IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.0429000

Refining Software Clustering: The Impact of Code Co-Changes on Architectural Reconstruction

STANA ADELINA DIANA[1] and SORA IOANA[2]
[1]Computer Science and Engineering Department "Politehnica" University of Timisoara, Romania (e-mail: stana.adelina.diana@gmail.com)
[2]Computer Science and Engineering Department "Politehnica" University of Timisoara, Romania (e-mail: ioana.sora@cs.upt.ro)

Corresponding author: Stana Adelina Diana (e-mail: stana.adelina.diana@gmail.com).

ABSTRACT Version control systems primarily offer support for tracking and managing changes in software code, but they can also provide a great deal of additional information about the managed software. Changes in multiple software entities simultaneously can imply that these entities are connected to one another. Software entities that change at the same time (code co-changes) are treated as a distinct category of software dependencies. In some cases, these co-changes are used together with other types of dependencies, such as structural dependencies or lexical dependencies, to enhance the understanding of a software system.
This paper proposes using code co-changes in software clustering for architectural reconstruction. Structural dependencies are the most commonly used type of dependencies in software clustering for architectural reconstruction, so we will use clustering solutions obtained from structural dependencies as the baseline for our evaluations. Our approach will be applied to five open-source projects found on GitHub. For each of the projects, we will compare and evaluate the clustering obtained by using only co-changes and the clustering obtained by using co-changes combined with structural dependencies with the baseline clustering generated solely from structural dependencies.

INDEX TERMS Architectural reconstruction, code co-changes, logical dependencies, software clustering, software dependencies, versioning system.

I. INTRODUCTION

Software systems often face a lack of documentation. Even if there was original documentation at the beginning of development, over the years it may become outdated or lost. Additionally, the original developers may leave the company, taking with them knowledge about how the software was designed. This situation challenges the teams when it comes to maintenance or modernization. In this context, recovering the system's architecture is essential. Understanding the system's architecture helps developers better evaluate and understand the nature and impact of changes they need to make. One technique to aid in reconstructing the system architecture is software clustering. Software clustering involves creating cohesive groups (modules) of software entities based on their dependencies and interactions.

Among the dependencies that can be used for software clustering are structural dependencies (relationships between entities based on code analysis), lexical dependencies (relationships based on naming conventions), and code co-changes/logical dependencies (relationships between entities extracted from the version control system), among others. Combining multiple types of dependencies, rather than relying on just one type, can be a good approach to generate better results. However, it requires fine-tuning the amount of dependencies used from each category and scaling the coefficients attached to them. Combining dependencies without considering these aspects might lead to results that are less effective than using an individual type of dependency alone.

In this paper, we assess whether using structural dependencies combined with logical dependencies can provide better results than using each type of dependency alone. The structural dependencies are used as they are extracted from static code analysis. The logical dependencies are filtered co-changes from the version control system. The reason behind filtering the co-changes and not using them as they are in the versioning system is to enhance their quality and make them easier to combine with structural dependencies, as their size can outnumber the structural dependencies [1].

To evaluate the results, we generate software clustering on five open-source projects and use two types of metrics for comparison. One of the metrics is MQ (Modularization Quality), which evaluates the modularization quality based on

the interaction between the modules and does not require any additional input besides the clustering result [2]. The other is the MoJo (Move and Join) metric, a commonly used metric for evaluating the similarity between two different software clustering results [3]. For this metric, we manually generate a base of comparison for the clustering result and compare it against this baseline.

In Section II, we review the related work and previous studies that used various dependencies for software clustering and their metrics for evaluation. Section III details the workflow and implementation of our approach, including the extraction and filtering of dependencies, and the clustering algorithm used. The results of our experiments on five open-source projects are presented in Section IV. Section V evaluates our results by using the Modularization Quality (MQ) metric and the Move and Join (MoJo) metric. We also manually analyze some of the clustering solutions. Finally, Section VI contains our conclusions, findings, and potential directions for future work.

A. ABBREVIATIONS AND ACRONYMS

The following abbreviations and acronyms are used throughout this article:

- **LD**: Logical Dependencies
- **SD**: Structural Dependencies
- **MQ**: Modularization Quality Metric
- **MoJo**: Move and Join Metric

Logical Dependencies (LD) refer to the relationships between software entities that have been extracted and filtered from the versioning system. If these entities are not filtered, they are simply referred to as co-changes. Structural Dependencies (SD), on the other hand, refer to the relationships between software entities extracted from static code analysis. Modularization Quality Metric (MQ) and Move and Join Metric (MoJo) are metrics used to evaluate software clustering results.

II. RELATED WORK

III. WORKFLOW AND IMPLEMENTATION

To achieve our goal of evaluating how the quality of clustering solutions is impacted by logical dependencies, we developed a Python tool capable of using any type of dependency, either alone or combined with other types of dependencies, as long as they are provided in CSV format. The tool clusters and evaluates software clustering solutions using either the MQ metric or the MoJo metric. In the following subsections, we present how we obtain the structural dependencies and logical dependencies used, the type of clustering algorithm we use, and the tool's workflow.

