IOSUD - Universitatea Politehnica Timişoa
Şcoala Doctorală de Studii Inginereşti

LEVERAGING VERSIONING SYSTEM INFORMATION TO IMPROVE METHODS AND TOOLS FOR LEGACY SOFTWARE ANALYSIS

Teză de doctorat – Rezumat

pentru obținerea titlului științific de doctor la
Universitatea Politehnica Timișoara
în domeniul de doctorat Calculatoare și Tehnologia Informației

autor ing. Adelina Diana STANA
conducător științific Prof. emerit dr. ing. Vladimir I. CREŢU

luna Septembrie anul 2025

This Ph.D. thesis addresses the analysis and improvement of methods for extracting, filtering, and integrating versioning system information in software engineering tasks. This thesis uses versioning information in the form of co-change patterns extracted from the versioning history of a software system, also called logical dependencies [1]. These dependencies can reveal relationships that are not visible in the system's static structure and can complement other types of dependencies, such as structural [2] and lexical dependencies [3]. The contributions of this thesis include the introduction of a new filtering metric for logical dependencies called connection strength, the development of a dedicated tool for logical dependency extraction and filtering, and the integration of logical dependencies into two software engineering tasks: key class detection [4,5] and software clustering for architectural reconstruction [6,7].

The first chapter presents the need for tools to analyze legacy software and the importance of using various types of dependencies as inputs in these tools to extend the knowledge base they rely on to improve their output. The maintenance of legacy systems is time-consuming because the codebase is large and documentation is often incomplete or outdated. Changes made in the wrong place can produce unwanted effects across the system, so it is important to understand how components are connected before making update [8]. To understand these connections, software dependencies are used. This chapter introduces three complementary dependency views: structural dependencies (explicit links in the code, such as calls and inheritance), lexical dependencies (similarities in names and comments), and logical dependencies (co-change patterns from version history).

The chapter also highlights the importance of filtering co-change patterns: raw co-change data can contain noise, such as bulk refactorings performed by developers, so filtering of logical dependencies is necessary to ensure their reliability when used alone or together with other dependency types in software engineering tasks [9,10].

Chapter 2 is divided into three parts. The first part presents the background on different types of software dependencies. The second part introduces version control concepts, which are necessary to better understand logical dependencies extraction. It also reviews existing approaches proposed by different authors for filtering logical dependencies (co-changes), including commit-size thresholds [9,10, 14], support/confidence metrics [11,12], and age-based thresholds [13]. The third part presents dependency-based applications relevant to the thesis. It covers two main applications, where this thesis focuses on the integration and validation of filtered logical dependencies from the versioning system: key-class detection, which identifies central classes in a system, and architecture recovery by clustering, which groups software entities into modules.

Chapter 3 presents the filters explored and the tool developed to extract and filter logical dependencies. The software dependency extraction tool analyzes source code to extract structural dependencies and parses version-control history (e.g., Git history) to extract co-change pairs. The extracted co-change pairs are then processed using different types of filters, which can be applied individually or in combination. The tool extracts both logical and structural dependencies, as this thesis also investigates the overlap and relationships between these two types of dependencies [.

The chapter then introduces different filtering methods for logical (co-change) dependencies. All filtering experiments are conducted on a set of 27 open-source projects with different sizes, complexities, and programming languages (Java and C#):

- *Commit size filter:* excludes co-change pairs from large commits, which often result from merges, restructuring, or formatting rather than functional changes. Different studies have suggested various thresholds for filtering commit sizes: some authors recommend excluding commits with more than 8 to 10 files, while others allow larger thresholds (e.g., 30 or even 100 files)[9,10,14]. In this thesis, multiple thresholds are explored for commit size ($cs \leq 5$; $cs \leq 10$; $cs \leq 20$; $cs < \infty$) to analyze how much of the original dependency base is lost with each threshold.
- *Support filter:* requires a minimum number of co-change occurrences for a pair to pass the filter. The support metric counts the number of commits including both entities. Previous studies used thresholds from 1 to 8 [11,12]. In this thesis, thresholds from 1 to 4 are explored to analyze their impact. Thresholds above 4 may reduce coincidental co-changes but can remove too many pairs in smaller projects.
- *Comment-only change filter:* excludes co-changing pairs from commits where only comments were changed, which are non-functional edits. Comment changes do not affect system behavior and are easily detected in diff files [9]. Two scenarios were evaluated: including comment-only changes and excluding them.
- *Connection strength filter:* a new metric introduced in this thesis, based on the confidence metric proposed by Zimmermann [11], which measures how two entities co-change relative to the total number of changes of one entity. The connection strength addresses confidence's limitation by scaling it with a system factor, giving higher weight to pairs that co-change more frequently relative to the system average. In this thesis, the filter is applied using 10 thresholds, starting at 10% and increasing in steps of 10% up to 100%.

