CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Analyzing the Usage of Traceability Links Within Open Source Software Projects

Bachelor of Science Thesis in Software Engineering and Management

Sebastian Fransson
Tim Jonasson

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

{Analyzing the usage of traceability links within open source software projects}
[Investigating the existence of explicit traceability links within projects that contain some form of modeling.]

Supervisor: Jan-Philipp Steghöfer
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

***Abstract*—Traceability is important when it comes to many different software engineering activities. It is used as a means of enabling smoother maintenance and maneuvering within a repository. Because of its importance, traceability within commercial development has long been a subject of research and development. Meanwhile traceability link existence and usage in open source software (OSS) is not as common if compared to commercial use. In this paper, we identify the usage of explicit traceability links within open source software that contained some form of modeling. We do so through representative repositories extracted from the Lindholmen Data Set, which at the time consisted of 24730 repositories based on Github. We investigate if these links exist as well as what the most common usage of these links are. Then we proceed to discuss the maintenance of traceability links over project lifetimes as well as how the responsibility of maintaining the traceability links is shared. The findings are that six out of our eight representative subjects contain some form of explicit link, although these links are for the most part not updated apart from a few exceptions. In the end the teams do not share the responsibilities of updating these traceability links but seemingly this task is performed only by a few select individuals. We have thus proven that there does indeed exist explicit types of traceability within OSS and that in some cases they are updated in regular intervals during the lifespan of the project. Through these results we present ideas for potential further research within the domain.**

Keywords: *Traceability links*; *Open source development*; *OSS*; *Modeling*; *Hierarchical clustering*; *Software maintenance;*

I. Introduction

Traceability in software engineering is the method of establishing a relation between different software artifacts, such as documentation and code, in an effort to create an easier way to maintain and navigate in a software project. The type of links called 'Traceability links' can come in many forms: they can be the connection between documentation and a part of the source code, or even how different requirements are implemented in the system [1]. In the case of a code-documentation link such as code-requirements, some type of reference between the artifacts will exist, such as a code comment stating that it partly fulfills the requirement of story X. Another option is to investigate the commit message where a code artifact was introduced, since issue tracking tools such as Jira have a specific identification attached to the story/issue which is then provided in the commit message. This identifier is ultimately used to track the story/issue progress [2]. Through this we can then see what requirements a certain code artifact fulfills.

The area of traceability in software engineering has emerged to become a hot topic over the last couple of years [3], [4]. The reason for this emerging interest lies in the fact that traceability is helpful to developers since there are many uses for it. As presented previously we have the maintenance and understanding aspect of these links [5], and these give way to less time spent scouring through projects when trying to find what changes affect what parts of the system. This is especially important for OSS projects, as anyone can pick up the code and start working on it. These traceability links have different forms, such as being implicit or explicit. What current research has shown so far is that OSS projects most commonly rely on implicit links that form between commits and issues, using the method previously mentioned.

There are various methods of finding implicit links within OSS projects, such as looking at the artifacts most commonly committed to a repository over a period of time [6]. One can then assume that these artifacts have some sort of connection, either in the sense of dependency or general documentation covering the area. There have been several studies on the usage of such traceability links already [1], [7], [8]. While these links are surely interesting and provide benefits to developers [9], we believe that there are more avenues that can strengthen the presence of traceability and its usage which have not been thoroughly investigated.

The focus of this paper is on the explicit links that exist between different software artifacts within an OSS project that contains some sort of modeling. These models add helpful information to the projects [10], [11], but it is not known how or if they are linked to the other software artifacts within the projects, e.g., the code, requirements and tests. Therefore our aim in this paper is to show which types of traceability links exist within OSS projects that make use of modeling. In addition, we aim to show how these traceability links are maintained throughout the lifespan of these OSS projects, as well as to investigate how developers share responsibilities in regards to maintaining these links.

The research questions that are posed to conduct these investigations, are the following:

- RQ1: Do explicit traceability links, other than commit-issue, exist within OSS projects using some form of modeling?
- RQ2: What traceability link types are most common in OSS projects that use some form of modeling?
- RQ3: Are traceability links maintained over the lifetime of the project?
- RQ4: Is the maintenance of traceability links performed by the entire team of committers?

