IOSUD - Universitatea Politehnica Timişoa
Şcoala Doctorală de Studii Inginereşti

UTILIZAREA INFORMAȚIILOR DIN SISTEMELE DE VERSIONARE PENTRU ÎMBUNĂTĂȚIREA METODELOR ȘI UNELTELOR DE ANALIZĂ A SOFTWARE-ULUI

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

Această teză de doctorat analizează și propune îmbunătățiri pentru metodele de extragere, filtrare și integrare a informațiilor din sistemele de versionare în activități de inginerie software. Teza utilizează informații de versionare sub forma corelațiilor de modificări extrase din istoricul unui sistem software, denumite în literatură dependențe logice [1]. Aceste dependențe pot pune în evidență relații care nu sunt vizibile în structura statică a sistemului și pot completa alte tipuri de dependențe, precum cele structurale [2] și lexicale [3]. Contribuțiile tezei includ introducerea unei noi metrici de filtrare pentru dependențele logice, denumită connection strength, dezvoltarea unui software dedicat pentru extragerea și filtrarea dependențelor logice și integrarea dependențelor logice în două activități de inginerie software: identificarea claselor cheie [4,5] și gruparea entităților in module (clustering) pentru reconstrucția arhitecturii [6,7].

Primul capitol prezintă necesitatea unor instrumente pentru analiza software-ului și importanța includerii mai multor tipuri de dependențe în aceste instrumente, pentru a extinde cunoștințele disponibile despre sistem și a îmbunătăți rezultatele acestor instrumente. Mentenanța sistemelor este costisitoare în timp deoarece baza de cod este mare, iar documentația este adesea incompletă sau învechită. Modificările efectuate în locuri nepotrivite pot produce efecte nedorite în întregul sistem, de aceea este important să se înțeleagă modul în care sunt conectate componentele înainte de a face modificări [8]. Pentru a înțelege aceste legături sunt folosite dependențele software. Acest capitol prezintă trei tipuri de dependențe: dependențe structurale (legături explicite în cod, cum ar fi apeluri sau moștenire), dependențe lexicale (similarități în nume și comentarii) și dependențe logice (tipare de co-modificare ale entităților software extrase din istoricul de versiuni).

Capitolul subliniază și importanța filtrării tiparelor de co-modificare: datele nefiltrate pot conține zgomot, de exemplu refactorizări la scară mare efectuate de dezvoltatori, astfel încât filtrarea dependențelor logice este necesară pentru a le asigura fiabilitatea atunci când sunt folosite singure sau împreună cu alte tipuri de dependențe în activități de inginerie software [9,10].

Capitolul 2 este împărțit în trei părți. Prima parte prezintă baza teoretică privind diferitele tipuri de dependențe software. A doua parte introduce conceptul de control al versiunilor, necesar pentru a înțelege extragerea dependențelor logice, și trece în revistă abordările existente propuse

de diverși autori pentru filtrarea acestor dependențe, precum: filtrarea pe baza dimensiunii commit-urilor (numărul de fișiere modificate in commit) [9,10,14], filtrarea pe baza metricilor de support/confidence [11,12] și filtrarea în funcție de vechimea commit-urilor [13]. A treia parte descrie aplicațiile bazate pe dependențe relevante pentru teză: identificarea claselor cheie, care extrage clasele centrale dintr-un sistem, și recuperarea arhitecturii prin clustering, care grupează entitățile software în module.

Capitolul 3 prezintă filtrele investigate și software-ul dezvoltat pentru extragerea și filtrarea dependențelor logice. Acest software analizează codul sursă pentru a obține dependențele structurale și parcurge istoricul Git pentru a extrage perechi de entități care se co-modifică. Perechile obținute sunt apoi prelucrate prin diverse filtre, aplicabile individual sau în combinație. Software-ul extrage atât dependențe logice, cât și structurale, deoarece teza investighează și suprapunerile și relațiile dintre aceste două tipuri de dependențe.

