CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1. Research scope and motivation

The domain of the proposed thesis is Automated Software Engineering. The thesis will develop methods for the analysis of legacy software systems, focusing on using historical information describing the evolution of the systems extracted from the versioning systems. The methods for analysis will integrate techniques based on computational algorithms as well as data-mining. As proof-of-concept, tool prototypes will implement the proposed methods and validate them by extensive experimentation on several cases of real-life systems.

1.2. Objectives of the thesis

The objective of this doctoral thesis is to investigate and improve methods for filtering logical dependencies extracted from version control systems and to validate the effectiveness of these methods. The filtered logical dependencies are used to enhance key class detection in software systems and to improve software architecture reconstruction, two areas that mostly depend on structural dependencies.

This research proposes techniques for extracting, filtering, and integrating logical dependencies into dependency models. Various metrics will be used to evaluate the impact of incorporating logical dependencies and assess their impact on key class detection and software architecture reconstruction. The goal is to demonstrate that logical dependencies, when appropriately filtered, provide valuable information that can complement or, in some cases, replace structural dependencies. To achieve this, the following objectives have been established:

[O1]: *Study and analyze the current state of research on logical dependencies to improve their extraction and filtering methods.*

This objective involved the following tasks:

- **[T1.1]**: Review existing methods for logical dependency extraction from version control systems.
- **[T1.2]**: Identify and evaluate factors that influence the validity of logical dependencies, such as commit size thresholds, minimum co-change occurrences, and comment changes, to develop effective filtering methods.
- **[T1.3]**: Based on the observations from T1.2, propose filtering methods for logical dependencies.
- **[T1.4]**: Investigate the interplay between logical and structural dependencies to understand their overlap and differences better.

The outcomes of this objective [O1] have been published in [A1] and [A2].

[O2]: *Develop a tool for extracting and filtering logical dependencies.*

This objective involved the following tasks:

- **[T2.1]**: Designing a tool for extracting logical dependencies based on co-changing entities with different configurable filters.
- **[T2.2]**: Implementing and testing the filtering mechanisms by outputting the results in a commonly used format.

This objective [O2] has been published and described in more detail in [A1] and [A2], the developed tool being further used in [A3], [A4], and [A5].

[O3]: *Integrate logical dependencies in key class detection.* This objective involved the following tasks:

- **[T3.1]** Extract logical dependencies from version control systems and apply the proposed filter (connection strength).
- **[T3.2]** Modify the baseline key class detection tool to incorporate logical dependencies.
- **[T3.3]** Evaluate the impact of combining logical and structural dependencies on key class detection performance.
- **[T3.4]** Evaluate the impact of using only logical dependencies for key class detection.
- **[T3.5]** Investigate the effect of filtering strategies (connection strength with different thresholds) on the key class detection results.

The outcomes of this activity [O3] have been published in [A3].

[O4]: *Refine software clustering using logical dependencies.* This objective involved the following tasks:

- **[T4.1]** Review and select suitable clustering algorithms for the experiments.
- **[T4.2]** Use only logical dependencies as input for software clustering.
- **[T4.3]** Combine logical and structural dependencies as input for clustering algorithms.
- **[T4.4]** Investigate the impact of different connection strength filter thresholds on clustering results by using clustering evaluation metrics: MQ and MoJoFM.
- **[T4.5]** Analyze the results: logical dependencies alone vs. combined dependencies with varying filter thresholds.

The outcomes of this objective [O4] have been published in [A4] and [A5].

1.3. Structure of the thesis

1.4. Main Contributions

The principal contributions of this thesis are:

- proposing methods for filtering logical dependencies extracted from version control systems to improve their reliability and usefulness; this includes introducing

a new metric for filtering logical dependencies, called connection strength;
- developing a tool for extracting and filtering logical dependencies;
- integrating logical dependencies into key class detection methodologies and analyzing the impact of different filtering strategies when using logical dependencies both independently and in combination with structural dependencies;
- integrating logical dependencies into software clustering and analyzing the impact of different filtering strategies when using logical dependencies both independently and in combination with structural dependencies; this analysis involves using three distinct clustering algorithms and two evaluation metrics to study the results;
- providing evidence that logical dependencies when appropriately filtered, can improve key class detection and software architecture reconstruction.

2. SOFTWARE DEPENDENCIES: CONCEPTS, APPLICATIONS, AND CURRENT RESEARCH

2.1. Software dependencies overview

2.1.1. Structural dependencies

A dependency is created by two elements that are in a relationship and indicates that an element of the relationship, in some manner, depends on the other element of the relationship [2], [3].

Structural dependencies can be found by analyzing the source code [4], [5]. There are several types of relationships between these source code entities and all those create *structural dependencies*:

Data Item Dependencies. Data items can be variables, records or structures. A dependency is created between two data items when the value held in the first data item is used or affects the value from the second.

```
Example:

int a = 10;
int b = a + 5;  // b depends on a
a = 20;         // changing a doesn't impact b
```

Data Type Dependencies. Data items are declared to be of a specific data type. Besides the built-in data types that every programming language has, developers can also create new types that they can use. Each time the data type definition is changed it will affect all the data items that are declared to be of that type.

```
Example:

class Rectangle { int width, height; }
//Change posibility: class Rectangle { int width, height, color; }

Rectangle r = new Rectangle();
r.width = 5;
r.height = 10;
// Existing code will fail to handle color in case of change
```

Subprogram Dependencies. A subprogram is a sequence of instructions that performs a certain task. Depending on the programming language a subprogram

may also be called a routine, a method, a function or a procedure. When a subprogram is changed, the developer must check the effects of that change in all the places that are calling that subprogram. Subprograms may also have dependencies with the data items that they receive as input or the data items that they are computing.

Example:

```
// First implementation:
class Calculator {
    double calculateArea(double side) {
        return side * side;
    }
}

// Modified implementation:
class Calculator {
    double calculateArea(double radius) {
        return Math.PI * radius * radius;
    }
}

Calculator calc = new Calculator();
// Code expecting square area now gets circle area.
double area = calc.calculateArea(5);
```

2.1.2. Lexical dependencies

Lexical dependencies, similar to structural dependencies, are extracted from the source code. The difference lies in the fact that lexical dependencies focus on finding pairs of entities that are similar (and thus connected) from a linguistic point of view. This means they are based on the textual content and naming conventions used in the code rather than explicit structural relationships. The lexical information about an entity (such as a class or interface) can be extracted from class names, method names, parameter names, code comments, source code statements, and others [6], [7].

For example, a class named `StructuralDependenciesEvaluator` and a class named `LexicalDependenciesEvaluation` can be related even if they do not directly share dependencies or their connection is not visible from a structural point of view.

To extract lexical dependencies, the code is tokenized, breaking it down into lexical tokens (e.g., words or symbols). A document containing these tokens is created for each source code file. Each document is then split into different parts, such as a class names part, attribute names part, parameter names part, and others. For instance, the class names part will contain the names of all classes found in the source file.

After the tokenization process and splitting into various parts, the similarity between the two documents is estimated.

Various methods can be used for similarity calculation, such as:

- *Term Frequency-Inverse Document Frequency (TF-IDF):* Weighs the importance of words based on their frequency across documents [6], [8].
- *Cosine Similarity:* Measures the cosine of the angle between two vectors representing the documents [7].

2.1.3. Semantical dependencies

Podgurski and Clarke define semantic dependencies in source code as follows: if changes in a statement s_1 impact the behavior of another statement s_2, then s_1 and s_2 are *semantically dependent* [9].

Semantic dependencies can be identified from the source code by constructing control flow graphs (CFGs). A control flow graph is a directed graph $G = (V, E)$, where:

- V is the set of vertices representing program statements (e.g., assignments, method calls, conditions),
- E is the set of edges representing possible transfers of control between statements.

A vertex $u \in V$ is semantically dependent on a vertex $v \in V$ if changes in v determine changes in u.

In object-oriented (OO) systems, semantic dependencies are dependencies that are not visible in the static structure of the code. There is no direct reference between two entities (e.g., classes or interfaces), but changes in one entity impact the behavior of the other [10, 11, 12].

Example:

Let A a class that uses an object of type I, and let B a class that implements I. For example, class A calls a method `performAction()` via an interface I, while class B implements the `performAction()` method.

If B changes its implementation of `performAction()`, the behavior of A might also change when it uses B. This means that A and B are semantically dependent.

2.1.4. Logical dedependencies

Logical dependencies (also known as logical coupling) can be discovered through software history analysis. They reveal relationships between entities that are not always present in the source code's static structure (i.e., structural dependencies).

Software engineering practice has shown that sometimes modules without explicit structural dependencies still appear to be related. *Co-evolution* represents the phenomenon where one component changes in response to changes in another component [13, 3]. Such changes can be found in the software history maintained by version control systems. Gall et al. defined *logical coupling* as the occurrence of modules that *repeatedly* update together during the evolution of the software system [14, 15, 16].

The concepts of logical coupling and logical dependencies have been applied in different analysis tasks related to software changes: for software change impact analysis [17]; for identifying potential ripple effects caused by software changes during maintenance and evolution [18, 19, 12, 20]; and for exploring their link to defects [21, 22].

An in-depth analysis of how logical dependencies are identified within the versioning system and their applications is provided in Section 2.3.

2.1.5. Additional dependencies

Besides the dependencies mentioned above, there are many other types of dependencies, such as temporal, package, external, and several others.

Temporal dependencies represent a type of dependency where certain operations in the code must occur in a specific order. These operations may belong to the same software component or span across different parts of the system [3].

Example:

```
var file = new File();
file.open("example.txt");
file.readContents();
file.close();
```

In this example, there is a temporal dependency between the `open`, `readContents`, and `close` methods. The `readContents` method cannot function correctly unless the `open` method is called first to open the file. Similarly, the `close` method should be called after `readContents` to release the file resources.

Package dependencies refer to the relationships between different software packages, usually managed using package managers [23].

External dependencies are the relationships between a software system's code and components outside the system, such as third-party services, libraries, APIs, or the programming language itself [24].

2.2. Software change and version control systems

2.2.1. Software change

Software systems have distinctive stages during their life: initial development, evolution, servicing, phase out, and close down [25], [26].

In the *evolution stage*, iterative changes are made. By changes, we mean additions (new software features), modifications (changes of requirements or misun-

derstood requirements), or deletions. There are two main reasons for the change: the software team's learning process and new requests from the client.

Suppose new changes are no longer easy to be made or are very difficult and time-consuming. In that case, the software enters the *servicing stage*, also called aging software, decayed software, and legacy [25], [27].

The main difference between changes made in the evolution and servicing stages is the effort to make changes. In the evolution stage, software changes are made easily and do not require much effort, while in the servicing stage, only a limited number of changes are made and require a lot of effort, so they are time-consuming [28, 29, 30].

The change mini-cycle consists of the following phases [31]:

- Phase 1: The change request. This usually comes from the software users, and it can also be a bug report or a request for additional functionality.
- Phase 2: The planning phase includes program comprehension and change impact analysis. Program comprehension is a mandatory prerequisite of the change, while change impact analysis indicates how costly the change will be. [32]
- Phase 3: The change implementation, restructuring for change, and change propagation.
- Phase 4: Verification and validation.
- Phase 5: Re-documentation.

Understanding these phases is important for ensuring the software system remains maintainable and reliable throughout its lifecycle.

2.2.2. Version control systems

Software evolution implies change which can be triggered either by new feature requests or bug reports [33]. As presented also in section 2.2.1, one phase of the change mini-cycle consists of change implementation and propagation (changing source code files). Usually, developers use version control when it comes to software development. Version control is a system that records changes to a file or set of files over time so that developers can recall specific versions of those files later [34]. Distributed version control systems (such as Git, Mercurial, Bazaar or Darcs) allows many developers to collaboratively develop their projects [35].

Below, we will review some of the operations supported by versioning systems. The operation names in the following text are specific to Git, as Git is the version control system used for data collection in the current work. However, similar operations exist in other versioning systems; for example, Subversion (SVN) and Mercurial also provide operations such as branching, merging, and committing changes.

Repository

A repository serves as a centralized storage location for project files. The entire codebase of a project can be stored within a single repository or be split into a main repository and various modules stored in multiple repositories. Git repositories

can be hosted on numerous platforms, such as GitHub, GitLab, Bitbucket, and others.

It is important to differentiate between Git and Git hosting services. Git is a version control system that allows developers to download and modify code on their local devices while hosting services like GitHub provide platforms for teams to host projects that utilize Git [36], [37].

Commits

Committing is an operation that allows developers to record the latest changes to the codebase in the repository. A commit saves the current state of the code, including changes made since the last commit.

The changes tracked include:

- *Additions:* New files or lines of code created.
- *Modifications:* Updates to existing lines of code or files.
- *Deletions:* Removal of files or lines of code that are no longer needed.

The *push* operation uploads these changes to the remote repository on the hosting service.

A unique identifier, also known as a commit hash, is assigned to each commit. This hash allows developers to track specific changes or revert to earlier code versions. When a developer commits the changes, a commit message is required. This message serves as documentation for the changes, enhancing code traceability and maintainability [36].

Figure 2.1 illustrates how changes are made to the code over time through commits, starting from the initial commit to the latest version.

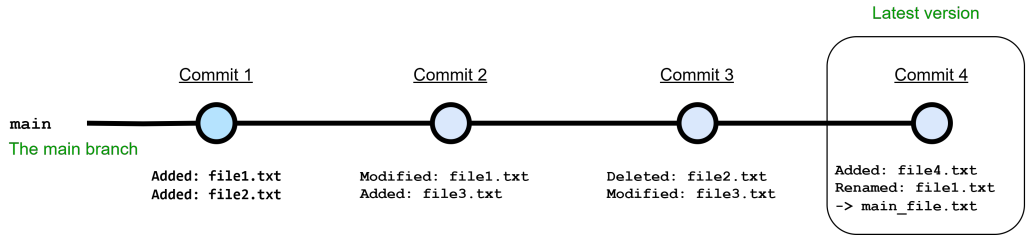

Figure 2.1: Tracking changes through commits.

Branches and Merging

By default, every repository comes with a main (or master) branch. This branch represents the central timeline for code changes and serves as the main integration point for stable code. Other branches can be created for parallel development from this branch.

Branches allow developers to encapsulate changes without affecting the main branch. In most software projects, it is standard practice for developers to use branches for development, with the main branch being locked for direct commits. Changes to the main branch can be integrated through merges from other branches.

Software projects often have multiple branches running in parallel, including branches from other branches. The main branch is not the only branch that supports branching; any branch can be a base for creating other branches.

The operation of integrating changes from one branch into another (usually into the main branch) is called *merging*.

There are three main types of merges in Git [36]:

- *Git Merge:* This is the most commonly used type of merging. It creates a new merge commit that combines all the changes from the branch being merged. Additionally, it retains the history of all the individual commits in the branch.

```
git checkout main
git merge feature-branch
```

- *Git Rebase and Merge:* This operation is typically used when the branch contains a single commit or a small number of commits. It moves all the commits from the source branch to the top of the target branch. The main disadvantage of this operation is that it rewrites the commit history.

```
git checkout feature-branch
git rebase main
git checkout main
git merge feature-branch
```

- *Git Squash and Merge:* This operation compresses all commits from the source branch into a single commit before merging it into the target branch. It results in a cleaner commit history but has the disadvantage of losing individual commits from the source branch.

```
git checkout main
git merge --squash feature-branch
git commit
```

Figure 2.2 illustrates the differences between these three types of merging.

Tagging

Another helpful operation in Git is tagging. Developers use the tagging operation to mark a specific code version at a particular commit. This is usually done for important milestones, such as a new release version or a stable build [36].

Tags provide a way to create a human-readable reference to a specific commit (e.g., `v1.0.0`), as the commit hash can be hard to remember (e.g., `a1b2c3d4e5f6g7h8i9`)

Git supports two types of tags:

- *Lightweight Tags:* Simple references to a commit that do not contain any additional metadata.
- *Annotated Tags:* These include additional metadata such as the author's name, date, and message, making them more suited for marking releases.

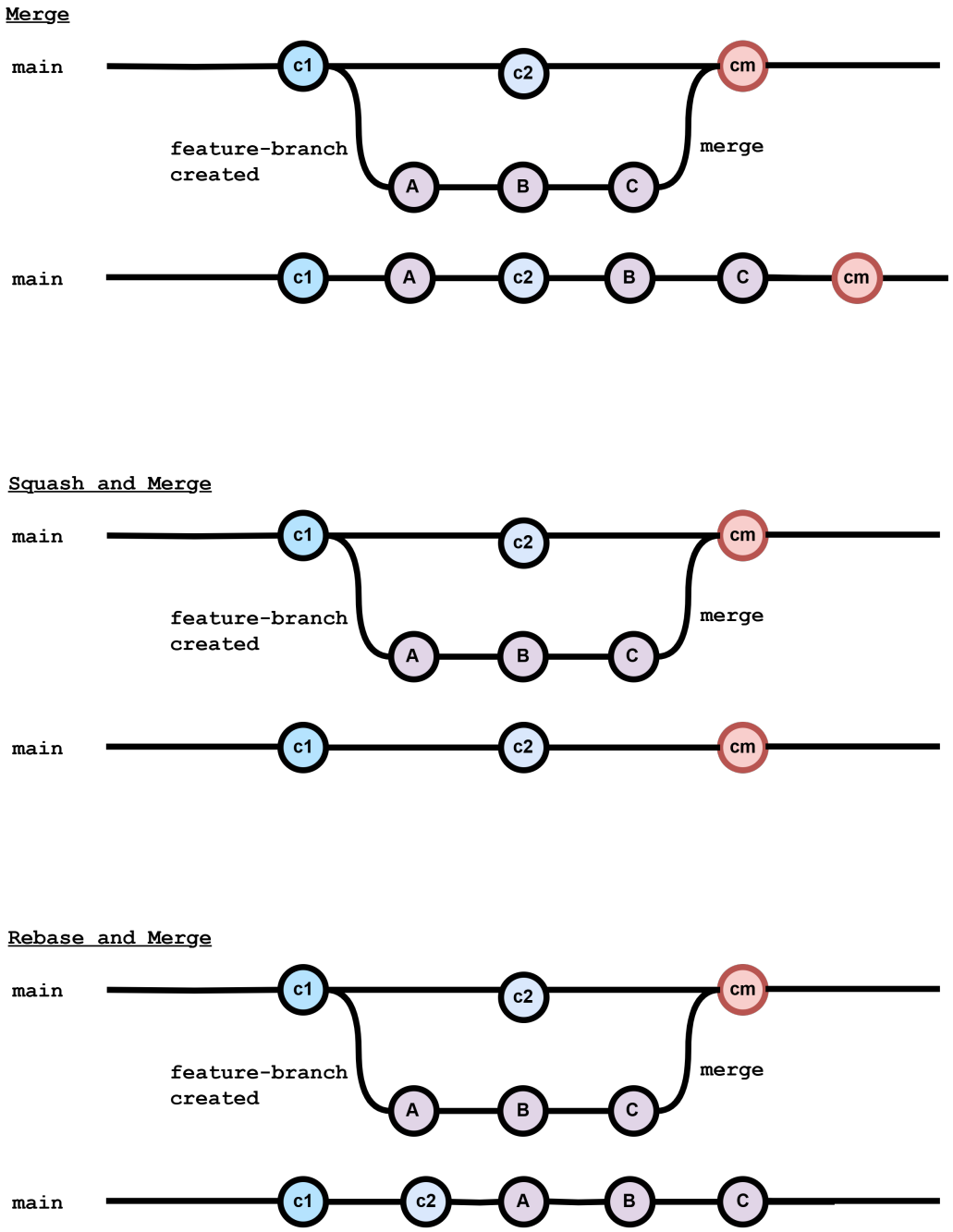

Figure 2.2: Comparison of Git merge types.

2.3. Current status of research on logical dependencies

2.3.1. Logical dependencies in software systems

The versioning system stores the long-term change history of every file in a

project. Each change made by an individual at a specific point in time is recorded as a commit [35]. This historical data provides insights into how software components evolve over time. Logical dependencies, also known as evolutionary coupling or co-changes, are derived from patterns of co-evolution, where one component changes in response to changes in another [13, 38].

Gall [14, 15, 16] identified *logical coupling* as the relationship between two modules that repeatedly change together during the historical evolution of a software system. Logical dependencies can be expressed at two levels of abstraction: module-level [39] and file/class-level [15, 40].

Several tools have been developed to detect and use logical dependencies. For instance, the **ROSE** tool, developed by Zimmermann et al. [22], mines version histories to suggest related code changes based on historical coupling between software entities. ROSE can predict 26% of further files to be changed and 15% of the functions or variables involved in future modifications.

Kagdi et al. [41, 42] developed **sqminer**, a tool based on the SPADE (Sequential Pattern Discovery Algorithm) to mine sequences of file changes for software change prediction. They also proposed an approach that combines conceptual couplings (derived from textual analysis of source code comments and identifiers) with logical dependencies (mined using the sqminer tool) to enhance software change impact analysis [41].

Moonen et al. [43, 44, 45] proposed an algorithm called **Tarmaq** that analyzes evolutionary coupling for predicting co-changes. Their results demonstrated that Tarmaq achieves better performance than the ROSE tool. Similarly, Mondal et al. [46] developed **HistoRank**, an algorithm that uses five different ranking strategies for prioritizing and filtering co-changes.

As shown above, logical dependencies have been widely used in co-change prediction. Additionally, they have been used for various other purposes, such as detecting code clones [47], identifying buggy code [48], evaluating the impact of software changes [17], and identifying potential ripple effects caused by software changes during maintenance and evolution [18, 19].

Despite their utility, logical dependency usage presents several challenges. One significant issue is the large volume of information generated during extraction, which must be processed and filtered to remove noise [49, 33, 50]. For instance, *Oliva and Gerosa* [19] found that the set of co-changed classes was much larger than the set of structurally coupled classes. At least 91% of logical dependencies involve files that are not structurally related. This implies that not all change dependencies are linked to structural dependencies and that other factors may cause software artifacts to be change-dependent.

Ajienka and Capiluppi also explored the interplay between the logical and structural coupling of software classes. In [51, 52], they conducted experiments on 79 open-source systems. For each system, they identified the sets of structural dependencies, logical dependencies, and the intersections of these sets. They studied the overlap between these dependencies, concluding that not all co-changed class pairs (logical dependencies) are also linked by structural dependencies. Another observation, which was not deeply investigated by the authors in [51, 52], is the ratio

between the total number of logical dependencies and structural dependencies in software systems. Based on their raw data, the average ratio across all analyzed projects is approximately 12.

Applications based on dependency analysis could benefit from incorporating non-structural dependencies alongside structural ones. For example, works that investigate various methods for architectural reconstruction [53, 54, 55], which rely mostly on structural dependency data, could enhance their dependency models by including logical dependencies.

Logical dependencies should integrate harmoniously with structural dependencies. Valid logical dependencies should not be excluded from the model, but structural dependencies should also not be overwhelmed by questionable logical dependencies. Therefore, to effectively incorporate logical dependencies into dependency models, co-changes must be filtered to remain only with a reduced and relevant set of valid logical dependencies.

2.3.2. Existing filtering techniques

Currently, there are no fixed rules for filtering extracted class co-changes to ensure they form a set of valid logical dependencies. Most studies using or investigating logical dependencies have applied various filtering techniques to the extracted co-changes. Below are some of the most commonly used filtering methods cited in the literature.

Commit Size

One of the most frequently used filters for co-change extraction is the commit size filter. Commit size refers to the number of code files modified in a specific commit. Large commits, which involve a significant number of files, are often the result of non-code-related tasks, such as branch merges or folder renaming. These operations can introduce irrelevant co-changing pairs of entities, adding noise. To solve this issue, commit size filtering is applied to the information extracted from version control systems.

Ajienka and Capiluppi [51] established a threshold of 10 files, discarding all commits with more than 10 modified files from their research.

Kagdi et al. also applied the same threshold, excluding commits with more than 10 source files from their analysis.

Ying et al. [56] set a higher threshold, excluding transactions involving more than 100 files.

Zimmermann et al. [22] configured the ROSE tool to exclude changes involving more than 30 files.

Moonen et al. [43] considered seven different transaction filtering sizes: 2, 4, 6, 8, 10, 20, and 30 (with 30 being the upper bound suggested by Zimmermann) and recommended excluding transactions larger than 8 files.

However, most of the works presented above do not discuss how the thresholds

were chosen, nor do they analyze the impact of different threshold values on the quantity and quality of the data extracted.

Support and confidence

Filtering based only on commit size is not enough. While this type of filtering can reduce the total number of extracted co-changes, it does so without guaranteeing that the remaining co-changes are more relevant.

Unrelated files can sometimes be updated in small commits due to human error. For instance, a file that was omitted from the current commit may be committed in the next commit together with some unrelated files. Such scenarios can introduce co-changing pairs that are not genuinely linked. To address this problem, a filter based on the occurrence rate of co-changing pairs should be applied. Co-changing pairs that occur multiple times are more likely to be dependent than those that appear only once [57].

Zimmermann et al. [22, 58] introduced the support and confidence metrics to measure the significance of co-changes based on the occurrence rate of co-changing pairs.

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 (4.1), 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).

$$\text{Confidence}(A \rightarrow B) = \frac{\text{Nr. of commits containing } A \text{ and } B}{\text{Nr. of commits containing } A} \tag{2.1}$$

The support metric represents the frequency of changes for an association rule and can take values from 0 to infinity. On the other hand, the confidence metric is defined within the interval [0, 1]. As in Equation (4.1), the confidence metric measures the likelihood of a co-change. It can never exceed 1 because Commits$(A \cap B)$ is always a subset of Commits(A).

For example, consider the case where entity A is modified in 10 commits and entity B is modified together with A in 7 of those commits. The confidence is then:

$$\text{Confidence}(A \rightarrow B) = \frac{7}{10} = 0.7.$$

If A and B are always modified together in all 10 commits involving A, the confidence would be:

$$\text{Confidence}(A \rightarrow B) = \frac{10}{10} = 1.$$

Many studies have used the support and confidence metrics. *Zimmermann*

et al. [22], in their ROSE tool, used different combinations of minimum support and confidence thresholds. They applied three minimum support thresholds, 1, 3, and 5, together with confidence thresholds ranging from 0.1 to 0.9, to predict future changes in software systems.

Ying et al. [56] recommend potentially relevant source code to developers during modification tasks. The support thresholds in their study vary based on the analysis, ranging from 5 to 30 (5, 10, 15, 20, 25, 30). For confidence, Ying et al. chose not to use this metric, as they consider it misleading when some files are changed much more frequently than others.