A. STRUCTURAL DEPENDENCIES

The structural dependencies are obtained using a tool from our previous research. While the tool primarily exports the key class ranking of a software system, it also has the capability to export the data in CSV format. The exported dependencies are directed and weighted based on the type of dependency they represent.

B. LOGICAL DEPENDENCIES

We refer to logical dependencies as the filtered co-changes between software entities. A co-change occurs when two or more software entities are modified together during the same commit in the version control system. Co-changes indicate that these entities are likely related or dependent on each other, directly or indirectly.

There is a degree of uncertainty associated with co-changes. Compared to structural dependencies, where the presence of a dependency is certain, co-changes are less reliable. For example, if the system was migrated from one version control system to another, the first commit will include all the entities from the system at that point in time. Should we consider all these entities as related to one another in this case? This would introduce false dependencies and reduce the likelihood of achieving accurate results when combining them with more reliable types of dependencies.

Even if we address the issue of the first commit, it can still happen that a developer resolves multiple unrelated issues in the same commit (even though this is not recommended by development processes).

To solve this problem, in our previous works, we refined some filtering methods to ensure that the co-changes that remain after filtering are more reliable and suitable for use with other dependencies or individually [4], [5], [6]. Based on our previous results, the filters we decided to use further in our research are the commit size filter and the strength filter. Both filters are used together, and the end result is the set of logical dependencies that we use to generate software clusters.

1) Commit Size Filter

The commit size filter filters out all co-changes that originate from commits that exceed a certain number of files.

We are interested in extracting dependencies from code commits that involve feature development or bug fixes because that is when developers change related code files. If multiple unrelated features or bug fixes are solved in a single commit, it will appear as though all the entities in those files are related, even if they are not.

One scenario where this issue arises is the first commit of a software system when it is ported from one versioning system to another. This commit will contain many changed code files, but these changes do not originate from any functionality change, so they generate numerous irrelevant co-changes for the system.

A similar scenario occurs with merge commits. A merge commit is created automatically when you perform a merge operation to integrate changes from one branch into another. After integration, all commits from the branch are added to the target branch, and on top of that, there is the merge commit containing all changes from the commits merged into a single commit. Since this commit contains only a merge of multiple smaller, related issues/features solved, it is better to gather

information from the smaller commits rather than from the overall merge commit.

What both scenarios above have in common is the large number of files involved in the commits. Based on our previous research and measurements regarding the number of files involved in a commit, we chose to set a threshold of 10 files [4], [5]. Therefore, all co-changes that originate from commits with more than 10 changed code files are filtered out.

2) Strength Filter

This filter focuses on the reliability of the co-changes. If a pair of co-changing entities appears only once in the entire history of the system, it might be less reliable than a pair that appears more frequently.

Zimmermann et al. introduced the support and confidence metrics to measure the significance of co-changes [7].

The *support metric* of a rule $(A \rightarrow B)$ where A is the antecedent and B is the consequent of the rule, is defined as the number of commits (transactions) in which both entities are changed together.

The *confidence metric* of $(A \rightarrow B)$, as defined in Equation (1) , focuses on the antecedent of the rule and is the number of commits together of both entities divided by the total number of commits of (A).

$$\text{Confidence}(A \rightarrow B) = \frac{\text{Nr. of commits containing } A \text{ and } B}{\text{Nr. of commits containing } A} \tag{1}$$

The confidence metric favors entities that change less and more frequently together, rather than entities that change more with a wider variation of other entities.

Assuming that (A) was changed in 10 commits and, of these 10 commits, 9 also included changes to (B), the confidence for the rule $(A \rightarrow B)$ is 0.9. On the other hand, if (C) was changed in 100 commits and, of these 100 commits, 50 also included changes to (D), the confidence for the rule $(C \rightarrow D)$ is 0.5. Therefore, in this scenario, we would have more confidence in the first pair $(A \rightarrow B)$ than in the second pair $(C \rightarrow D)$, even though the second pair has more than five times more updates together.

To favor entities that are involved in more commits together, we calculated a *system factor*. This system factor is the mean value of the support metric values for all entity pairs.

The system factor is multiplied with the calculated confidence metric value. In addition, since we plan to use the metric values as weights, together with the weights of the structural dependencies, we multiply by 100 to scale the metric value to be supraunitary, and we clip the results between 0 and 100.

We refer to this addition to the original calculation formula as strength metric, and it is defined in Equation (2).

$$\text{strength}(A \rightarrow B) = \text{confidence}(A \rightarrow B) \times 100 \times \text{system factor} \tag{2}$$

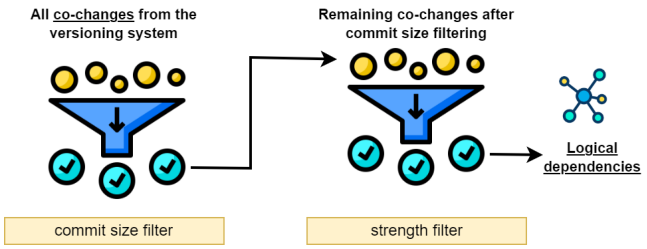

FIGURE 1. Filter application process

3) Filter Application Process

The overall filter application process is illustrated in Fig. 1. We begin by extracting all co-changes from the versioning system, and the first filter applied is the commit size filter. The commit size filter has a strict threshold of 10 files, meaning that any co-changes from commits involving more than 10 files are filtered out.

