Integrating Logical Dependencies in Software Clustering: A case study on Apache Ant

1st Adelina Stana
Computer Science and Engineering Department
"Politehnica" University of Timisoara
Timişoara, România
stana.adelina.diana@gmail.com

2nd Ioana Şora
Computer Science and Engineering Department
"Politehnica" University of Timisoara
Timişoara, România
ioana.sora@cs.upt.ro

***Abstract*—Extracted code co-changes from versioning systems have multiple applications across numerous fields, including fault detection, software reconstruction, key class identification, among others. This paper will focus on the influence of code co-changes on software clustering for architectural reconstruction. Specifically, we will analyze their impact on the clustering solution of Apache Ant in order to assess whether co-changes usage enhances the quality of the obtained solution.**

***Index Terms*—logical dependencies, logical coupling, mining software repositories, code co-change; co-changing entities, software evolution, clustering**

I. Introduction

The software architecture helps developers in gaining a better understanding of the system and its expected behavior. Additionally, it is also of great help in change management. By knowing the existing system architecture, project managers can assess whether a requested change can be easily implemented or not.

Architecture reconstruction appears in contexts where a software system lacks documentation entirely, or when existing documentation fails to accurately reflect changes within the system. This process involves identifying the modules or subsystems within the system, which is done through software clustering. Additionally, architectural reconstruction can be used for validating whether a documented modularization aligns with the actual structure of the system.

Previous research has revealed that dependencies extracted from versioning systems are distinct from those extracted from code, which are called logical dependencies, implying that using them could enhance our understanding of the system [1], [2], [3].

We use logical dependencies to enhance the results of clustering methods that previously relied solely on dependencies extracted from code.

In Section II, we detail how we extract logical dependencies from the versioning system and how different filter methods affect the logical dependencies size and reliability. Section III discusses previous approaches to software clustering, including the inputs and evaluation methods used by other authors. Our methodology for generating clustering solutions based on logical dependencies and our evaluation approach are presented in Section IV. Section V presents our findings, while Section VI provides a more thorough analysis of two solutions, one incorporating logical dependencies and the other without. Finally, our conclusions and plans for future work are summarized in Section VII.

II. Logical dependencies

During development processes, numerous software entities are changed. It has been observed that entities changing together are not only those structuraly dependent on one another (can be found by static code analysis) but also include entities that are functionally dependent on one another. This functional dependency, however, cannot be observed by examining the code.

This type of entities, that consistently change together throughout development activities, are called logical dependencies or coupling. This concept was initially introduced by Gall et al. [4] and has numerous fields of application.

A problem with dependencies extracted from the versioning system is their potential to become excessively numerous. This often results from commits that contain numerous files, thereby generating thousands of dependencies from a single commit. In our previous research, we examined several software systems and determined that commits involving a large number of files are frequently unrelated to code changes. [5].

Additionally, the reliability of co-changes is another problem; certain entities may only change together once throughout the entire versioning history, making them less reliable than entities that change together hundreds of times, for instance.

Previous research uses two known metrics in order to solve this problems: *support* and *confidence*, defined as in equations (1) and (2). Given a dependency where $A \rightarrow B$, the support metric measures the frequency of updates that two entities share within the versioning system. The confidence metric calculates the proportion of updates between two entities relative to the total updates of the antecedent (A) or consequent (B) entity [1], [2], [6].

$$\text{support}(A \rightarrow B) = \text{freq}_{\text{total commits}}(A \cup B) \tag{1}$$

$$\text{confidence}(A \rightarrow B) = \frac{\text{support}(A \rightarrow B)}{\text{freq}_{\text{total commits}}(A)} \tag{2}$$

In our previous research, we focused on identifying metrics that enhance the reliability of dependencies extracted from the versioning system [7], [5]. One of the metrics is the *commit size metric*, which involves extracting dependencies from commits that do not exceed a specific commit size limit, thereby reducing the possibility of extracting an overly large number of dependencies. Additionally, we developed the strength metric, which serves as a refinement of the more known *confidence metric* [8].

The strength metric, as defined in equation (4), was developed to obtain a better reflection of the system and its values. For instance, when using the confidence metric, two entities that update together only once, and this is also their only update, will have the highest possible score on the confidence metric, a result that is not desirable. On the other hand, entities that undergo hundreds of updates together, and even more updates with other entities, will receive a lower confidence metric score. Thus, the confidence metric will favor entities that update less and always together.