Kagdi et al. [41], in their studies on software change prediction, used minimum support thresholds of 1, 2, 4, and 8. For confidence, they considered all possible values greater than zero.

Mandal et al. [47], in their study on detecting clones and analyzing their evolution, developed a tool called MARC (Mining Association Rules among Clones). The tool detects code clones and ranks their change-proneness based on support and confidence values. In their experiments, they applied a minimum support threshold of 1.

Oliva and Gerosa [19, 59] investigated the interplay between structural and logical dependencies, using a modified version of the support and confidence metrics introduced by Zimmermann, adjusted to a set of commits. The study found that the support interval [1, 7] contains 90% of all logical dependencies, while the interval [0, 10] contains 99.9% of logical dependencies. The interval [11, 31] represents only 0.1% of logical dependencies, with high variation observed across coupling levels.

The confidence metric was divided into three categories: low logical coupling (0.00–0.33), medium logical coupling (0.33–0.66), and high logical coupling (0.66–1.00). The study concluded that classes with high logical coupling were the least influenced by structural coupling.

Ajienka and Capiluppi [52], in their study on the overlappings between structural and logical dependencies used a support threshold of 0.1 and a confidence threshold of 0.01.

History length and age

Moonen et al. investigated the influence of history length (the number of analyzed transactions) and history age (the number of transactions that have occurred since the last co-change) when mining evolutionary coupling. Their study, which analyzed over 540,000 commits, found no evidence to support the idea that there is an upper limit to the amount of history that can be used for mining evolutionary coupling before outdated knowledge starts to negatively affect the quality of the extracted information [44].

2.4. Applications of software dependencies

This section reviews several applications of software dependencies (e.g., struc-

tural, lexical, semantical, logical dependencies). These dependencies play an important role in various software engineering tasks, architecture reconstruction, clone identification, and more.

Architecture reconstruction. Currently, software systems contain tens of thousands of lines of code and are updated daily by multiple developers. The software architecture is important for understanding and maintaining a system. Often, code updates are made without checking or updating the architecture.

These kinds of updates cause the architecture to drift from the reality of the code over time. Therefore, reconstructing the architecture and verifying if it still matches the actual implementation is important [60, 26, 61]. Architecture reconstruction has mainly been done using structural dependencies [62], [55], [63], but recent works also include semantical and logical dependencies [6], [8], [64].

Identifying Clones. Research suggests that a considerable portion (around 5-10%) of the source code in large-scale software is duplicate code ("clones"). Source code is often duplicated for a variety of reasons: programmers may simply reuse a piece of code by copying and pasting, or they may "reinvent the wheel" [65], [66]. Detection and removal of clones can significantly decrease software maintenance costs [67], [68].

Structural dependencies can be used to detect code clones. For example, Cordy and Roy created the NiCad tool, which receives the entire code base of a project as input to detect project clones [69]. The same authors also used versioning system information to link clones to bug-fix commits from the versioning system [70]. Other researchers have used semantic dependencies and machine learning techniques to identify clones [71], while others have used lexical dependencies extracted from code comments [72].

Code Smells. Fowler defined code smells citebookFowler as patterns generally associated with bad design and poor programming practices. Originally, code smells were used to identify areas in software that may require refactoring [73]. Studies have found that code smells can impact software comprehension and increase changes and faults in the system [74], [75], [76].

Examples of code smells include:

- **Large Class:** A class with many fields and methods, making it difficult to maintain or understand.
- **Feature Envy:** Methods that access more methods and fields of another class than of their class.
- **Data Class:** Classes that contain only fields and no meaningful functionality.
- **God Class:** A class that centralizes too much responsibility, often not respecting the Single Responsibility Principle.
- **Refused Bequest:** A subclass that inherits many fields or methods from its parent but leaves them unused.
- **Parallel Inheritance:** Every time a subclass is added to one class, a subclass must also be added to another class.
- **Shotgun Surgery:** A single change requires modifications in multiple classes.

Code smell detection approaches often use structural information extracted

from static code analysis. For example, Marinescu [77] proposed a method to identify smells like God Class and Data Class based on structural metrics. Recent works have used machine learning algorithms in combination with structural metrics to improve the detection of code smells [78, 79].

Certain smells, however, are more effectively detected using information from versioning systems, which capture logical dependencies. For instance, smells like *Parallel Inheritance* and *Shotgun Surgery* have been successfully identified by analyzing co-change patterns in version control systems [80].

Key Classes. The concept of key classes was first introduced by Zaidman et al. [81], referring to classes found in documents that provide an architectural overview of the system or an introduction to its structure. Tahvildari and Kontogiannis provided a more detailed definition of the key classes concept: *"Usually, the most important concepts of a system are implemented by very few key classes which can be characterized by specific properties. These classes, which we refer to as key classes, manage many other classes or use them in order to implement their functionality. The key classes are tightly coupled with other parts of the system."* [82].

Key class identification can be performed using different algorithms with various inputs. Most of the research is based on using structural dependencies [55], [83], [84], [1], [81], [85], class diagrams [86], or *dynamic dependencies* obtained through runtime analysis [81].

Comprehension. Software comprehension is the process of gaining knowledge about a software system. An increased understanding of the software system helps activities such as bug correction, enhancement, reuse, and documentation [87], [88].

Previous studies show that the proportion of resources and time allocated to maintenance activities can vary from 50% to 75% [89]. Within the maintenance process, the biggest effort is dedicated to understanding the system.

To support software comprehension, various tools have been developed to help this process. For example, the COSPEX tool developed by Gupta et al. uses source code analysis to help novice developers better comprehend their tasks [90]. Şora proposes a tool that uses structural dependencies and graph-based ranking to generate an executive summary highlighting the most important classes in the software system [83].

Fault Localization.

Debugging software is an expensive and mostly a manual process. Among all debugging activities, fault localization is the most time-consuming task [91].

Software developers typically locate faults in their programs through a manual process. This process begins when developers observe failures in the program. They select a dataset to inject into the system, which is a set of data likely to replicate previous failures or trigger new ones, and set breakpoints using a debugger. They then monitor the system's state until a failure occurs and backtrack from the failure state to identify the root cause of the fault [92, 93].

Several tools and methods have been developed to support in the fault local-

ization process. A commonly used technique is running coverage tests to evaluate the likelihood of faults in different parts of the source code. Code statements or methods having more failing tests than passing tests are labeled more likely to contain faults. An example of such a tool is GZoltar, developed by Campos et al. [94].

More recent studies have suggested enhancing fault localization techniques by incorporating information from versioning systems in addition to source code analysis. Wen et al. empirically demonstrated that versioning system data could improve fault localization. Their approach identifies code entities modified by more ”bug-inducing commits” than regular commits, labeling them as potential fault sources [95].

Defect Prediction.

Fault localization is the process of identifying elements responsible for software failures reported by users or discovered by developers. Defect prediction, on the other hand, predicts elements that are likely to be fault-prone before faults occur. Due to this difference, fault localization and defect prediction are studied as two separate research areas [96, 97].

Research has shown that modules or entities most likely to contain defects can be identified based on structural metrics (e.g., lines of code, cyclomatic complexity) and versioning system data (e.g., frequency of changes, modified code in a source file) [98], [99].

3. FILTERING LOGICAL DEPENDENCIES

3.1. Extracting structural dependencies

A dependency is created between two elements that are in a relationship and indicates that an element of the relationship, in some manner, depends on the other element of the relationship [2], [3].

Structural dependencies can be found by analyzing the source code [4], [5], [100]. A structural dependency between two classes A and B is given by the fact that A statically depends on B, meaning that A cannot be compiled without knowing about B. In object oriented systems, this dependency can be given by many types of relationships between the two classes: A extends B, A implements B, A has attributes of type B, A has methods which have type B in their signature, A uses local variables of type B, A calls methods of B.

We use an external tool called srcML [101] to convert all source code files from the current release into XML files. All the information about classes, methods, calls to other classes are extracted by parsing the XML files and building a dependency data structure [88], [102]. We choose the srcML format because it has the same markup for different programming languages and can ease the parsing of source code written in various programming languages such as Java, C++, and C#.

3.2. Extracting co-changing pairs

Logical dependencies (a.k.a logical coupling) can be found by software history analysis and can reveal relationships that are not always present in the source code (structural dependencies).

TBA

3.3. Tool for measuring software dependencies

To establish structural and logical dependencies, we developed a tool that takes as input the source code repository URL of a given system and extracts from it the software dependencies [103]. From a workflow point of view, we can identify 3 major types of activities that the tool does: downloads the required data from the git repository, extracts from the source code the structural dependencies and, extracts and filters the co-changing pairs from the repository's commit history. Figure 3.1 represents the activities mentioned above. Each block represents a different activity.

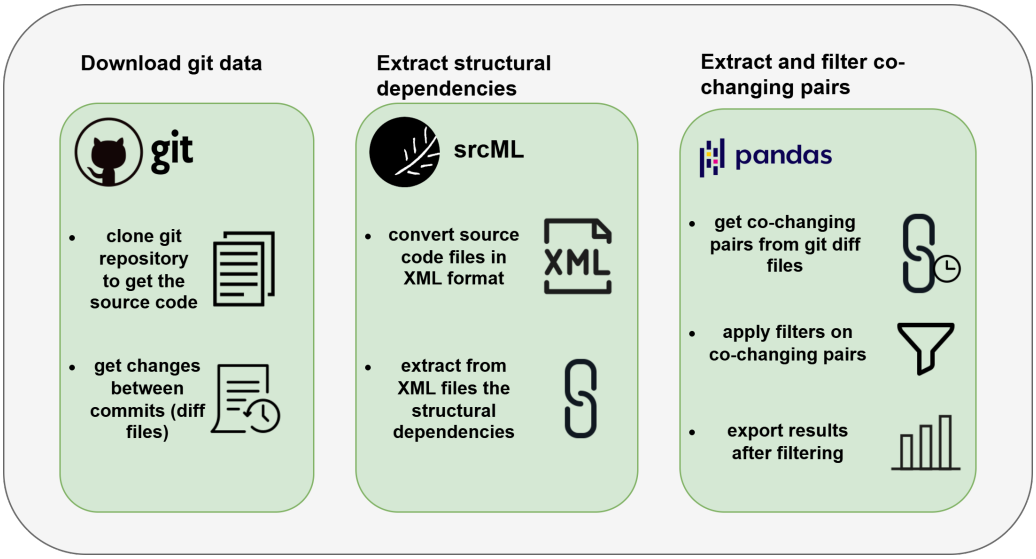

Figure 3.1: Tool workflow and major activities.

Download git data.

The source code repository provides us all the needed information to extract both types of dependencies. It holds the code of the system but also the change history of the system. We use the source code for structural dependencies extraction and the change history for co-changing pairs extraction. To get the source code files and the change history, we first need to know the repository URL from GitHub (GitHub is a Git repository cloud-based hosting service). With the GitHub URL and a series of Git commands, the tool can download all the necessary data for dependencies extraction.

As we can see in figure 3.2, the *"clone"* command will download a Git repository to your local computer, including the source code files. The *"diff"* command will get the differences between two existing commits in the Git repository. The tool gets the Git repository and the source code files by executing the "clone" command. Afterward, it gets all the existing commits within the Git repository. The commits are ordered by date, beginning with the oldest one and ending with the most recent one. The tool executes the "diff" command between each commit and its parent (the previous commit). The "diff" command generates a text file that contains the differences between the two commits: code differences, the number of files changed and changed file names.

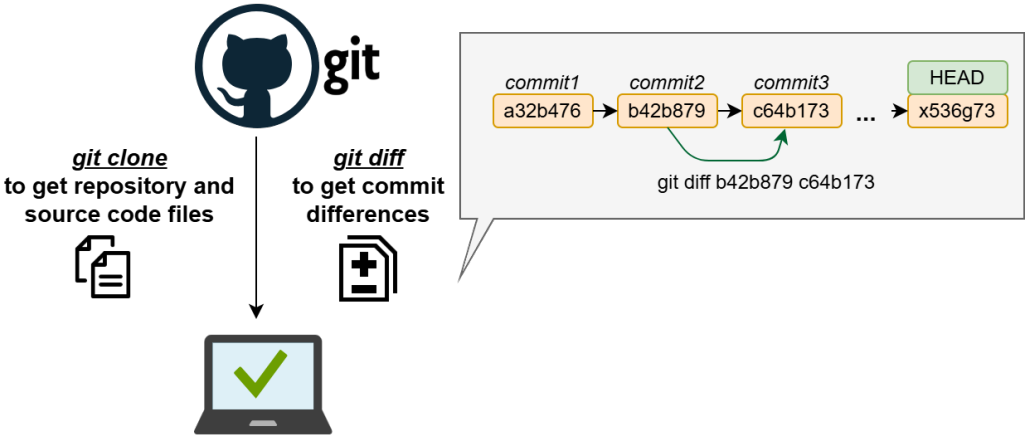

Figure 3.2: Commands used to download the required data from GitHub.

Extract structural dependencies.

To extract the structural dependencies from the source code files the tool converts each source code file into srcML format using an open-source tool called srcML. The srcML format is an XML representation for source code. Each markup tag identifies elements of the abstract syntax for the language [101]. After conversion, the tool parses each file and identifies all the defined entities (class, interface, enum, struct) within the file. It also identifies all the entities that are used by the entities defined. The connection between both types of entities mentioned above constitutes a structural dependency.

Extract and filter co-changing pairs.

The process of extracting and filtering the co-changing pairs is represented in figure 3.3. For co-changing pairs extraction, the tool parses each generated diff file. For each file, the tool gets the number of changed files and the name of the files. After structural dependencies extraction, the tool knows all the software entities contained in a file. Two entities from two changed files form a co-changing pair. After all the co-changing pairs of one diff file are extracted, the tool moves to the next diff file and extracts the set of co-changing pairs.

As will be presented in more details in sections 3.5.1, 3.5.2, and 3.5.3, not every co-changing pair extracted is a logical dependency. For a co-changing pair to be labeled as a logical dependency, it has to meet some criteria. Each criterion constitutes a filter that a co-changing pair has to pass in order to be called logical dependency. The filters are implemented in the tool and can be combined. The input for each filter is the set of co-changing pairs extracted, and the output is the remaining co-changing pairs that respect the filter criterion.

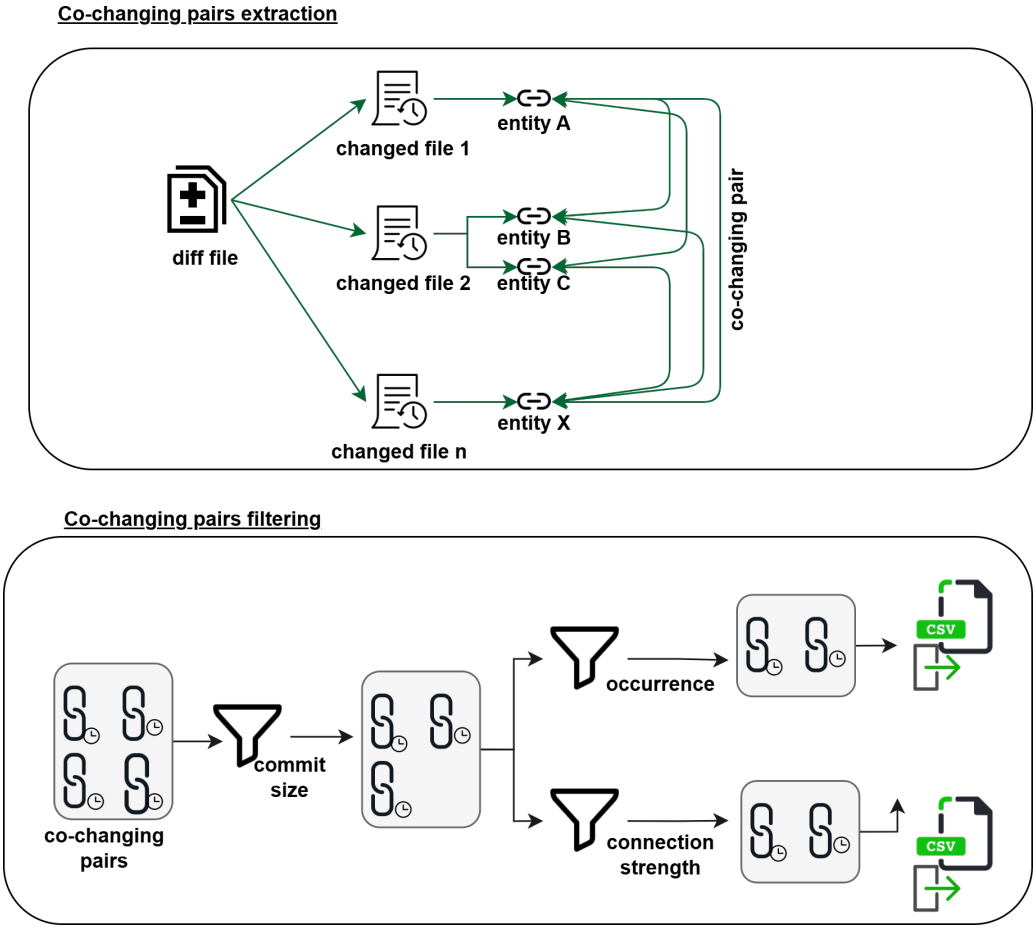

Figure 3.3: Co-changing pairs extraction and filtering.

3.4. Data set used

We have analyzed a set of open-source projects found on GitHub[1] [61] in order to extract the structural and logical dependencies between classes. Table 3.1 enumerates all the systems studied. The 1st column assigns the projects IDs; 2nd column shows the project name; 3rd column shows the number of entities(classes and interfaces) extracted; 4th column shows the number of most recent commits analyzed from the active branch of each project and the 5th shows the language in which the project was developed.

[1]http://github.com/

Table 3.1: Summary of open source projects studied.

ID	Project	Nr. of entites	Nr. of commits	Type
1	bluecove	2685	894	java
2	aima-java	5232	1006	java
3	powermock	2801	949	java
4	restfb	3350	1391	java
5	rxjava	21097	4398	java
6	metro-jax-ws	6482	2927	java
7	mockito	5189	3330	java
8	grizzly	10687	3113	java
9	shipkit	639	1563	java
10	OpenClinica	9655	3276	java
11	robolectric	8922	5912	java
12	aeron	4159	5977	java
13	antlr4	4747	4431	java
14	mcidasv	3272	4136	java
15	ShareX	4289	5485	csharp
16	aspnetboilerplate	9712	4323	csharp
17	orleans	16963	3995	csharp
18	cli	2063	4488	csharp
19	cake	12260	2518	csharp
20	Avalonia	16732	5264	csharp
21	EntityFrameworkCore	50179	5210	csharp
22	jellyfin	8764	5433	csharp
23	PowerShell	2405	3250	csharp
24	WeiXinMPSDK	7075	5729	csharp
25	ArchiSteamFarm	702	2497	csharp
26	VisualStudio	4869	5039	csharp
27	CppSharp	17060	4522	csharp

3.5. Overview in types of filters used

3.5.1. Filtering based on the size of commit transactions

As presented in section 3.2, according to surveys, co-changing pairs are not used because of their size. One system can have millions of co-changing pairs. With this filtering type, we not only want to decrease the total size of the extracted co-changing pairs. But also to be one step closer to the identification of the logical dependencies among the co-changing pairs. In this step, we want to filter the co-changing pairs extracted after commit size (cs). This means that the co-changing pairs are extracted only from commits that involve fewer files than an established threshold number.

Different works have chosen fixed threshold values for the maximum number of files accepted in a commit. Cappiluppi and Ajienka, in their works [51], [52] only take into consideration commits with less then 10 source code files changed in building the logical dependencies.

The research of Beck et al [38] only takes in consideration transactions with up to 25 files. The research [19] provided also a quantitative analysis of the number of files per revision; Based on the analysis of 40,518 revisions, the mean value obtained

for the number of files in a revision is 6 files. However, standard deviation value shows that the dispersion is high.

We analyzed the overall transaction size trend for 27 open-source csharp and java systems with a total of 74 332 commits. The results are presented in Figure 3.4 and in table 3.2, based on them we can say that 90% of the total commit transactions made are with less than 10 source code files changed. This percent allows us to say that setting a threshold of 10 files for the maximum size of the commit transactions will not affect so much the total number of commit transactions from the systems since it will still remain 90% of the commit transactions from where we can extract co-changing pairs [103].

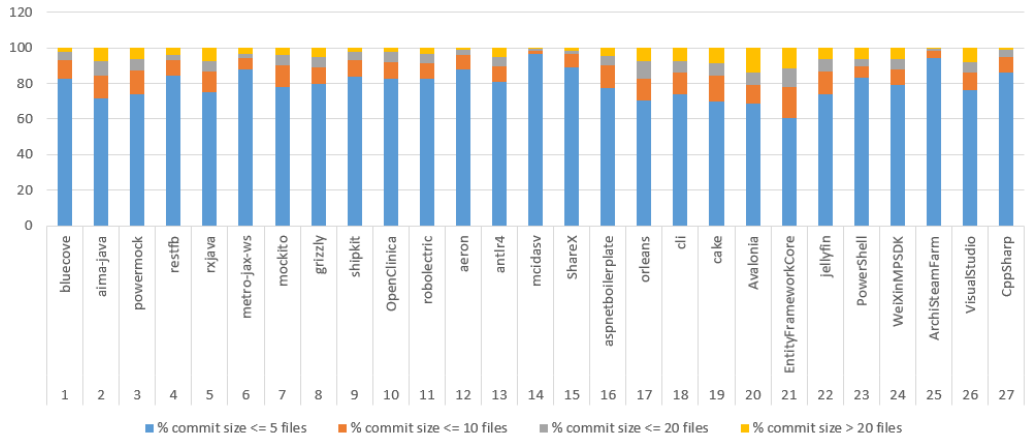

Figure 3.4: Commit transaction size(cs) trend in percentages.

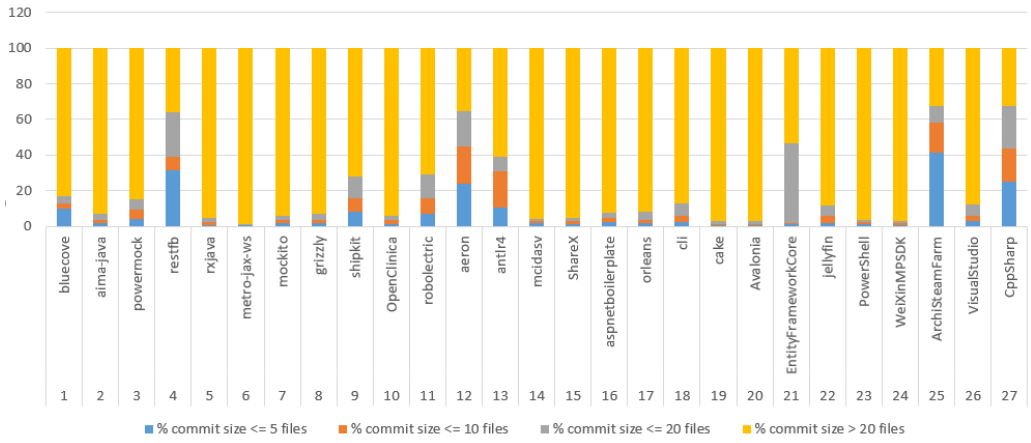

Figure 3.5: Percentages of LD extracted from each commit transaction size(cs) group.

As we can see in Figure 3.5 even though only 5% of the commit transactions have more than 20 files changed ($20 < cs < inf$) they generate in average 80% of

the total amount of co-changing pairs extracted from the systems. The high number of co-changing pairs extracted from such a small number of commit transactions is caused by the number of files involved in those commit transactions.

One single commit transaction can lead to a large amount of co-changing pairs. For example in RxJava we have commit transactions with 1030 source code files, this means that those commits can generate ${}^{n}C_{k} = \frac{n!}{k!(n-k)!} = \frac{1030!}{2!(1028)!} = 529935$ logical dependencies. By setting a threshold on the commit transaction size we can avoid the introduction of those co-changing pairs into the system.

So filtering 10% of the total amount of commit transactions can lead to a significant decrease of the amount of co-changing pairs and that is why we choose the value of 10 files as our fixed threshold for the maximum size of a commit transaction [103].

Table 3.2: Commit transaction size(cs) trend and average per system.

Nr.	Project	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$	Avg
1	bluecove	738	97	37	22	4.9
2	aima-java	733	134	74	65	7.24
3	powermock	685	128	66	70	9.61
4	restfb	1160	127	44	60	9.9
5	rxjava	3395	447	253	303	8.46
6	metro-jax-ws	2583	198	78	68	4.33
7	mockito	2522	433	222	153	6.33
8	grizzly	2487	302	180	144	5.28
9	shipkit	1311	151	64	37	4.26
10	OpenClinica	2837	250	119	70	3.31
11	robolectric	4827	503	264	318	7.43
12	aeron	4844	684	300	149	4.6
13	antlr4	3426	437	304	264	8.5
14	mcidasv	3996	81	35	24	2.47
15	ShareX	4731	529	145	80	4.69
16	aspnetboilerplate	3208	569	321	225	6.61
17	orleans	2780	518	369	328	8.95
18	cli	3377	551	308	252	6.43
19	cake	1785	359	174	200	9.89
20	Avalonia	3806	641	371	446	8.43
21	EntityFrameworkCore	2866	878	644	822	15.38
22	jellyfin	4007	662	419	345	6.25
23	PowerShell	2702	224	133	191	7.33
24	WeiXinMPSDK	4604	526	296	303	9.01
25	ArchiSteamFarm	2357	92	28	20	2.24
26	VisualStudio	3902	521	295	321	6.71
27	CppSharp	3870	390	203	59	3.28

3.5.2. Filtering based on number of occurrences

In the previous section, we filtered the co-changing pairs based on the commit size. Even though the number of extracted co-changing pairs was reduced, this type of filtering will not guarantee that the remaining co-changing pairs can pass as logical dependencies. One occurrence of a co-change pair can be a valid logical dependency, but can also be a coincidence.

Taking into consideration only co-changing pairs with multiple occurrences as valid dependencies can lead to more accurate results. But, if the project studied has a relatively small amount of commits, the probability to find multiple updates of the same classes at the same time is less likely to happen, so filtering after the number of occurrences can lead to filtering all the co-changes extracted.

