, the example project can be easily (with small adaptations) then substituted with a real working product, such as E4L website.

Moreover, several features can be added in the future. For example, it will be useful to generate a change-log file, which would reflect the changes, which are not yet merged into public version of repository, and the authors of these changes.

Also, some details of work can be improved. The two critical aspects for the moment are:

1) Github limited resources. The project was implemented platform-specifically for the moment. Github provides limited resources for CI/CD execution times with a free version, while on the University's facilities there are no restrictions. But, the university's cloud code storage is based on Gilab platform, so in the future, some sufficient changes are required to make it work platform-independently.
2) Execution time. For the moment, each time when a pull request synchronization is happening, a new instance of Copybara is building. It takes a significant time for this operation, so this is a point to improve in future works, before deploying this tool in production environment.
3) Code wrap-up. From now on, the whole software consists of several scripts, which requires additional environment set up to work correctly. It would be a good

feature if the installation process of this synchronization tool was made simpler.

Overall, the software works satisfactorily and passes all requirements.

Acknowledgment

The authors would like to thank the BiCS management and education team for the amazing work done.

6. Conclusion

To sum up, in is important to mention that the goals and objectives set for this BSP have been accomplished. The project, which was done by student during the semester studies, helped to develop his skills in writing both scientific and technical reports.

In the scientific deliverable, a definition of open-source software was given and the procedure to accomplish all the requirements to turn software into open-source was presented. Moreover, this guideline was applied for a real-world case, e4l project, created within University of Luxembourg, which allows to assess the quality of provided scientific knowledge.

In the technical deliverable, some tools were introduced, which are supposed to create a convenient working environment for open source project maintainer. These tools allows to review and reject pull requests from public contributors within the private environment in order not to expose some private aspects of the project. This feature was the main requirement to this piece of software and it was accomplished, providing a Proof of Concept prototype of such environment.

Nevertheless, there is always things to improve, and the project could be updated and ameliorated if there would be more time.

7. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person's work without acknowledging the source
  6) Paraphrasing another person's work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
  8) Using another person's unpublished work without attribution and permission ('stealing')
  9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

  Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[BiCS(2021)] BiCS Bachelor Semester Project Report Template. https://github.com/nicolasguelfi/lu.uni.course.bics.global University of Luxembourg, BiCS - Bachelor in Computer Science (2021).

[BiCS(2021)] Bachelor in Computer Science: BiCS Semester Projects Reference Document. Technical report, University of Luxembourg (2021)

[Armstrong and Green(2017)] J Scott Armstrong and Kesten C Green. Guidelines for science: Evidence and checklists. *Scholarly Commons*, pages 1–24, 2017. https://repository.upenn.edu/marketing_papers/181/

8. Appendix

All images and additional material go there.

8.1. State of the Art

References

[1] Fedor Chikhachev. Source code of BSP-S2 technical deliverable. link.
[2] Github Community. The legal side of open source. link.
[3] Github Community. Starting an open source project. link.
[4] E4L. Energy4Life (e4l) website. link.
[5] Google. Copybara: a tool for transforming and moving code between repositories. link.
[6] nexB. Scancode: tool for scanning code for 3rd-party packages. link.
[7] OSI. OSI approved licenses. link.
[8] Open Source Initiative (OSI). The open source definition. link.
[9] Bruce Perens et al. The open source definition. *Open sources: voices from the open source revolution*, 1:171–188, 1999.
[10] SAP SE. CLA assistant. link.
[11] SPI. Debian Social Contract. link.