The strength metric is calculated by multiplying the confidence metric value with a system factor, which is determined by the average number of updates for all entities in the system. In this way, a very high confidence score resulting from only a few updates will be adjusted downwards, while a low confidence score that is based on a big number of updates will be increased.

$$\text{system factor for } (A \to B) = \frac{\text{support } (A \to B)}{\text{system mean}} \tag{3}$$

$$\text{strength}(A \to B) = \frac{\text{support}(A \to B) \times 100}{\text{freq}_{\text{total commits}}(A)} \times \text{system factor} \tag{4}$$

The confidence metric score ranges from 0 to 1, with 1 representing the best value. On the other hand, the strength metric ranges from 0 to 100, where 100 represents the best possible score.

Co-changes that meet both the commit size metric threshold and the strength metric threshold are referred to as *logical dependencies*.

III. Related work

Software clustering is the process of organizing software entities into groups(clusters) that correspond to the systems modules. There are numerous algorithms that can be used for software clustering such as hierarchical algorithms like Minimum Spanning Tree and Louvain that cluster software entities by their hierarchical relationships [9], [10]. Partitioning algorithms, like k-means, that organize data into clusters by similarity [11]. Density-based algorithms, such as DBSCAN, focus on areas of higher density to form clusters [12], and many others [13].

Regarding the input data used by software clustering algorithms, some approaches rely solely on data derived from code dependencies (structural dependencies) to establish clusters [10], [14]. The Bunch tool, developed by Mitchell and Mancoridis, utilizes source code analysis along with hill-climbing and genetic algorithms to create clusters from the code data [15], [16], [17].

Other approaches use lexical dependencies extracted from code comments [18] or the name of the source files [19] [20].

A more recent approach involves using data from the system's historical changes in order to gain more knowledge about the system [21]. Co-changes have been used to analyze how the system's modularity evolves over time [22] or how co-changes impact the packaging restructuring [23].

Prajapati et al. used structural dependencies, lexical dependencies and co-changes from the versioning system to enhance system modularization [24].

When it comes to evaluating the obatined clustering solutions, if a reference solution exists, metrics can be applied to evaluate the similarity between the reference clustering solution and the obtained clusterings. One such metric is the MOJO distance, MOJO measures the minimum number of Move and Join operations required to transform one clustering solution into another [25].

If no reference solution exists, then metrics that evaluate the cohesion of the clustering solution, based on the input graph, can be used. One such metric is the Modularization Quality (MQ) metric. As defined by Mitchell and Mancoridis [26], [27], is extensively used to assess the results of clustering techniques. While it is primarily applied to clusters formed from structural dependencies [15], [16], [17], it can be used also for evaluating clusters derived from other types of dependencies [24].

IV. Method

To generate and evaluate the results of our approach, we initially create a clustering solution based on structural dependencies alone. Then we incorporate dependencies extracted from the versioning system with the structural dependencies to produce a second clustering solution. Lastly, we construct a clustering solution that relies entirely on logical dependencies, and we compare all three solutions obtained.

A. Dependencies extraction

To obtain the logical dependencies for further use, we use a Python tool developed in our previous research [8]. This tool retrieves all necessary data from GitHub [28] using git commands and then processes it. The workfow of this tool is presented in Figure 1.

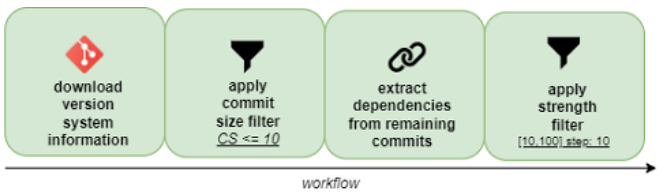

Fig. 1. Logical dependencies extraction workflow

The initial phase involves applying a commit size filter to exclude all commits that involve changes to more than 10

files. Following, the tool creates dependencies based on these commits, establishing a dependency link between each entity in a modified file and all entities in other files modified by the same commit.