Următoarea parte a capitolului descrie metodele de filtrare pentru dependențele logice. Toate experimentele de filtrare au fost realizate pe un set de 27 de proiecte open-source, cu dimensiuni, complexități și limbaje diferite (Java și C#):

- *Filtrul pe dimensiunea commit-ului*: exclude perechile de co-modificare provenite din commit-uri mari, care adesea reprezintă restructurări sau reformatări, nu modificări funcționale. Diverse studii recomandă valori diferite pentru acest filtru: unii autori propun excluderea commit-urilor cu mai mult de 8-10 fișiere, alții folosesc praguri mai mari (de 30 sau chiar 100 de fișiere) [9,10,14]. În teza de față au fost testate mai multe valori de prag pentru dimensiunea commit-ului (cs (commit size) ≤ 5; $cs \leq 10$; $cs \leq 20$; $cs < \infty$) pentru a măsura ce parte din baza inițială de dependențe se pierde la fiecare filtrare.
- *Filtrul de suport*: cere un număr minim de apariții împreună pentru ca o pereche să treacă filtrul. Metrica de suport numără commit-urile care conțin ambele entități. Studii anterioare au folosit valori de prag între 1 și 8 [11,12]; în această teză s-au folosit praguri între 1 și 4 pentru a evalua impactul. Valorile peste 4 pentru această metrică pot reduce co-modificările întâmplătoare, dar pot elimina multe perechi în proiectele mici.
- *Filtrul de comentarii:* exclude perechile care apar în commit-uri în care s-au modificat numai comentarii (editări non-funcționale). Modificările comentariilor nu afectează comportamentul sistemului și sunt ușor detectabile [9]. Au fost analizate două scenarii: includerea și excluderea perechilor de entități rezultate din modificări care afectează exclusiv comentariile.
- *Filtrul connection strength:* o metrică nouă introdusă în această teză, derivată din ideea metricii de confidence propuse de Zimmermann [11], care măsoară cât de des două entități se modifică împreună în raport cu totalul modificărilor uneia dintre ele. Connection strength atenuează limitarea metricii de confidence prin scalare cu un factor la nivel de sistem, acordând o pondere mai mare perechilor care co-apar mai frecvent în raport cu media sistemului. În lucrare filtrul a fost aplicat pe 10 praguri, începând de la 10% și crescând în trepte de 10% până la 100%.

Experimentele din capitol arată că o combinație între valoarea setată pentru filtrul de dimensiune a commit-ului, de 10 fișiere, și filtrul de connection strength produce dependențe logice mai fiabile. Rezultatele evidențiază că peste 90% dintre commit-uri implică 10 sau mai puține fișiere, astfel încât majoritatea commit-urilor rămân disponibile pentru extragere. Spre deosebire de filtrul de dimensiune a commit-ului, valoarea setată pentru filtrul de connection strength nu este fixată în acest capitol; experimentele ulterioare privind integrarea dependențelor logice în activități de inginerie software explorează diferite intervale de valori pentru acest filtru pentru a identifica acele valori care oferă cele mai bune rezultate pentru diverse sisteme.

Capitolul 4 explică cum se combină dependențele structurale și logice într-un model unificat. Sunt prezentate experimente și discuții despre suprapunerea celor două tipuri de dependențe, arătând că, deși există anumite suprapuneri, fiecare tip oferă informații complementare despre sistem. Capitolul propune factori de ponderare pentru ambele tipuri de dependențe: pentru dependențele structurale se alocă ponderi diferite, deoarece nu toate relațiile structurale au același impact asupra arhitecturii și comportamentului sistemului; pentru dependențele logice, ponderea fiecărei perechi este determinată de numărul de commit-uri în care ambele entități au fost modificate simultan. Capitolul descrie apoi construirea unui graf de dependențe combinat, care va fi folosit în experimentele următoare.

