Integrating Logical Dependencies in Software Clustering: A case of 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 Sora
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. 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 where the documentation hasn't been updated to reflect changes in the system.

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

We intend to use dependencies obtained from the versioning system to enhance the results of clustering methods that previously relied solely on connections extracted from code.

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

II. Logical dependencies

During development processes, numerous software entities are changed. It has been observed that entities changing together are not only those with structural relationships but also include entities that are functionally dependent on one another.

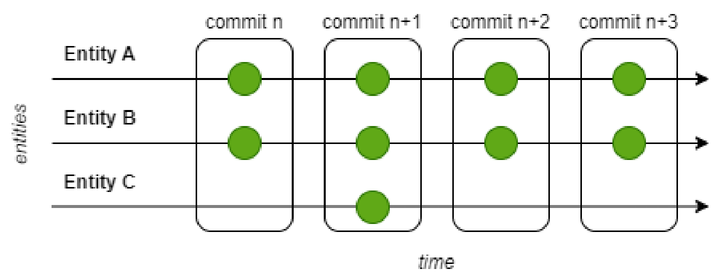

Fig. 1. Overview on updates of different entities

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. 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. Figure 1 ilustrates this scenario.

Previous research uses two known metrics in order to solve this problems: *support* and *confidence*. 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], [5].

In our previous research, we focused on identifying metrics that enhance the reliability of dependencies extracted from the versioning system [6], [7]. 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 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 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.

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) based on their similarity. 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 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].

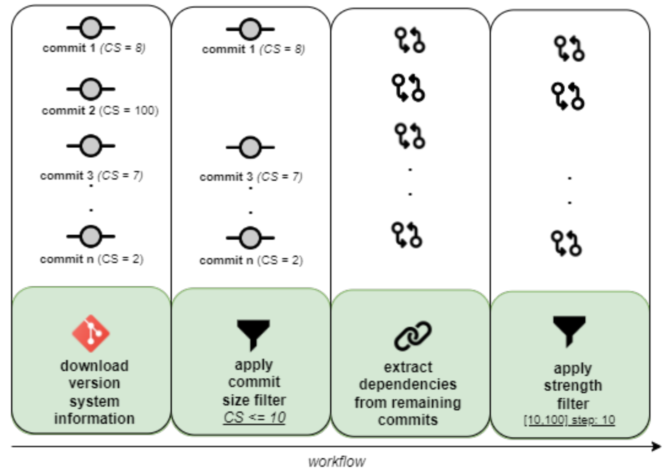

Fig. 2. Logical depenedencies extraction workflow

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 obtain the logical dependencies for further use, we will use the same 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 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 chapter 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). Finally, we exported the results in comma separated values format file (.csv). The above described workflow is illustrated in figure 2.

Following the generation of logical dependencies, we integrate them into software clustering techniques. 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 Minimum Spanning Tree (MST) and Louvain Clustering

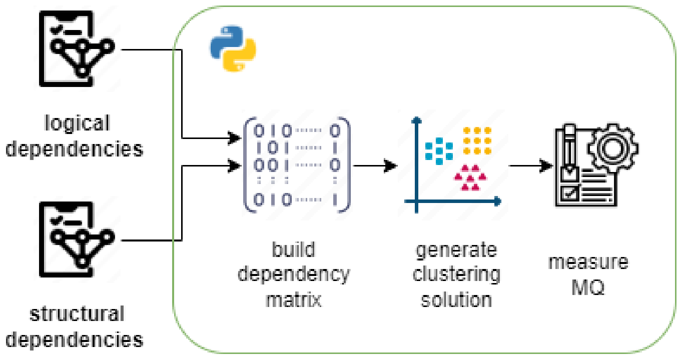

Fig. 3. Clustering solution creation process diagram

algorithms. The resulting clustering solution is then evaluated using the Modularity Quality (MQ) metric [26].

A. Clustering algorithms: MST Clustering

Minimum Spanning Tree Clustering is a hierarchical clustering method based on the construction of a minimum spanning tree from the weighted graph formed by the data connections of the system [29].

B. Clustering algorithms: Louvain Clustering

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].

In order to compare the clustering solutions and their asociated MQ metric, we generated clustering solutions in three 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 the cluster generation. The third scenario workflow is ilustrated in the figure 3.

To generate and compare the clustering solutions and their corresponding Modularity Quality (MQ) metric, we created clustering solutions for three distinct scenarios: the first scenario utilizes solely structural dependencies, the second exclusively employs logical dependencies, and the third combines both logical and structural dependencies to fill the dependency matrix, which is then utilized in generating the clusters. The process for the third scenario is depicted in Figure 3.

V. Results

The results of the scenarios mentioned in chapter IV are presented in table I and II.

!INFO: not yet done.