The co-changes that remain after applying the commit size filter are then processed using the strength filter. The strength filter uses multiple thresholds, specifically 10 different thresholds. We start with a threshold of 10 and increment it by 10 until we reach a maximum value of 100. The reason for not using a fixed threshold is to assess how different strength thresholds affect our cluster generation.

To extract and filter the co-changes, we used a previously developed tool [4]. This tool takes as input the GitHub repository address and the threshold values for commit and strength filters. The tool clones the repository, downloads all commit diffs starting from the first commit, examines all files changed in each commit to identify which entities have changed in those files, and creates undirected co-change dependencies between all changed entities within a commit.

The commit size filter is applied to these undirected co-change dependencies, since the metric value for $(A \rightarrow B)$ is the same as for $(B \rightarrow A)$. For the strength filter, each co-change dependency is converted into a directed co-change dependency, so for each $(A \rightarrow B)$ dependency we have both $(A \rightarrow B)$ and $(B \rightarrow A)$. This conversion is necessary because, as mentioned in the previous section, the confidence filter, upon which the strength filter is built, evaluates the antecedent of the rule. Thus, the metric value for $(A \rightarrow B)$ differs from the metric value for $(B \rightarrow A)$.

The remaining dependencies after applying the strength filter are then exported to a CSV file for further use.

C. LOUVAIN CLUSTERING ALGORITHM

The Louvain algorithm was originally developed by Blondel et al. and is used for finding community partitions (clusters) in large networks. The algorithm begins with a weighted network of N nodes, initially assigning each node to its own cluster, resulting in N clusters. For each node, the algorithm evaluates the modularity gain from moving the node to the cluster of each of its neighbors. Based on the results, the node is moved to the cluster with the maximum positive modularity

gain. This process is repeated for all nodes until no further improvement in modularity is possible [8].

The modularity of a cluster is a value that ranges from -1 to 1, which measures the density of connections inside clusters compared to connections between clusters [9].

D. CLUSTERING RESULT EVALUATION

1) Modularity Quality Metric
2) MoJo Metric

E. TOOL WORKFLOW OVERVIEW

For cluster generation and result evaluation, we developed a tool that accepts one or multiple dependency files as input and outputs the cluster results along with the values for both the MoJo and MQ metrics. The dependencies are stored in a CSV file with the format: antecedent, consequent, weight. The tool reads each line, adding the antecedent and consequent as nodes in a directed graph, with the weight serving as the edge weight. If multiple files are processed and the same dependency is found in them, the weights are summed.

Based on the constructed directed graph, the Louvain clustering algorithm is performed, and the solution is evaluated using the MQ metric. We then manually generated baseline solutions for clustering to perform MoJo metric evaluation, comparing the baseline solution with the generated Louvain clustering solution.

IV. RESULTS

V. EVALUATION

VI. CONCLUSION

FIRST Add text here

SECOND Add text here

REFERENCES

[1] Ajienka, Nemitari & Capiluppi, Andrea. (2017). Understanding the Interplay between the Logical and Structural Coupling of Software Classes. Journal of Systems and Software. 134. 10.1016/j.jss.2017.08.042.

[2] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242), 1998, pp. 45–52.

[3] V. Tzerpos and R. C. Holt, "MoJo: a distance metric for software clusterings," Sixth Working Conference on Reverse Engineering (Cat. No.PR00303), Atlanta, GA, USA, 1999, pp. 187-193, doi: 10.1109/WCRE.1999.806959.

[4] Stana, Adelina-Diana & Şora, Ioana. (2023). Logical dependencies: Extraction from the versioning system and usage in key classes detection. Computer Science and Information Systems. 20. 25-25. 10.2298/CSIS220518025S.

[5] Stana, Adelina-Diana & Şora, Ioana. (2019). Analyzing information from versioning systems to detect logical dependencies in software systems. 000015-000020. 10.1109/SACI46893.2019.9111582.

[6] Stana, Adelina-Diana & Şora, Ioana. (2019). Identifying Logical Dependencies from Co-Changing Classes. 486-493. 10.5220/0007758104860493.

[7] T. Zimmermann, P. Weibgerber, S. Diehl and A. Zeller, "Mining version histories to guide software changes," Proceedings. 26th International Conference on Software Engineering, Edinburgh, UK, 2004, pp. 563-572, doi: 10.1109/ICSE.2004.1317478.

[8] Blondel, Vincent & Guillaume, Jean-Loup & Lambiotte, Renaud & Lefebvre, Etienne. (2008). Fast Unfolding of Communities in Large Networks. Journal of Statistical Mechanics Theory and Experiment. 2008. 10.1088/1742-5468/2008/10/P10008.

[9] Newman, Mark. (2006). Newman MEJ.. Modularity and community structure in networks. Proc Natl Acad Sci USA 103: 8577-8582. Proceedings of the National Academy of Sciences of the United States of America. 103. 8577-82. 10.1073/pnas.0601602103.