Capitolul 5 evaluează utilizarea dependențelor logice pentru identificarea claselor cheie (clasele centrale ale sistemului). A fost adaptat un sistem software anterior [7] pentru detectarea claselor cheie, care inițial folosea doar dependențe structurale, astfel încât să includă și dependențe logice. Experimentele au fost rulate pe trei sisteme open-source (Apache Ant, Tomcat Catalina, Hibernate), comparând trei scenarii: numai dependențe structurale, numai dependențe logice (filtrate cu diverse valori pentru filtrul de connection strength) și combinația celor două. Rezultatele arată că integrarea dependențelor logice cu cele structurale conduce la o detectare mai eficientă a claselor cheie comparativ cu utilizarea exclusivă a dependențelor structurale. Performanța maximă a fost obținută când dependențele structurale au fost folosite împreună cu dependențele logice filtrate cu valori de connection strength între 40% și 70%. Folosirea exclusivă a dependențelor logice a oferit, de asemenea, rezultate bune, doar ușor inferioare abordării combinate și comparabile cu cele structurale. Concluzia este că dependențele logice filtrate completează dependențele structurale și ajută la identificarea unor clase importante care pot fi omise când se folosește doar analiza structurală.

Capitolul 6 analizează impactul dependențelor logice în gruparea entităților software-ului in module (clustering) pentru reconstrucția arhitecturii. Pentru evaluare au fost folosiți trei algoritmi de clustering pe grafuri (Louvain, Leiden, DBSCAN), iar calitatea soluțiilor a fost măsurată prin Modularization Quality (MQ) [15] și MoJoFM [16,17]. Au fost testate aceleași trei scenarii ca în capitolul precedent: numai dependențe structurale, numai dependențe logice (filtrate cu diverse praguri pentru filtrul de connection strength) și combinația celor două. Experimentele au fost efectuate pe patru proiecte Java open-source: Apache Ant, Apache Tomcat, Hibernate ORM și Gson. Rezultatele arată că, în majoritatea cazurilor, combinația dependențelor structurale și logice conduce la scoruri MQ și MoJoFM mai bune decât folosirea exclusivă a dependențelor structurale. La valori de connection strength între 10% și 40%, combinația depășește constant

utilizarea doar a dependențelor structurale. Utilizarea exclusivă a dependențelor logice poate obține cele mai bune scoruri MQ și MoJoFM dintre cele trei scenarii, dar numai la valori foarte ridicate pentru filtrul de connection strength, care acoperă o porțiune mică din sistem. La valori mai scăzute (10-40%), dependențele logice oferă un bun compromis între acoperirea sistemului și calitatea clusterizării, oferind suficiente dependențe pentru a îmbunătăți rezultatele fără a introduce prea mult zgomot. În ansamblu, experimentele arată că dependențele logice filtrate ajută recuperarea arhitecturii atât când sunt combinate cu dependențele structurale (acoperire completă), cât și când sunt folosite singure (acoperire parțială).

În Capitolul 7 sunt prezentate contribuțiile, concluziile și direcțiile viitoare. Principalele contribuții ale tezei sunt:

- propunerea unor metode pentru filtrarea dependențelor logice, pentru a le crește utilitatea, precum și introducerea noii metrici connection strength; [18],[19]
- dezvoltarea unui software pentru extragerea și filtrarea dependențelor logice; [18],[19],[20]
- integrarea dependențelor logice în detectarea claselor cheie și analiza impactului strategiilor de filtrare când dependențele logice sunt utilizate independent sau împreună cu dependențele structurale; [20]
- integrarea dependențelor logice în software clustering (gruparea entităților software în module); analiza a implicat folosirea a trei algoritmi de clustering și două metrici de evaluare. [21],[22]

Experimentele privind identificarea claselor cheie și reconstrucția arhitecturii prin clustering arată că dependențele logice îmbunătățesc rezultatele ambelor activități când sunt combinate cu dependențele structurale și dau rezultate bune și cand sunt folosite independent. Totuși, aplicarea unor valori stricte de filtrare poate face ca dependențele logice să acopere doar un subset redus din entitățile sistemului, ceea ce poate fi problematic pentru activități care necesită o acoperire completă a sistemului, cum este clustering-ul. Atunci când nu este necesară o acoperire completă, se pot folosi valori mai stricte de filtrare pentru a obține doar cele mai relevante dependențe. Un alt avantaj al dependențelor logice este independența față de limbaj, astfel încât atunci când extragerea dependențelor structurale este dificilă, dependențele logice pot oferi în continuare o reprezentare relevantă a sistemului.

Bibliografie

[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 Şora. 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 Şora, 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 Şora. 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 Şora, 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 Şora, 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 Şora, 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.