We have performed a series of analyses on the test systems, incrementing the threshold value occurrence (occ) from 1 to 4. The co-changing pairs are extracted only for commits with the commit transaction size less or equal to 10. For each threshold mentioned above, the extracted co-changing pairs are filtered again by the occurrence threshold established. All the co-changing pairs that do not exceed the minimum number of occurrences are discarded.

The results of the analysis are presented in Table 3.3 as percentages of co-changing pairs that are also structural dependencies and Table 3.4 as ratio of the number of co-changing pairs to the number of structural dependencies (SD).

Table 3.3: Percentage of co-changing pairs that are also structural dependencies.

ID	$occ \geq 1$	$occ \geq 2$	$occ \geq 3$	$occ \geq 4$
1	7,13	7,77	7,99	19,71
2	19,54	25,76	29,55	32,16
3	6,66	8,58	11,82	14,87
4	1,16	1,17	0,91	0,80
5	3,99	3,96	7,75	7,49
6	13,92	20,16	22,91	22,77
7	8,38	9,28	14,93	14,58
8	6,70	9,73	14,20	15,60
9	16,98	23,34	29,22	32,89
10	8,94	9,15	11,05	10,59
11	4,99	6,92	8,88	11,08
12	13,19	17,15	18,60	19,57
13	2,43	5,59	8,33	8,21
14	13,27	18,88	19,02	19,28
15	12,90	21,95	25,51	27,01
16	13,33	17,34	18,53	16,24
17	6,09	6,18	6,41	6,44
18	9,73	10,60	14,27	18,80
19	10,26	13,54	13,64	12,60
20	12,83	18,36	21,00	25,72
21	2,86	4,65	5,70	4,98
22	5,20	6,56	8,18	8,90
23	8,23	13,64	17,04	17,65
24	6,77	10,89	14,47	16,05
25	9,85	10,15	11,65	11,33
26	8,65	10,79	12,78	14,34
27	7,04	8,78	9,87	10,08
Avg	8,93	11,88	14,23	15,55

Based on Table 3.3 we can say that only a small percentage of the extracted co-changing pairs are also structural dependencies. This is consistent with the findings of related works [51], [52]. The percentage of co-changing pairs that are also structural dependencies increases with the minimum number of occurrences because the number of co-changing pairs from the systems decreases with the minimum number

Table 3.4: Ratio of number of co-changing pairs to number of structural dependencies.

ID	$occ \geq 1$	$occ \geq 2$	$occ \geq 3$	$occ \geq 4$
1	4,13	1,94	1,23	0,26
2	0,81	0,33	0,16	0,10
3	5,12	1,93	0,78	0,38
4	53,36	42,00	38,31	36,30
5	4,27	2,90	0,88	0,72
6	1,07	0,46	0,30	0,23
7	4,09	2,38	0,99	0,73
8	4,06	1,57	0,76	0,49
9	3,64	2,03	1,14	0,77
10	1,41	1,01	0,47	0,34
11	7,91	4,47	2,93	2,03
12	3,92	2,15	1,47	1,07
13	10,15	3,18	1,22	1,03
14	3,07	1,53	1,16	0,97
15	2,34	0,84	0,48	0,33
16	1,21	0,47	0,26	0,19
17	2,99	1,83	1,11	0,84
18	2,26	1,37	0,67	0,40
19	2,32	1,38	0,76	0,67
20	1,24	0,58	0,35	0,18
21	5,33	2,12	1,27	1,05
22	3,38	1,88	0,99	0,74
23	3,62	1,22	0,76	0,37
24	2,57	1,22	0,67	0,46
25	7,47	5,36	4,16	3,73
26	4,03	2,16	1,50	1,15
27	7,46	4,26	2,99	2,43
Avg	5,67	3,43	2,51	2,15

of occurrences. We calculate the overlapping between co-changing pairs and structural dependencies not only because we want to get an idea of how many structural dependencies are reflected in the versioning system through co-changing pairs, but also because we want to eliminate co-changing pairs that are structural dependencies since they don't bring any new information about the system.

We stopped the minimum occurrences threshold to 4 because we observed that for systems with ID 2, 6, 10, and 16 from Table 3.4 the ratio number is lower than 1, which means that the number of structural dependencies is higher than the number of co-changing pairs. On the other hand, for systems with ID 4, 11, 25, 27, the threshold of 4 for a minimum number of occurrences does not change the discrepancy between the number of co-changing pairs and structural dependencies.

If we try to go higher with the occurrences threshold, we will risk filtering all the existing co-changing pairs for some systems. So, filtering with a threshold of 4 for the minimum number of occurrences will indeed filter the logical dependencies, but for some of the systems, the remaining number of co-changing pairs will still be significantly higher compared to the number of structural dependencies.

3.5.3. Filtering based on connection strength

In section 3.5.1 we filtered the co-changing pairs extracted from the versioning system history based on the commit size. Based on the results obtained, we decided to filter out all co-changing pairs extracted from commits with more than 10 files changed.

In section 3.5.2, we added a new filtering rule based on the occurrence of a co-changing pair. The new filter is applied to the co-changing pairs resulted after commit size filtering. In this case, the filtering method proved insufficient due to the size diversity of the systems. One important conclusion drawn from the occurrence number filtering is that setting a hard threshold for a filter is not always a good idea. One threshold value can be too much for a small-sized system and too little for a medium-sized system.

To avoid the above problem, we decided to introduce another filter complementary to the commit size filter described in section 3.5.1. This filter focuses on the connection strength of a co-changing pair. In this section, we will filter out all the co-changing pairs that are not strongly connected.

To determine the connection strength of a pair, we first need to calculate the connection factors for both entities that form a co-changing pair. Assuming that we have a co-changing pair formed by entities A and B, the connection factor of entity A with entity B is the percentage from the total commits involving A that contains entity B. The connection factor of entity B with entity A is the percentage from the total commits involving B that contain also entity A.

$$connection\ factor\ for\ A = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ A} \tag{3.1}$$

$$connection\ factor\ for\ B = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ B} \tag{3.2}$$

As a practical example, if the pair formed by A and B update together 7 times and the total number of commits involving A is 20 and involving B is 7. The factor for A is 35 and for B is 100. The factor of 100 is the maximum factor that you can have and means that in all the commits involving B, also A is present.

Due to the fact that the factors obtained can vary from 0 to 100, for this filter, we begin with a threshold value of 10 and increment it by 10 until we reach 100.

The co-changing pairs are filtered out based on two scenarios:

- factor A and factor B $\geq threshold$%
- factor A or factor B $\geq threshold$%

In table 3.5 we have on the columns the ratio between the number of structural dependencies and the number of co-changing pairs that resulted after filtering out pairs that have at least one factor below the specified threshold in the column header. In table 3.6 we have on the columns the ratio between the number of structural dependencies and the number of co-changing pairs that resulted after filtering out pairs that have both factors below the specified threshold in the column header.

Table 3.5: Ratio of number of filtered co-changing pairs to number of SD, when factor A and factor B $\geq threshold$%

Project	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 100%
bluecove	1.326	0.658	0.433	0.401	0.244	0.199	0.195	0.022	0.011	0.011
aima-java	0.266	0.137	0.070	0.044	0.036	0.019	0.005	0.004	0.003	0.003
powermock	0.505	0.243	0.147	0.086	0.061	0.031	0.031	0.031	0.031	0.031
restfb	0.822	0.163	0.045	0.017	0.011	0.002	0.001	0.001	0.001	0.001
rxjava	0.234	0.119	0.054	0.037	0.034	0.018	0.013	0.011	0.007	0.007
metro-jax-ws	0.227	0.155	0.101	0.077	0.070	0.036	0.018	0.017	0.016	0.016
mockito	1.590	0.804	0.357	0.288	0.215	0.088	0.052	0.036	0.032	0.032
grizzly	2.073	0.293	0.170	0.111	0.093	0.050	0.039	0.034	0.021	0.007
shipkit	1.495	0.479	0.271	0.142	0.108	0.059	0.047	0.011	0.008	0.008
OpenClinica	0.253	0.135	0.093	0.078	0.062	0.042	0.024	0.019	0.019	0.017
robolectric	0.114	0.086	0.064	0.037	0.027	0.025	0.001	0.000	0.000	0.000
aeron	0.277	0.136	0.085	0.069	0.053	0.045	0.039	0.015	0.007	0.004
antlr4	11.363	0.721	0.031	0.010	0.007	0.004	0.000	0.000	0.000	0.000
mcidasv	3.225	0.805	0.660	0.533	0.493	0.454	0.386	0.356	0.005	0.005
ShareX	6.097	0.725	0.663	0.564	0.500	0.242	0.176	0.170	0.001	0.001
aspnetboilerplate	1.302	0.333	0.219	0.146	0.094	0.045	0.014	0.008	0.007	0.007
orleans	0.816	0.640	0.551	0.503	0.496	0.196	0.159	0.152	0.142	0.142
cli	1.676	0.233	0.159	0.118	0.102	0.062	0.058	0.029	0.026	0.026
cake	2.335	0.753	0.614	0.337	0.075	0.021	0.007	0.004	0.004	0.004
Avalonia	0.846	0.117	0.098	0.018	0.013	0.002	0.001	0.001	0.001	0.001
EntityFrameworkCore	3.377	1.691	1.608	1.584	1.576	1.310	0.001	0.001	0.001	0.001
jellyfin	0.132	0.006	0.003	0.002	0.002	0.000	0.000	0.000	0.000	0.000
PowerShell	1.732	1.299	0.158	0.053	0.007	0.001	0.000	0.000	0.000	0.000
WeiXinMPSDK	3.295	0.334	0.188	0.061	0.017	0.006	0.003	0.001	0.000	0.000
ArchiSteamFarm	0.897	0.479	0.429	0.423	0.412	0.403	0.339	0.009	0.001	0.000
VisualStudio	1.281	0.090	0.053	0.028	0.020	0.013	0.006	0.001	0.001	0.001
CppSharp	99.528	1.020	0.992	0.980	0.972	0.927	0.078	0.075	0.073	0.072

We calculate the ratio number between the co-changing pairs and the structural dependencies because we want to evaluate the size of the extracted co-changing pairs compared to the size of the structural dependencies from the system. According to surveys [50], [62], the main reason why logical dependencies (a.k.a filtered co-changes) are not used together with structural dependencies is because of their size. So, it is important to us to get at each filtering step an overview regarding the ratio between co-changes size and structural dependencies size.

From the results presented in tables 3.5 and 3.6 we conclude that the number of co-changing pairs is drastically reduced. In most cases, the number of structural dependencies surpasses the number of co-changing pairs that remain after filtering. But, we do the filtering not only to reduce the size of the co-changing pairs extracted. We do the filtering of co-changing pairs extracted to make sure that the remaining co-changing pairs are indeed logically dependent.

If we filter out all the co-changing pairs that do not update at least half of the time together (factor A and factor B ≥ 50%) we remain with a decent quantity of co-changing pairs. Given the size of the output and the connection strength of the co-changing pairs, the remaining co-changing pairs can be considered, at this point, to be logically dependent.

Table 3.6: Ratio of number of filtered co-changing pairs to number of SD, when factor A or factor B $\geq threshold$%

Project	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 100%
bluecove	1.312	1.181	0.700	0.599	0.419	0.235	0.219	0.046	0.045	0.045
aima-java	0.430	0.280	0.176	0.118	0.103	0.056	0.022	0.020	0.020	0.020
powermock	0.508	0.328	0.234	0.179	0.150	0.092	0.091	0.091	0.091	0.091
restfb	0.662	0.336	0.122	0.067	0.059	0.016	0.015	0.015	0.015	0.015
rxjava	0.279	0.206	0.145	0.100	0.099	0.047	0.044	0.039	0.034	0.034
metro-jax-ws	0.271	0.261	0.204	0.172	0.160	0.106	0.082	0.081	0.080	0.080
mockito	2.481	1.521	0.904	0.623	0.411	0.199	0.128	0.107	0.101	0.101
grizzly	1.332	0.838	0.515	0.320	0.288	0.142	0.117	0.106	0.090	0.076
shipkit	1.376	1.083	0.725	0.515	0.424	0.191	0.149	0.105	0.094	0.094
OpenClinica	0.830	0.434	0.314	0.256	0.217	0.130	0.093	0.082	0.080	0.072
robolectric	0.366	0.122	0.088	0.046	0.031	0.027	0.003	0.002	0.002	0.002
aeron	0.781	0.449	0.265	0.190	0.160	0.096	0.062	0.031	0.021	0.018
antlr4	11.363	0.798	0.055	0.022	0.011	0.007	0.002	0.002	0.002	0.002
mcidasv	1.932	1.203	0.858	0.682	0.579	0.473	0.396	0.365	0.013	0.013
ShareX	2.681	1.292	0.916	0.730	0.593	0.287	0.210	0.201	0.017	0.017
aspnetboilerplate	1.055	0.759	0.493	0.364	0.273	0.130	0.067	0.050	0.046	0.046
orleans	1.120	0.962	0.849	0.750	0.744	0.559	0.482	0.476	0.466	0.466
cli	1.676	0.762	0.560	0.434	0.375	0.269	0.237	0.149	0.142	0.142
cake	1.883	1.197	1.001	0.541	0.185	0.103	0.019	0.013	0.013	0.013
Avalonia	0.510	0.224	0.138	0.037	0.028	0.011	0.006	0.003	0.003	0.003
EntityFrameworkCore	2.636	1.888	1.695	1.623	1.608	1.317	0.006	0.006	0.006	0.006
jellyfin	0.132	0.030	0.016	0.011	0.008	0.003	0.002	0.002	0.002	0.002
PowerShell	3.454	1.648	0.232	0.081	0.021	0.004	0.003	0.003	0.003	0.003
WeiXinMPSDK	1.342	0.603	0.327	0.144	0.080	0.047	0.015	0.008	0.007	0.007
ArchiSteamFarm	5.472	1.416	0.830	0.677	0.575	0.450	0.353	0.023	0.016	0.014
VisualStudio	1.281	0.236	0.142	0.092	0.060	0.040	0.031	0.020	0.019	0.019
CppSharp	55.038	1.343	1.106	1.044	1.030	0.983	0.449	0.443	0.441	0.439

3.6. Overlaps between structural and logical dependencies

A logical dependency can be also a structural dependency and vice-versa, so studying the overlapping between logical and structural dependencies while filtering is important since the intention is to introduce those logical dependencies among with structural dependencies in architectural reconstruction systems. Current studies have shown a relatively small percentage of overlapping between them with and without any kind of filtering [51]. This means that a lot of non related entities update together in the versioning system, the goal here is to establish the factors that determine such a small percentage of overlapping [104].

Since we are first extracting co-changing pairs and only after various filters we call the remaining co-changing pairs logically dependent, we will be studying the overlapping between the remaining co-changing pairs after each filtering stage and the structural dependencies. For each system, we extracted the structural dependencies and the co-changing pairs and determined the overlap between the two dependencies sets, in various experimental conditions.

One variable experimental condition is whether changes located in comments contribute towards logical dependencies. This condition distinguishes between two different cases:

- with comments: a change in source code files is counted as a co-changing pair,

even if the change is inside comments in all files
- without comments: commits that changed source code files only by editing comments are ignored

In all cases, we varied the following threshold values:

- commit size (cs): the maximum size of commit transactions which are accepted to generate co-changes. The values for this threshold were 5, 10, 20 and no threshold (infinity).
- number of occurrences (occ): the minimum number of repeated occurrences for a co-change to be counted as logical dependency. The values for this threshold were 1, 2, 3 and 4.

The six tables below present the synthesis of our experiments. We have computed the following values:

- the mean ratio of the number of co-changes to the number of structural dependencies (SD)
- the mean percentage of structural dependencies that are also co-changes (calculated from the number of overlaps divided to the number of structural dependencies)
- the mean percentage of co-changes that are also structural dependencies (calculated from the number of overlaps divided to the number of co-changes)

In all the six tables, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12 we have on columns the values used for the commit size cs, while on rows we have the values for the number of occurrences threshold occ. The tables contain median values obtained for experiments done under all combinations of the two threshold values, on all test systems. In all tables, the upper right corner corresponds to the most relaxed filtering conditions, while the lower left corner corresponds to the most restrictive filtering conditions.

Table 3.7: Ratio of number of co-changes to number of SD, case with comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	3,39	5,67	9,00	80,31
$occ \geq 2$	2,24	3,47	5,02	60,14
$occ \geq 3$	1,04	2,53	3,52	44,68
$occ \geq 4$	0,90	2,16	2,88	33,47

Table 3.8: Ratio of number of co-changes to number of SD, case without comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	3,24	5,33	7,90	67,16
$occ \geq 2$	1,35	3,27	4,72	47,39
$occ \geq 3$	1,00	1,67	2,49	32,39
$occ \geq 4$	0,43	1,26	1,93	22,15

In order to assess the influence of comments, we compare pairwise Tables 3.7 and 3.8, Tables 3.9 and 3.10 and Tables 3.11 and 3.12. We observe that, although there are some differences between pairs of measurements done in similar conditions with and without comments, the differences are not significant.

Table 3.9: Percentage of SD that are also co-changes, case with comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	19,75	29,86	39,29	76,59
$occ \geq 2$	12,50	20,20	27,68	66,11
$occ \geq 3$	8,49	14,22	19,94	55,99
$occ \geq 4$	6,58	10,95	15,76	47,12

Table 3.10: Percentage of SD that are also co-changes, case without comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	18,88	28,47	37,44	71,12
$occ \geq 2$	11,87	19,03	25,93	59,58
$occ \geq 3$	8,00	13,09	18,15	48,65
$occ \geq 4$	5,85	9,94	14,27	39,07

Table 3.11: Percentage of co-changes that are also SD, case with comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	12,02	8,86	6,72	1,79
$occ \geq 2$	15,05	11,71	9,38	2,21
$occ \geq 3$	17,45	13,97	11,57	2,86
$occ \geq 4$	18,96	15,28	12,94	3,67

Table 3.12: Percentage of co-changes that are also SD, case without comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	12,05	9,02	6,98	1,93
$occ \geq 2$	15,08	12,03	9,66	2,42
$occ \geq 3$	17,78	14,37	12,24	3,28
$occ \geq 4$	19,22	15,59	13,30	4,21

Table 3.13: Percentage of SD that are also co-changing pairs after connection strength filtering.

Condition	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
factor A and factor B	11.20	6.80	4.44	3.25	2.58	1.74	1.16	0.57	0.35	0.33
factor A or factor B	15.94	11.02	7.56	5.59	4.52	2.90	2.00	1.33	1.04	1.02

Table 3.14: Percentage of co-changing pairs that are SD after connection strength filtering.

Condition	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
factor A and factor B	10.95	20.61	23.73	26.75	28.57	33.31	33.43	38.34	42.52	39.41
factor A or factor B	12.19	16.85	19.41	20.70	21.63	22.84	21.86	23.08	24.00	22.73

On the other hand, the overlap between structural and co-changes is given by the number of pairs of classes that have both structural and co-change dependencies. We evaluate this overlap as a percentage relative to the number of structural dependencies in Tables 3.9,3.10 and 3.13, respectively as a percentage relative to the number of co-changes in Tables 3.11,3.12, 3.14.

A first observation from Tables 3.9, 3.10, and 3.13 is that not all pairs of classes with structural dependencies co-change. The biggest value for the percentage of structural dependencies that are also co-changes is 76.5% obtained in the case when no filterings are done.

From Tables 3.11, 3.12, and 3.14 we notice that the percentage of co-changes which are also structural is always low to very low. This means that most co-changes are recorded between classes that have no structural dependencies to each other [104].

4. COMBINING STRUCTURAL AND LOGICAL DEPENDENCIES

Software clustering relies on various dependencies to identify relationships between software entities. Structural dependencies have been mostly used due to their reliability [54]. However, recent research has started incorporating other types of dependencies besides structural dependencies [6], [105], [106]. This section will present an overview of structural and logical dependencies, focusing on how they are extracted.

4.1. Structural Dependencies Weights

Structural dependencies are important for understanding the architecture of a software system because they reveal how different modules interact at the code level. In our research, we extract structural dependencies using a tool from our previous work [57]. This tool analyzes the source code to identify various relationships between software entities and exports them in CSV format.

Structural dependencies do not all have the same level of influence on a software system's architecture and behavior. For instance, the relationship between a variable and the class that uses it is not the same as the relationship between a class and the interface it implements. To reflect these differences, we assign different weights to each type of dependency.

The dependency types and weights were previously defined in related works on clustering [53], [1].

Table 4.1 shows the weights assigned to different categories of structural dependencies, as proposed in previous works.

Weight	Dependency types
4	Interface realization
3	Inheritance, parameter, return type, field, cast, type binding
2	Method call, field access, instantiation
1	Local variable

Table 4.1: Weights assigned to different structural dependency types. [1]

The weights are assigned based on the following considerations:

Weight 4 – Interface Realization: Assigned the highest weight because it signifies a strong architectural relationship. Implementing an interface means classes are expected to provide specific functionalities.

Weight 3 – Inheritance, Parameter, Return Type, Field, Cast, Type Binding: These dependencies represent significant connections between entities. They include

inheritance relationships and shared data or types, which affect the behavior and properties of entities.

Weight 2 – Method Call, Field Access, Instantiation: These indicate interactions between classes but are less impactful than higher weights. They involve using methods or fields of other classes or creating instances. When a method call, field access, or instantiation occurs multiple times between the same pair of entities, the weight is multiplied by the number of occurrences. For example, if Class A calls a method in Class B three times, the assigned weight would be 6 (weight 2 multiplied by 3).

Weight 1 – Local Variable: Given the lowest weight, local variables are the most basic level of interaction.

4.2. Logical Dependencies Weights

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 directly or indirectly related or dependent on each other.

Co-changes are associated with a degree of uncertainty. Compared to structural dependencies, where 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 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, a developer can still resolve multiple unrelated issues in the same commit (even though development processes do not recommend this).

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 [57], [103], [104]. 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 result is the set of logical dependencies that we use to generate software clusters.

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 that 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, generating numerous irrelevant co-changes for the system.

A similar scenario occurs with merge commits. A merge commit is automatically created when developers 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.

Both scenarios above have in common 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 set a threshold of 20 files [57], [**?**]. Therefore, all co-changes originating from commits with more than 20 changed code files are filtered out.

Strength Filter

This filter focuses on the reliability of the co-changes. If a pair of co-changing entities appears only once in the system's history, 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 [22].

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 (4.1), 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).

$$\text{Confidence}(A \rightarrow B) = \frac{\text{Nr. of commits containing } A \text{ and } B}{\text{Nr. of commits containing } A} \tag{4.1}$$

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

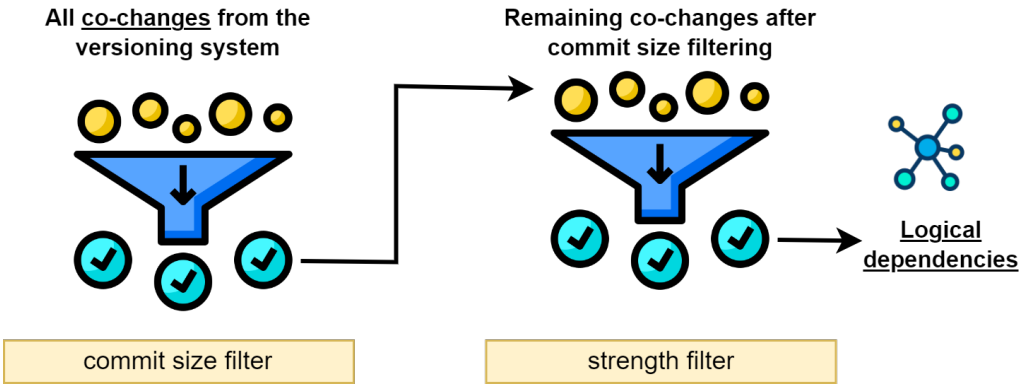

Figure 4.1: Filter application process

pair $(C \rightarrow D)$, even though the second pair has more than five times more updates together.

To favor entities 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 by 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 the strength metric, and it is defined in Equation (4.2).

$$\text{strength}(A \rightarrow B) = \text{confidence}(A \rightarrow B) \times 100 \times \text{system factor} \tag{4.2}$$

Filter Application Process

Fig. 4.1 illustrates the overall filter application process. 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 20 files, meaning that any co-changes from commits involving more than 20 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, precisely 10 different thresholds. We start with a threshold of 10 and increment it by 10 until we reach a maximum value of 100. We do not use a fixed threshold to assess how different strength thresholds affect our cluster generation.

Dependency Extraction and Filtering Tool

To extract and filter the co-changes, we used a previously developed tool [57]. This tool takes the GitHub repository address and the threshold values for commit and strength filters as input. 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 evaluates the rule's antecedent. Thus, the metric value for $(A \rightarrow B)$ differs from the metric value for $(B \rightarrow A)$.

After applying the filters, the remaining dependencies are exported to a CSV file for further use.

It is important to note that the strength metric is only used for filtering and is *not considered as a weight* of the dependencies. The *weight assigned to each dependency is the number of commits in which both entities were updated together*.

4.3. Combining Structural and Logical Dependencies

When structural dependencies (SD) and logical dependencies (LD) are combined in software clustering, both types of relationships are represented within the same graph.

Each entity in the system is represented as a node in the graph, and the dependencies between them are represented as directed weighted edges.

SD and LD weights are combined when the same pair of entities appear in both dependencies. In this case, the weights from SD and LD are summed, giving more influence to those entity pairs. When a pair of entities appear only in SD or only in LD, the edge is added to the graph together with its corresponding weight.

Figure 4.2 illustrates combining structural and logical dependencies in the same dependency graph. The structural dependencies between `House`, `OrangeCat`, and `CatBehavior` entities are visible from the source code analysis.

However, the combination of SD and LD reveals additional insights. One important observation is the logical dependency between `House` and `OrangeCat`, which is not observed from the structural analysis. This relation is extracted from version control and filtered using a 60% strength filter. The strength metric reveals that `House` and `OrangeCat` have a significant co-change value of 75.0, usually associated with a strong relationship.

