

Logical dependencies: extraction from the versioning system and an example of usage *

Adelina Diana Stana¹ and Ioana Șora²

¹ Stana Adelina Diana

Politehnica University, Piața Victoriei Nr. 2, ; 300006 Timișoara, jud. Timiș, România
stana.adelina.diana@gmail.com

² Șora Ioana

Politehnica University, Piața Victoriei Nr. 2, ; 300006 Timișoara, jud. Timiș, România
ioana.sora@cs.upt.ro

Abstract. The version control system of every software product can provide important information about how the system is connected. In this study, we first propose a language-independent method to collect and filter dependencies from the version control, and second, we use the results obtained in the first step to identify key classes from three software systems. To identify the key classes, we are using the dependencies extracted from the version control system together with dependencies from the source code, but also separate. Based on the results obtained we can say that, compared with the results obtained by using only dependencies extracted from code, the mix between both types of dependencies do not provide dramatically different results, only small improvements. And, by using only dependencies from the version control system, we obtained results that did not surpass the results previously mentioned, but are still acceptable. Even though the logical dependencies results are only acceptable, this might open an important opportunity for software systems that use dynamically typed languages such as JavaScript, Objective-C, Python, Ruby, or systems that use multiple languages. These types of systems, for which the code dependencies are harder to obtain, can use the dependencies extracted from the version control to gain better knowledge about the system.

Keywords: logical dependencies; logical coupling; mining software repositories; versioning system; key classes; co-changing entities; software evolution.

1. Introduction

The version control (also known as source control) system that tracks changes in source code during software development can provide useful information about the system's details. The usage of information extracted from the version control system is not something new, previous works have used version control information to detect design issues [30], predict fault incidence among modules [12], [3] or guide software changes [7], [2]. In software engineering literature, concepts like evolutionary coupling, evolutionary dependencies, logical dependencies, or logical coupling refer to the same sort of relationship among software entities. That relationship is extracted from the version control system

* If this is an extended version of a conference paper, it should be clearly stated here.

1 and can mean that the entities from the source code files that are changing together,
 2 evolve together, and might depend on one another. Studies show that dependency rela-
 3 tionships found in the source code overlap only in a small percentage with dependency
 4 relationships found in the version control system, and suggest that these two types of rela-
 5 tionships can be used together [14], [1]. But, in practice, dependencies extracted from the
 6 version management system are rarely used due to the size of the information extracted
 7 [21]. A relatively small source code repository with roundabout one thousand commits
 8 can lead to millions of connections. In this paper, we intend to speed up the processing
 9 time, reduce the size of connections extracted from the version control and, increase the
 10 confidence that the connections obtained might be related by applying a set of filters with
 11 different thresholds to the information extracted. In order to validate the results obtained,
 12 and to see if the filtering methods had or not had a favorable effect on the final result,
 13 we want to identify the key classes of different systems. The identification of key classes
 14 had been previously performed by using structural dependencies, so, we intend to use the
 15 results obtained together with structural dependencies, and also separate, and see how the
 16 final results fluctuate.

17 The paper is organized as follows: Section 2 introduces the concepts of logical de-
 18 pendencies and the methods of obtaining them. Section 3 introduces the concept of key
 19 classes and the new approach of using logical dependencies to detect key classes. Section
 20 4 defines the data set used, and presents the new results obtained with the data set. Finally,
 21 section 5 discusses the conclusions based on the results obtained.

22 **2. The concept of logical dependencies**

23 **2.1. State of the art**

24 The concept of logical coupling (dependency) was first introduced by Gall et al. [11].
 25 They defined the logical dependency between two software entities (classes, modules,
 26 interfaces, etc.) as the fact that the entities repeatedly change together during the histor-
 27 ical evolution of a software system. Since then, logical dependencies have been used in
 28 multiple areas of software engineering, most commonly in fault and change prediction.
 29 Besides the studies on how logical dependencies can help gain knowledge about software
 30 systems, some studies also focused on the interplay between logical and structural de-
 31 pendencies. Ajienka et al. and Olivia et al. studied the interplay between structural and
 32 logical dependencies, and they concluded that, in most cases, structural dependencies do
 33 not lead to logical dependencies [14], [15], [1]. The above affirmation is also supported
 34 by Lanza et al., who consider that logical dependencies are important because they can
 35 reveal dependencies that are not visible via code analysis [6].