The links or lack thereof would provide another helping hand in increasing maintainability as well as how such links can increase understanding around a piece of open source software. In addition it would be the first time that traceability links related to modelling artifacts will be investigated in the context of OSS projects. Therefore we present our findings in hope of providing this type of strengthening information that can aid future research.

Finally we wish to investigate the maintenance of these traceability links in order to show how developers within OSS projects handle and distribute responsibilities when it comes to the upkeep of said links. Through this we will provide an insight into how well traceability links are handled by developers both in regards to the upkeep of the actual links as well as the presence or absence of shared responsibility regarding them.

The remainder of this paper is structured as follows; Section II discusses related studies within the area of traceability links and how such links might be established and found. Section III introduces methodology on how we conducted the research as well as the limitations of our study, while Section IV presents the results and findings of our investigation into OSS repositories. Section V presents a discussion on the findings of the paper. Finally, Section VI concludes the paper and presents possible areas of future investigations.

II. Related Work

A. Traceability in practice

One of the key challenges regarding traceability has always been "the return of investment", meaning that given the large amount of resources required for the establishment, traceability does not always provide developers with an immediate sense of value. Not all of the established links are sure to yield the expected results. However, the few that do will make maintaining the software cheaper [12].

There are several different trace link recovery approaches that have been proposed in order to support trace link creation. These different approaches can be classified into three different "types" as follows; "manual", "semi-automatic" and "fully-automatic" [12]. By semi-automatic it is meant that parts of the process are conducted manually while others are performed automatically. For the most part, the process follows the fully-automatic version, at least up until a certain point. After the automatic traceability extraction has been done, the researchers themselves go through the candidate links and remove the ones deemed incorrect as well as adding links missed by the tool. This type of method can in theory be the "best of both worlds".

Meanwhile, the automatic process generally gives an analyst the ability to input the initial settings but does not allow the analyst any further hand in the decision making process. However this type of tracing does not ensure the correctness of the established traceability relationship as there are no evaluation of the findings by hand. Another issue with using an automatic process is the fact that precision of such an approach is rather low [13].

Finally we have the manual tracing process, which is the process that we have chosen to use for this paper. In this type of process the analyst themselves choose the way to approach the problem out of many different approaches. The analyst themselves go through code elements and documents in order to find and establish eventual traceability elements. This process however is highly time consuming and in the cases of larger projects this type of tracing quickly becomes unfeasible, additionally, there is no guarantee that the analyst will find links in a consistent manner as the process quickly proves a boring task and this tends to make the process error-prone [12].

B. Existence of traceability links in OSS

The usage and understanding of traceability links can be helpful for many different types of tasks, such as program comprehension, maintenance, requirement tracing, impact analysis, and reuse of existing software. There are studies that show that traceability links help with increasing accuracy when performing these types of maintenance tasks, such as presented by Jaber et al. [14]. Finding these links however can prove challenging and time consuming. Antoniol et al. [1] proposes a way to recover traceability links using a technique based on Information Retrieval (IR) methods. The paper suggests that developers often use descriptive naming while creating different software artifacts relating to the code, such as variable names and class names. Then using this information one can proceed to use these names as keywords for searching through documentations and finding related documents. However as previously presented, this type of process runs the risk of low precision.

The authors then propose to use a probabilistic model or a vector space model to rank the different documents by relevance to the search query. Another way to establish such links is through the usage of the latent semantic indexing method or LSI for short [15]. In this paper the authors state that their method of recovering traceability links requires less processing than the papers in their related works section, while also staying independent from factors such as language and paradigms. And thus they conclude that it is better suited for automation. There is no discernible way however to fully automatically find traceability links in a reliable manner. Decisions on the links found by tools are ultimately made by the user [16].

Our focus is related to what we can see from the work of Kadgi et al. [17]. It is shown that software repositories do indeed contain forms and types of traceability links that we can establish between different artifacts when dealing with open source software projects, using methods described in their paper. They propose looking into the change-sets of the projects that they investigate, using the information stored there to relate different artifacts such as through frequency of occurrence together which is a common method used when dealing with open source software repositories. Although this can be done, the usage of commits and issue tracking is the norm for most of the traceability link establishment research being done in the field of OSS [18]. Therefore it is a method which we will not be using, but instead merely allow ourselves to be inspired by.