When SD and LD overlap, such as between `OrangeCat` and `CatBehavior`, their weights are summed. This summation increases the weight of the dependency, making

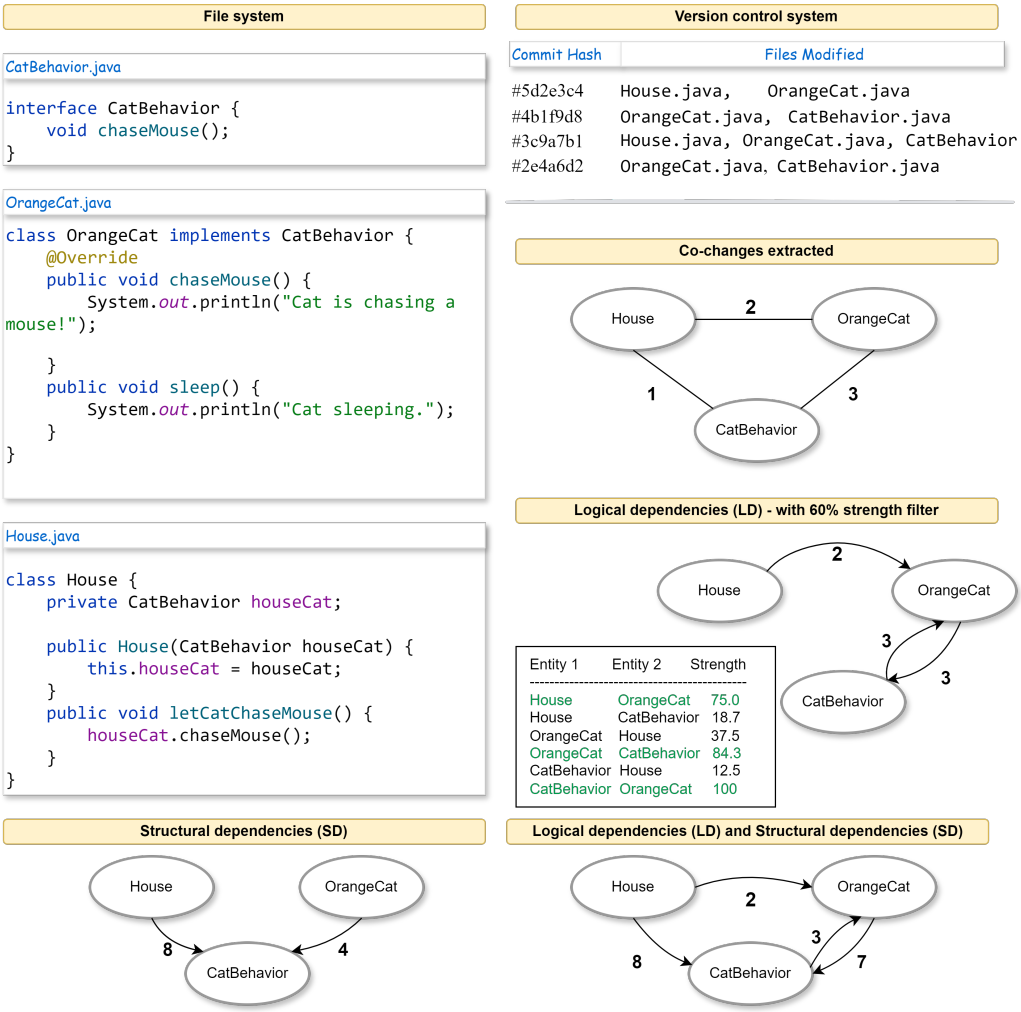

Figure 4.2: Dependency Graph: Combining structural and logical dependencies.

it more important in the dependency graph.

New structure:

```
Chapter 3: Research methodology
    3.1 Overview of research approach
    3.2 Data collection
        3.2.1 Extracting structural dependencies
        3.2.2 Extracting logical dependencies (co-changing pairs)
        3.2.3 Description of data sets used
    3.3 Tool development
        3.3.1 Tool for measuring software dependencies
```

5. LOGICAL DEPENDENCIES IN KEY CLASS DETECTION

5.1. Introduction

Zaidman et al [81] were the first to introduce the concept of key classes and it refers to classes that can be found in documents written to provide an architectural overview of the system or an introduction to the system structure. Tahvildari and Kontogiannis have a more detailed definition regarding key classes concept: "Usually, the most important concepts of a system are implemented by very few key classes which can be characterized by the specific properties. These classes, which we refer to as key classes, manage many other classes or use them in order to implement their functionality. The key classes are tightly coupled with other parts of the system. Additionally, they tend to be rather complex, since they implement much of the legacy system's functionality" [82]. Also, other researchers use a similar concept as the one defined by Zaidman but under different terms like important classes [107] or central software classes [108].

The key class identification can be done by using different algorithms with different inputs. In the research of Osman et al., the key class identification is made by using a machine learning algorithm and class diagrams as input for the algorithm [86]. Thung et al. builds on top of Osman et al.'s approach and adds network metrics and optimistic classification in order to detect key classes [85].

Zaidman et al. use a webmining algorithm and dynamic analysis of the source code to identify the key classes [81].

Sora et al. use a page ranking algorithm for finding key classes and static analysis of the source code [55], [83], [84]. In [1] the authors use in addition to the previous research also other class attributes to identify important classes. The page ranking algorithm is a customization of PageRank, the algorithm used to rank web pages [109]. The PageRank algorithm works based on a recommendation system. If one node has a connection with another node, then it recommends the second node. In previous works, connections are established based on structural dependencies extracted from static code analysis. If A has a structural dependency with B, then A recommends B, and also B recommends A.

The ranking algorithm ranks all the classes from the source code of the system analyzed according to their importance. To identify the important classes from the rest of the classes a threshold for TOP classes from the top of the ranking is set. The TOP threshold value can go from 1 to the total number of classes found in the system.

Some researchers [81], [110], [111] consider that 15% of the total number of classes of the system is a suited value for the TOP threshold. Other researchers [1] consider that 15% of the total number of classes is a too high value for the TOP threshold and suggest that a value in the range of 20–30 is better.

5.2. Metrics for results evaluation

To evaluate the quality of the key classes ranking algorithm and solution produced, the key classes found by the algorithm are compared with a reference solution.

The reference solution is extracted from the developer documentation. Classes mentioned in the documentation are considered key classes and form the reference solution (ground truth) used for validation [112].

For the comparison between both solutions, is used a classification model. The quality of the solution produced is evaluated by using metrics that evaluate the performance of the classification model, such as Precision-Recall and Receiver Operating Characteristic Area Under Curve (ROC-AUC).

A classification model (or ”classifier”) is a mapping between expected results and predicted results [113], [114]. Both results can be labeled as positive or negative, which leads us to the confusion matrix from figure 5.1. The confusion matrix has the

Expected Result \ Predicted Result	Positive	Negative
Positive	***T**rue* ***P**ositive*	***F**alse* ***P**ositive*
Negative	***F**alse* ***N**egative*	***T**rue* ***N**egative*

Figure 5.1: Confusion matrix

following outcomes:

- *true positive*, if the expected result is positive and the predicted result is also positive.
- *false positive*, if the expected result is positive but the predicted result is negative.
- *false negative*, if the expected result is negative but the predicted result is positive.
- *true negative*, if the expected result is negative and the predicted result is also negative.

Precision-recall

Precision is the ratio of True Positives to all the positives of the result set.

$$precision = \frac{TP}{TP + FN} \tag{5.1}$$

The recall is the ratio of True Positives to all the positives of the reference set.

$$recall = \frac{TP}{TP + FP} \tag{5.2}$$

As mentioned in section 5.1, to distinguish the key classes from the rest of the classes a TOP threshold is used. Some researchers consider that 15% of the total classes is the best value for the TOP threshold and others consider that the value should be in the range of 20-30.

The precision-recall metric is suited if the threshold value is fixed. If the threshold value is variable, then metrics that capture the behavior over all possible values must be used. Such metric is the Receiver Operating Characteristic metric.

Receiver Operating Characteristic Area Under Curve

The ROC graph is a two-dimensional graph that has on the X-axis plotted the false positive rate and on the Y-axis the true positive rate. By plotting the true positive rate and the false positive rate at thresholds that vary between a minimum and a maximum possible value we obtain the ROC curve. The area under the ROC curve is called Area Under the Curve (AUC).

The true positive rate of a classifier is calculated as the division between the number of true positive results identified and all the positive results identified:

$$True\ positive\ rate(TPR) = \frac{TP}{TP + FN} \tag{5.3}$$

The false positive rate of a classifier is calculated as the division between the number of false positive results identified and all the negative results identified:

$$False\ positive\ rate(FPR) = \frac{FP}{FP + TN} \tag{5.4}$$

In multiple related works, the ROC-AUC metric has been used to evaluate the results for finding key classes of software systems. For a classifier to be considered good, its ROC-AUC metric value should be as close to 1 as possible, when the value is 1 then the classifier is considered to be perfect.

Osman et al. obtained in their research an average Area Under the Receiver Operating Characteristic Curve (ROC-AUC) score of 0.750 [86]. Thung et al. obtained an average ROC-AUC score of 0.825 [85] and Sora et al. obtained an average ROC-AUC score of 0.894 [1].

5.3. Data set used

In this section, we will look over all the systems studied in the baseline research presented in section 5.4.1, and we will try to identify the systems that could be used also in our current research involving logical dependencies.

The research of I. Sora et al [1] takes into consideration structural public dependencies that are extracted using static analysis techniques and was performed on the object-oriented systems presented in table 5.1.

The requirements for a system to qualify as suited for investigations using logical dependencies are: has to be on GitHub, has to have release tags to identify the version, and also has to have an increased number of commits. From the total of 14 object-oriented systems listed in the paper [1], 13 of them have repositories in Github 5.2. And from the found repositories we identified only 6 repositories that have the same release tag as the specified version from table 5.1. It is important to identify the correct release tag for each repository to limit the commits further analyzed by date. Only commits that were made until the specified release are considered and analyzed. The commits number found on the remaining 6 repositories varies from 19108 commits for Tomcat Catalina to 149 commits for JHotDraw. In order to have more accurate results, we need a significant number of commits, so we reached the conclusion that only 3 systems can be used for key classes detection using logical dependencies: Apache Ant, Hibernate, and Tomcat Catalina. From all the systems mentioned in table 5.1 Apache Ant is the most used and analyzed in other works [104], [115], [116], [117].

Table 5.1: Analyzed software systems in previous research paper.

ID	System	Description	Version
Sl	Apache Ant	Java library and command line tool that drive the build processes as targets and extension points depending upon each other	1.6.1
S2	Argo UML	UML modelling tool with support for all UML diagrams.	0.9.5
S3	GWT Portlets	Open source web framework for building GWT (Google Web Toolkit) Applications.	0.9.5 beta
S4	Hibernate	Persistence framework for Java.	5.2.12
S5	javaclient	Java distributed application for playing with robots	2.0.0
S6	jEdit	Java mature text editor for programmers.	5.1.0
S7	JGAP	Genetic Algorithms and Genetic Programming Java library.	3.6.3
S8	JHotDraw	JHotDraw is a two-dimensional graphics framework for structured drawing editors that is written in Java.	6.0b.1
S9	JMeter	JMeter is a Java application designed to load test functional behavior and measure performance	2.0.1
S10	Log4j	Logging Service	2.10.0
S11	Mars	The Mars Simulation Project is a Java project that models and simulates human settlements on Mars planet	3.06.0
S12	Maze	The Maze-solver project simulates an artificial intelligence algorithm on a maze	1.0.0
S13	Neuroph	Neuroph is a Java neural network framework.	2.2.0
S14	Tomcat Catalina	The Apache Tomcat project is an open-source implementation of JavaServlet and JavaServerPages technologies	9.0.4
S15	Wro4J	The Wro4J is a web resource (JS and CSS) optimizer for Java.	1.6.3

Table 5.2: Found systems and versions of the systems in GitHub.

ID	System	Version	Release Tag name	Commits number
Sl	Apache Ant	1.6.1	rel/1.6.1	6713
S2	Argo UML	0.9.5	not found	0
S3	GWT Portlets	0.9.5 beta	not found	0
S4	Hibernate	5.2.12	5.2.12	6733
S5	javaclient	2.0.0	not found	0
S6	jEdit	5.1.0	not found	0
S7	JGAP	3.6.3	not found	0
S8	JHotDraw	6.0b.1	not found	149
S9	JMeter	2.0.1	v2_1_1	2506
S10	Log4j	2.10.0	v1_2_10-recalled	634
S11	Mars	3.06.0	not found	0
S12	Maze	1.0.0	not found	0
S13	Neuroph	2.2.0	not found	0
S14	Tomcat Catalina	9.0.4	9.0.4	19108
S15	Wro4J	1.6.3	v1.6.3	2871

5.4. Measurements using logical dependencies

As we mentioned in the beginning the purpose is to check if the logical dependencies can improve key class detection.

As presented in section 5.4.1, and section 5.1 the key class detection was done by using structural dependencies of the system. In this section, we will use the same tool used in the baseline approach presented in section 5.4.1, and we will add a new input to it, the logical dependencies.

Below is a comparison between the new approach and baseline approach, how we collect the logical dependencies, the results obtained previously, and the new results obtained. The new results are separated into two categories, the results obtained by using structural and logical dependencies and the results obtained by using only logical dependencies.

5.4.1. Baseline approach

We use the research of I. Sora et al [1] as a baseline for our research involving the usage of logical dependencies to find key classes. The baseline approach uses a tool that takes as an input the source code of the system and applies ranking strategies to rank the classes according to their importance.

In order to rank the classes according to their importance, different class metrics are used [110], [81], [111]. Below are presented some of the class metrics used in the baseline approach in order to rank the classes according to their importance.

Class attributes that characterize key classes

The metrics used in the baseline research can be grouped into the following categories:

- class size metrics: number of fields (NoF), number of methods (NoM), global size (Size = NoF+NoM).
- class connection metrics, any structural dependency between two classes:
  - CONN-IN, the number of distinct classes that use a class;
  - CONN-OUT, the total number of distinct classes that are used by a class;
  - CONN-TOTAL, the total number of distinct classes that a class uses or are used by a class (CONN-IN + CONN-OUT).
  - CONN-IN-W, the total weight of distinct classes that use a class.
  - CONN-OUT-W, the total weight of distinct classes that are used by a class.
  - CONN-TOTAL-W, the total weight of all connections of the class (CONN-IN-W + CONN-OUT-W) [1].
- class pagerank values, previous research use pagerank values computed on both directed and undirected, weighted and unweighted graphs:
  - PR - value computed on the directed and unweighted graph;
  - PR-W - value computed on the directed and weighted graph;
  - PR-U - value computed on the undirected and unweighted graph;
  - PR-U-W - value computed on the undirected and weighted graph;
  - PR-U2-W - value computed on the weighted graph with back-recommendations [55], [83], [1], [84].

Based on the class attributes presented, all the classes of the system are ranked. To differentiate the important (key) classes from the rest of the classes, a TOP threshold for the top classes found is set. The threshold vary between 20 and 30 classes.

The baseline approach not only identifies the key classes but also evaluates the performance of the solution produced. The same approach as the one presented in section 5.2 is used for the evaluation of the results. The key classes found by the ranking algorithm are compared with a reference solution that is extracted from the developer documentation by using a classification model.

The true positives (TP) are the classes found in the reference solution and also in the top TOP ranked classes. False positives (FP) are the classes that are not in the reference solution but are in the TOP ranked classes. True Negatives (TN) are classes that are found neither in the reference solution nor in the TOP ranked classes. False Negatives (FN) are classes that are found in the reference solution but not found in the TOP ranked classes.

Due to the fact that the TOP threshold is varied, the Receiver Operating Characteristic Area Under Curve metric is used for the evaluation of the results.

The entire workflow of the baseline approach that was presented above is also presented in figure 5.2.

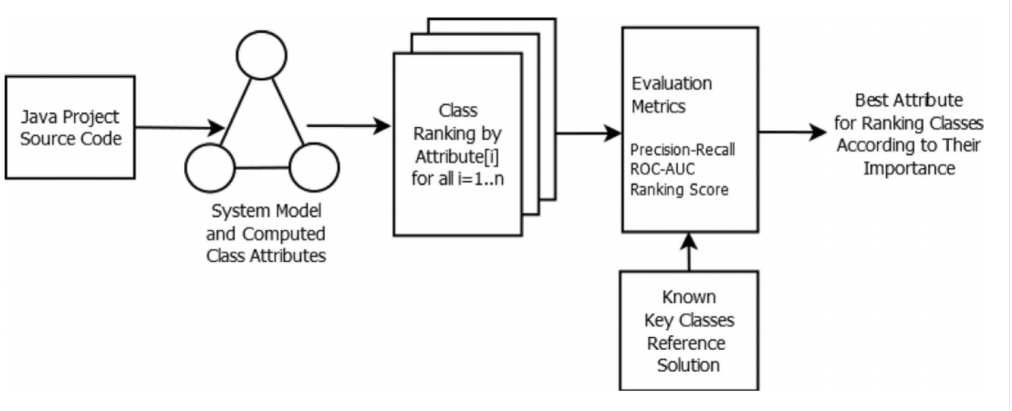

Figure 5.2: Overview of the baseline approach. Reprinted from "Finding key classes in object-oriented software systems by techniques based on static analysis." by Ioana Sora and Ciprian-Bogdan Chirila, 2019, Information and Software Technology, 116:106176. Reprinted with permission.

5.4.2. Comparison with the baseline approach

The baseline approach uses a tool that takes as input the source code of the system to identify the key classes and the reference solution to evaluate the quality of the solution. We modified the tool such that it can also take as input the logical dependencies.

In order to rank the classes according to their importance, the tool uses different class metrics. The list of the metrics used in the baseline approach is presented in section 5.4.1. The difference in the metrics used compared with the baseline approach is that we use a subset of those metrics. The reason why we are not using all the metrics is that the extracted logical dependencies are undirected. The metrics used by the current approach are CONN-TOTAL, CONN-TOTAL-W, PR-U, PR-U-W, and PR-U2-W.

We did not change the rest of the workflow of the tool. Meaning that the TOP threshold is varied between 20 and 30 and the resulting solution is evaluated by using the ROC-AUC metric. The goal being a ROC-AUC (Receiver Operating Characteristic - Area Under the Curve) metric value as close to 1 as possible.

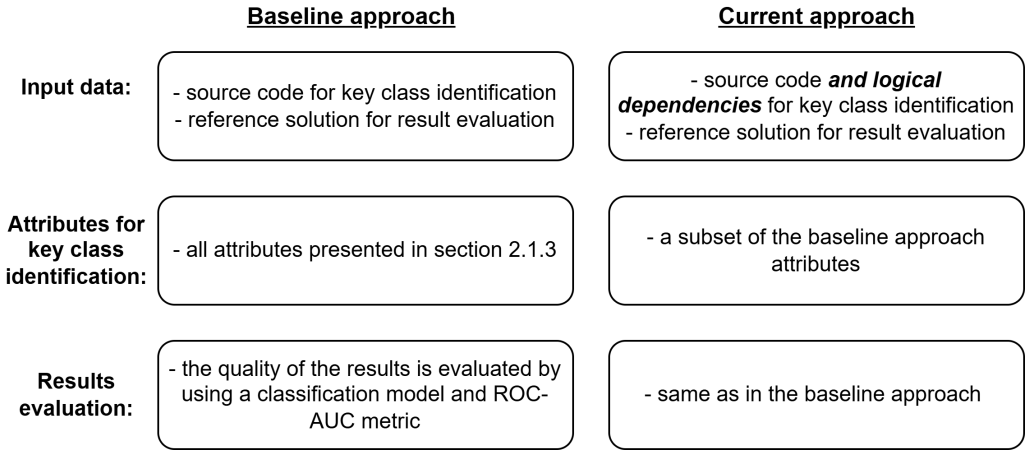

Figure 5.3: Comparison between the new approach and the baseline

5.4.3. Logical dependencies collection and current workflow used

The logical dependencies are those co-changing pairs extracted from the versioning system history that remain after filtering. The filtering part consists of applying two filters: the filter based on commit size and the filter based on connection strength.

To determine the connection strength of a pair, we first need to calculate the connection factors for both entities that form a co-changing pair. Assuming that we have a co-changing pair formed by entities A and B, the connection factor of entity A with entity B is the percentage from the total commits involving A that contains entity B. The connection factor of entity B with entity A is the percentage from the total commits involving B that contain also entity A.

$$connection\ factor\ for\ A = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ A} \tag{5.5}$$

$$connection\ factor\ for\ B = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ B} \tag{5.6}$$

We calculated the connection factor for each entity involved in a co-changing pair and filtered the co-changing pairs based on it. The rule set is that both entities had to have a connection factor with each other greater than the threshold value.

After the filtering part, the remaining co-changing pairs, now called logical dependencies, are exported in CSV files.

The entire process of extracting co-changing pairs from the versioning system, filter them, and export the remaining ones into CSV files is done with a tool written in Python.

The next step is to use the exported logical dependencies for key classes detection. In order to do that we used the same key class detection tool used in the previous research presented in section 5.4.1. We adapted the tool to be able to process also logical dependencies because previously the tool used only structural dependencies extracted from the source code of the software systems. The workflow is presented in figure 5.4

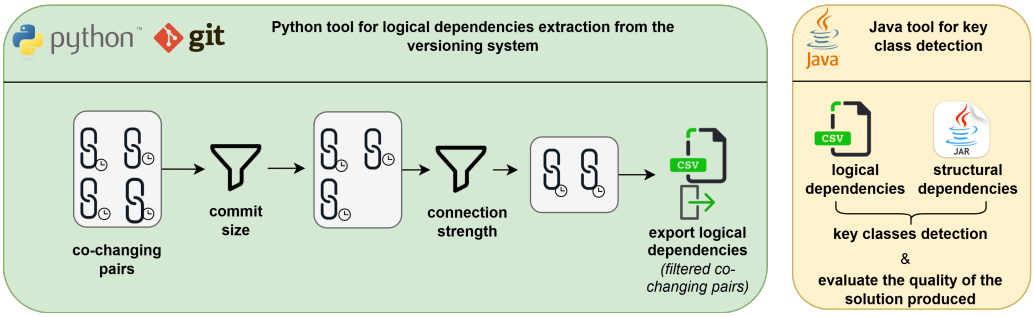

Figure 5.4: Workflow for key classes detection

5.4.4. Measurements using only the baseline approach

In table 5.3 are presented the ROC-AUC values for different attributes computed for the systems Ant, Tomcat Catalina, and Hibernate by using the baseline approach. We intend to compare these values with the new values obtained by using also logical dependencies in key class detection.

Table 5.3: ROC-AUC metric values extracted.

Metrics	Ant	Tomcat Catalina	Hibernate
PR_U2_W	0.95823	0.92341	0.95823
PR	0.94944	0.92670	0.94944
PR_U	0.95060	0.93220	0.95060
CONN_TOTAL_W	0.94437	0.92595	0.94437
CONN_TOTAL	0.94630	0.93903	0.94630

5.4.5. Measurements using combined structural and logical dependencies

The tool used in the baseline approach runs a graph-ranking algorithm. The graph used contains the structural dependencies extracted from static source code analysis. Each edge in the graph represents a dependency, the entities that form a structural dependency are represented as vertices in the graph. As mentioned in section 5.4.2, we modified the tool to read also logical dependencies and add them to the graph. In this section, we add in the graph the logical dependencies together with the structural dependencies.

In tables 5.4, 5.5, and 5.6, on each line, we have the metric that is calculated and on each column, we have the connection strength threshold that was applied to

the logical dependencies used in identifying the key classes. We started with logical dependencies that have a connection strength greater than 10%, which means that in at least 10% of the commits involving A or B, A and B update together. Then we increased the threshold value by 10 until we remained only with entities that update in all the commits together. The last column contains the results obtained previously by the tool by only using structural dependencies.

As for the new results obtained by combining structural and logical dependencies, highlighted with orange are the values that are close to the previously registered values but did not surpass them. Highlighted with green are values that are better than the previously registered values. At this step, we can also observe that for all three systems measured in tables 5.4, 5.5, and 5.6, the best values obtained are for connection strength between 40-70%.

Table 5.4: Measurements for Ant using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.924	0.925	0.926	0.927	0.927	0.927	0.929	0.928	0.928	0.928	0.929
PR	0.914	0.854	0.851	0.866	0.876	0.882	0.887	0.854	0.852	0.852	0.855
PR_U	0.910	0.930	0.933	0.933	0.935	0.934	0.939	0.933	0.933	0.933	0.933
CON_T_W	0.924	0.928	0.931	0.932	0.933	0.934	0.936	0.934	0.934	0.934	0.934
CON_T	0.840	0.886	0.904	0.909	0.915	0.923	0.932	0.935	0.936	0.936	0.942

Table 5.5: Measurements for Tomcat using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.910	0.917	0.923	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.923
PR	0.811	0.800	0.815	0.834	0.847	0.852	0.853	0.858	0.858	0.858	0.927
PR_U	0.910	0.921	0.931	0.933	0.933	0.932	0.933	0.932	0.932	0.932	0.932
CON_T_W	0.914	0.920	0.924	0.926	0.926	0.926	0.926	0.926	0.926	0.926	0.926
CON_T	0.868	0.906	0.930	0.936	0.937	0.938	0.938	0.938	0.938	0.938	0.939

Table 5.6: Measurements for Hibernate using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.954	0.957	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.958
PR	0.929	0.929	0.933	0.939	0.939	0.946	0.947	0.947	0.947	0.947	0.949
PR_U	0.942	0.947	0.948	0.949	0.949	0.950	0.950	0.950	0.950	0.950	0.951
CON_T_W	0.939	0.942	0.943	0.944	0.944	0.945	0.945	0.945	0.945	0.945	0.944
CON_T	0.924	0.933	0.938	0.941	0.941	0.944	0.945	0.945	0.945	0.945	0.946

5.4.6. Measurements using only logical dependencies

In the previous section, we added in the graph based on which the ranking algorithm works the logical and structural dependencies. In the current section, we will add only the logical dependencies to the graph.

In tables 5.7, 5.8, and 5.9, are presented the results obtained by using only logical dependencies to detect key classes. The measurements obtained are not as good as using logical and structural dependencies combined or using only structural dependencies. But, all the values obtained are above 0.5, which means that a good part of the key classes is detected by only using logical dependencies. As mentioned in section 5.2, a classifier is good if it has the ROC-AUC value as close to 1 as possible.

One possible explanation for the less performing results is that the key classes may have a better design than the rest of the classes, which means that are less prone to change. If the key classes are less prone to change, this implies that the number of dependencies extracted from the versioning system can be less than for other classes.