36 In previous research, the *support* and *confidence* metrics were used to measure the
 37 strength of a logical dependency. The logical dependencies are commonly represented as
 38 directed association rules, the association rule between A and B ($A \rightarrow B$) means that
 39 changes in entity A cause changes in entity B, where A is the antecedent, and B is the
 40 consequent of the rule. The support metric counts the number of commits in which both
 41 entities of an association rule change together. The confidence metric is the ratio between
 42 the support metric and the total number of commits in which the antecedent of the rule
 43 was involved.

1 By applying different thresholds to the metrics presented above, the logical dependen-
 2 cies to further use where selected [15], [1], [30].

3 2.2. Current approach

4 To avoid confusion, we call *co-changing pairs* all the association rules of one system.
 5 The association rules are formed between two software entities that update together in
 6 the same commit. For example, a commit that contains seven entities will generate 21
 7 co-changing pairs ($C_k^n = \frac{n!}{k!(n-k)!} = \frac{7!}{2!(5)!} = 21$).

8 The *logical dependencies* are the association rules whose metrics fulfill certain condi-
 9 tions. So, the logical dependencies are a subset of the co-changing pairs.

10 The conditions that need to be met by a co-changing pair to be considered a logical
 11 dependency are called *filters*. Like in other research regarding logical dependencies, our
 12 filters are thresholds applied to the metrics of association rules.

13 Previously, we tried to filter logical dependencies from co-changing pairs by applying
 14 filters like the occurrence filter and commit size filter [23], [24]. The commit size filter,
 15 presented in more detail in section 2.3, and used by other authors [1], proved to be helpful,
 16 and it will be also used for this paper. But, we cannot say the same for the occurrence filter,
 17 the filter consisted of different thresholds applied to the support metric and proved to not
 18 work well for systems with few commits.

19 So currently, we aim to refine the filtering method with a new filter that can be applied
 20 for all sorts of commit history sizes. This new filter, presented in section 2.4, will be used
 21 together with the commit size filter to filter logical dependencies from co-changing pairs.
 22 The entire process of extracting co-changing pairs from the versioning system, filtering
 23 them to obtain logical dependencies, and exporting the results is done with a tool written
 24 in Python. The workflow is presented in figure 1.

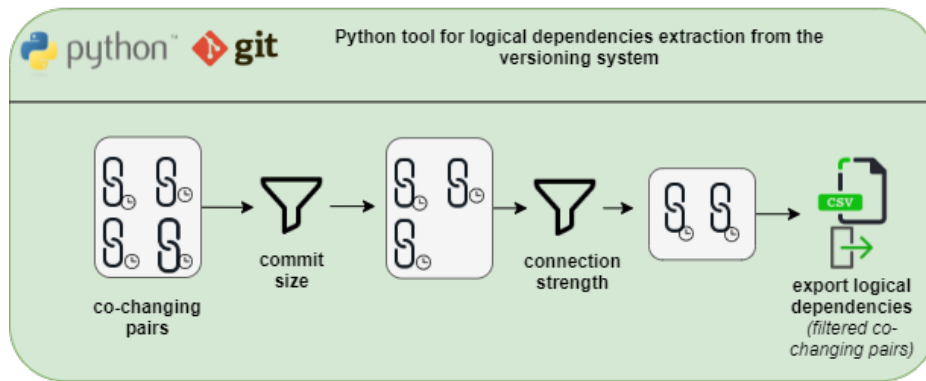


Fig. 1. Workflow for logical dependencies extraction.