Next, the tool calculates the strength metric as described in section II. It then filters out any dependencies falling below the set strength metric threshold. For our experiments, we initiated with a strength metric threshold of 10, incrementing in steps of 10 up to 100 (the maximum value for the metric).Since the structural dependencies are oriented dependencies, we also export oriented logical dependencies. For entities A and B that update together, we calculate the strength metric for $A \rightarrow$B and $B \rightarrow$A, and export the dependency orientation that is above the set threshold. Finally, the results are exported in comma separated values format file (.csv).

Not only do logical dependencies require specialized extraction, but also the relationships among various code components, known as structural dependencies, necessitate specialized analysis tools for their extraction from the codebase. We export the structural dependencies in the same .csv format as logical dependencies, utilizing a tool that was previously developed [8], [29].

B. Louvain Clustering algorithm

Following the generation of logical dependencies, we use them as input for software clustering. For this, a Python script was developed to extract both structural and logical dependencies from their respective files, forming a dependency matrix. This matrix is used as the input for the Louvain Clustering algorithm. The workflow of the tool is presented in figure 2.

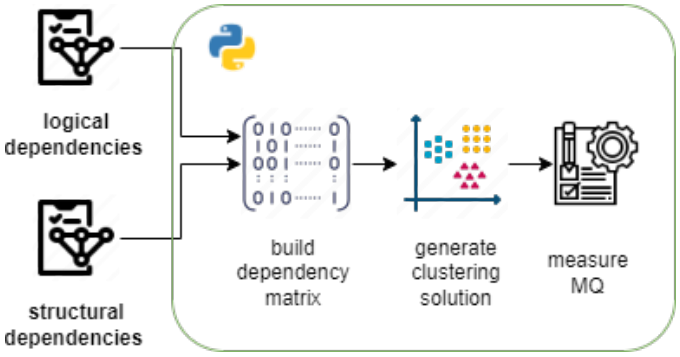

Fig. 2. Clustering solution creation process diagram

Louvain Clustering is a community detection algorithm designed for finding clusters or communities in complex networks. The Louvain method involves a greedy algorithm that moves nodes between clusters to obtain clusters that are highly interconnected [30].

C. Evaluation using MQ metric

The resulting clustering solution is then evaluated using the Modularity Quality (MQ) metric . The MQ metric can vary between -1 and 1, where -1 means no cohesion within the modules, and 1 means no coupling between the modules [26].

To compare the clustering solutions and their MQ evaluation, we generated clustering solutions under three different scenarios. The first one by using only structural dependencies, the second one by using only logical dependencies, and the third one by using logical and structural dependencies to populate the dependency matrix that is further used in cluster generation.

V. Results

The results of the scenarios mentioned in section IV are presented in tables I and II. In both tables, the first row shows the results from the clustering analysis relying solely on structural dependencies (scenario 1, which is used as a baseline for comparison). We highlighted the results of structural dependencies in both tables for a more easy comparison with the results involving also logical dependencies.

Table I presents the results when using only logical dependencies for software clustering. All rows, except the first one, represent the measurements obtained from logical dependencies extracted with varying strength thresholds.

TABLE I
Louvain clustering results for LD only

Dataset	Entities Count	Cluster count	MQ metric
SD only	517	14	0.085
LD strength 10%	320	75	0.383
LD strength 20%	215	53	0.547
LD strength 30%	174	44	0.558
LD strength 40%	152	40	0.58
LD strength 50%	138	35	0.604
LD strength 60%	120	34	0.587
LD strength 70%	106	32	0.577
LD strength 80%	92	29	0.576
LD strength 90%	79	24	0.606
LD strength 100%	64	19	0.611

Table II presents the results when using logical dependencies combined with structural dependencies for software clustering. All rows, except the first one, represent the measurements obtained from logical dependencies extracted with varying strength thresholds combined with structural dependencies extracted from static code analysis.

In both tables, the second column shows the total count of different entities forming dependencies in the system. The thirth column indicates the number of clusters obtained after injecting the dependencies to the Lovian algorithm. Lastly, the forth column indicates the MQ metric result computed for on the obtained clusters.

Can be observed in Table I that the MQ results for clustering solutions based solely on logical dependencies (LD) are not better than those based solely on structural dependencies (SD). In Column 2, it is noticeable that the total number of entities used decreases as the strength threshold increases. This trend is expected because stricter thresholds filter out more entities. But it is noticeable that this affects the overall quality of the clustering solutions obtained.