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

TABLE I
Comparison of Louvain and MST clustering results for LD

Dataset	**Entities Count**	**Cluster count** Louvain	MST	**MQ metric** Louvain	MST
SD only	517	14	228	0.085	0.084
LD strength 10%	517	272	353	0.047	0.031
LD strength 20%	517	355	360	0.04	0.037
LD strength 30%	517	387	404	0.036	0.033
LD strength 40%	517	405	413	0.034	0.033
LD strength 50%	517	414	422	0.03	0.029
LD strength 60%	517	431	439	0.029	0.027
LD strength 70%	517	443	444	0.027	0.026
LD strength 80%	517	454	460	0.024	0.022
LD strength 90%	517	462	463	0.021	0.021
LD strength 100%	517	472	480	0.019	0.016

TABLE II
Comparison of Louvain and MST clustering results for LD combined with SD

Dataset	**Entities Count**	**Cluster count** Louvain	MST	**MQ metric** Louvain	MST
SD only	517	14	228	0.085	0.084
SD LD strength 10%	517	15	74	0.087	0.054
SD LD strength 20%	517	13	8	0.071	0.059
SD LD strength 30%	517	13	14	0.071	0.083
SD LD strength 40%	517	13	8	0.071	0.109
SD LD strength 50%	517	13	8	0.071	0.109
SD LD strength 60%	517	13	8	0.071	0.109
SD LD strength 70%	517	13	10	0.071	0.155
SD LD strength 80%	517	13	6	0.071	0.177
SD LD strength 90%	517	13	2	0.071	0.129
SD LD strength 100%	517	13	8	0.072	0.166

0.071, which is an improvement over the 0.085 MQ metric obtained when considering only structural dependencies.

We manualy compared two clustering solutions, the clustering solutions 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, in order to asses if the second solution is better or not.

The entities listed below are placed in different clusters: taskdefs.Available$FileDir, taskdefs.Concat, taskdefs.Concat$1, taskdefs.Concat$MultiReader, taskdefs.Concat$TextElement, taskdefs.Javadoc$AccessType, util.WeakishReference, util.WeakishReference$HardReference.

We will refer to the clusters resulted from the structural dependencies as *Cluster A* and to those from logical and structural dependencies as *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, Cluster B places them 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).

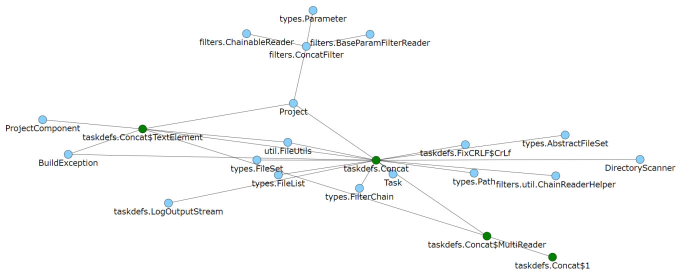

Fig. 4. Ant dependencies (LD and SD) of Concat class

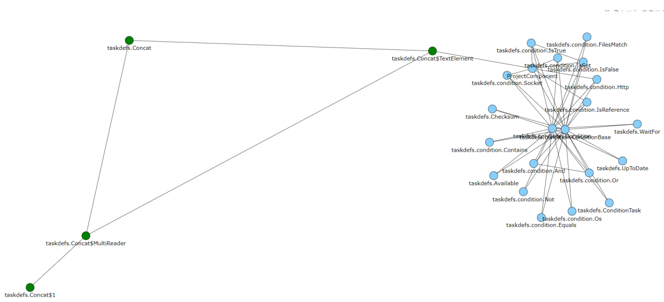

Fig. 5. Cluster A

The placement of the Concat class together with its inner classes in a different cluster can be motivated based on it's usage and purpose; according to the documentation : "This class contains the 'concat' task, used to concatenate a series of files into a single stream." [31]. So, from our point of view, it is better placed in Cluster B, then in Cluster A. Figure 4 represents the connections that Concat has in the versioning and file system. Figure 5 represents Cluster A connections and figure 6 represents Cluster B connections .

VII. Conclusion and Future work

Based the outcomes detailed in Chapter V and the analysis in Chapter VI, it is clear that incorporating logical dependencies improves the quality of clustering solutions. For our forthcoming work, we intend to conduct further experiments across various projects to validate these findings. Additionally, we will try to create a reference clustering solution 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].

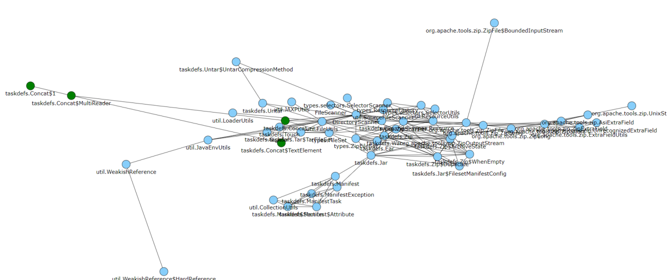

Fig. 6. Cluster B

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] 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

[6] 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.

[7] 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.

[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] M. Yu, A. Hillebrand, P. Tewarie, J. Meier, B. Dijk, P. Mieghem, and C. Stam, "Hierarchical clustering in minimum spanning trees," *Chaos (Woodbury, N.Y.)*, vol. 25, p. 023107, 02 2015.

[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.