2.3. Commit size filter

The commit size filter filters out all co-changing pairs from commits with more than 10 files changed. We consider that commits with more than 10 files changed tend to be code unrelated; we studied the commit size trend from several git open-source repositories, and we concluded that most of the commits contain less than ten files. On average, only 10 % of the total commits have more than ten files changed.

This filter will also prevent the volume of data processed from going out of proportion. In some of the repositories studied, we found commits with more than 1000 files; these commits could generate over half a million co-changing pairs if the commit size filter is not applied.

2.4. Connection strength filter

The connection strength filter is new for our research regarding logical dependencies identification, and it is based on our experience with the occurrence filter. An important conclusion drawn from the results obtained with the occurrence filter is that setting a hard threshold for a filter is not always a good idea. A certain threshold can work well with a medium/large-sized system but, when applied to a small-sized system, can reduce the co-changes filtered to 0. To avoid this type of situation, we evaluated a filter that considers the system's specifications.

As we previously mentioned, a filter has two components: the metrics computed for each co-changing pair (association rule) and the threshold values. The connection strength filter's metrics derive from the support and the confidence metric.

For an association rule (co-changing pair) formed from the ascendent A and consequent B ($A \rightarrow B$), the support count is the total number of commits in which both entities are involved,

$$support(A \rightarrow B) = freq_{total\ commits}(A \cup B) \quad (1)$$

and the confidence is the ratio between the support and the frequency of the ascendent of the rule.

$$confidence(A \rightarrow B) = \frac{support(A \rightarrow B)}{freq_{total\ commits}(A)} \quad (2)$$

The only problem with the confidence metric, as it is defined above, is that it does not include the big picture of the system. The best value for the confidence metric is 1, meaning that in all commits in which entity A is present, entity B is also there. If, for example, we have a co-changing pair $A \rightarrow B$, and A updates only once in the entire history, and in that time, updates together with B, then the confidence metric associated with the co-changing pair will be 1 (the best value possible). That is not a fair value compared with other scenarios. For example, we can have the co-changing pair $A \rightarrow B$, and A updates 100 in the entire history from which 80 times updates together with B, leading to a confidence value of 0.8. Even though in the second scenario we have a confidence value smaller than in the first scenario, the second scenario could lead to a more trustworthy connection.

Figures 2 and 3 intend to offer the big picture of two systems, one small-sized (Ant) and one medium-sized(Hibernate). In both figures, the dots represent the maximum number of updates of one entity with another, and the blue line represents the average occurrence value of the system. It can be observed that both systems have multiple entities that update only once (the dark line at the bottom), meaning that we might have many confidence values of 1 (highest value possible) for entities that update only once together.

To consider also the big picture of the system, we defined a new metric for a co-changing pair (association rule), called the connection strength metric. The metric considers the mean value of all the occurrences between two entities from the system. Also, the values obtained are clipped between 0 and 100, where 100 is the best value to obtain and 0 is the worst.

$$systemfactorfor(A \rightarrow B) = \frac{support(A \rightarrow B)}{system\ mean} \quad (3)$$

$$strength(A \rightarrow B) = \frac{support(A \rightarrow B) * 100}{freq_{total\ commits}(A)} * system\ factor \quad (4)$$

By using the strength metric, if we consider again the two scenarios presented above, and a system mean value of 10, we will have the following values: for the scenario in which entity A and B update only once, and in that one update, update together, the strength metric value is 10. For the scenario in which entity A updates 100 in the entire history from which 80 times updates together with B, the strength metric value is 100.