TABLE II
LOUVAIN CLUSTERING RESULTS FOR LD AND SD COMBINED

Dataset	Entities Count	Cluster count	MQ metric
SD only	517	14	0.085
SD LD strength 10%	517	15	0.087
SD LD strength 20%	517	13	0.071
SD LD strength 30%	517	13	0.071
SD LD strength 40%	517	13	0.071
SD LD strength 50%	517	13	0.071
SD LD strength 60%	517	13	0.071
SD LD strength 70%	517	13	0.071
SD LD strength 80%	517	13	0.071
SD LD strength 90%	517	13	0.071
SD LD strength 100%	517	13	0.072

In a scenario where structural dependencies aren't accessible for extraction, logical dependencies can be used as a substitute, but with less strict filtering compared to when they're intended to be combined with structural dependencies.

On the other hand, in Table II can be observed that starting with the strength threshold of 20% the MQ results for clustering solutions based on logical dependencies (LD) combined with structural dependencies (SD) are better than those based solely on structural dependencies (SD).

VI. DISCUSSION

Based on the results from table II, we can observe that the combined approach of structural dependencies and logical dependencies gives a Modularity Quality (MQ) metric of 0.071, which is an improvement over the 0.085 MQ metric obtained when considering only structural dependencies.

Beyond the positive result indicated by the MQ metric, we searched for further validation by human software engineering expert opinion. After thoroughly studying and understanding the analyzed system source code and documentation, we evaluated the remodularization proposals resulting from the two clustering solutions.

The two clustering solutions compared are the clustering solution obtained only from structural dependencies, in comparison to the clustering solution obtained from using both structural and logical dependencies, filtered with a threshold of 20% for strength.

The clustering solution relying solely on structural dependencies consists of 14 clusters, while the solution using both structural and logical dependencies consists of 13 clusters, both solutions involve the same number of entities (517). The entities listed below are placed in different clusters:

- taskdefs.Available$FileDir
- taskdefs.Concat and its inner classes taskdefs.Concat$1, taskdefs.Concat$ MultiReader, taskdefs.Concat$ TextElement
- taskdefs.Javadoc$AccessType
- util.WeakishReference and its inner class util.WeakishReference$HardReference
- taskdefs.Replace and its inner classes taskdefs.Replace$ NestedString, taskdefs.Replace$ Replacefilter

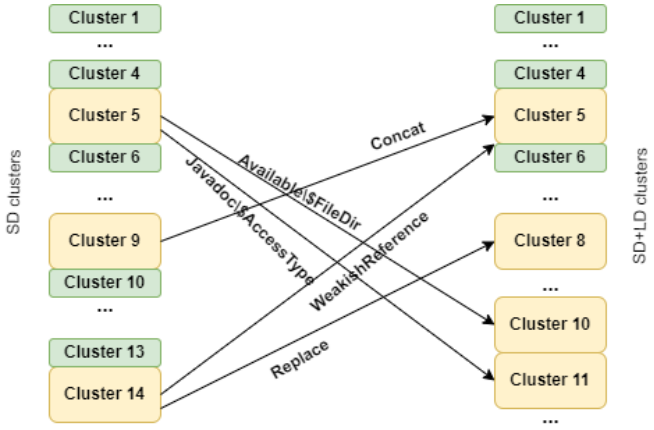

Fig. 3. Migration of entities between clusters

The migration of entities between clusters is ilustared in Figure 3. Given that the inner classes are shifted from one cluster to another in identical way as the outer class, we omitted the inner classes from the diagram.

As the cluster number itself is not significant and may vary across different script runs (labels might vary), we will refer to each individual cluster resulting from structural dependencies as *Cluster A* and to the ones resulting from both logical and structural dependencies as *Cluster B*.

A. taskdefs.Concat and its inner classes

To have a better overview into how and why entities are transferred between clusters, we depicted Concat's logical and structural connections in Figure 4. Additionally, Figure 5 illustrates connections within Cluster A, while Figure 6 does the same for Cluster B.

In Cluster A, the Concat class and its inner classes (Concat$1, Concat$MultiReader, Concat$TextElement) are placed together with conditions like Available, And, Or, IsTrue, Equals, IsReference, Contains.