Table 5.7: Measurements for Ant using only logical dependencies

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.720	0.627	0.718	0.703	0.732	0.824	0.852	0.881	0.876	0.876	0.929
PR	0.720	0.627	0.718	0.703	0.732	0.824	0.852	0.881	0.876	0.876	0.855
PR_U	0.720	0.627	0.718	0.703	0.732	0.824	0.852	0.881	0.876	0.876	0.933
CON_T_W	0.722	0.581	0.644	0.676	0.727	0.819	0.842	0.874	0.876	0.876	0.934
CON_T	0.722	0.581	0.644	0.676	0.727	0.819	0.842	0.874	0.876	0.876	0.942

Table 5.8: Measurements for Tomcat using only logical dependencies

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Previous
PR_U2_W	0.672	0.656	0.645	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.923
PR	0.685	0.643	0.642	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.927
PR_U	0.685	0.643	0.644	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.932
CON_T_W	0.694	0.636	0.636	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.926
CON_T	0.654	0.611	0.636	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.939

Table 5.9: Measurements for Hibernate using only logical dependencies

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.657	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.958
PR	0.644	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.949
PR_U	0.644	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.951
CON_T_W	0.649	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.944
CON_T	0.644	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.946

5.5. Correlation between details of the systems and results

In this section, we discuss about the correlation between the details of the systems and the results obtained in section 5.4.

The reason why we are doing this correlation is to find if there are some links between the details of the systems and the results obtained.

The results obtained are presented in figures 5.5 - 5.10. We are using plots to display the results obtained to have a clearer view of how the results fluctuate over different thresholds values.

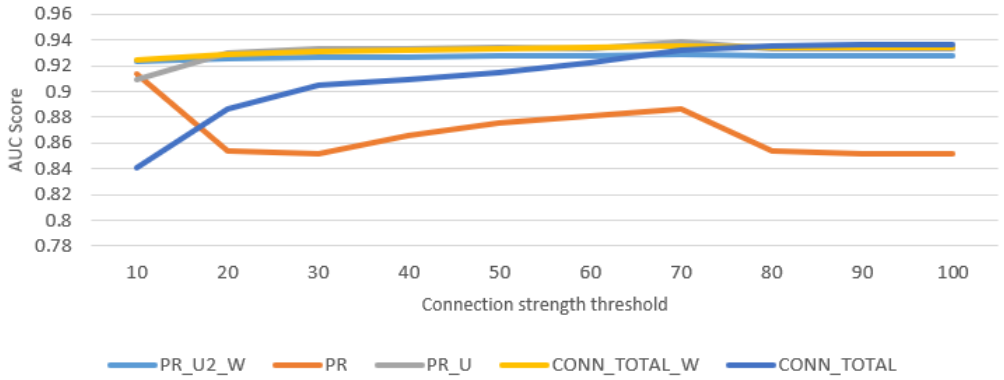

Figure 5.5: Variation of AUC score when varying connection strength threshold for Ant. Results for structural and logical dependencies combined.

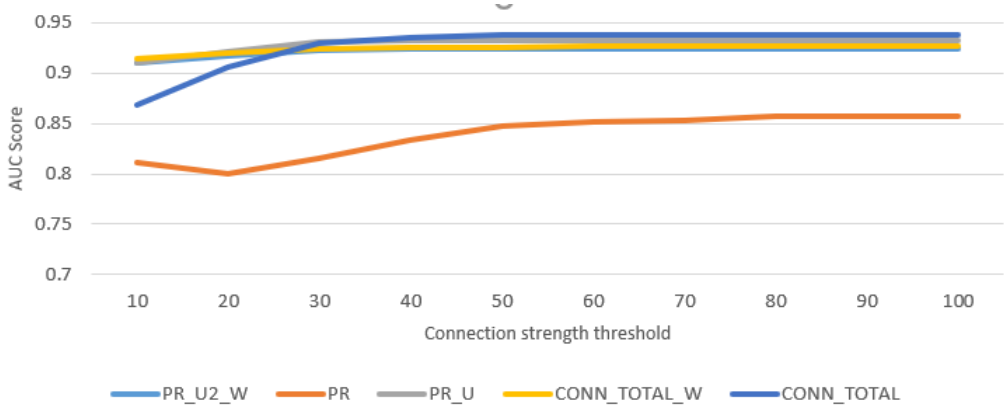

Figure 5.6: Variation of AUC score when varying connection strength threshold for Tomcat. Results for structural and logical dependencies combined.

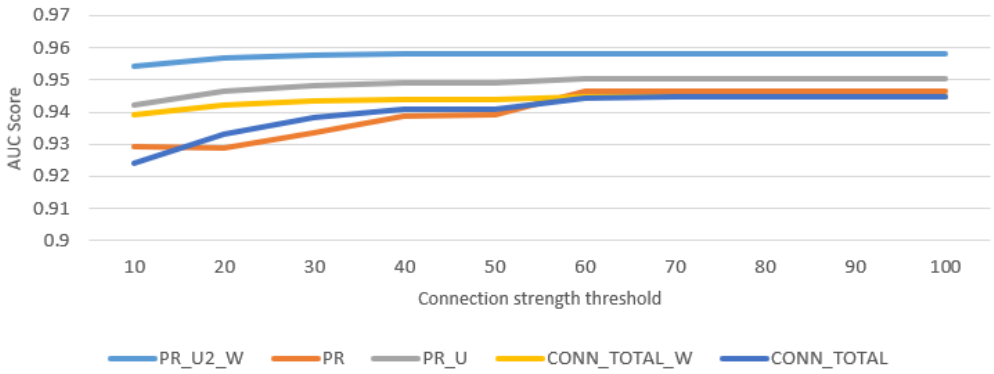

Figure 5.7: Variation of AUC score when varying connection strength threshold for Hibernate. Results for structural and logical dependencies combined.

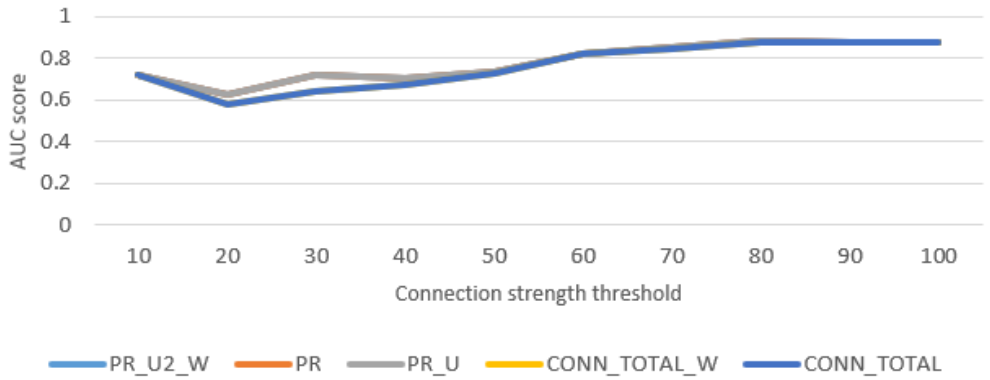

Figure 5.8: Variation of AUC score when varying connection strength threshold for Ant. Results for logical dependencies only.

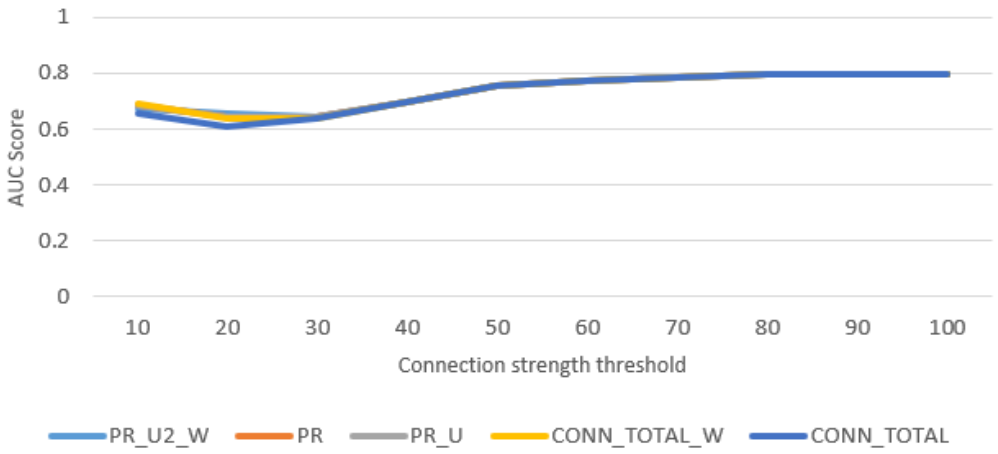

Figure 5.9: Variation of AUC score when varying connection strength threshold for Tomcat. Results for logical dependencies only.

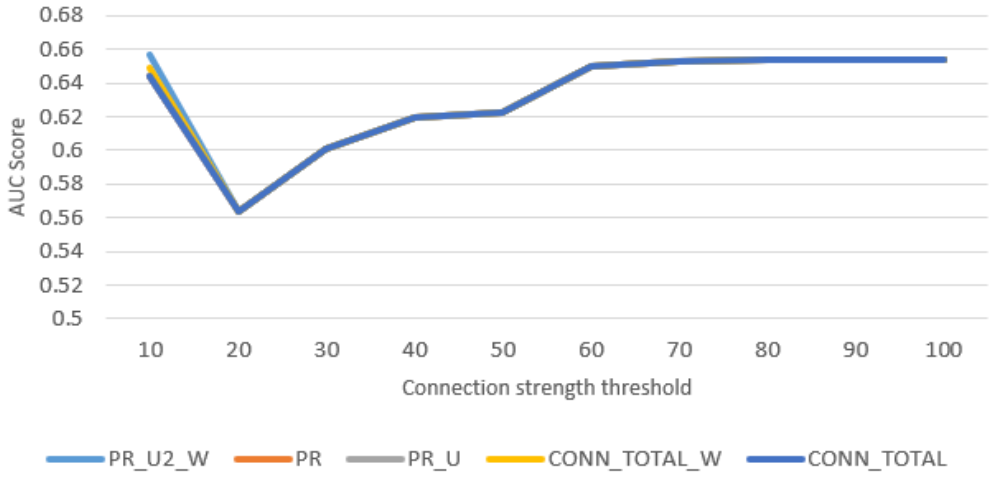

Figure 5.10: Variation of AUC score when varying connection strength threshold for Hibernate. Results for logical dependencies only.

The details of the systems are presented in two tables. In table 5.10 are the overlappings between structural and logical dependencies expressed in percentages. Each column represents the percentage of logical dependencies that are also structural, for each column the logical dependencies are obtained by applying a different connection strength filter. The connection strength filter begins at 10, meaning that in at least 10 % of the total commits involving two entities, the entities update together. We increase the connection strength filter by 10 up until we reach 100, meaning that in all the commits that involve one entity, the other entity is present also.

In table 5.11 are the ratio numbers between structural dependencies and logical dependencies. We added this table in order to highlight how different the total number of both dependencies is.

Table 5.10: Percentage of logical dependencies that are also structural dependencies

System	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
Ant	25.202	34.419	36.385	34.656	33.528	33.333	28.659	33.333	35.294	35.294
Tomcat Catalina	4.059	22.089	25.000	25.758	25.926	37.525	47.368	55.285	75.000	76.923
Hibernate	6.546	26.607	29.565	32.374	32.543	45.170	44.980	42.473	42.473	42.473

Table 5.11: Ratio between structural and logical dependencies (SD/LD)

System	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
Ant	1.315	3.284	4.972	5.603	6.175	10.697	12.915	27.154	41.529	41.529
Tomcat Catalina	0.120	0.923	1.313	1.531	1.619	3.177	7.092	13.146	67.375	124.385
Hibernate	1.037	6.391	10.037	14.947	18.940	54.248	83.442	111.704	111.704	111.704

In figures 5.5, 5.6 and 5.7 are the measurements obtained by using structural and logical dependencies combined. In all three figures, the measurements at the beginning are smaller than the rest. Once with the increasing of the threshold value also the measurements begin to increase. Meaning that better results for key class detection are found. The best measurements are when the threshold value is between 40 and 60, after that, the measurements tend to decrease a little bit and stay at that fixed value.

A possible explanation of the results fluctuation and then capping is that if we are looking at table 5.11 we can see that at the beginning, the total number of logical dependencies used is close to the number of existing structural dependencies. The high volume of logical dependencies introduced might cause an erroneous detection of the key classes, in consequence, smaller measurements. When the threshold begins to be more restrictive and the total number of logical dependencies used begins to decrease, the key classes detection starts to improve. This improvement stops after the threshold value reaches 60%. If we look again at table 5.11 we can see that after 60% the number of structural dependencies outnumbers the number of logical dependencies up to 124 times in some cases. In addition, if we look at table 5.10 we can see that the remaining logical dependencies overlap a lot with the structural dependencies, so we are not introducing too much new information.

So, the number of logical dependencies used is so small that it doesn't influence the key class identification. Since the structural dependencies used don't change, we obtain the same results for different threshold values.

In figures 5.8, 5.9 and 5.10 are the measurements obtained by using only logical dependencies. Initially, we expected to see a Gaussian curve, but instead, we see a bell curve. We think that in the beginning, we use a high number of logical dependencies in key class detection, among those logical dependencies is an important number of key classes and also an important number of other classes. But the number of other classes does not influence the key classes detection. When we start to increase the value of the threshold and filter more the logical dependencies, we also filter some of the initial detected key classes and remain with a significant number of other classes. In this case, the other classes that remain influence the measurements, causing the worst-performing solutions. Some of the key classes are strongly

connected in the versioning system, and even for higher threshold values don't get filtered out. Meanwhile, the rest of the classes that are not key classes get filtered out for higher threshold values which leads to better performing measurements when the threshold value are above 60%.

5.6. Comparison of the extracted data with fan-in and fan-out metric

Fan-in and fan-out are coupling metrics. The fan-in of entity A is the total number of entities that call functions of A. The fan-out of A is the total number of entities called by A [118].

In tables 5.12, 5.13, and 5.14 we can find the metrics detalis for each documented key class of each system. The first column represents the name of each key class, the second column represents the fan_in values for each key class, the third column represents the fan_out values, the fourth column represents the number of entities that call functions of that key class plus the number of entities that are called by the key class (fan_in and fan_out combined), and the fifth column represents the number of logical dependencies in which an entity is involved.

For Ant, we can see in table 5.12 that all the key classes have logical dependencies with other classes. The LD_NUMBER means the number of logical dependencies of an entity. The key classes with the most LD number are Project and IntrospectionHelper, these two entities can be found also in table 5.15 in which we did a top 10 entities that have a logical dependency with other entities. This means that some key classes are involved in software change quite often and can be observed via system history.

Table 5.12: Measurements for Ant key classes

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	Project	191	23	214	157
2	Target	28	6	34	78
3	UnknownElement	17	13	30	90
4	RuntimeConfigurable	17	13	30	118
5	IntrospectionHelper	18	24	42	143
6	Main	1	13	14	82
7	TaskContainer	11	1	12	21
8	ProjectHelper2$ElementHandler	1	12	13	30
9	Task	110	7	117	88
10	ProjectHelper	16	8	24	101

For Tomcat Catalina, same as for Ant, we can see in table 5.13 that all the key classes have logical dependencies. The key classes with the most LD number are StandardContext and Request, these two entities can also be found in table 5.16 in which we did a top 10 entities that have the most logical dependencies with other entities for Tomcat Catalina.

For Hibernate things are a little bit different, as we can see in table 5.14, key classes like Criterion, Projection, or Transaction have 0 logical dependencies, meaning that those key classes are not involved in any software change. One possible expla-

nation for this is that for Hibernate the architecture is designed in such way that the core is not often touched by change.

Table 5.13: Measurements for Tomcat Catalina key classes.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	Context	74	8	82	126
2	Request	48	28	76	215
3	Container	51	8	59	64
4	Response	38	12	50	90
5	StandardContext	11	38	49	216
6	FANector	23	9	32	89
7	Session	29	2	31	28
8	Valve	29	2	31	19
9	Wrapper	29	1	30	36
10	Manager	25	3	28	31
11	Host	26	1	27	44
12	Service	20	6	26	51
13	Engine	23	2	25	1
14	Realm	18	6	24	21
15	CoyoteAdapter	1	22	23	140
16	StandardHost	8	15	23	88
17	LifecycleListener	21	1	22	3
18	StandardEngine	2	19	21	57
19	Pipeline	19	2	21	20
20	Server	16	4	20	49
21	HostConfig	3	15	18	79
22	StandardWrapper	5	13	18	92
23	StandardService	3	12	15	81
24	Catalina	2	13	15	94
25	Loader	14	1	15	18
26	StandardServer	2	12	14	94
27	StandardPipeline	1	10	11	62
28	Bootstrap	3	3	6	41

Table 5.14: Measurements for Hibernate key classes.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	SessionFactoryImplementor	438	43	481	51
2	Type	444	5	449	0
3	Table	89	29	118	82
4	SessionImplementor	52	12	64	14
5	Criteria	45	12	57	15
6	Column	46	10	56	20
7	Session	31	21	52	52
8	Query	12	28	40	0
9	Configuration	1	38	39	115
10	SessionFactory	24	12	36	33
11	Criterion	30	3	33	0
12	Projection	11	3	14	0
13	FANectionProvider	12	2	14	0
14	Transaction	11	1	12	0

In tables 5.15, 5.16, and 5.17 we can find the top 10 entities with logical dependencies. The first column represents the name of each top 10 entity, the second column represents the fan_in values, the third column represents the fan_out values, the fourth column represents the fan_in and fan_out combined, and the fifth column represents the number of logical dependencies in which the entity is involved.

We did these top 10 tables to offer an overview of the highest registered numbers for LD for each system. As we mentioned before, some of the key classes are also present in these tables, but not all of them.

In table 5.17 we can find the top 10 measurements for Hibernate, most of the table is occupied by inner classes of AbstractEntityPersister. This is expected behavior since class AbstractEntityPersister is also present. This behavior is caused by the impossibility to separate the updates done for a class from its inner classes in the versioning system. So, each time AbstractEntityPersister records a change, also the inner classes are considered to have changed.

Table 5.15: Top 10 measurements for Ant.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	Project	191	23	214	157
2	Project$AntRefTable	1	2	3	157
3	Path	39	13	52	147
4	Path$PathElement	3	2	5	147
5	IntrospectionHelper	18	24	42	143
6	IntrospectionHelper$AttributeSetter	8	1	9	143
7	IntrospectionHelper$Creator	3	5	8	143
8	IntrospectionHelper$NestedCreator	7	1	8	143
9	Ant	2	15	17	136
10	Ant$Reference	3	1	4	136

Table 5.16: Top 10 measurements for Tomcat Catalina.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	StandardContext	11	38	49	216
2	StandardContext$ContextFilterMaps	0	0	0	216
3	StandardContext$NoPluggabilityServletContext	0	0	0	216
4	Request	48	28	76	215
5	Request$SpecialAttributeAdapter	0	0	0	215
6	ApplicationContext	3	22	25	158
7	ApplicationContext$DispatchData	0	0	0	158
8	ContextConfig	3	26	29	143
9	ContextConfig$DefaultWebXmlCacheEntry	0	0	0	143
10	ContextConfig$JavaClassCacheEntry	0	0	0	143

Table 5.17: Top 10 measurements for Hibernate.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NR
1	AvailableSettings	1	0	1	205
2	AbstractEntityPersister	9	143	152	190
3	AbstractEntityPersister$CacheEntryHelper	0	0	0	190
4	AbstractEntityPersister$InclusionChecker	0	0	0	190
5	AbstractEntityPersister$NoopCacheEntryHelper	0	0	0	190
6	AbstractEntityPersister$ReferenceCacheEntryHelper	0	0	0	190
7	AbstractEntityPersister$StandardCacheEntryHelper	0	0	0	190
8	AbstractEntityPersister$StructuredCacheEntryHelper	0	0	0	190
9	Dialect	265	104	369	176
10	SessionFactoryImpl$SessionBuilderImpl	1	25	26	167

Overall, by looking at the comparisons between FAN_IN, FAN_OUT, FAN_TOTAL, and the logical dependencies in which a class is involved we could not determine a direct connection between them. Nither we can say that one influences the other.

We consider that even though the metrics are not related directly, they could be all used together to get a better view of the system connections.

6. LOGICAL DEPENDENCIES IN ARCHITECTURAL RECONSTRUCTION

We explore using code co-changes as input for software clustering for architectural reconstruction. Since structural dependencies are the most commonly used dependencies in software clustering, we investigate whether integrating them with code co-changes provides better results than using either dependency type alone.

Our experiments are applied to four open-source Java projects from GitHub. For each project, we apply three distinct clustering algorithms (Louvain, Leiden, and DBSCAN) and evaluate their performance using two clustering evaluation metrics. These metrics allow a comparison between clustering based solely on code co-changes and clustering that integrates both co-changes and structural dependencies, offering a better understanding of how these co-changes influence software architecture reconstruction.

6.1. Introduction

Software systems often need more documentation. Even if there was original documentation at the beginning of development, it may become outdated over the years. 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 evaluate better and understand the nature and impact of changes they must make. One technique to help 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), and others.

This paper assesses the impact of logical dependencies in software clustering alone and combination with structural dependencies. The structural dependencies are used as they are extracted from static code analysis, while the logical dependencies are filtered co-changes obtained from the version control system [119]. The co-changes are filtered to enhance their reliability and remove noise caused by large commits with many files unrelated to development activities (e.g., formatting changes) or rare co-changes that may not indicate a true dependency [**?**].

The following research questions guide our investigation:

- **RQ1:** Does using structural dependencies (SD) combined with logical dependencies (LD) improve software clustering results compared to traditional approaches using only structural dependencies (SD)?
- **RQ2:** Can using only logical dependencies (LD) produce good software clustering results?
- **RQ3:** How do different filtering settings for logical dependencies (LD) impact clustering results, and which filtering settings provide the best performance?

To answer these research questions, we apply three different clustering algorithms (Louvain, Leiden, and DBSCAN) to different open-source projects. We then evaluate the results using two metrics: MQ (Modularization Quality) [120] and MoJoFM (Move and Join e**F**fectiveness **M**easure) [121]. The MoJoFM metric is used for external evaluation, evaluating against the perspective of the system's architect or developers. The MQ metric is used for internal evaluation based on the software structure itself. These two metrics allow us to compare the effectiveness of using structural and logical dependencies alone and combined. This comparison helps clarify how different dependencies and filtering choices affect clustering results.

6.2. Related work

Several studies have explored the use of different types of dependencies in software clustering, applying different algorithms to improve clustering results and using various metrics to evaluate the results obtained.

Tzerpos and Holt developed ACDC (Algorithm for Comprehension-Driven Clustering). This pattern-driven clustering algorithm uses subsystem structures such as source file patterns, directory patterns, system graph patterns, and support library patterns to detect similarities and create clusters [122]. For result evaluation, the authors introduced the MoJo metric, which counts the minimum number of move and join operations required to transform one clustering result into another, assessing how close one clustering solution is to another [123], [124]. Later, Wen and Tzerpos introduced the MoJoFM metric, an enhanced version of the original MoJo distance metric for more effective measurements, as presented in more detail in subsection 6.3.2 [121].

Corazza et al. [6], [8] used lexical dependencies derived from code comments, class names, attribute names, and parameter names, applying Hierarchical Agglomerative Clustering (HAC) to group-related entities. For evaluating the results, the authors used a metric based on the MoJo distance metric and NED (Non-Extremity Cluster Distribution), which measures that the formed clusters are not too large or too small.

Andritsos and Tzerpos [124] used structural dependencies and nonstructural attributes, such as file names and developer names, and proposed the LIMBO algorithm, a hierarchical clustering algorithm for clustering software systems. They used the MoJo distance metric to evaluate the algorithm's output.

Anquetil et al. [105] also used lexical information, including file names, routine names, included files, and comments. They applied an n-gram-based clustering approach to detect semantic similarities between entities and evaluated the results using precision and recall metrics.

Maletic and Marcus [64] propose an approach to software clustering that uses semantic dependencies extracted using Latent Semantic Indexing (LSI), a technique for identifying similarities between software components. They apply the minimal spanning tree (MST) algorithm for clustering and evaluate the results using metrics based on both semantic and structural information.

Wu et al. [125] conducted a comparative study of six clustering algorithms using structural dependencies on five software systems. Four of the algorithms are based on agglomerative clustering, one on program comprehension patterns, and one algorithm is a customized version of Bunch [120]. The performance of these algorithms was evaluated using the MoJo metric and NED (Non-Extreme Distribution).

Mancoridis and Mitchell [120], [126], [127] developed the Bunch tool for software clustering and used structural dependencies as input. The tool applies clustering algorithms to the structural dependency graph and outputs the system's organization. For evaluation, the authors introduced the Modularization Quality (MQ) metric, described in more detail in Section 6.3.2, and is also used in our current experiments as an evaluation metric.

Prajapati et al. [106] propose a many-objective SBSR (search-based software remodularization) approach with an improved definition of objective functions based on lexical, structural, and change-history dependencies. The authors evaluate their approach on several open-source software systems using the MoJoFM metric for external evaluation and the MQ metric for internal evaluation.

Şora et al. [54], [53] developed the ARTs (Architecture Reconstruction Tool Suite) for their experiments on improving software architecture reconstruction through clustering. The tool suite implements various clustering algorithms, such as minimum spanning tree-based, metric-based, search-based, and hierarchical clustering, primarily using structural dependencies as input. The research focuses on identifying the right factors for direct coupling between classes, indirect coupling, and layered architecture. The results of applying these different factors are evaluated using the MoJo distance metric.

Silva et al. [128] investigated using solely co-change dependencies as input for the Chameleon algorithm, an agglomerative hierarchical clustering method, to identify clusters. For evaluation, the authors used distribution maps to compare the clusters generated from co-change dependencies with the system's package structure.

6.3. Methodology and implementation

In this section, we present the methodology used to evaluate the impact of logical dependencies on the quality of software clustering solutions.

First, we describe the clustering algorithms used in our experiments: Louvain, Leiden, and DBSCAN. Next, we introduce the evaluation metrics used to assess the quality of the clustering results. Finally, we present the workflow and implementation of the tool developed for this research, which is built to process structural and logical dependencies, apply the selected clustering algorithms, and compute the evaluation metrics.

6.3.1. Clustering algorithms

Louvain

The Louvain algorithm was originally developed by Blondel et al. and is used to find 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 gained 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 [129], [130].

Leiden

The Leiden algorithm, developed by Traag et al., is an improvement over the Louvain algorithm for community detection in large networks. Like Louvain, the Leiden algorithm begins with each node assigned to its own cluster and iteratively moves nodes between clusters to optimize modularity. However, the Leiden algorithm addresses some problems of the Louvain method, particularly regarding poorly connected communities and runtime performance issues [131] [132].