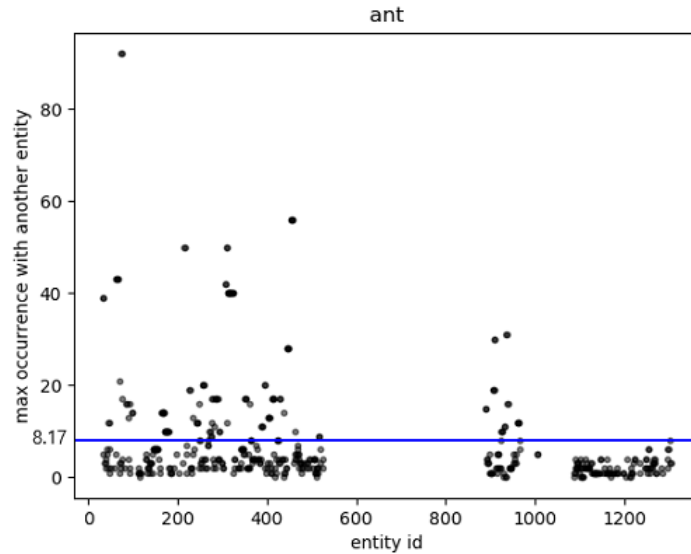


Fig. 2. Overview of the number of occurrences in Ant.

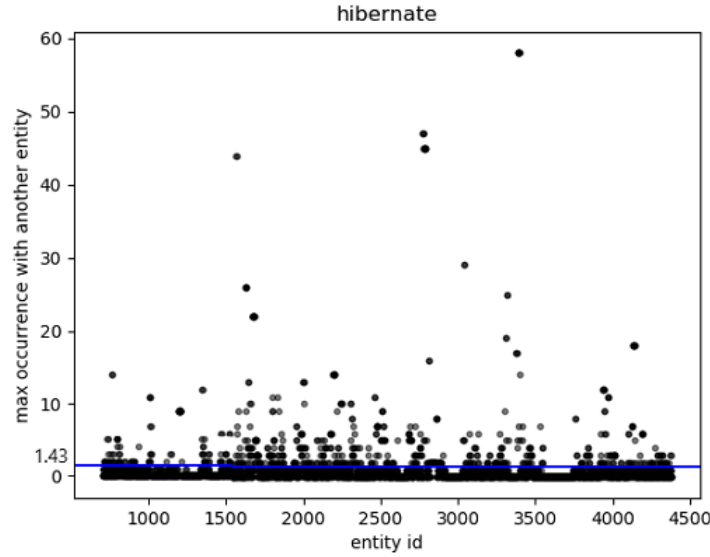


Fig. 3. Overview of the number of occurrences in Hibernate.

1 Since the values can vary from 0 to 100, the filter threshold values begin at 10 and are
 2 incremented by 10, until 100. We do not settle for one value because we want to see how
 3 the threshold values affect the number of remaining co-changing pairs and the output of
 4 their usage.

5 In figure 4 we plotted the number of structural dependencies, co-changing pairs before
 6 filtering, and co-changing pairs after filtering for two systems, one small-sized and one
 7 medium-sized. With the connection strength filter, the small-sized system didn't lose all
 8 the co-changing pairs once with the filtering. We compare the number of remaining co-
 9 changing pairs with the number of structural dependencies because according to surveys
 10 [21], [10], the main reason why logical dependencies (filtered co-changes) are not used
 11 together with structural dependencies is because of their size. So, it is essential to get
 12 an overview of the comparison between the co-changing pairs number and the structural
 13 dependencies number at each filtering step.

14 We call the co-changing pairs that remain after filtering, logical dependencies. After
 15 this step, we will use the logical dependencies obtained with different threshold values
 16 and see which threshold value performs the best. Up until now, we only looked at the size
 17 of the resulting logical dependencies and decided if a filter and its threshold are good or
 18 not. Now, we can also look at the results obtained by using the logical dependencies and
 19 decide.

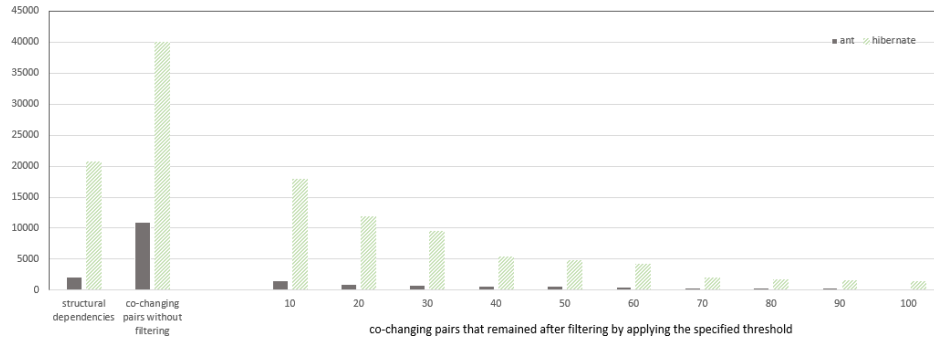


Fig. 4. Overview of the impact of connection strength filtering on the number of co-changing pairs.