On the other hand, in Cluster B are placed with classes associated with file manipulation and archive operations such as Ear, Jar, War, and Zip, as well as utility classes for file handling like FileUtils and JavaEnvUtils, and entities for zip file processing (ZipEntry, ZipFile). This placement is due to the logical dependencies that Concat class has with FileUtils and FileSet in the versioning system.

To assess whether the placement of Concat in Cluster B is better then in Cluster A, we referred to the official Ant documentation. According to the documentation: "This class contains the 'concat' task, used to concatenate a series of files into a single stream" [31]. Therefore, judging by its usage and purpose according to the documentation, positioning the Concat class along with its inner classes in Cluster B is more suitable than in Cluster A.

B. taskdefs.Available$FileDir

In Cluster A the entity 'taskdefs.Available$FileDir' is in the same cluster with entities that are related to the build

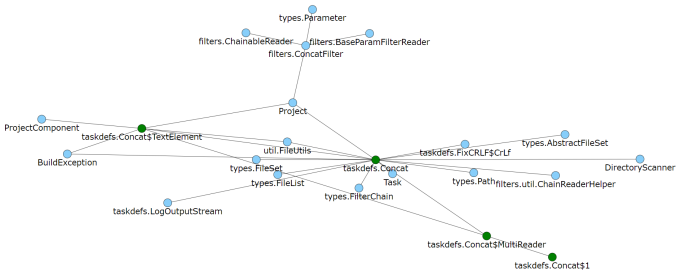

Fig. 4. Dependencies (LD and SD) of Concat class

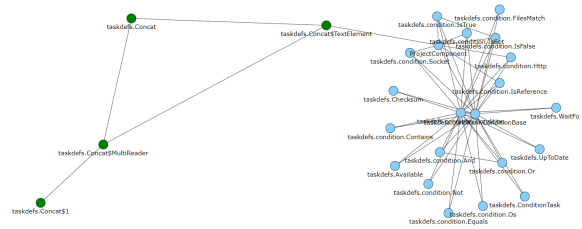

Fig. 5. Placement of Concat in ClusterA (SD); cluster size: 25

process (ProjectHelper, TaskAdapter, ComponentHelper), but not with entities that have any relation to condition checks or file existence evaluations or with its outer class.

Cluster B contains entities that are related to condition checking (And, Contains, Equals, Or, UpToDate), and also with the outer class of 'taskdefs.Available$FileDir', 'taskdefs.Available'. The movement of entities from Cluster A to Cluster B was influenced by the logical dependencies between 'taskdefs.Available' and 'taskdefs.Available$FileDir'.

The documentation description for 'taskdefs.Available' states: "Will set the given property if the requested resource is available at runtime. This task may also be used as a condition by the condition task." [31]. This reinforces its placement in Cluster B, which is centered around task definitions and conditions related to build processes.