Chapter experiments conclude that combining a 10-file commit-size threshold with a connection-strength threshold provides more reliable logical dependencies. The experiments show that over 90% of commits involve 10 or fewer files, so most commits remain available for extraction. Unlike commit size, the connection-strength threshold is not fixed in this chapter; instead, the following experiments on logical dependencies integration in software engineering tasks explore different threshold intervals to evaluate which values perform best across different systems.

Chapter 4 presents how to merge the structural and logical dependencies extracted by the tool into a unified model. It also presents experiments and discussion about the overlap between the two dependency types, showing that even if some overlap exists, they also provide different information about the systems. The chapter then proposes weighting schemes for both types of dependencies. For structural dependencies, different weights are assigned because not all have the same impact on the system architecture and behavior. For logical dependencies, the weight of each entity pair is the number of commits in which both entities were updated together. The chapter then describes the creation of a combined dependency graph with both dependency types, which will be further used in the experiments.

Chapter 5 evaluates the usage of logical dependencies for identifying key classes (the central classes managing core functionality). A previously developed tool [7] for key class detection that relied only on structural dependencies was modified to also use logical dependencies. The experiments were run on three open-source systems (Apache Ant, Tomcat Catalina, Hibernate) comparing three scenarios: using only structural dependencies, only logical dependencies (filtered with various connection-strength metric thresholds), or both combined. Results show that combining logical and structural dependencies improves detection compared to structural dependencies alone. The best performance was achieved when using structural dependencies combined with logical dependencies filtered with connection-strength thresholds between 40% and 70%. Using only logical dependencies still gave good results, with only slightly lower results than the combined approach and comparable to structural baselines. The conclusion is that filtered logical dependencies complement structural dependencies and help highlight important classes that might be missed when using structural dependencies alone.

Chapter 6 evaluates the usage of logical dependencies in software clustering for architectural reconstruction. To assess their impact, three graph-clustering algorithms (Louvain, Leiden, DBSCAN) are used, and quality is measured by Modularization Quality (MQ) [15] and MoJoFM (Move and Join eFfectiveness Measure) metrics [16, 17]. The same three scenarios as in key class detection are tested: using only structural dependencies, only logical dependencies (filtered with various connection-strength thresholds), or both combined. Experiments were performed on four open-source Java projects: Apache Ant, Apache Tomcat, Hibernate ORM, and Gson. The results show that combining structural and logical dependencies achieves higher MQ and MoJoFM scores than using structural dependencies alone in almost all cases. At strength thresholds of 10% to 40%, the combination consistently outperforms using only structural dependencies. Using only logical dependencies can achieve the best MQ and MoJoFM results among the three scenarios, but only at very high strength thresholds that cover a small portion of the system. At lower strength thresholds (10-40%), logical dependencies provide a good balance

between system coverage and clustering quality, including enough dependencies to improve results without adding noise. Overall, the experiments show that filtered logical dependencies help architecture recovery both when combined with structural dependencies (full system coverage) and alone (partial system coverage).

In Chapter 7, the contributions, conclusions, and future work are presented. The main contributions of this thesis are:

- proposed methods for filtering logical dependencies extracted from version control systems to improve their reliability and usefulness, including the introduction of a new metric for filtering logical dependencies, called connection strength; [18], [19]
- developed a tool for extracting and filtering logical dependencies; [18], [19], [20]
- integrated logical dependencies into key class detection and analyzed the impact of different filtering strategies when using logical dependencies both independently and in combination with structural dependencies; [20]
- integrated logical dependencies into software clustering; this analysis involved using three distinct clustering algorithms and two evaluation metrics; [21], [22]

The experiments on detecting key classes and reconstructing software architecture through clustering show that logical dependencies improve the results of both tasks when combined with structural dependencies and produce good results on their own. However, when strict filtering thresholds are applied, logical dependencies may cover only a small subset of system entities, which can be an issue for engineering tasks that require a complete system overview, such as clustering. When a full system overview is not needed, higher strength thresholds can be applied to obtain only the strongest dependencies, as in key-class detection. Logical dependencies also have the advantage of being language-independent, so in cases where structural dependencies are difficult to extract, logical dependencies can still provide a meaningful overview of the system.