The Leiden algorithm introduces a refinement phase that ensures communities are locally optimally clustered and well-connected. This refinement step distinguishes the Leiden algorithm from Louvain.

DBSCAN

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm, introduced by Ester et al., is a density-based clustering algorithm for identifying clusters of arbitrary shape and detecting noise in data [133], [132].

DBSCAN operates based on two main parameters:

- **Eps**: It defines the radius within which to search for neighboring points.
- **MinPts**: The minimum number of points required for a dense region. It determines the minimum number of neighbors a point should have to be considered a core point.

The algorithm classifies points into three categories:

1. **Core Points**: Points that have at least *MinPts* neighbors within a radius of *Eps*. These points are located in the interior of a cluster.
2. **Border Points**: Points that have fewer than *MinPts* neighbors within a radius of *Eps* but are in the Eps-neighborhood of a core point. They are located on the edge of a cluster.

3. **Noise**: Points that are neither core points nor border points.

The DBSCAN algorithm starts by visiting an arbitrary point in the dataset. If the point is a core point, the algorithm starts a new cluster and retrieves all reachable points from this core point. All points are then marked as part of the cluster. If the point is a border point, it moves to the next point in the dataset. This process is repeated until all points have been visited.

DBSCAN can be applied for software clustering by considering software entities as data points. A distance measure based on dependency weights can be used to compute the neighborhood between entities.

6.3.2. Clustering result evaluation

We evaluate the clustering results using two metrics: the Modularity Quality (MQ) metric and the Move and Join Effectiveness Measure (MoJoFM) metric. Each provides a different perspective on the quality of the clustering solutions.

Modularity Quality metric

Mancoridis et al. introduced the Modularity Quality (MQ) metric to evaluate the modularization quality of a clustering solution based on the interaction between modules (clusters) [**?**],[120]. It evaluates the difference between connections within clusters and connections between different clusters.

The MQ of a graph partitioned into k clusters, where A_i is the Intra-Connectivity of the i-th cluster and E_{ij} is the Inter-Connectivity between the i-th and j-th clusters, is calculated using Equation (6.1) [126].

$$MQ = \left(\frac{1}{k}\sum_{i=1}^{k} A_i\right) - \left(\frac{1}{k(k-1)}\sum_{i,j=1}^{k} E_{ij}\right) \tag{6.1}$$

The MQ metric's value ranges between -1 and 1. A value of -1 means that the clusters have more connections between the clusters than within the clusters, while a value of 1 means that there are more connections within clusters than between clusters. A good clustering solution should have an MQ value close to 1, since this indicates that the clusters are more cohesive internally and have fewer connections to other clusters.

The MQ metric is useful because it does not require additional input besides the clustering result. It relies on the structure of the clustered entities and their interactions.

MoJoFM metric

Wen and Tzerpos introduced the MoJoFM metric to evaluate the similarity between two different software clustering results [121]. The metric is based on the MoJo metric, which measures the absolute minimum number of *Move* and *Join* operations required to transform one clustering solution into another [123], [121]. However, MoJoFM provides a similarity measure ranging between 0% and 100%, where 100% indicates identical clustering solutions.

The MoJoFM metric is calculated using Equation (6.2):

$$\text{MoJoFM}(A, B) = \left(1 - \frac{\text{mno}(A, B)}{\text{max}(\text{mno}(\forall A, B))}\right) \times 100\% \tag{6.2}$$

Where:

- $\text{mno}(A, B)$ is the minimum number of *Move* and *Join* operations required to transform clustering solution A into clustering solution B.
- $\text{max}(\text{mno}(\forall A, B))$ is the maximum possible number of such operations required to transform any clustering A into clustering B.

To use the metric, we first need to generate a reference clustering solution for comparison. We manually created this reference based on our analysis of the codebase.

Using the MoJoFM metric, we can evaluate the similarity between the generated and reference clustering solutions. This metric is useful when combining multiple dependencies because it measures the similarity between the obtained clustering solutions and the same reference.

6.3.3. Tool workflow for software clustering and evaluation

To evaluate how logical dependencies impact the quality of clustering solutions, we developed a Python tool capable of using any type of dependency, either alone or combined with other types of dependencies, as long as it is provided in CSV format. The tool clusters and evaluates software clustering solutions using the MQ or MoJoFM metrics.

Input

The tool takes one or multiple dependency CSV files as input and the reference solution required for the MoJoFM metric. We designed the tool to accept multiple dependency files so that we can generate clustering solutions based on either a single type of dependency (structural or logical) or a combination of both.

Since the MoJoFM metric requires a reference solution to evaluate the obtained clustering solutions, we manually inspected the code and created reference clustering solutions, which we then provided as input for the tool.

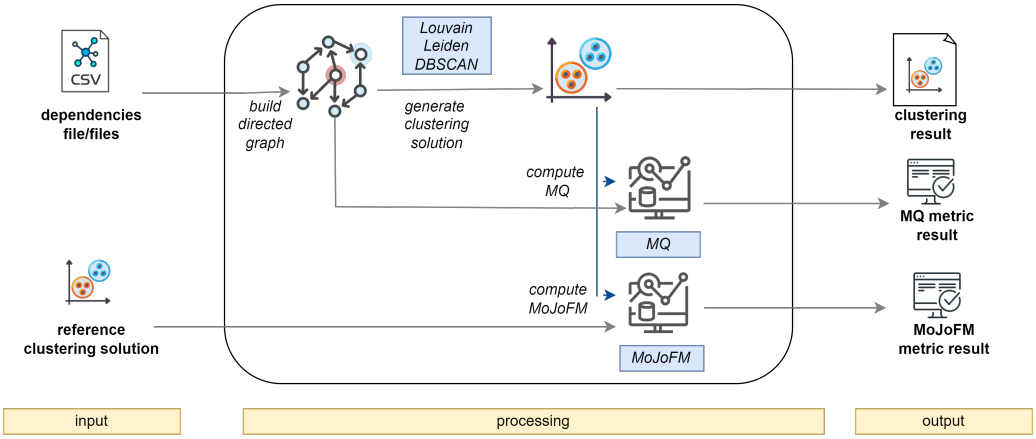

Figure 6.1: Tool workflow overview: input, processing and output.

Processing

The dependencies are saved in the CSV file in the following format: `antecedent of a dependency, consequent of a dependency, weight`. The tool reads each line, adds the antecedent and consequent as nodes in a directed graph, and creates an edge between them, with the weight from the CSV file becoming the edge weight. The edge weights are summed if multiple dependency files are processed and the same dependency is found in multiple files.

The workflow of applying the clustering algorithms and performing the evaluations is shown in Figure 6.1. After all dependencies are read, the directed graph is passed to the clustering algorithms: Louvain, Leiden, and DBSCAN. Each algorithm generates its own clustering result. The results from each algorithm are then evaluated using the MQ metric and the MoJoFM metric. The MQ metric requires the directed graph and the clustering result, while the MoJoFM metric requires the reference clustering solution provided as input and the clustering result.

Output

After applying each clustering algorithm and completing both evaluations, we export the clustering result, the number of clusters from the clustering solution, and the MQ and MoJoFM metrics values.

6.4. Data set used in experimental analysis

In Table 6.1, we have synthesized all the information about the four projects used in our experiments. The 'Project Name' column contains the names of the software projects sourced from GitHub. The 'Release Tag' column contains the specific release

Table 6.1: Overview of projects used in experimental analysis

Project Name	Release Tag	Commits	GitHub Repository Link	Repository Description
Apache Ant	1.10.13	14,917	https://github.com/apache/ant	Java build tool for automating software tasks.
Apache Tomcat	8.5.93	22,698	https://github.com/apache/tomcat	Java web server and servlet container.
Hibernate ORM	6.2.14	16,609	https://github.com/hibernate/hibernate-orm	Java ORM framework for database management.
Gson	gson-parent-2.10.1	1,772	https://github.com/google/gson	Java library for JSON serialization and deserialization.

tag of the project that was analyzed. We processed all the commits for logical dependency extraction, from the first commit to the commit associated with the specified tag. We extracted the dependencies from the code of that specific tag for structural dependencies. The 'Number of Commits' column provides the total number of commits used for logical dependencies extraction. The 'GitHub Repository Link' column includes the URL link to the project's repository on GitHub. Finally, the 'Repository Description' column briefly describes the project's purpose and functionality.

We mainly chose projects with more than 10,000 commits in their commit history so that the logical dependencies extraction can be done on a more extensive information base. However, we selected Gson, which has a relatively small commit history (1,772 commits), to determine if our experiments work with a smaller information base.

Table 6.2 presents the commit statistics for the studied projects. The columns represent the percentage of commits with under 5 files modified, between 5 and 10 files, between 10 and 20 files, and above 20 files modified. We can observe that most commits have under 5 files changed, with Apache Tomcat having more than 90% of the commits with less than 5 files changed. On the opposite side, only a few commits involve more than 20 files changed, Hibernate ORM having the highest percentage at 8.39%.

Table 6.2: Commit statistics for studied projects

Project Name	Number of files changed			
	Under 5	5-10	10-20	Above 20
Apache Ant	83.83%	7.50%	4.17%	4.50%
Apache Tomcat	90.95%	5.44%	2.04%	1.58%
Hibernate ORM	71.74%	12.37%	7.50%	8.39%
Gson	83.63%	9.85%	3.70%	2.81%

6.5. Experimental plan and results

6.5.1. Experimental plan

Tool runs

To assess the impact of logical dependencies and to answer the research questions from section 6.1, we run the tool presented in Section 6.3.3 in three different scenarios for all the projects from table 6.1. All three scenarios are illustrated in Fig. 6.2.

In the first scenario, we run the tool once, providing only the system's structural dependencies as input for the clustering algorithm.

In the second scenario, we run the tool ten times, using only logical dependencies as input. We perform ten runs because we generate logical dependencies with different threshold values for the strength filter. We start with a threshold of 10 and increase it in steps of 10 up to 100, where 100 is the maximum value for the threshold.

In the third scenario, we combine logical with structural dependencies. Similar to the second scenario, we ran the tool ten times using structural and logical dependencies generated with different strength thresholds.

6.5.2. Results

The experimental results are presented in this subsection in four tables, each corresponding to a different project. Table 6.3 presents the results for Apache Ant, Table 6.4 presents the results for Apache Tomcat, Table 6.5 presents the results for Hibernate ORM, and Table 6.6 presents the results for Gson.

Each table includes the following columns:

- **Dependency Type:** The types of dependencies used are as follows: SD for Structural Dependencies, LD for Logical Dependencies, and SD+LD for their combination. The strength threshold used is specified in parentheses right after LD.
- **Entities Count:** The total number of software entities (such as classes, interfaces, enums) involved in clustering.
- **System Coverage:** Considering that the total number of entities extracted from the codebase — which represents the entities forming structural dependencies (SD) — constitutes the entire set of entities in the system (the first line of each table), we calculated the percentage of entities present in the filtered logical dependencies (LD) relative to the total number of known codebase entities.
- **Louvain/ Leiden/ DBSCAN:** The clustering algorithms used in the experiments.
- **Nr. of Clusters:** The number of clusters from the clustering solution.
- **MQ (Modularization Quality):** The result obtained when applying the MQ evaluation metric to the clustering solution.
- **MoJoFM:** The result obtained when applying the MoJoFM evaluation metric to

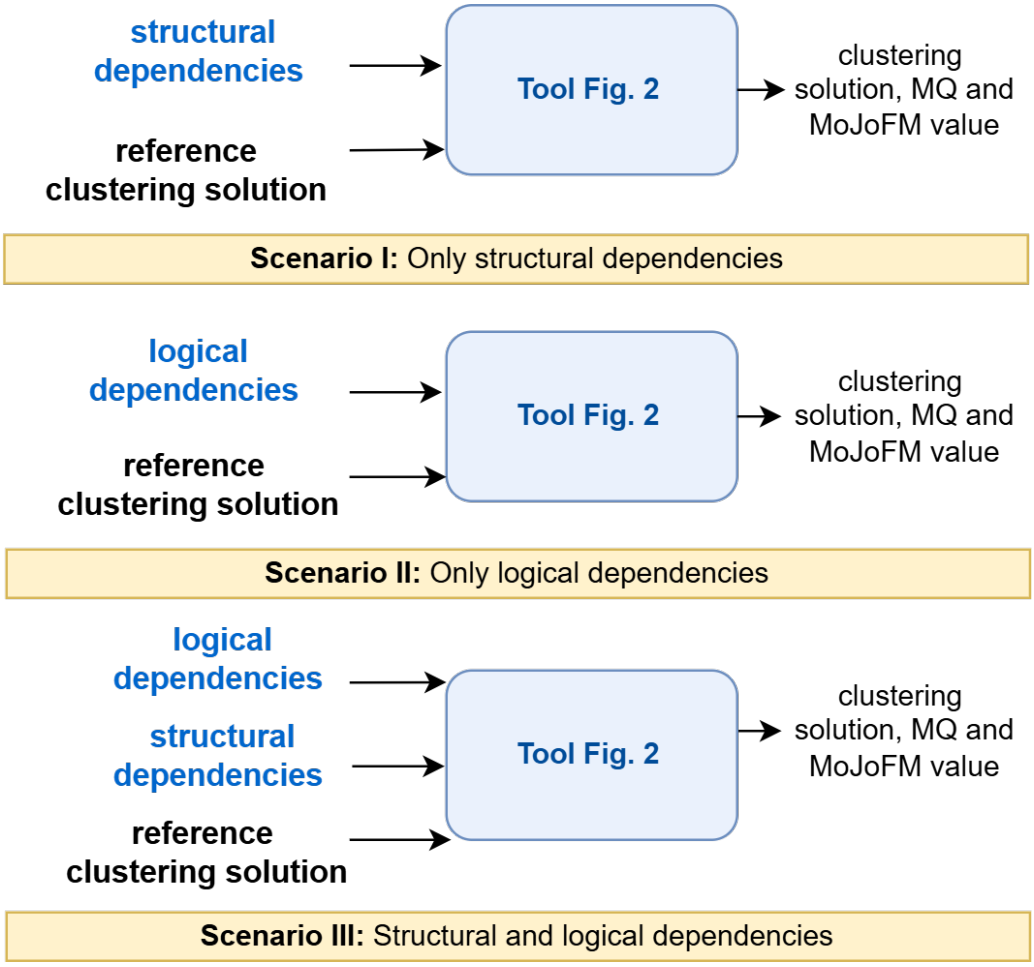

Figure 6.2: Experimental scenarios for analyzing the impact of logical dependencies on clustering quality

the clustering solution.

The rows in each table represent different dependency types and strength filter thresholds used in the clustering experiments.

To better understand the impact of different dependency types on software clustering, we also analyzed the average weights assigned to structural dependencies (SD) and logical dependencies (LD) across the studied projects. Table 6.7 presents these average dependency weights. The first row shows the average weights for SD, which remain constant across all strength thresholds, while the other rows show the average weights for logical dependencies at different strength thresholds.

Table 6.3: Clustering results based on different dependency types and strength filter thresholds for repository: https://github.com/apache/ant

Dependency Type (strength threshold)	Entities Count	System Coverage (%)	Louvain			Leiden			DBSCAN		
			Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
SD	517	100.00	14	0.114	46.02	14	0.101	52.99	34	0.144	25.1
LD (10)	320	61.89	55	0.506	65.57	55	0.506	65.57	30	0.435	39.02
LD (20)	215	41.58	53	0.547	68	53	0.547	68	23	0.505	53.5
LD (30)	174	33.65	44	0.558	71.7	44	0.558	71.7	19	0.585	50
LD (40)	152	29.40	40	0.580	71.53	40	0.580	71.53	19	0.602	53.06
LD (50)	138	26.69	35	0.604	73.98	35	0.604	73.98	17	0.633	56.1
LD (60)	120	23.21	34	0.587	70.48	34	0.587	70.48	14	0.650	51.43
LD (70)	106	20.50	32	0.577	71.43	32	0.577	71.43	11	0.661	51.65
LD (80)	92	17.79	29	0.576	70.13	29	0.576	70.13	9	0.709	50.65
LD (90)	79	15.28	24	0.606	71.88	24	0.606	71.88	8	0.705	56.6
LD (100)	64	12.37	19	0.611	75.51	19	0.611	75.51	6	0.691	56.93
SD+LD (10)	517	100.00	18	0.355	55.18	15	0.254	54.98	37	0.147	25.9
SD+LD (20)	517	100.00	17	0.318	52.39	19	0.365	53.78	32	0.149	26.49
SD+LD (30)	517	100.00	16	0.282	53.19	16	0.265	54.78	30	0.159	24.5
SD+LD (40)	517	100.00	17	0.340	51.99	17	0.317	53.19	31	0.146	24.7
SD+LD (50)	517	100.00	15	0.248	52.59	19	0.298	56.77	31	0.146	24.7
SD+LD (60)	517	100.00	16	0.244	50.8	16	0.271	54.38	32	0.155	25.1
SD+LD (70)	517	100.00	15	0.238	51.00	18	0.281	52.99	32	0.155	25.1
SD+LD (80)	517	100.00	13	0.246	45.22	15	0.255	45.82	32	0.155	25.1
SD+LD (90)	517	100.00	14	0.258	46.02	16	0.268	47.01	32	0.155	25.1
SD+LD (100)	517	100.00	15	0.214	50.8	15	0.227	50.4	32	0.155	25.1

Table 6.4: Clustering results based on different dependency types and strength filter thresholds for repository: https://github.com/apache/tomcat

Dependency Type (strength threshold)	Entities Count	System Coverage (%)	Louvain			Leiden			DBSCAN		
			Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
SD	662	100.00	26	0.186	77.76	24	0.184	76.99	43	0.142	73.31
LD (10)	406	61.33	42	0.505	72.47	42	0.505	72.47	60	0.393	67.93
LD (20)	303	45.77	45	0.538	68.26	45	0.538	67.24	41	0.510	72.7
LD (30)	249	37.61	46	0.532	69.87	46	0.532	69.87	32	0.561	80.33
LD (40)	208	31.42	42	0.590	69.70	42	0.591	70.71	28	0.572	83.84
LD (50)	198	29.91	44	0.604	70.21	44	0.604	70.21	22	0.631	85.11
LD (60)	177	26.74	45	0.601	70.66	45	0.601	70.66	18	0.662	85.63
LD (70)	164	24.77	45	0.598	75.32	45	0.598	75.32	17	0.676	88.96
LD (80)	127	19.18	36	0.618	79.49	36	0.618	79.49	15	0.713	89.74
LD (90)	116	17.52	32	0.623	81.13	32	0.623	81.13	14	0.718	89.62
LD (100)	110	16.62	30	0.640	85.00	30	0.640	85.00	13	0.735	89.00
SD+LD(10)	662	100.00	28	0.324	78.99	28	0.324	78.99	40	0.161	74.23
SD+LD(20)	662	100.00	31	0.287	78.22	30	0.320	80.06	50	0.189	73.31
SD+LD(30)	662	100.00	32	0.296	79.92	32	0.277	75.77	45	0.209	73.47
SD+LD(40)	662	100.00	34	0.292	79.91	32	0.326	78.22	43	0.198	73.47
SD+LD(50)	662	100.00	33	0.294	76.53	35	0.301	76.23	43	0.196	73.31
SD+LD(60)	662	100.00	35	0.304	77.15	33	0.286	76.84	41	0.177	73.62
SD+LD(70)	662	100.00	34	0.282	76.69	34	0.292	77.45	41	0.166	73.62
SD+LD(80)	662	100.00	34	0.283	76.23	33	0.282	76.38	42	0.153	73.47
SD+LD(90)	662	100.00	31	0.311	78.99	31	0.311	78.99	43	0.153	73.31
SD+LD(100)	662	100.00	31	0.311	78.83	31	0.305	78.37	43	0.153	73.31

Table 6.5: Clustering results based on different dependency types and strength filter thresholds for repository: https://github.com/hibernate/hibernate-orm

https://github.com/hibernate/hibernate-orm

Dependency Type (strength threshold)	Entities Count	System Coverage (%)	Louvain			Leiden			DBSCAN		
			Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
SD	4414	100.00	30	0.09	52.23	23	0.071	52.44	373	0.128	46.32
LD (10)	1450	32.85	44	0.389	57.22	45	0.39	58.22	99	0.395	57.08
LD (20)	1325	30.02	66	0.397	62.66	66	0.397	62.66	151	0.378	63.36
LD (30)	1222	27.68	66	0.38	62.45	67	0.38	63.04	148	0.378	65.42
LD (40)	915	20.73	84	0.417	63.68	85	0.412	63.56	110	0.382	66.9
LD (50)	900	20.39	84	0.409	64.56	84	0.409	64.56	105	0.386	67.02
LD (60)	848	19.21	82	0.406	63.26	81	0.41	63.39	104	0.379	65.13
LD (70)	459	10.40	89	0.516	69.08	89	0.516	69.08	41	0.467	58.21
LD (80)	450	10.19	91	0.506	68.64	91	0.506	68.64	39	0.479	60.49
LD (90)	432	9.79	92	0.492	66.93	92	0.492	66.93	40	0.473	58.66
LD (100)	356	8.07	81	0.524	65.92	81	0.524	65.92	29	0.537	58.2
SD+LD (10)	4414	100.00	19	0.096	53.93	19	0.099	52.28	282	0.121	46.01
SD+LD (20)	4414	100.00	21	0.126	52.85	23	0.122	56.21	309	0.135	47.4
SD+LD (30)	4414	100.00	26	0.121	55.76	26	0.15	54.54	317	0.135	49.45
SD+LD (40)	4414	100.00	27	0.182	54.57	28	0.163	55.89	350	0.134	49.35
SD+LD (50)	4414	100.00	26	0.16	52.37	24	0.147	53.31	350	0.134	49.37
SD+LD (60)	4414	100.00	26	0.161	52.35	27	0.153	53.19	352	0.135	49.31
SD+LD (70)	4414	100.00	28	0.139	52.78	29	0.154	54.34	366	0.13	47.13
SD+LD (80)	4414	100.00	28	0.142	52.83	28	0.147	53.35	366	0.13	47.72
SD+LD (90)	4414	100.00	28	0.136	52.62	30	0.153	53.83	365	0.13	47.72
SD+LD (100)	4414	100.00	30	0.128	52.78	28	0.114	55.23	365	0.128	47.75

Table 6.6: Clustering results based on different dependency types and strength filter thresholds for repository: https://github.com/google/gson

https://github.com/google/gson

Dependency Type (strength threshold)	Entities Count	System Cover (%)	Louvain			Leiden			DBSCAN		
			Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
gson SD	210	100.00	10	0.139	53.47	9	0.129	55.94	23	0.127	51.88
gson LD (10)	66	31.43	10	0.565	62.07	9	0.572	60.34	19	0.399	68.97
gson LD (20)	50	23.81	11	0.547	64.29	11	0.547	64.29	9	0.523	59.52
gson LD (30)	41	19.52	12	0.544	63.64	12	0.544	63.64	6	0.606	66.67
gson LD (40)	31	14.76	8	0.635	69.57	8	0.635	69.57	6	0.612	69.57
gson LD (50)	31	14.76	8	0.600	69.57	8	0.600	69.57	6	0.565	60.87
gson LD (60)	28	13.33	8	0.552	65.00	8	0.552	65.00	5	0.584	60.00
gson LD (70)	26	12.38	7	0.579	66.67	7	0.579	66.67	5	0.586	55.56
gson LD (80)	18	8.57	5	0.590	60.00	5	0.590	60.00	4	0.544	40.00
gson LD (90)	18	8.57	5	0.590	60.00	5	0.590	60.00	4	0.544	40.00
gson LD (100)	18	8.57	5	0.590	60.00	5	0.590	60.00	4	0.544	40.00
gson SD+LD(10)	210	100.00	11	0.317	64.36	11	0.317	64.36	20	0.172	63.86
gson SD+LD(20)	210	100.00	11	0.259	61.39	11	0.259	61.39	17	0.136	53.96
gson SD+LD(30)	210	100.00	11	0.277	61.39	11	0.277	61.39	20	0.136	55.94
gson SD+LD(40)	210	100.00	10	0.277	61.39	10	0.277	61.39	20	0.135	55.94
gson SD+LD(50)	210	100.00	10	0.270	60.40	11	0.270	60.89	20	0.135	55.94
gson SD+LD(60)	210	100.00	9	0.296	61.39	10	0.290	61.88	20	0.135	55.94
gson SD+LD(70)	210	100.00	8	0.295	59.41	8	0.295	59.41	20	0.135	55.94
gson SD+LD(80)	210	100.00	7	0.267	58.91	8	0.263	59.41	21	0.134	55.45
gson SD+LD(90)	210	100.00	7	0.267	58.91	7	0.267	58.91	21	0.134	55.45
gson SD+LD(100)	210	100.00	7	0.267	58.91	8	0.263	59.41	21	0.134	55.45

Dependency type	Ant	Tomcat	Hibernate	Gson
SD	5.91	6.91	5.41	5.24
LD(10)	11.17	12.82	2.45	14.15
LD(20)	16.01	19.65	3.00	19.10
LD(30)	18.08	23.56	3.27	27.58
LD(40)	19.08	25.57	4.63	29.85
LD(50)	19.94	26.31	4.80	29.97
LD(60)	24.26	28.91	5.14	33.93
LD(70)	26.70	30.35	9.53	34.37
LD(80)	30.83	35.33	10.18	43.00
LD(90)	32.11	36.90	10.47	43.00
LD(100)	33.93	37.04	12.00	43.00

Table 6.7: Average weights of Structural Dependencies (SD) and Logical Dependencies (LD).

6.6. Evaluation

The overall analysis of all the results from subsection 6.5.2 indicates that combining structural and logical dependencies (SD+LD) provides better clustering solutions than using structural dependencies (SD) alone, covering 100% of the system, meaning that no entity is missed during cluster generation. On the other hand, logical dependencies (LD) alone result in better clustering quality metrics compared to both SD and SD+LD, but they do not cover the entire system.

The best results for SD+LD are observed with a strength threshold between 10-40%. For LD only, the best results are obtained at a 100% strength threshold. The overall trend shows that for LD only, the MQ metric increases in value with a higher strength threshold, indicating more cohesive clusters, while the MoJo metric decreases, indicating that fewer transformations are needed to reach the expected clustering. For SD+LD, the best MQ and MoJo values are obtained at lower strength thresholds, and then both metrics indicate a less effective clustering solution obtained with higher strength thresholds.