III. Methodology

The purpose of this paper is to show the usage of traceability links through the collection and analysis of traceability link data, which is contained within OSS projects that have some measure of modeling present. The projects selected had to meet the criteria of containing at least one type of model in order to be incorporated into the final results. The Lindholmen Data Set CSV dump was used as the population for this study [19]. At the time the data-set contained a list of 24730 GitHub projects that contain some form of modeling. The following methodology will be focusing on cutting this population down into only the most relevant group for our research. This group were created by imposing a number of restrictions upon the

data-set that helped to sort out the repositories which met our requirements. Then by way of clustering we selected representatives that were used as the subjects of our research, together with additional data extracted from the resulting representative repositories.

A. Data collection

Because of the data-set being so large, sampling on the population had to be done to decrease this population to a more manageable size. To do this we applied an agglomerative cluster sampling method [20], in order to divide the repositories into smaller clusters based on a few selected criteria. These criteria are as follows:

- Number of models in a project.
- Number of files in a project.
- Number of updates to .uml and .xmi models in a project.
- Latest commit not earlier than the beginning of 2019.

The number of models a project contains is already available through the Lindholmen Data Set and was retrieved from there.

A number of bash scripts were then created to extract the aforementioned information about the repositories while on the master branch. These scripts extracted data directly from the repositories as the information was not present within the selected data-set.

The script created to extract the number of updates was used to look for the occurrences of .uml and .xmi in the committed files of each commit. To calculate the number of times models had been updated in a single repository, we took the number of occurrences of the models found in the commits and then subtracted it by the number of models in the repository which was documented in the Lindholmen Data Set CSV dump. This was done to remove the occurrences where a model was first added to the repository.

The dates were then organized and checked so that the repositories whose last commit was made before January of 2019 could be removed, as we wanted to have projects that had been updated fairly recently. The reason for having this kind of restriction is so that we have repositories that can reflect the current status of OSS projects. We did not check the quality of the models as the only thing that we are concerned about is that they actually exist and have had some form of update throughout the lifespan of the project. Additionally as we were pressed for time regarding the manual information gathering and evaluation, we did not concern ourselves with the state of the models. This was because it would not have been feasible when looking at the sheer number of them being found within certain repositories. We therefore assume that all models that are present within the chosen repositories are relevant to our research as long as they are contained within a repository that has been updated within the year of 2019.

The clusters were created with the help of agglomerative hierarchical clustering [13], [20], using the Ward's method [21]. Before the start of the clustering, there was a need for the data to be normalized. This was to make sure that each of the three data points, i.e. number of models, number of files and number of updates had the same interval for their values when calculating the distance between clusters. In each step of the clustering the closest neighbors were merged into a new cluster for the next iteration of the clustering. To choose which two clusters to merge, we calculated the distance between each of the clusters. By having the number of models, number of files, and the number of updated .uml and .xmi models in each project as our three data points, we calculated the Euclidean distance [22] between the different clusters and as dictated by the Ward's method, merged the closest clusters together.

During this process we decided to set the break-even point to a final cluster count of eight clusters. The reason for this was that we wished to have a manageable number of repositories to check, that were still a representative of the population, as well as the fact that we did not have the knowledge of what might be the right number of clusters for our case. Therefore we follow the example of what was done in the study by Rath et al. [20].

When the clusters had been formed through this type of sorting, we executed a random sampling method on each of the clusters as a way to select the final eight representatives [13].

In order to answer RQ1 and RQ2 we proceeded to manually search the representative projects for explicit traceability information which are related between several parts of the projects. These parts being: between models and code, between models and requirements, between models and tests, between code and tests and finally between code and requirements. Note that traceability information such as between commits and issues were not considered as it was not within the scope of our intended research.

The links between models and code were located by manually searching through code artifacts for specific mentions of models, which forms an explicit traceability link between the code and the model. The models were also searched through for artifacts mentioning different parts of code such as class names or packages.

The links between requirements and models were located through mentions of the models in the specific requirements, such as an update to a certain model in relation to a code change.

The links between models and tests were located by looking through the test comments for mentions of the models such as sequence diagrams and class-diagrams etc. The links between code and tests were found through searching the testing artifacts for mentions of the related code artifacts such as through the test naming and test-suite documentation [23].

Finally, the links between code and requirements were located through an investigation of the code and requirement artifacts for any explicitly mentioned relation.