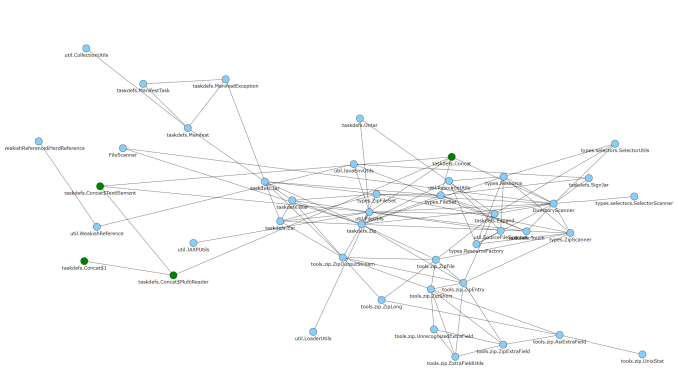

Fig. 6. Placement of Concat in ClusterB (SD and LD); cluster size: 52

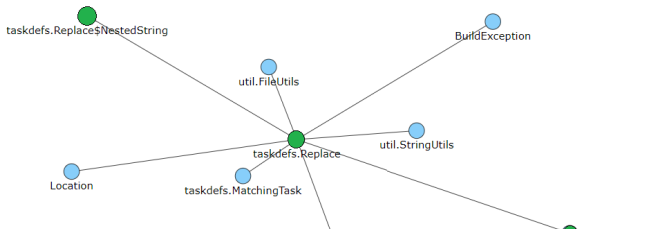

Fig. 7. Ant dependencies (LD and SD) of Replace and its inner classes

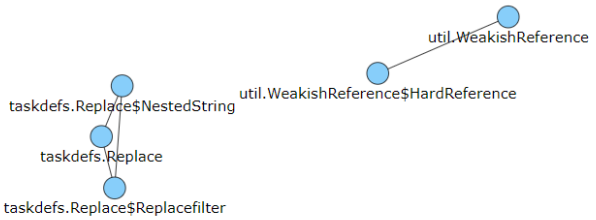

Fig. 8. Placement of Replace in ClusterA (SD); cluster size: 5

C. *taskdefs.Replace and its inner classes*

We depicted the logical and structural connections of Replace in Figure 7. Figure 8 illustrates connections within Cluster A, while Figure 9 does the same for Cluster B.

Replace and its inner classes are placed in Cluster B with entities with which they share common functionality and purpose, such as Copydir, Delete, DependSet, or MatchingTask. These entities are involved in similar tasks and operations. On the other hand, the placement in Cluster A, with WeakishReference and its inner class doesn't seem good, as these entities are completly unrelated.

The movement of entities from Cluster A to Cluster B was influenced by the strong logical dependency between 'taskdefs.Replace' and 'taskdefs.MatchingTask'.

Also, the cluster from figure 8 is the 14th cluster that

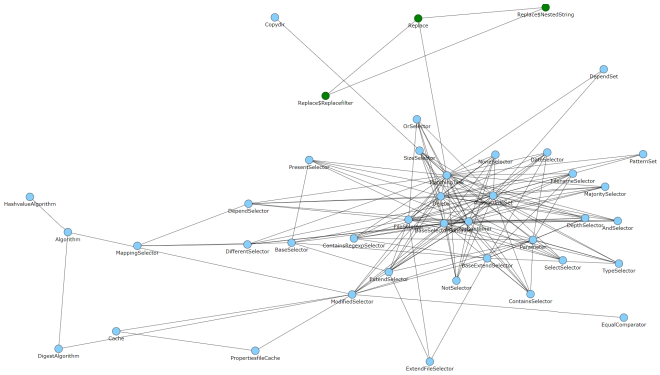

Fig. 9. Placement of Replace in ClusterB (SD and LD); cluster size: 42

is missing from the clustering solution based on logical and structural dependencies. We obtained 14 clusters from structural dependecies only and 13 from logical combined with structural dependencies. This particular cluster is essentially comprised of two unrelated entities: Replace and WeakishReference. Due to their logical dependencies, both entities are grouped within larger clusters that share similar functionalities in the SD and LD solution.

VII. Conclusion and Future work

Based the outcomes from Section V and the analysis from Section VI, it is clear that incorporating logical dependencies improves the quality of clustering solutions. The improvement is obtained from the insights provided by the logical dependencies within the system. This reinforces the idea that dependencies extracted from the versioning system contribute with new information that isn't otherwise available through code analysis. Information that can be used together or separately, along with other types of dependencies, to enhance the overall effectiveness of software clustering techniques.

For our forthcoming work, we intend to conduct further experiments across various projects to validate these findings. Additionally, we will try to create reference clustering solutions for these projects, based on their code and documentation. We will then evaluate the clustering results against the reference solution by utilizing the MOJO metric for comparison [25].

References

[1] G. A. Oliva and M. A. Gerosa, "Experience report: How do structural dependencies influence change propagation? an empirical study," in *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, 2015, pp. 250–260. [Online]. Available: https://doi.org/10.1109/ISSRE.2015.7381818

[2] N. Ajienka and A. Capiluppi, "Understanding the interplay between the logical and structural coupling of software classes," *Journal of Systems and Software*, vol. 134, pp. 120–137, 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2017.08.042

[3] G. A. Oliva and M. A. Gerosa, "On the interplay between structural and logical dependencies in open-source software," in *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering*, ser. SBES '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 144–153. [Online]. Available: https://doi.org/10.1109/SBES.2011.39

[4] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 190–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853338

[5] A. D. Stana. and I. Şora., "Identifying logical dependencies from co-changing classes," in *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, INSTICC. SciTePress, 2019, pp. 486–493.