3. Key classes: baseline versus current approach

3.1. State of the art

Zaidman et al. [29] 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 the 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” [25].

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 [18]. Thung et al. build on top of Osman et al.’s approach and add network metrics and optimistic classification in order to detect key classes [26].

Zaidman et al. use a web mining algorithm and dynamic analysis of the source code to identify the key classes [29].

3.2. Baseline approach

We use the research of I. Şora et al [17] as a baseline for our research involving the usage of logical dependencies to find key classes.

Şora et al. used the static analysis of the source code, a page ranking algorithm and other class attributes to find key classes [4], [16], [5], [22],[17]. The page ranking algorithm is a customization of PageRank, the algorithm used to rank web pages [19] and 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.

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

5 3.3. Current approach

6 The baseline approach uses a tool that takes as an input the source code of the system and
 7 applies ranking strategies to rank the classes according to their importance. We modified
 8 the tool used by the baseline approach to take also the logical dependencies as input; the
 9 rest of the workflow is the same as in the baseline approach (figure 5).

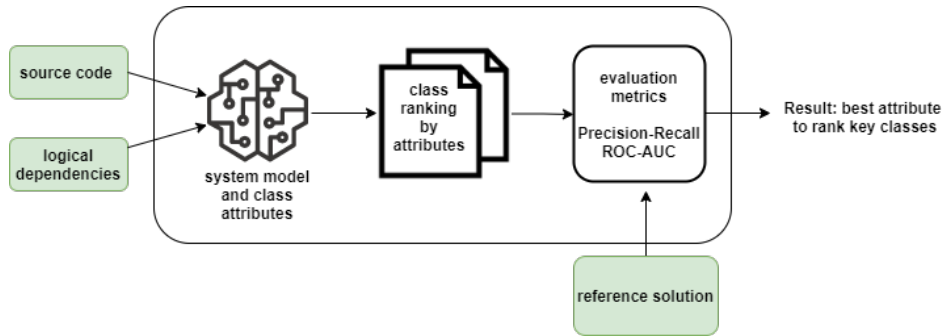


Fig. 5. Overview of the current approach.

10 In order to rank the classes according to their importance, different class metrics are
 11 used [8], [29], [20]. Below are some of the class metrics used in the baseline approach
 12 and our current research to rank the classes according to their importance.

13 The class metrics used can be separated into two categories: class connection metrics
 14 and class PageRank values. The class connection metrics are CONN-TOTAL-W, which is
 15 the total weight of all connections of the class, and CONN-TOTAL, the total number of
 16 distinct classes that a class uses or are used by a class [17].

17 Previous research used PageRank values computed on both directed and undirected,
 18 weighted and unweighted graphs. In the current research, we use the PR, which is the
 19 PageRank value computed on the directed and unweighted graph. The PR-U, which is
 20 the value computed on the undirected and unweighted graph, and PR-U2-W, the value
 21 computed on the weighted graph with back-recommendations [4], [16], [17], [22].

22 3.4. Metrics for results evaluation

23 To evaluate the quality of the key classes ranking algorithm, and solution produced, the
 24 key classes found by the algorithm are compared with a reference solution. The reference
 25 solution is extracted from the developer documentation. The classes mentioned in the
 26 documentation are considered key classes and form the reference solution (ground truth)
 27 used for validation [27].