Beyond looking at these artifacts there was also a possibility of external text or spreadsheet documents existing in the repositories that would link the previously mentioned artifacts together. These types of explicit links are called external links, while there are also those that exist inside of the artifacts

themselves, such as in code comments. Such links are called internal links.

In order to answer RQ3 and RQ4, we extracted additional information from the repositories such as the number of contributors maintaining the traceability links that we found, as well as the number of updates made to them throughout the lifetime of the project. In order to retrieve the number of contributors to files where trace links were present, we decided to manually look through all files containing traceability links and documenting all unique contributors on those files as well as in which of the commits to these files the actual links were updated. Then in order for us to retrieve the information regarding the number of updates made to traceability links throughout their lifespan, we proceeded to manually go through the files containing traceability links and documenting the date each commit was made.

B. Data analysis

The data gathered from the repositories during the data collection which regard to RQ1 and RQ2 was first grouped after their type of traceability link before any analysis on them could be done. The groups that they were divided into are the following five:

- Between models and code
- Between models and requirements
- Between models and tests
- Between code and tests
- Between code and requirements

When the traceability links that had been found they were successfully sorted between the five different groups, the goal was to then to through these groups show what type of links are present as well as which are the most recurring ones between repositories in OSS development.

The gathered information regarding the contributors to the found traceability links, as well as the number of updates made to these links, were then sorted by relevance and presented within tables and through subsections highlighting each repository. One depicting the repositories with number of committers per link as well as the updates of these links within text form.

C. Validity threats

1) Construct Validity: As we have a relatively small base population after the restrictions put on the used data-set, we can see a threat to the construct validity in what is commonly known as a random sampling error. As we are using such a small representative population there will always be some likelihood of encountering a random sampling error as the sample population can never really match the entirety of the population.

2) Internal Validity: Threats towards the internal validity have been mitigated through the usage of both the agglomerative clustering as a way to normalize clusters so that they may be on the same "height", as well as the random sampling that was made on the resulting clusters afterwards in order to fairly select representatives of each cluster that would be used in the study.

3) External Validity: External validity holds many different types of validity that are closely related to empirical research, such as the *population validity*. The population validity in our case we believe to be uncompromised as we performed random sampling upon the population in order to retrieve representatives, which means that we have given each datapoint within the population an equal chance of being selected. As we only use data extracted from repositories situated on Github, we do not have the issue of live subjects compromising the research through predispositions and opinions or possible occurrences of the "Hawthorne effect" [24].

4) Limitations: The conducted study has a very limited time-frame, as such we do not have the ability to secure that we will provide a definitive answer in the end. To manage with the short time-frame of this study we have chosen to perform the study on a small sample size of OSS projects. There is no way to get around this problem in this study but in the case of future research on this subject that has a longer time-frame it will be possible to do a more throughout study of OSS projects that contains modeling documents.

IV. Results

A. Existence and types of traceability links

After filtering away the repositories that do not contain .uml and .xmi files, 7256 of the original 24730 projects from the Lindholmen Data Set were left. Next we imposed the restriction of the repositories having to have been updated in 2019, through looking at the latest commits of the various repositories with the help of an automated script. We ended up with 218 repositories out of the 7256 that were eligible.

These 218 repositories were then used to perform the hierarchical agglomerative clustering, so that we could create the necessary clusters for the final random sampling, from which we retrieved representative repositories which we then searched for explicit traceability links.

Fig. 1 itself is a representation of the clustering procedure, i.e. how the different clusters are merged into what is finally one large cluster. As we are performing an hierarchical agglomerative clustering, we then start with all the 218 repositories within their own clusters and then we start to merge them using the calculated Euclidean distance between the different clusters, thus forming new and larger clusters that are to be merged for the next iteration. The dendrogram then aids with showing the process of selecting the final eight clusters that we used in order to pick representatives for the population. Additionally, we then use it in order to show the final size of the eight clusters that we settled with.