[6] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999460

[7] A. D. Stana and I. Şora, "Analyzing information from versioning systems to detect logical dependencies in software systems," in *2019 IEEE 13th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 2019, pp. 000 015–000 020.

[8] A.-D. Stana and I. Şora, "Logical dependencies: Extraction from the versioning system and usage in key classes detection," *Computer Science and Information Systems*, vol. 20, pp. 25–25, 01 2023.

[9] F. Murtagh and P. Contreras, "Algorithms for hierarchical clustering: An overview," *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, vol. 2, pp. 86–97, 01 2012.

[10] I. Şora, "Software architecture reconstruction through clustering: Finding the right similarity factors," in *Proceedings of the 1st International Workshop in Software Evolution and Modernization - Volume 1: SEM, (ENASE 2013)*, INSTICC. SciTePress, 2013, pp. 45–54.

[11] S. Na, L. Xumin, and G. Yong, "Research on k-means clustering algorithm: An improved k-means clustering algorithm," in *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*, 2010, pp. 63–67.

[12] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: ordering points to identify the clustering structure," *SIGMOD Rec.*, vol. 28, no. 2, p. 49–60, jun 1999. [Online]. Available: https://doi.org/10.1145/304181.304187

[13] D. Xu and Y. jie Tian, "A comprehensive survey of clustering algorithms," *Annals of Data Science*, vol. 2, pp. 165 – 193, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:54134680

[14] V. Tzerpos and R. Holt, "Accd: an algorithm for comprehension-driven clustering," in *Proceedings Seventh Working Conference on Reverse Engineering*, 2000, pp. 258–267.

[15] S. Mancoridis, B. Mitchell, Y.-F. Chen, and E. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," 04 1999.

[16] B. Mitchell and S. Mancoridis, "On the evaluation of the bunch search-based software modularization algorithm," *Soft Comput.*, vol. 12, pp. 77–93, 01 2008.

[17] ——, "Demonstration proposal: Clustering module dependency graphs of software systems using the bunch tool," 04 1999.

[18] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 35–44.

[19] N. Anquetil and T. C. Lethbridge, "Recovering software architecture from the names of source files," *J. Softw. Maintenance Res. Pract.*, vol. 11, pp. 201–221, 1999. [Online]. Available: https://api.semanticscholar.org/CorpusID:6986218

[20] N. Anquetil and T. Lethbridge, "File clustering using naming conventions for legacy systems," 03 1998.

[21] L. Silva, M. Valente, and M. Maia, "Co-change clusters: Extraction and application on assessing software modularity," *Transactions on Aspect-Oriented Software Development*, 03 2015.

[22] R. Benkoczi, D. Gaur, S. Hossain, and M. A. Khan, "A design structure matrix approach for measuring co-change-modularity of software products," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 331–335. [Online]. Available: https://doi.org/10.1145/3196398.3196409

[23] A. Parashar and J. Chhabra, "Package-restructuring based on software change history," *National Academy Science Letters*, vol. 40, 04 2016.

[24] A. Prajapati, A. Parashar, and A. Rathee, "Multi-dimensional information-driven many-objective software remodularization approach," *Frontiers of Computer Science*, vol. 17, 11 2022.

[25] V. Tzerpos and R. Holt, "Mojo: a distance metric for software clusterings," in *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, 1999, pp. 187–193.

[26] 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.

[27] B. Mitchell and S. Mancoridis, "Comparing the decompositions produced by software clustering algorithms using similarity measurements," in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, 2001, pp. 744–753.

[28] A. S. Foundation, "Apache ant," https://github.com/apache/ant, 2024, gitHub repository.

[29] I. Şora and C.-B. Chirila, "Finding key classes in object-oriented software systems by techniques based on static analysis," *Information and Software Technology*, vol. 116, p. 106176, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919301727

[30] S. Harenberg, G. Bello, L. Gjeltema, S. Ranshous, J. Harlalka, R. Seay, K. Padmanabhan, and N. Samatova, "Community detection in large-scale networks: A survey and empirical evaluation," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 6, 11 2014.

[31] Apache Ant Project, “Apache Ant Concat Task Documentation,” https://ant.apache.org/manual/api/org/apache/tools/ant/taskdefs/Concat.html, accessed on February 14, 2024.