For the comparison between both solutions, is used a classification model. The quality of the solution produced is evaluated by using the Receiver Operating Characteristic Area Under Curve (ROC-AUC) metric, a metric that evaluates the performance of a classification model.

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:

$$\text{True positive rate}(TPR) = \frac{TP}{TP + FN} \quad (5)$$

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:

$$\text{False positive rate}(FPR) = \frac{FP}{FP + TN} \quad (6)$$

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.

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 [18]. Thung et al. obtained an average ROC-AUC score of 0.825 [26] and Şora et al. (the baseline) obtained an average ROC-AUC score of 0.894 [17].

4. Experimental results using logical dependencies

As presented in section 3, the key class detection was previously done only by using the structural dependencies of the system. In this section, we will use the same tool used in the baseline approach presented in section 3, and we will add a new input to it, the logical dependencies.

In subsection 4.1, we present the data set used to generate new results and, in subsection 4.2 we present the previously obtained results. Subsection 4.3 presents the conclusions and results obtained by using logical and structural dependencies together, subsection 4.4 presents the conclusions and results obtained by using only logical dependencies. And finally, subsection 4.5 presents a comparison between results obtained with confidence metric versus results obtained with strength metric.

4.1. Data set used

The research of I. Sora et al. take into consideration structural dependencies that are extracted using static analysis techniques, and is performed on the object-oriented systems presented in table 1 [17].

The requirements for a system to qualify as suited for investigations using logical dependencies are: has to be on GitHub, has to have releases to identify a specific version (previous research was done only on specific releases), and also, has to have a significant number of commits. From the total of 14 object-oriented systems listed in the baseline [17], 13 of them have repositories in Github 1. And from the found repositories, only 6 repositories have the same release tag as the specified version in previous research. 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 1 Apache Ant is the most used and analyzed in other works [24], [9], [28], [13].

Table 1. Found systems and versions of the systems in GitHub.

ID	System	Version	Release Tag name	Commits number
S1	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

4.2. Measurements using only the baseline approach

In table 2 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 2. 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	

4.3. Measurements using combined structural and logical dependencies

The tool used in the baseline approach runs a graph-ranking algorithm on a graph that contains all 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 3, we modified the tool to be able to take logical dependencies as input. For this subsection's measurements, we add in the graph that contains structural dependencies the logical dependencies. Since it is a weighted graph, if a structural dependency is also a logical dependency, then the final weight of the connection is the sum of the current weight and the connection strength metric associated with the logical dependency.

In tables 3, 4, and 5, on each line, we have the computed key class metric, and on each column, the connection strength threshold that was applied to the co-changing pairs.

We started with logical dependencies that have a connection strength metric greater than 10, then we increased the value by 10 until 100. The last column of the table contains the results previously obtained by the tool by only using structural dependencies (the results presented in section 4.2).

The results obtained by combining structural and logical dependencies are close to the previously registered values but, in most cases, do not surpass them. Underlined are the values that are better than the previously registered values. We can also observe that for all three systems, the best values obtained are for connection strength between 40-70.

Table 3. Measurements for Ant using structural and logical dependencies combined

Metrics	≥ 10	≥ 20	≥ 30	≥ 40	≥ 50	≥ 60	≥ 70	≥ 80	≥ 90	≥ 100	Baseline
PR_U2_W	0.877	0.880	0.883	0.888	0.884	0.880	0.901	0.924	0.900	0.891	0.929
PR	<u>0.955</u>	<u>0.932</u>	<u>0.936</u>	<u>0.936</u>	0.880	0.884	0.887	0.889	0.888	0.890	0.855
PR_U	0.933	<u>0.937</u>	<u>0.936</u>	<u>0.939</u>	<u>0.940</u>	<u>0.939</u>	<u>0.941</u>	<u>0.943</u>	<u>0.942</u>	<u>0.940</u>	0.933
CON_T_W	0.841	0.839	0.836	0.838	0.835	0.849	0.859	0.872	0.870	0.874	0.934
CON_T	0.920	0.919	0.921	0.923	0.923	0.932	0.934	0.939	0.937	0.937	0.942

Some other details about the systems are presented in tables 6 and 7. In table 6 are the overlappings between structural and logical dependencies expressed in percentages. Each column represents the percentage of logical dependencies that are also structural, one column for each connection strength filter threshold. The values obtained confirm that indeed, the logical dependencies overlap with structural dependencies in a small percentage, and they must be treated as different dependencies.