Through performing this type of clustering, as well as drawing up the resulting dendrogram, as seen in Fig. 1, we then picked out the height to settle at where we would have a manageable amount of clusters that we could make use of. We did this by analyzing the dendrogram visually and decided to settle at the height of seven where we ended up with eight

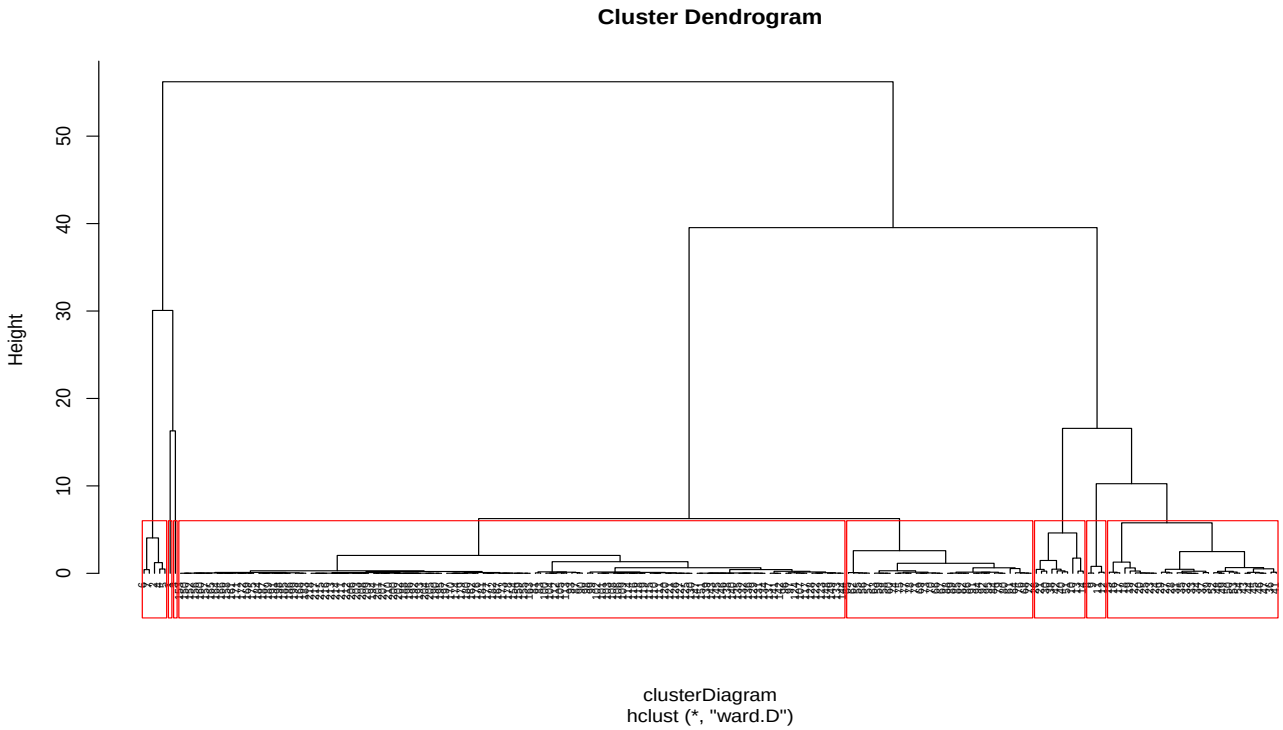

Fig. 1. Clustering procedure as well as marking the choosing of the eight clusters to represent the population in red

total clusters, which was our original goal, so that we could pick representatives from each cluster through the use of the aforementioned random sampling method. The boundaries of these clusters are marked in red.

From a search that we performed by going through the eight representative repositories, we found that there were explicit links established in six out of eight repositories. We continued with sorting the explicit links that we had found through the search of the repositories into the categories mentioned in Section III-B.

Table I shows how many links were found in each of the eight representative repositories, as well as what type of links they were.

The models to code links that we found were all located in a small number of files that contained all links of this type. These files were located in Repository C and were presented in .pdf format, which described the diagrams and how they related the different coding artifacts. This strengthened and also aided in creating the links between models and code. In the case for Repository G, all the links were located in the same file, a .png file that showed the model with documentation on which files it was linked to.

Between code and tests the links were found through the same method for all four repositories that contained them. The name of the test classes were usually linking to the code through the naming of the test classes, and as for the remaining links, there were comments in the testing code that specified which file and part of the code that was being tested.

As for the last type of link that we found, code to requirements links, the only such traceability link that we found was located in a comment of the code were it was specified that the code was created according to the requirements from a specified web-link.