Bibliography

[1] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. Proceedings of the International Conference on Software Maintenance, pages 190–, 1998.

[2] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. IEEE Transactions on Software Engineering, 35:864–878, 2009.

[3] Amarjeet Prajapati and Jitender Chhabra. Improving modular structure of software system using structural and lexical dependency. Information and Software Technology, 82, 10 2016.

[4] Andy Zaidman and Serge Demeyer. Automatic identification of key classes in a software system using webmining techniques. Journal of Software Maintenance and Evolution: Research and Practice, 20(6):387–417, 2008.

[5] Ioana Șora. A PageRank based recommender system for identifying key classes in software systems. 2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelligence and Informatics (SACI), pages 495–500, May 2015.

[6] Ioana Șora, Gabriel Glodean, and Mihai Gligor. Software architecture reconstruction: An approach based on combining graph clustering and partitioning. Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on, pages 259–264, May 2010.

[7] Ioana Șora. Software architecture reconstruction through clustering: Finding the right similarity factors. Proceedings of the 1st International Workshop in Software Evolution and Modernization - Volume 1: SEM, (ENASE 2013), pages 45–54, 2013.

[8] Wesley KG Assunção, Luciano Marchezan, Lawrence Arkoh, Alexander Egyed, and Rudolf Ramler. Contemporary software modernization: Strategies, driving forces, and research opportunities. ACM Transactions on Software Engineering and Methodology, 34(5):1–35, 2025.

[9] Nemitari Ajienka and Andrea Capiluppi. Understanding the interplay between the logical and structural coupling of software classes. Journal of Systems and Software, 134:120–137, 2017.

[10] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. Proceedings of the 26th International Conference on Software Engineering, pages 563–572, 2004.

[11] Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings., pages 73–83, 2003.

[12] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. IEEE Transactions on Software Engineering, 30(9):574–586, 2004.

[13] Leon Moonen, Thomas Rolfsnes, Dave Binkley, and Stefano Di Alesio. What are the effects of history length and age on mining software change impact? Empirical Software Engineering, 23, 08 2018.

[14] Leon Moonen, Stefano Di Alesio, David Binkley, and Thomas Rolfsnes. Practical guidelines for change recommendation using association rule mining. 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 732–743, 2016.

[15] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. Proceedings of the IEEE International Conference on Software Maintenance (ICSM'99), pages 50–59, 1999

[16] V. Tzerpos and R. C. Holt. Mojo: a distance metric for software clusterings. Proceedings of the Sixth Working Conference on Reverse Engineering, pages 187–193, 1999.

[17] Zhihua Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. Proceedings of the 12th IEEE International Workshop on Program Comprehension, pages 194–203, 2004.

[18] Stana, Adelina-Diana, and Șora, Ioana. (2019). Analyzing Information from Versioning Systems to Detect Logical Dependencies in Software Systems. In Proceedings of the 2019 International Symposium on Applied Computational Intelligence and Informatics (SACI), pp. 15–20. DOI: 10.1109/SACI46893.2019.9111582.

[19] Stana, Adelina-Diana, and Șora, Ioana. (2019). Identifying Logical Dependencies from Co-Changing Classes. In Proceedings of the 2019 International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), pp. 486–493. DOI: 10.5220/0007758104860493.

[20] Stana, Adelina-Diana, and Șora, Ioana. (2023). Logical Dependencies: Extraction from the Versioning System and Usage in Key Classes Detection. International Conference on System Theory, Control and Computing (ComSIS), pp. 25–25. DOI: 10.2298/CSIS220518025S.

[21] Stana, Adelina-Diana, and Șora, Ioana. (2024). Integrating Logical Dependencies in Software Clustering: A Case Study on Apache Ant. In Proceedings of the 2024 International Conference on Control Systems and Computer Science (ICSTCC), pp. 113–118. DOI: 10.1109/ICSTCC62912.2024.10744671.

[22] Stana, Adelina-Diana, and Șora, Ioana. (2025). Refining Software Clustering: The Impact of Code Co-Changes on Architectural Reconstruction. In IEEE Access, vol. 13, pp. 132126–132145. DOI: 10.1109/ACCESS.2025.3592777.