Table 4. Measurements for Tomcat using structural and logical dependencies combined

Metrics	≥ 10	≥ 20	≥ 30	≥ 40	≥ 50	≥ 60	≥ 70	≥ 80	≥ 90	≥ 100	Baseline
PR_U2_W	0.862	0.883	0.898	0.901	0.907	0.909	0.910	0.916	0.918	0.918	0.923
PR	0.879	0.885	0.888	0.882	0.869	0.869	0.863	0.863	0.863	0.863	0.927
PR_U	0.924	0.930	0.931	0.932	0.932	0.932	0.932	0.932	0.932	0.932	0.932
CON_T_W	0.868	0.888	0.901	0.909	0.914	0.917	0.918	0.923	0.925	0.925	0.926
CON_T	0.925	0.934	0.937	0.938	0.938	0.938	0.938	0.938	0.938	0.938	0.939

Table 5. Measurements for Hibernate using structural and logical dependencies combined

Metrics	≥ 10	≥ 20	≥ 30	≥ 40	≥ 50	≥ 60	≥ 70	≥ 80	≥ 90	≥ 100	Baseline
PR_U2_W	0.903	0.909	0.916	0.928	0.930	0.932	0.946	0.947	0.947	0.949	0.958
PR	0.956	0.959	0.961	0.962	0.962	0.962	0.953	0.953	0.953	0.954	0.949
PR_U	0.937	0.941	0.943	0.947	0.948	0.948	0.950	0.950	0.950	0.950	0.951
CON_T_W	0.864	0.872	0.879	0.896	0.898	0.900	0.929	0.930	0.931	0.934	0.944
CON_T	0.920	0.927	0.932	0.940	0.940	0.940	0.945	0.945	0.945	0.945	0.946

1 In table 7 are the ratio numbers between structural dependencies and logical depen-
2 dependencies. We added this table to highlight how different the total number of both depen-
3 dependencies is.

Table 6. Percentage of logical dependencies that are also structural dependencies

System	≥ 10	≥ 20	≥ 30	≥ 40	≥ 50	≥ 60	≥ 70	≥ 80	≥ 90	≥ 100
Ant	17.628	19.872	20.461	20.858	21.078	23.913	24.688	21.807	20.000	19.776
Tomcat Catalina	10.331	14.931	15.862	16.221	16.427	16.302	16.598	18.336	19.207	19.149
Hibernate	8.005	8.971	9.755	12.060	12.348	12.254	18.426	19.105	18.836	19.371

Table 7. Ratio between structural and logical dependencies (SD/LD)

System	≥ 10	≥ 20	≥ 30	≥ 40	≥ 50	≥ 60	≥ 70	≥ 80	≥ 90	≥ 100
Ant	1.373	2.251	2.870	3.133	3.461	4.604	5.282	6.598	7.060	7.903
Tomcat Catalina	0.445	0.936	1.302	1.543	1.660	1.967	2.218	3.057	3.376	3.440
Hibernate	1.159	1.747	2.184	3.867	4.283	4.877	10.547	11.920	12.464	14.851

4 In most cases, for all systems, the results tend to become better once with the in-
5 creasing value of the connection strength threshold up until one point, after which, the
6 results obtained begin to drop. If we look at table A, we can observe that high values
7 for the connection strength filter reduce the number of logical dependencies. For exam-
8 ple, in Hibernate, the value 70 for the connection strength threshold makes the structural
9 dependencies outnumber 10 times the logical dependencies.