We then ended up with a total set of 44 links between code and tests, while 501 links were found to exist between models and code as well as one link between code and requirements across all the representative repositories. The number of links of each type found in each repository is presented in Table I. We can also see the rarity in the total number of files that exist in all of the searched repositories vs the combined number of links that we found, if we were to do this comparison we would end up with 0,004 links per file, which indicates a very small percentage of links present within such OSS projects.

B. Updates of traceability links in repositories

Presented in the following sub-subsections is information regarding the six repositories that contained some form of explicit links.

1) Repository A: Repository A had four files containing traceability links. Three files which contained six test-code links combined, as well as one file containing one code-requirements link. These files were not created at the same time but instead were created during different years (2014, 2015, 2017 and 2018). All of the links except the one created during 2018 have been updated approximately once a year since their creation. However there have been no updates during 2019 so far.

TABLE I
THE NUMBER OF LINKS IN EACH REPOSITORY

Link type	Models-code	Models-requirements	Models-tests	Code-tests	Code-Requirements
Repository A	0	0	0	6	1
Repository B	0	0	0	0	0
Repository C	493	0	0	0	0
Repository D	0	0	0	6	0
Repository E	0	0	0	17	0
Repository F	0	0	0	15	0
Repository G	8	0	0	0	0
Repository H	0	0	0	0	0

TABLE II
THE NUMBER OF FILES CONTAINING TRACEABILITY LINKS ORDERED BY THE NUMBER OF UNIQUE CONTRIBUTORS IN EACH REPOSITORY

Number of contributors	Total	Total of unique	1 unique	2 unique	3 unique	4 unique	5 unique	6 unique
Repository A	24	7	0	1	1	0	1	1
Repository C	4	1	18	0	0	0	0	0
Repository D	3	3	1	5	0	0	0	0
Repository E	8	5	4	5	3	1	1	0
Repository F	13	5	6	1	5	0	0	0
Repository G	2	1	1	0	0	0	0	0

2) Repository C: Repository C had 18 files containing traceability links. All of these files externally established links between model artifacts and code artifacts. However 16 of them were added at the same date in 2013, and the remaining two were created in 2016. The naming of these files imply that the documents were created outside of GitHub and then added to the GitHub repository when finished. This is because they have dates present in the name of the files and no updates have been made on any of the files since they were added to the repository.

3) Repository D: Repository D had six files containing traceability links. These six files all contained one test-code link. Five of these were added at the same time and then proceeded to get two updates, all at the same time, and then they have not been touched since they were created in 2012. However the remaining file, which was created in 2018, is a code artifact that contains test-code links and has been updated several times since then and has also been updated during 2019.

4) Repository E: Repository E had 14 files containing traceability links. These 14 files all contained 17 test-code links combined. Out of these 14, seven were created at the same date during 2015. They have since then been updated several times, sometimes together and sometimes separately. But overall they have been updated approximately once a year until late 2018. Out of the remaining files, four were created during 2018 and have not received any updates. Two files have also been updated several times a year since their creation in 2016 until early 2019. The final file was created during 2019 and has been updated several times since then.

5) Repository F: Repository F had twelve files containing traceability links. These twelve files all contained 15 test-code links combined. Out of these twelve files, six were created during 2013 and 2014 that have yet to receive any updates since then. One file was created during 2013 and the remaining five during 2012. The ones created during 2012 got a few updates during the same year but then were not updated again until 2017 when they got updates around the same time as the link created in 2013.

6) Repository G: Repository G had one file containing eight traceability links. These eight links were all model-code. This file was created during 2015 and has not received any updates since.

C. Contributors on traceability links

In Table II the data for the number of unique contributors on each link is shown as well as the total amount of contributors for the entire project.

The data shows that there is rarely a case where all of the contributors are participating in updating the traceability links within the repository. Out of the six repositories that we found traceability links in, only one of the repositories had all their contributors participating in the maintenance of a traceability link. However, this repository had a low number of total contributors, in total three. As for the other repositories, one of them only had two contributors but both of them did not participate in the maintenance of the traceability links. For the remaining four repositories that contained traceability links, they had a higher number of contributors than the two other repositories, but the number of contributors that participated in the maintenance of the traceability links were less than the total amount of contributors found in these repositories.