We analyze each project in detail in the sections below and address the research questions.

6.6.1. Detailed evaluation

Apache Ant

The clustering results for Apache Ant (Table 6.3) show that the combined structural and logical dependencies (SD+LD) achieved the best values with a strength threshold between 10% and 30%. The highest value for the MQ metric is reached with Leiden at a strength threshold of 20%, and the highest value for MoJoFM is also reached with Leiden at a strength threshold of 10%.

Compared with the SD-only results, all SD+LD clustering solutions for all algorithms show better MQ metric values, with the highest MQ value for SD+LD being more than three times greater than the corresponding SD-only value. Similarly, the MoJoFM metric shows better results than SD-only. However, it does not always outperform the MoJoFM metric applied to SD-only data.

Logical dependencies (LD) alone produced the highest MQ and MoJoFM values at the 100% strength threshold for both Leiden and Louvain, with the obtained metric values being higher than those of SD-only and SD+LD. However, the percentage of entities covered is significantly lower (LD(100) covers only 12.37% of the system). If we look at LD(10), where there is a 61.89% coverage of the system, which is more compared to LD(100), both metrics still perform better than SD-only and SD+LD(10). However, there is still a gap until 100% coverage.

From the clustering algorithm performance point of view, Leiden obtains the best evaluation metrics for all scenarios, followed by Louvain and DBSCAN.

One interesting observation is that, based on the LD-only results, where the metrics results improved with higher strength thresholds, the SD+LD results did not

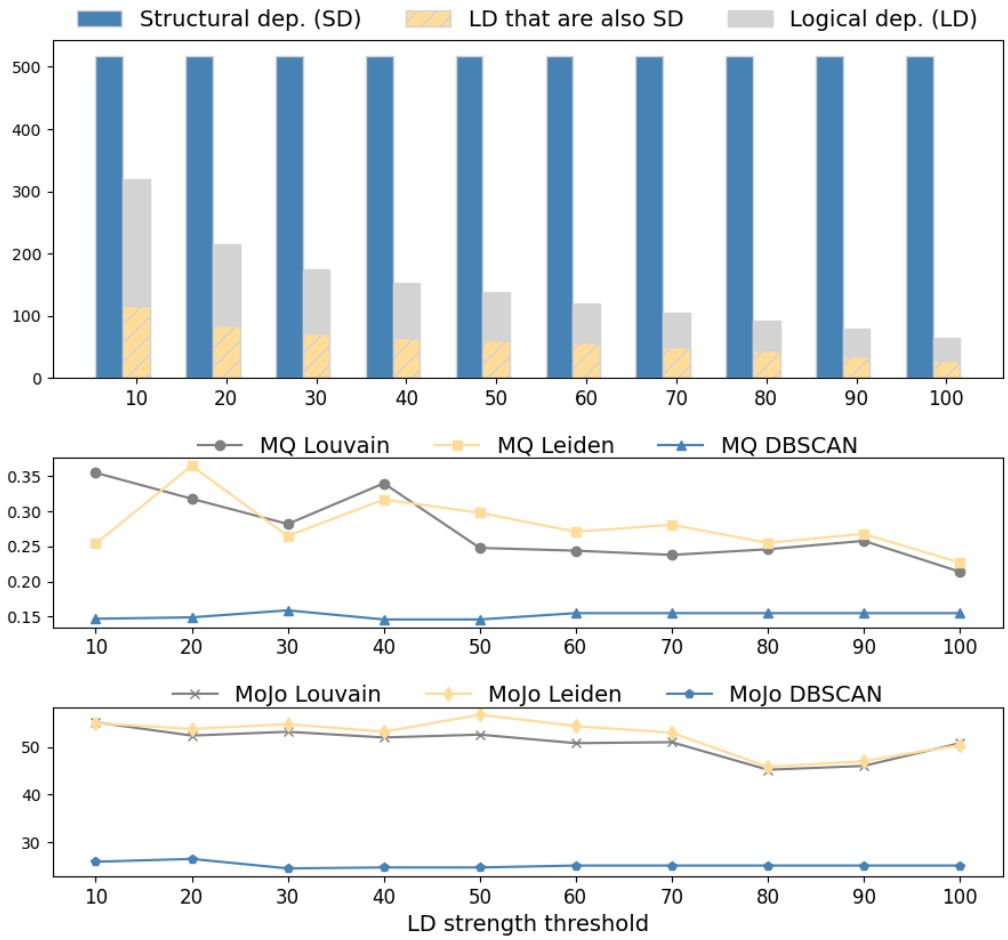

Figure 6.3: Apache Ant: Overlap between structural and logical dependencies and its correlation with clustering metrics.

follow the same pattern. On the opposite, the SD+LD metric results decline with a higher strength threshold. An explanation for this behavior may lie in the overlap between structural and logical dependencies. As presented in Figure 6.3, the number of LD decreases with a stricter strength threshold compared to the number of SD, and the overlap between the two types of dependencies increases.

In our previous works, we studied how these two types of dependencies overlap [57], [103]. The reason behind those studies was to check how much new information we can get from using logical dependencies and how much is already present via structural dependencies.

Our overall findings were that with stricter filtering of logical dependencies, we obtain a higher percentage of overlap between the two dependencies, reaching at

most 50% of logical dependencies that are also structural dependencies.

So, we consider that the reason why SD+LD clustering solutions decline in performance with a higher strength threshold is that less and less new additional information is added to the system (logical dependencies that are not structural dependencies), causing the clustering solution to start resembling the performance of the SD-only solution. In Figure 6.3, we can see that LD(10) represents 61% of the quantity of SD, while LD(100) is only at 12%, with half of them being duplicated with SD.

Apache Tomcat

For Apache Tomcat (Table 6.4), the best results for SD+LD were obtained with strength thresholds between 10% and 40% across all algorithms. The Leiden algorithm achieved the best result for the MQ metric at a strength threshold of 40%, while the best MoJoFM result was obtained at a threshold of 20%, also with the Leiden algorithm. Compared with the SD-only results, the peak MQ values almost double the SD-only values. Like Apache Ant, the MQ values for all strength thresholds are higher than those for SD-only. While MoJoFM is not better for all thresholds, it still improves compared with the SD-only results.

The LD-only results show the highest MQ and MoJoFM values at LD(100) for the Louvain and Leiden algorithms. However, as with the Apache Ant results, coverage remains an issue. LD(100) covers only 16.62% of the system, lower than the coverage from SD-only or SD+LD combinations. On the other hand, LD(10), which covers 61.33% of the system, still has better clustering solutions compared to SD-only, based on both MQ and MoJoFM results.

We observe the same decline in results with a stricter strength threshold for SD+LD. As with the previous system, these results can again be connected to the percentage of LD that also overlaps with SD and the decreasing number of LD compared to SD once the strength threshold becomes stricter. As shown in Figure 6.4, LD filtered with a 10% strength threshold overlaps with SD by approximately 22%, while at a 100% strength threshold, the overlap increases to approximately 39%.

To ensure that the decline in performance for SD+LD with a stricter strength threshold is indeed caused by the fact that LD are significantly fewer than SD, and SD duplicates that part of them at higher thresholds, we added an additional experiment to our study. In this experiment, whose results can be found in Table 6.8, we increased the weights associated with LD(100) for Apache Tomcat to confirm that we are dealing with an LD quantity problem rather than a weight problem.

Therefore, in this experiment, we increased the weight assigned to each logical dependency filtered with a 100% strength threshold from the Apache Tomcat system by values ranging from 1 to 5 and re-ran scenario III from Fig. 6.2.

To maintain consistency, we used the same columns as in the other result tables (6.3, 6.4, 6.5, 6.6), with the addition of two new columns:

- **Multiplication Factor:** The value by which each logical dependency weight is

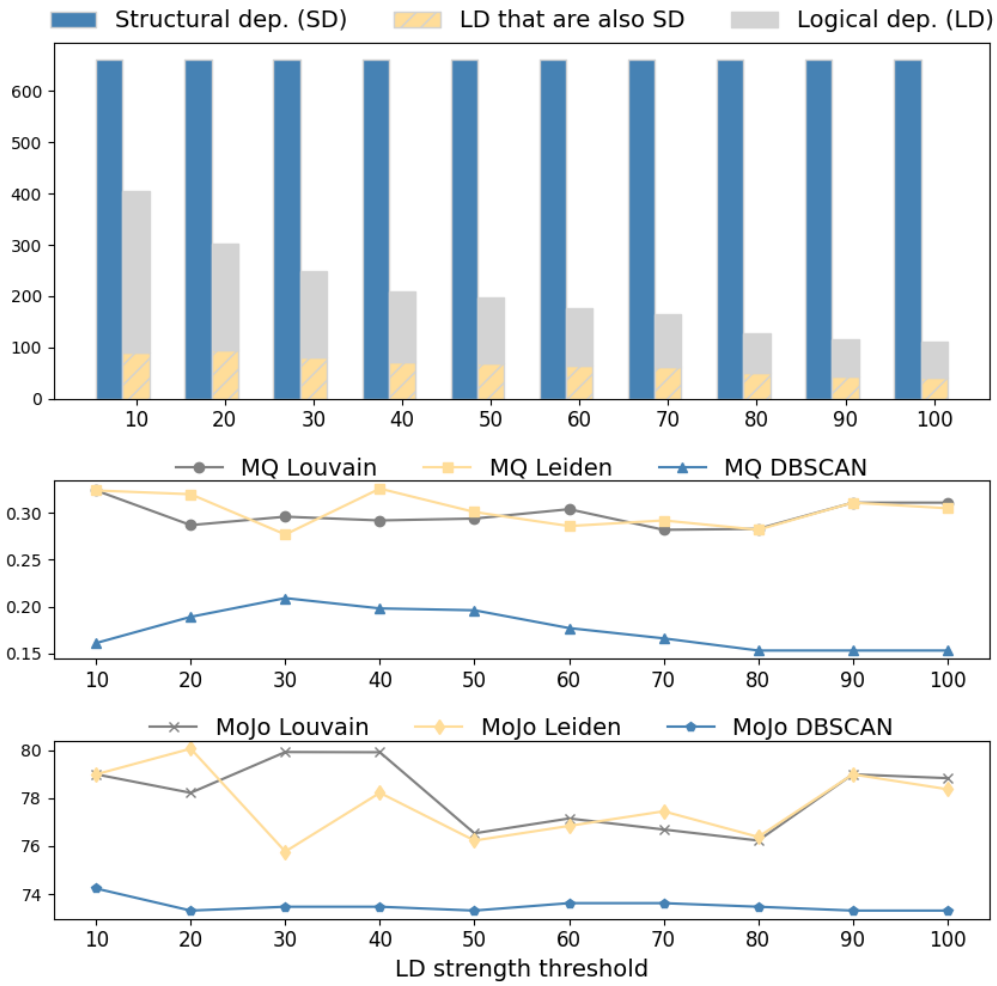

Figure 6.4: Apache Tomcat: Overlap between structural and logical dependencies and its correlation with clustering metrics.

multiplied.

- **Avg Weight:** The average weight assigned to each type of dependency used.

In Table 6.7, which presents the average weights associated with the dependencies across all systems, we can see that for Tomcat, the average weight for LD(10) is already almost double the SD average weight. For LD(100), the average weight is approximately five times higher than that of SD.

Based on the metric values obtained for multiplication factors of 2 to 5, we can see that after increasing the weights assigned to LD, the metric values improve only slightly, with changes recorded at the second decimal: a 0.02 improvement for Louvain and 0.07 for Leiden. The results for DBSCAN remain unchanged due to the

Table 6.8: Impact of multiplication factors on clustering results for LD(100) in Apache Tomcat

Multiplication Factor	Avg. weight		Louvain			Leiden			DBSCAN		
	SD	LD	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
1	6.91	37.04	31	0.311	78.83	31	0.305	78.37	43	0.153	73.31
2	6.91	74.08	33	0.295	73.57	30	0.301	72.33	43	0.153	73.31
3	6.91	111.12	34	0.313	74.19	33	0.309	72.80	43	0.153	73.31
4	6.91	148.16	34	0.312	73.88	33	0.312	72.49	43	0.153	73.31
5	6.91	185.20	34	0.306	73.88	33	0.308	72.18	43	0.153	73.31

fixed values of MinPts and Eps and the already high LD weights for LD(100).

We can conclude from the experiment with weights that the issue is the quantity of dependencies. SD outnumbers LD, making LD information less impactful on the clustering solution.

Hibernate ORM

Hibernate ORM is the second largest system after Apache Tomcat regarding the number of commits analyzed, with 16,609 commits considered for LD extraction. Additionally, it is the largest in terms of system size, with 4,414 entities (Table 6.1).

Based on the results from Table 6.5, the SD+LD combination with a strength threshold of 40 performs best for this system. Louvain achieves the best MQ metric at this threshold, while Leiden achieves the best MoJoFM metric.

LD-only produced the best MQ values at 100% strength and the best MoJoFM values at 70% strength for both Louvain and Leiden. Compared to the previous systems, where both best values were recorded at the same strength threshold, Hibernate shows an earlier peak for MoJoFM. The system coverage is likely a factor contributing to this. Hibernate LD[100] covers only 8.07% of the system, the lowest percentage among all systems studied. This low percentage can be linked to the number of commits compared to the number of entities. For Apache Tomcat, there were 22,698 commits and 662 entities, while for Hibernate, there were 16,609 commits and 4,414 entities. This indicates that not all entities had a chance to be updated in the version control system.

This observation is also reflected in Table 6.7, where Hibernate has the lowest average weights for LD compared to SD across all systems. In other systems, LD(10) starts with almost double the average weight compared to SD, while Hibernate's LD(10) average weight is less than half of the SD average weight.

Hibernate has the lowest overlap percentage between LD and SD, as shown in Figure 6.5. Similar to the other systems, the performance for MQ and MoJoFM decreases for SD+LD as the strength threshold becomes stricter.

The results for Hibernate highlight the challenge of achieving better clustering in larger systems with fewer commits relative to their size.

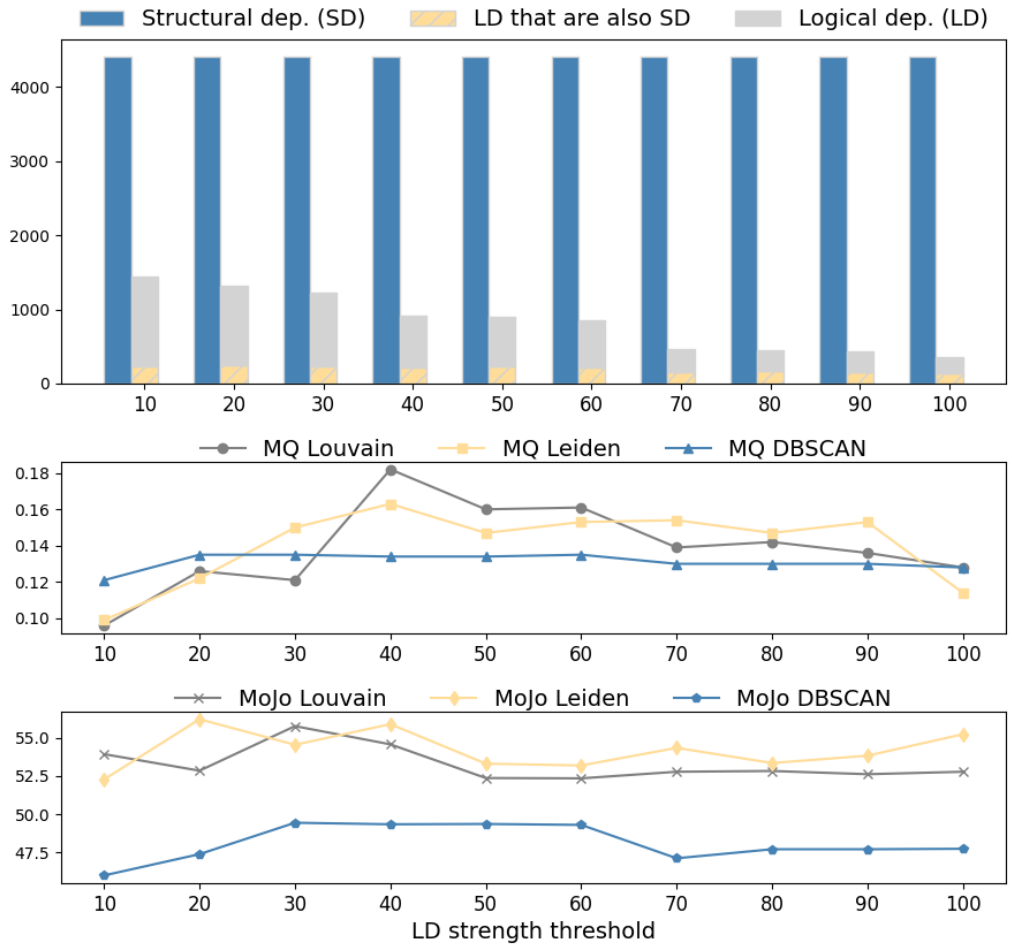

Figure 6.5: Hibernate ORM: Overlap between structural and logical dependencies and its correlation with clustering metrics.

Google Gson

Gson has the smallest number of commits analyzed, with 1,772 commits considered for LD extraction, and it is also the smallest in terms of system size, with 210 entities involved in clustering (Table 6.1).

Based on the results from Table 6.6, the SD+LD combination with a strength threshold of 10 achieved the best results for both MQ and MoJoFM. Like Apache Ant and Tomcat, all SD+LD combinations achieve better MQ values than SD-only.

LD-only produced the best results for MQ at 40% strength for both Louvain and Leiden, and the best MoJoFM value was also observed at the same threshold for

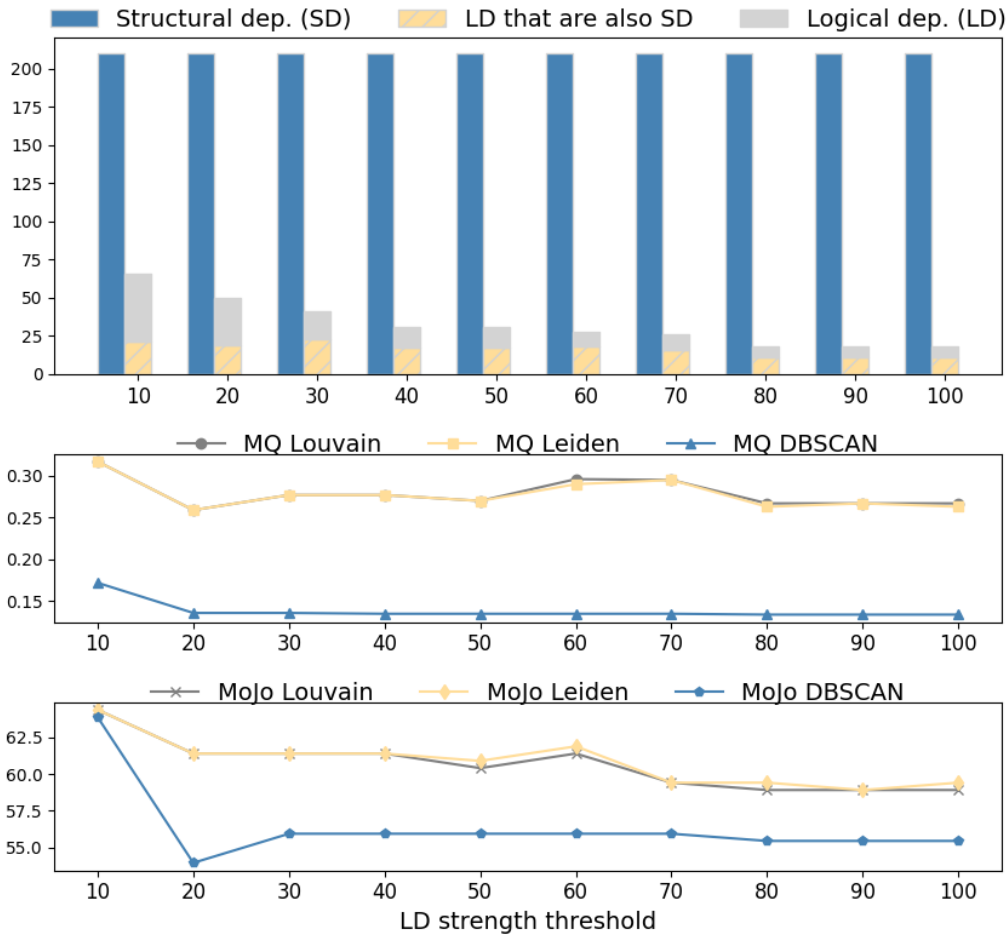

Figure 6.6: Google Gson: Overlap between structural and logical dependencies and its correlation with clustering metrics.

both algorithms. It is the only system where the best MQ result for LD-only occurs at a lower strength threshold than 100%. This is due to the very low number of entities remaining in the system at 100% (only 18 out of 210).

In this particular system, it is more visible that the Leiden clustering algorithm does not improve the Louvain algorithm in some scenarios. This observation is based on the fact that the values obtained for both MQ and MoJoFM metrics are the same in most cases for the Gson system for both algorithms.

It can also be observed that Gson has identical metric values for MQ and MoJoFM across multiple strength thresholds. Again, the small number of commits and the size of the system contribute to the stability of these metrics.

Gson also has relatively high overlap rates between LD and SD compared to the other systems, as shown in Figure 6.6. Despite the constant values, the trend of decreasing performance for SD+LD with stricter strength filtering for LD is also present in Gson.

The results for this system highlight the difficulty of achieving better clustering solutions using logical dependencies in smaller systems with fewer commits. However, even for a small system like Gson, an improvement is still visible when using logical dependencies.

6.6.2. Discussion on Ant clustering

Based on the results from Table **??**, we can observe that the combined approach of structural dependencies and logical dependencies gives the highest Modularity Quality (MQ) metric of 0.227 with a strength threshold of 30%, which is an improvement over the 0.08 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 30% for strength.

The clustering solution relying solely on structural dependencies consists of 12 clusters, while the solution using both structural and logical dependencies consists of 15 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

The migration of entities between clusters is illustrated in Figure 6.7. Given that the inner classes are shifted from one cluster to another in the same 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*.

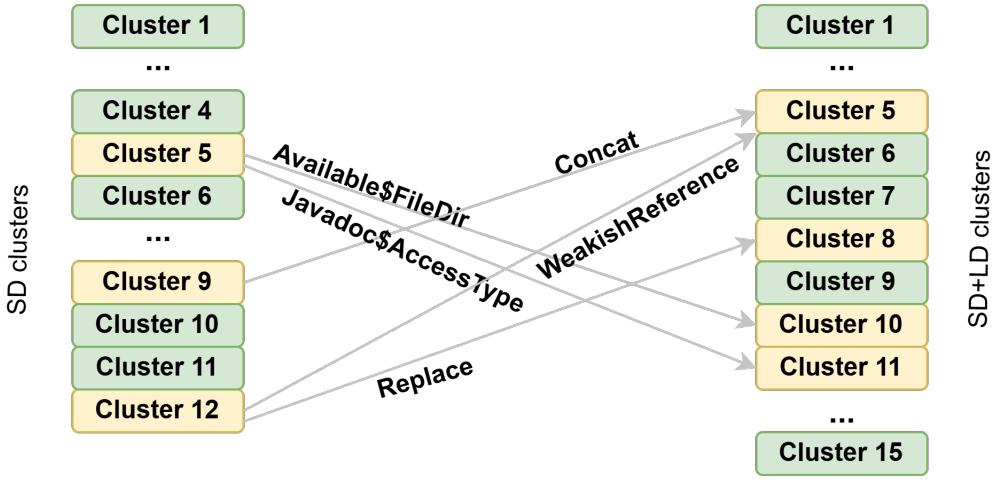

Figure 6.7: Migration of entities between clusters

taskdefs.Concat and its inner classes

To have a better overview of how and why entities are transferred between clusters, we depicted Concat's logical and structural connections in Figure 6.8. Additionally, Figure 6.9 illustrates connections within Cluster A, while Figure 6.10 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, they 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 than 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" [**?**]. 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.

taskdefs.Available$FileDir

In Cluster A the entity 'taskdefs.Available$FileDir' is in the same cluster with entities that are related to the build process (ProjectHelper, TaskAdapter, ComponentHelper), but not with entities that have any relation to condition checks or file existence eval-

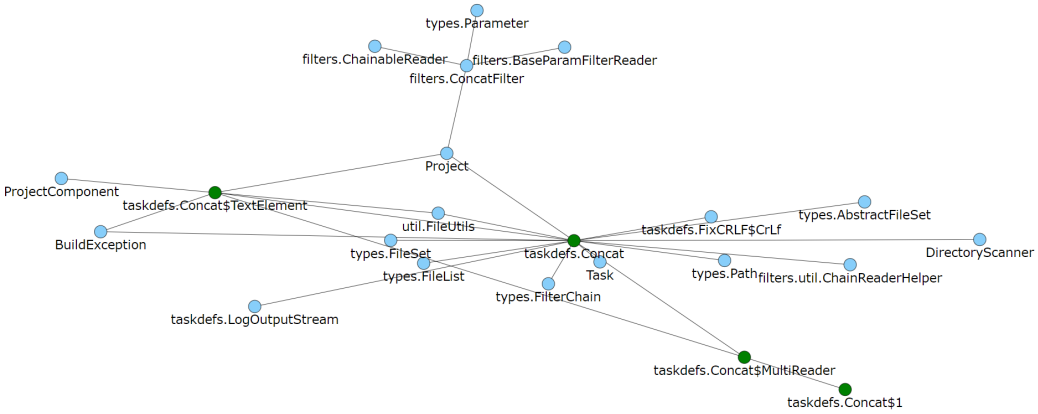

Figure 6.8: Dependencies (LD and SD) of Concat class

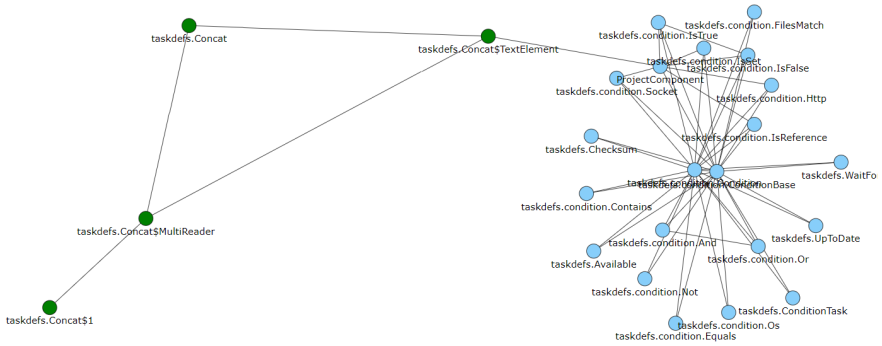

Figure 6.9: Placement of Concat in ClusterA (SD); cluster size: 25

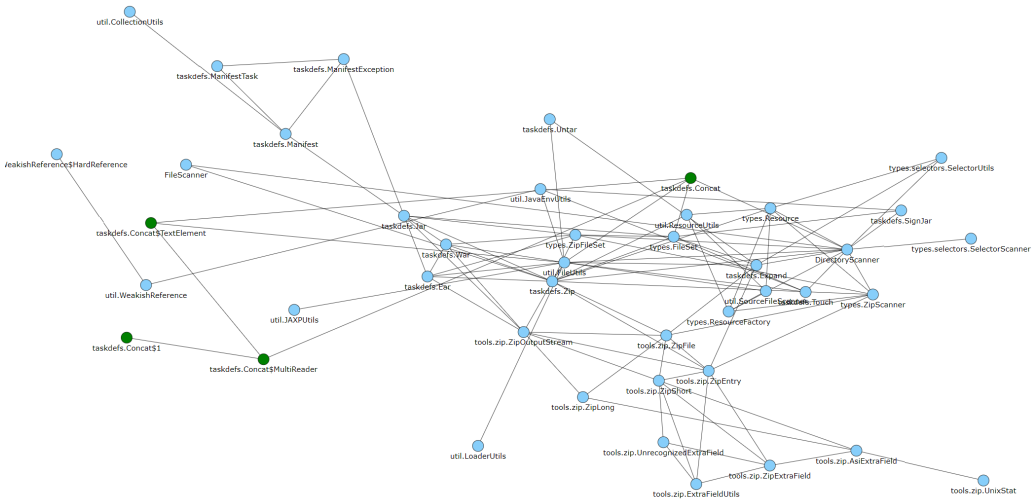

Figure 6.10: Placement of Concat in ClusterB (SD and LD); cluster size: 52

uations 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."[**?**]. This reinforces its placement in Cluster B, which is centered around task definitions and conditions related to build processes.

taskdefs.Replace and its inner classes

We depicted the logical and structural connections of Replace in Figure 6.11. Figure 6.12 illustrates connections within Cluster A, while Figure 6.13 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'.