From the data we can also see that there is rarely a case where our sample repositories have a consistent number of contributors that are maintaining the traceability links. In Table II it is shown that the repository with the highest number of unique contributors on traceability links has some links with two contributors but also has up to six unique contributors on one link.

V. Discussion

A. RQ1 & RQ2

The two questions that we are trying to answer in this subsection are "Do explicit traceability links, other than commit-issue, exist within OSS projects using some form of modeling?" and "What traceability link types are most common in OSS projects that use some form of modeling?".

We can see from the amount of links that could be found in the projects that were searched that different types of explicit traceability links are not very popularly used within these types of OSS projects, which makes it surprising that we found traceability links within six out of the eight repositories that were within our sample of representatives. Even though models are used and in many cases updated quite a number of times, there is still nothing that is linking them together in a way that would make sense from a models to code class standpoint, with the exception of two cluster samples. With this we are referring to the fact that in six out of our eight samples, we found no correlation between the models and code that could link them together. The test to code links that were found followed the same pattern across the repositories, which is understandable since it is of modern coding standard to name test files appropriately after what file they are testing, as well as showing in code comments what the specific test is indeed testing. This however, is not the only type of link that we have seen being used, as comments that refer to the names of classes under test also appear from time to time, showing another way to link two such artifacts together in an explicit manner.

While there were indeed quite a few links that were models to code, it did appear that it is not the most popularly used link type, as most of the links were found within a single repository which was using them extensively. Even though the model-code links were numerous, the most popularly used link type across the cluster samples was the test-code links that came from naming conventions and comments depicting what code artifact it tested.

As for models to requirement links we failed to find any instance of this type of link. From this we can not conclude anything other than that we failed to prove that they exist. The same goes for the models to test links that we also did not find in any of the repositories.

We did however manage to find one link between code and requirements. This link was formed through the code comment stating that it fulfilled a certain requirement. In addition, there was a web-link attached in the comment that linked to the specified requirement. From this one link we can conclude that there are links between code and requirements within OSS, but we do not know how widely they are used. There is a possibility that there are no code to requirements links other than the one that we found. But since we found one link then we believe that there is a possibility of there being more of this type of link within other OSS projects. Taking all of these points into account we can see that the overall usage of explicit traceability links is scarce and mainly used when creating tests for pieces of code within the system.

B. RQ3

In this subsection we are discussing the research question "Are trace links maintained over the lifetime of the project?".

From the results that we have gathered and shown in the previous section it becomes apparent that not many traceability links are maintained and also not very frequently. As most of the links that are maintained get updated around once every year, which could also be argued to be a reflection upon how often the files that have these links are being changed which might not necessarily mean that the link itself has been altered at all. The question which we are trying to answer with these results is after all if traceability links are updated during the lifetime of the project, and some definitely are updated. However, considering the question of how actively they are maintained, it can be seen that the developers do in fact update these links fairly rarely.

C. RQ4

This subsection is about research question four, "Is the maintenance of trace links performed by the entire team of committers?".

As shown in Table II, there are several people that are participating in maintaining the traceability links. However we can also see that for our sample there is no relation between the total number of contributors and the number of people that are maintaining the traceability links. As such it can be seen that not all the members of the teams are taking an active role in maintaining the traceability links. Through this we can argue that it does not seem to be something that the developers within OSS believe to have the need of being a shared responsibility between team members, but rather is restricted to be kept by a select number of individuals.

VI. Conclusion

We can see through the gathered results and the discussion presented that OSS projects do indeed use explicit traceability links within certain areas. We can also see that the most frequent type of traceability link is the one that ties together testing artifacts and code artifacts. These links establish themselves through comments pointing towards code artifacts, specifically what parts of the code that is being tested, and in some cases why. We can also see that the models-code is in a close second with the overwhelming amount of links found, even though they were only present in two out of eight repositories. Meanwhile the test-code links were found in four, although significantly fewer in number.

Although these links are established with purpose of aiding in maintaining the software, the links themselves are not often updated during the lifetime of the repository. This can be pointed out to be because the features are not used, bad implementation or just simply because of neglect. Additionally we see that the upkeep of explicit traceability links is not a shared activity among the contributors on an OSS project.

Instead it is done by a few select individuals that execute their upkeep with what was found to be a usual frequency of around one year between each update.