The cluster from figure 6.12 is the 12th cluster from the clustering solution

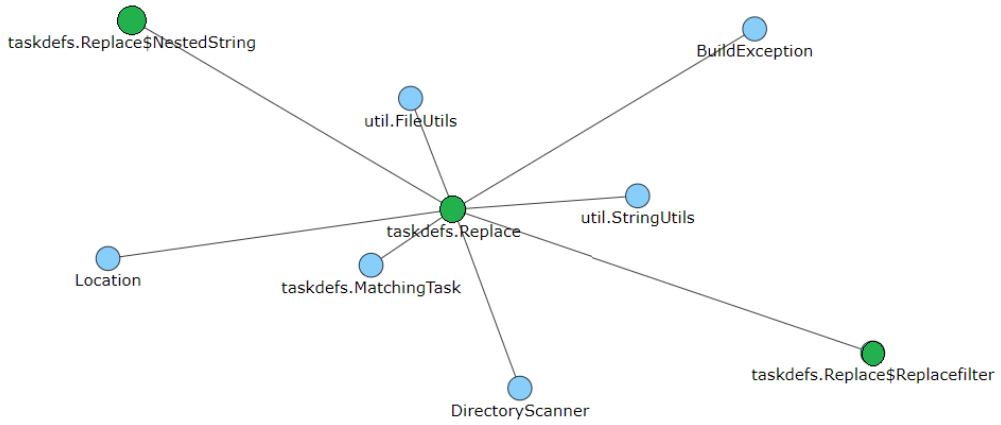

Figure 6.11: Ant dependencies (LD and SD) of Replace and its inner classes

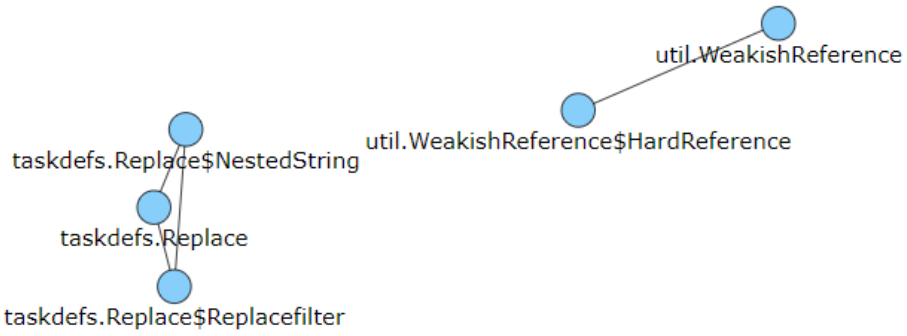

Figure 6.12: Placement of Replace in ClusterA (SD); cluster size: 5

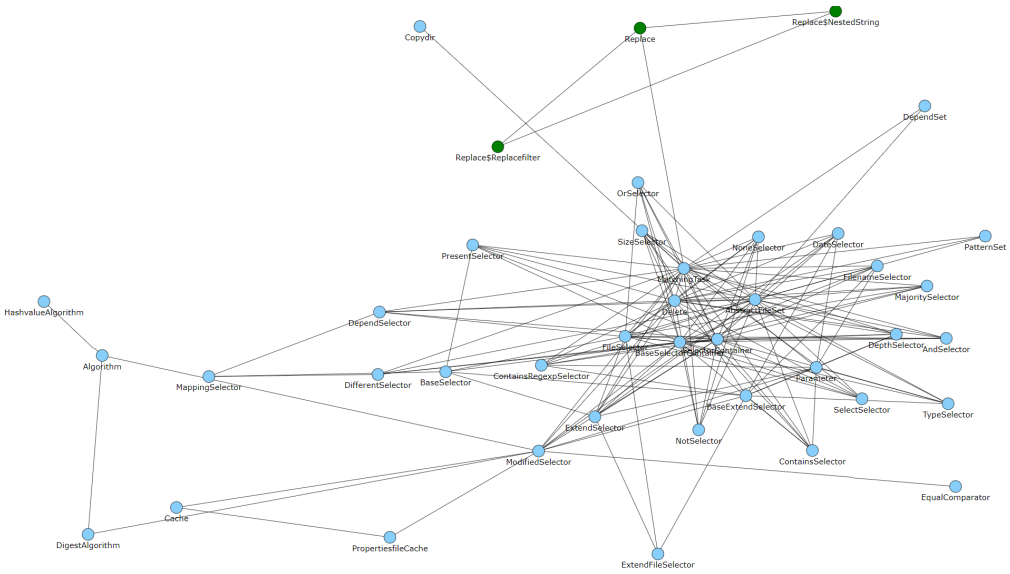

Figure 6.13: Placement of Replace in ClusterB (SD and LD); cluster size: 42

based on 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 [134].

6.6.3. Research questions and findings

In this section, we will answer our research questions based on the results of our experiments.

RQ1: *Does using structural dependencies (SD) combined with logical dependencies (LD) improve software clustering results compared to traditional approaches using only structural dependencies (SD)?*

Based on the results from all four systems analyzed, the combination of SD and LD performed better than SD-only according to the MQ and MoJoFM metrics, confirming that using SD+LD improves software clustering results. Across different strength thresholds, SD+LD achieved higher MQ and MoJoFM values for all clustering algorithms, indicating better modularization. The filtering thresholds applied to logical dependencies influence the effectiveness of combining SD and LD. Lower strength thresholds (10% to 40%) resulted in better MQ and MoJoFM values across all clustering algorithms compared with higher thresholds.

At higher strength thresholds, the MQ and MoJoFM results decline. This decline occurs because stricter strength thresholds significantly reduce the amount of logical dependencies. This reduction leads to fewer new relationships introduced into the clustering process. Our previous research on the overlap between SD and LD showed

that higher strength thresholds correlate with increased overlap between the two dependencies. This overlap indicates that at higher strength thresholds, not much new information is added to the system besides what is already introduced by structural dependencies, reducing the impact of logical dependencies on clustering results.

However, when considering a lower strength threshold, the relationship between the system size and the number of commits in the version history becomes an important factor.

In the case of Hibernate, a stricter strength threshold was needed to achieve the best metric values compared to other systems. Although the metrics obtained at a 10% strength threshold for LD are better than SD-only, the system reached peak metric values at 40% threshold across all algorithms.

With 16,609 commits and 4,414 entities, Hibernate has a significantly lower average number of commits per entity than Ant (14,917 commits and 517 entities) or Tomcat (22,698 commits and 662 entities). This lower ratio means that each entity in Hibernate is, on average, involved in fewer commits. As a result, the co-change data extracted for logical dependencies are sparser and contain more noise at lower strength thresholds.

A stricter strength threshold (e.g., 40%) filters out these weaker logical dependencies.

In contrast, systems like Ant and Tomcat, with higher ratios of commits to entities, obtain better results with logical dependencies at a 10% strength threshold because entities participate in more commits on average.

In conclusion, with an appropriate threshold, combining SD with LD leads to better clustering results than using SD alone.

RQ2: *Can using only logical dependencies (LD) produce good software clustering results?*

Using LD-only produced good clustering results, especially at higher strength thresholds. LD(100) produced the highest MQ and MoJoFM values for most systems compared with SD-only and SD+LD. However, the coverage of LD-only is significantly lower at these higher thresholds. For all systems, after filtering with 100% strength threshold, the system coverage of the remaining logical dependencies is less than 17% of the total known dependencies in the system.

Thus, while LD(100) provides the highest metric results compared to SD+LD and SD-only, it represents only a small subset of the system's dependencies.

On the other hand, LD(10) has, in most of the cases, better metric results than those for SD-only and SD+LD with better system coverage. Apache Ant and Tomcat LD(10) cover more than 60% of the system, while for Hibernate and Gson, the coverage is slightly above 30%.

Therefore, LD(10) can be an alternative to SD-only or SD+LD, especially if structural dependencies are not available.

It is also important to consider that LD-only performance at higher strength

thresholds depends on the system's characteristics, such as the number of commits and entities. For Gson project, the performance at a 100% strength threshold is not as good as for the other systems, reaching its peak at a 40% threshold. This is due to the low number of entities remaining at higher thresholds, with only 18 entities at the 80% to 100% strength thresholds, and the relatively small number of commits considered.

In summary, while LD-only can produce good clustering results, especially at higher strength thresholds, its limited coverage reduces its usability, as the clustering is intended for the entire system, not just a small subset. LD-only offers a good alternative to SD-only at lower strength thresholds, providing acceptable coverage.

RQ3: *How do different filtering settings for logical dependencies (LD) impact clustering results, and which filtering settings provide the best performance?*

The impact of different filtering settings on clustering performance was observed across all systems. For LD-only clustering, lower strength thresholds like 10% provided good system coverage but had lower MQ and MoJoFM values compared to higher thresholds like 100%, where the best metric results were often achieved. However, at these higher thresholds, the system coverage was significantly reduced.

The best performance was generally observed with strength thresholds between 10% and 40% for the combination of SD and LD (SD+LD). At these thresholds, the clustering solutions achieved higher MQ and MoJoFM values than SD alone.

It is important to select the optimal filtering threshold. Logical dependencies filtered with lower strength thresholds include more relationships, introducing more knowledge that could improve clustering results. However, a too-low threshold may sometimes introduce noise, especially in systems with a low commits-to-entities ratio. On the other hand, higher thresholds significantly reduce noise but can exclude valuable dependencies and decrease coverage.

The optimal filtering threshold may vary depending on system characteristics (number of commits, size of the system), so it is important to consider these factors when filtering logical dependencies.

7. CONCLUSION AND FUTURE WORK

7.1. Summary of research findings

7.2. Contributions

7.3. Future work

BIBLIOGRAPHY

[1] Ioana Şora and Ciprian-Bogdan Chirila. Finding key classes in object-oriented software systems by techniques based on static analysis. *Information and Software Technology*, 116:106176, 2019.

[2] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[3] 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.

[4] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 167–176, New York, NY, USA, 2005. ACM.

[5] Trosky B. Callo Arias, Pieter van der Spek, and Paris Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16(5):544–586, Oct 2011.

[6] Anna Corazza, Sergio Di Martino, Valerio Maggio, and Giuseppe Scanniello. Investigating the use of lexical information for software system clustering. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 35–44, 2011.

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

[8] Anna Corazza, Sergio Di Martino, and Giuseppe Scanniello. A probabilistic based approach towards software system clustering. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 88–96, 2010.

[9] A. Podgurski and L.A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[10] Alberto Costa Neto, Marcio de Medeiros Ribeiro, Marcos Dosea, Rodrigo Bonifacio, and Paulo Borba. Semantic dependencies and modularity of aspect-oriented software. In *Proceedings of the 29th International Conference on Software Engineering Workshops*, ICSEW '07, page 171, USA, 2007. IEEE Computer Society.

[11] Cezar Sas and Andrea Capiluppi. Using structural and semantic information to identify software components. 02 2021.

[12] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, Feb 2009.

[13] Liguo Yu. Understanding component co-evolution with a study on linux. *Empirical Softw. Engg.*, 12(2):123–141, April 2007.

[14] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 190–, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IWPSE '03, pages 13–, Washington, DC, USA, 2003. IEEE Computer Society.

[16] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. An empirical study on the developers' perception of software coupling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 692–701, May 2013.

[17] Xiaoxia Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 664–665, May 2005.

[18] Gustavo Ansaldi Oliva and Marco Aurélio 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*, pages 250–260, 2015.

[19] Gustavo Ansaldi Oliva and Marco Aurelio Gerosa. On the interplay between structural and logical dependencies in open-source software. In *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering*, SBES '11, pages 144–153, Washington, DC, USA, 2011. IEEE Computer Society.

[20] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *2010 17th Working Conference on Reverse Engineering*, pages 119–128, Oct 2010.

[21] Igor Scaliante Wiese, Rodrigo Takashi Kuroda, Reginaldo Re, Gustavo Ansaldi Oliva, and Marco Aurélio Gerosa. An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project. In Ernesto Damiani, Fulvio Frati, Dirk Riehle, and Anthony I. Wasserman, editors, *Open Source Systems: Adoption and Impact*, pages 3–12, Cham, 2015. Springer International Publishing.

[22] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

[23] Guang-yi Tang and Hong-wei Xuan. Research on measurement of software package dependency based on component. *Journal of Software*, 7, 09 2012.

[24] Dharmalingam Ganesan. Adam: External dependency-driven architecture discovery and analysis of quality attributes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23, 03 2014.

[25] Franz Lehner. Software life cycle management based on a phase distinction method. *Microprocessing and Microprogramming*, 32:603–608, 08 1991.

[26] K H. Bennett, Dh Le, and Vaclav Rajlich. The staged model of the software lifecycle: A new perspective on software evolution. 05 2000.

[27] K. Bennett. Legacy systems: coping with success. *IEEE Software*, 12(1):19–23, Jan 1995.

[28] Keith H. Bennett and Vaclav Rajlich. Software maintenance and evolution: a roadmap. pages 73–87, 05 2000.

[29] Vaclav Rajlich. Modeling software evolution by evolving interoperation graphs. *Ann. Software Eng.*, 9:235–248, 05 2000.

[30] Gerardo Canfora and Massimiliano Di Penta. New frontiers of reverse engineering. pages 326 – 341, 06 2007.

[31] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC '78.*, pages 60–65, Nov 1978.

[32] S A. Bohner and R S. Arnold. Software change impact analysis. *IEEE Computer Society*, 1, 01 1996.

[33] Hongji Yang and Martin Ward. Successful evolution of software systems. 01 2003.

[34] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control With Subversion for Subversion 1.6: The Official Guide And Reference Manual*. CreateSpace, Paramount, CA, 2010.

[35] S. Li, H. Tsukiji, and K. Takano. Analysis of software developer activity on a distributed version control system. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 701–707, March 2016.

[36] Git Contributors. *Git Documentation*, 2024.

[37] GitHub, Inc. Github, 2024.

[38] Fabian Beck and Stephan Diehl. On the congruence of modularity and code coupling. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 354–364, New York, NY, USA, 2011. ACM.

[39] Noriko Hanakawa. Visualization for software evolution based on logical coupling and module coupling. In *14th Asia-Pacific Software Engineering Conference (APSEC'07)*, pages 214–221, 2007.

[40] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving evolvability through refactoring. volume 30, 07 2005.

[41] Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18, 10 2012.

[42] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. Mining sequences of changed-files from version histories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 47–53, New York, NY, USA, 2006. Association for Computing Machinery.

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

[44] 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.

[45] Thomas Rolfsnes, Stefano Di Alesio, Razieh Behjati, Leon Moonen, and Dave W. Binkley. Generalizing the analysis of evolutionary coupling for software change impact analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 201–212, 2016.

[46] Manishankar Mondal, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. Historank: History-based ranking of co-change candidates. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 240–250, 2020.

[47] Manishankar Mandal, Chanchal K. Roy, and Kevin A. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 114–123, 2014.

[48] Chakkrit Tantithamthavorn, Akinori Ihara, and Ken-Ichi Matsumoto. Using co-change histories to improve bug localization performance. In *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 543–548, 2013.

[49] Marco D'Ambros and Michele Lanza. Reverse engineering with logical coupling. In *2006 13th Working Conference on Reverse Engineering*, pages 189–198, 2006.

[50] Mark Shtern and Vassilios Tzerpos. Clustering methodologies for software engineering. *Adv. Soft. Eng.*, 2012:1:1–1:1, January 2012.

[51] 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.

[52] Nemitari Ajienka, Andrea Capiluppi, and Steve Counsell. An empirical study on the interplay between semantic coupling and co-change of software classes. *Empirical Software Engineering*, 23(3):1791–1825, 2018.

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

[54] Ioana 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)*, pages 45–54. INSTICC, SciTePress, 2013.

[55] Ioana Şora. Helping program comprehension of large software systems by identifying their most important classes. In *Evaluation of Novel Approaches to Software Engineering - 10th International Conference, ENASE 2015, Barcelona, Spain, April 29-30, 2015, Revised Selected Papers*, pages 122–140. Springer International Publishing, 2015.

[56] 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.

[57] Adelina-Diana Stana and Ioana Şora. Logical dependencies: Extraction from the versioning system and usage in key classes detection. *Computer Science and Information Systems*, 20:25–25, 2023.

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

[59] Gustavo Ansaldi Oliva and Marco Aurélio Gerosa. Experience report: How do structural dependencies influence change propagation? an empirical study. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 250–260, 2015.

[60] Kamran Sartipi. Software architecture recovery based on pattern matching. 09 2003.

[61] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, Oct 2016.

[62] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[63] L Bass, P Clements, and Rick Kazman. Software architecture in practice 2nd edition. 01 2003.

[64] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic (lexical) and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 103–112, 2001.

[65] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. pages 244–, 01 1996.

[66] Andrian Marcus and J.I. Maletic. Identification of high-level concept clones in source code. pages 107– 114, 12 2001.

[67] Ira Baxter, Andrew Yahin, Leonardo de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. volume 368-377, pages 368–377, 01 1998.

[68] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28:654– 670, 08 2002.

[69] James R. Cordy and Chanchal K. Roy. The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220, 2011.

[70] Md Saidur Rahman and Chanchal K. Roy. On the relationships between stability and bug-proneness of code clones: An empirical study. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 131–140, 2017.

[71] Abdullah Sheneamer, Hanan Hazazi, Swarup Roy, and Jugal Kalita. Schemes for labeling semantic code clones using machine learning. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 981–985, 2017.

[72] Arianna Blasi and Alessandra Gorla. Replicomment: Identifying clones in code comments. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 320–3203, 2018.

[73] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. 11 2002.

[74] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190, March 2011.

[75] F. Khomh, M. Di Penta, and Y. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84, Oct 2009.

[76] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17:243–275, 06 2012.

[77] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359, 2004.

[78] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V. Mäntylä. Code smell detection: Towards a machine learning-based approach. In *2013 IEEE International Conference on Software Maintenance*, pages 396–399, 2013.

[79] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology*, 99:1–10, 2018.

[80] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, May 2015.

[81] 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.

[82] L. Tahvildari and K. Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *J. Softw. Maintenance Res. Pract.*, 16:331–361, 2004.

[83] Ioana Şora. Finding the right needles in hay - helping program comprehension of large software systems. In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 129–140. INSTICC, SciTePress, 2015.

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

[85] Ferdian Thung, David Lo, Mohd Hafeez Osman, and Michel R. V. Chaudron. Condensing class diagrams by analyzing design and network metrics using optimistic classification. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 110–121, New York, NY, USA, 2014. Association for Computing Machinery.

[86] M. H. Osman, M. R. V. Chaudron, and P. v. d. Putten. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In *2013 IEEE International Conference on Software Maintenance*, pages 140–149, 2013.

[87] Spencer Rugaber. Program comprehension. 08 1997.

[88] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based lightweight C++ fact extractor. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 134–143, May 2003.

[89] Bennet Lientz, E Burton Swanson, and Gerry E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21:466–471, 06 1978.

[90] Nakshatra Gupta, Ashutosh Rajput, and Sridhar Chimalakonda. Cospex: A program comprehension tool for novice programmers. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 41–45, 2022.

[91] Iris Vessey. Expertise in debugging computer programs. 12 1984.

[92] James A. Jones and Mary Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. pages 273–282, 01 2005.

[93] Holger Cleve and Andreas Zeller. Locating causes of program failures. pages 342– 351, 06 2005.

[94] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381, 2012.

[95] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering*, 47(11):2348–2368, 2021.

[96] Jeongju Sohn. Bridging fault localisation and defect prediction. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 214–217, 2020.

[97] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[98] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, Feb 1991.

[99] V. Y. Shen, Tze-jie Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software—an empirical study. *IEEE Transactions on Software Engineering*, SE-11(4):317–324, April 1985.

[100] David Binkley. Source code analysis: A road map. pages 104–119, 06 2007.

[101] srcml; www.srcml.org.

[102] Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. Lightweight transformation and fact extraction with the srcML toolkit. In *Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, SCAM '11, pages 173–184, Washington, DC, USA, 2011. IEEE Computer Society.

[103] Stana Adelina and Şora Ioana. Analyzing information from versioning systems to detect logical dependencies in software systems. In *International Symposium on Applied Computational Intelligence and Informatics (SACI)*, May 2019.

[104] Adelina Diana Stana. and Ioana 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,*, pages 486–493. INSTICC, SciTePress, 2019.

[105] Nicolas Anquetil and Timothy Lethbridge. File clustering using naming conventions for legacy systems. *Proceedings of the Second Working Conference on Reverse Engineering*, 1998.

[106] Amarjeet Prajapati, Anshu Parashar, and Amit Rathee. Multi-dimensional information-driven many-objective software remodularization approach. *Frontiers of Computer Science in China*, 17(3):173209, 2023.

[107] P. Meyer, H. Siy, and S. Bhowmick. Identifying important classes of large software systems through k-core decomposition. *Adv. Complex Syst.*, 17, 2014.

[108] D. Steidl, B. Hummel, and E. Juergens. Using network analysis for recommendation of central software classes. In *2012 19th Working Conference on Reverse Engineering*, pages 93–102, 2012.

[109] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[110] Yi Ding, B. Li, and Peng He. An improved approach to identifying key classes in weighted software network. *Mathematical Problems in Engineering*, 2016:1–9, 2016.

[111] Weifeng Pan, Beibei Song, Kangshun Li, and Kejun Zhang. Identifying key classes in object-oriented software using generalized k-core decomposition. *Future Generation Computer Systems*, 81:188–202, 2018.

[112] X. Yang, D. Lo, X. Xia, and J. Sun. Condensing class diagrams with minimal manual labeling cost. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 22–31, 2016.

[113] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006. ROC Analysis in Pattern Recognition.

[114] Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

[115] L. do Nascimento Vale and M. de A. Maia. Keecle: Mining key architecturally relevant classes using dynamic analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 566–570, 2015.

[116] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 134–142, 2005.

[117] M. Kamran, M. Ali, and B. Akbar. Identification of core architecture classes for object-oriented software systems. *Journal of Applied Computer Science & Mathematics*, 10:21–25, 2016.

[118] A. Mubarak, S. Counsell, and R. M. Hierons. An evolutionary study of fan-in and fan-out metrics in oss. In *2010 Fourth International Conference on Research Challenges in Information Science (RCIS)*, pages 473–482, 2010.

[119] Gustavo Oliva and Marco Aurelio Gerosa. A method for the identification of logical dependencies. In *Proceedings of the 7th International Conference on Global Software Engineering (ICGSEW)*, pages 70–72, 2012.

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

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

[122] V. Tzerpos and R. C. Holt. Accd: an algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, 2000.

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

[124] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, February 2005.

[125] A. E. Hassan Wu and R. C. Holt. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 525–535, 2005.

[126] 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 of the 6th International Workshop on Program Comprehension (IWPC'98)*, pages 45–52, 1998.

[127] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, March 2006.

[128] Luciana Silva, Marco Valente, and Marcelo Maia. Co-change clusters: Extraction and application on assessing software modularity. *Transactions on Aspect-Oriented Software Development*, 2015.

[129] Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008.

[130] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E, Statistical, Nonlinear, and Soft Matter Physics*, 80(5), 2009.

[131] V. Traag, L. Waltman, and Nees Jan van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific Reports*, 9:5233, 2019.

[132] T. Bonald, N. de Lara, Q. Lutz, and B. Charpentier. Scikit-network: Graph analysis in python. *Journal of Machine Learning Research*, 21(185):1–6, 2020.

[133] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 226–231, 1996.

[134] Adelina-Diana Stana and Ioana Şora. Integrating logical dependencies in software clustering: A case study on apache ant. pages 113–118, 10 2024.

List of published papers

[A1]. **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.

[A2]. **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.

[A3]. **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.

[A4]. **Stana, Adelina-Diana**, and Şora, 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.

[A5]. **Stana, Adelina-Diana**, and Şora, Ioana. (2024). *Refining Software Clustering: The Impact of Code Co-Changes on Architectural Reconstruction*. IEEE Access, pp. xx–xx. DOI: 10.xxxx/ACCESS.xxxxxxx.