We see possibilities to extend this research further in the future by investigating the developer stance towards traceability and its practical usage. In addition information can be gathered regarding the knowledge of traceability within the open source community, and on how traceability links have helped the contributors when contributing to OSS projects. This research can be done by performing interviews or sending out a simple questionnaire to the developers of OSS projects that contain some sorts of explicit traceability links.

REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE transactions on software engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[2] M. Ortu, G. Destefanis, B. Adams, A. Murgia, M. Marchesi, and R. Tonelli, "The jira repository dataset: Understanding social aspects of software development," in *Proceedings of the 11th international conference on predictive models and data analytics in software engineering*, ACM, 2015, p. 1.

[3] G. Spanoudakis and A. Zisman, "Software traceability: A roadmap," in *Handbook Of Software Engineering And Knowledge Engineering: Vol 3: Recent Advances*, World Scientific, 2005, pp. 395–428.

[4] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, and J. Maletic, "The grand challenge of traceability (v1. 0)," in *Software and Systems Traceability*, Springer, 2012, pp. 343–409.

[5] P. Mäder and A. Egyed, "Do developers benefit from requirements traceability when evolving and maintaining a software system?" *Empirical Software Engineering*, vol. 20, no. 2, pp. 413–441, 2015.

[6] H. Kagdi, J. I. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*, IEEE, 2007, pp. 145–154.

[7] S. K. Sundaram, J. H. Hayes, A. Dekhtyar, and E. A. Holbrook, "Assessing traceability of software engineering artifacts," *Requirements engineering*, vol. 15, no. 3, pp. 313–335, 2010.

[8] M. Rath, D. Lo, and P. Mäder, "Analyzing requirements and traceability information to improve bug localization," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, IEEE, 2018, pp. 442–453.

[9] A. Egyed and P. Grünbacher, "Supporting software understanding with automated requirements traceability," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 05, pp. 783–810, 2005.

[10] O. Badreddin, T. C. Lethbridge, and M. Elassar, "Modeling practices in open source software," in *IFIP International Conference on Open Source Systems*, Springer, 2013, pp. 127–139.

[11] R. Hebig, T. H. Quang, M. R. Chaudron, G. Robles, and M. A. Fernandez, "The quest for open source projects that use uml: Mining github," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ACM, 2016, pp. 173–183.

[12] J. Cleland-Huang, O. Gotel, A. Zisman, *et al.*, *Software and systems traceability*, 3. Springer, 2012, vol. 2.

[13] G. G. Van Ryzin, "Cluster analysis as a basis for purposive sampling of projects in case study evaluations," *Evaluation Practice*, vol. 16, no. 2, pp. 109–119, 1995.

[14] K. Jaber, B. Sharif, and C. Liu, "A study on the effect of traceability links in software maintenance," *IEEE Access*, vol. 1, pp. 726–741, 2013.

[15] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th international conference on software engineering*, IEEE Computer Society, 2003, pp. 125–135.

[16] A. Marcus, X. Xie, and D. Poshyvanyk, "When and how to visualize traceability links?" In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, ACM, 2005, pp. 56–61.

[17] H. Kagdi and J. Maletic, "Software repositories: A source for traceability links," in *International Workshop on Traceability in Emerging Forms of Software Engineering (GCT/TEFSEâ07)*, 2007, pp. 32–39.

[18] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," in *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, ACM, 2011, pp. 31–37.

[19] *Lindholmen data set*, http://oss.models-db.com/, Accessed: 2019-02-26.

[20] M. Rath, M. T. Tomova, and P. Mäder, "Selecting open source projects for traceability case studies," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*, Springer, 2019, pp. 229–242.

[21] F. Murtagh and P. Legendre, "Ward's hierarchical clustering method: Clustering criterion and agglomerative algorithm," *arXiv preprint arXiv:1111.6285*, 2011.

[22] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and image processing*, vol. 14, no. 3, pp. 227–248, 1980.

[23] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, IEEE, 2009, pp. 209–218.

[24] R. McCarney, J. Warner, S. Iliffe, R. Van Haselen, M. Griffin, and P. Fisher, "The hawthorne effect: A randomised, controlled trial," *BMC medical research methodology*, vol. 7, no. 1, p. 30, 2007.