10 We can identify three scenarios based tables 3, 4, 5 and 7: in the first scenario, the
11 connection strength threshold is too small, and we remain with a lot of logical depen-

dencies after filtering. The high volume of logical dependencies introduced in the graph might cause an erroneous detection of the key classes, in consequence, less performing measurements/results. This affirmation is sustained by the fact that when the threshold begins to be more restrictive, and the total number of logical dependencies begins to decrease, the key classes detection starts to improve. The second scenario assumes that the connection strength threshold is too big, significantly decreasing the number of logical dependencies. In this case, the logical dependencies introduced in the graph are too few to improve the detection, and instead, will create noise in the graph. This leads us to the third scenario, in which the connection strength threshold is just right, not too small because this will introduce too many logical dependencies in the graph and produce less performing results. And not too high, because this will decrease too much the number of logical dependencies, producing less performing results.

The 'just right' value can differ from one system to another, depending on the size of the system. If we look at Ant (the smaller size system), we can see that the results begin to decrease sooner than for Hibernate. On average, all the systems perform well between 40 and 70 for the connection strength threshold value.

4.4. Measurements using only logical dependencies

Table 8. Measurements for Ant using only logical dependencies

Metrics	≥ 10	≥ 20	≥ 30	≥ 40	≥ 50	≥ 60	≥ 70	≥ 80	≥ 90	≥ 100	Baseline
PR_U2.W	0.679	0.695	0.738	0.799	0.822	0.883	0.890	0.901	0.846	0.862	0.929
PR	0.868	0.776	0.767	0.825	0.822	0.850	0.834	0.863	0.844	0.860	0.855
PR_U	0.801	0.792	0.757	0.806	0.822	0.854	0.856	0.867	0.848	0.860	0.933
CON_T.W	0.819	0.825	0.818	0.817	0.813	0.828	0.843	0.861	0.845	0.854	0.934
CON_T	0.856	0.836	0.819	0.803	0.801	0.816	0.831	0.855	0.840	0.851	0.942

Table 9. Measurements for Tomcat using only logical dependencies

Metrics	≥ 10	≥ 20	≥ 30	≥ 40	≥ 50	≥ 60	≥ 70	≥ 80	≥ 90	≥ 100	Baseline
PR_U2.W	0.775	0.810	0.834	0.828	0.819	0.815	0.805	0.816	0.820	0.813	0.923
PR	0.813	0.813	0.836	0.831	0.820	0.814	0.804	0.816	0.820	0.813	0.927
PR_U	0.772	0.815	0.835	0.831	0.820	0.814	0.804	0.816	0.819	0.813	0.932
CON_T.W	0.805	0.823	0.842	0.835	0.822	0.815	0.805	0.817	0.820	0.813	0.926
CON_T	0.787	0.812	0.835	0.832	0.821	0.814	0.804	0.817	0.820	0.813	0.939

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

In tables 8, 9, and 10, are presented the results obtained by using only logical dependencies to detect key classes.

Table 10. Measurements for Hibernate using only logical dependencies

Metrics	≥ 10	≥ 20	≥ 30	≥ 40	≥ 50	≥ 60	≥ 70	≥ 80	≥ 90	≥ 100	Baseline
PR_U2_W	0.721	0.733	0.743	0.700	0.700	0.703	0.741	0.742	0.744	0.751	0.958
PR	0.735	0.747	0.756	0.704	0.702	0.706	0.745	0.745	0.746	0.752	0.949
PR_U	0.738	0.740	0.749	0.699	0.701	0.704	0.744	0.743	0.745	0.752	0.951
CON_T_W	0.730	0.739	0.747	0.701	0.702	0.706	0.746	0.747	0.748	0.754	0.944
CON_T	0.740	0.743	0.750	0.700	0.700	0.704	0.746	0.746	0.747	0.753	0.946

Table 11. Ant key classes update overview.

Key class name	Commit count	Max occurrence with another entity
org.apache.tools.ant.Task	40	13
org.apache.tools.ant.Target	39	16
org.apache.tools.ant.IntrospectionHelper	52	43
org.apache.tools.ant.RuntimeConfigurable	38	16
org.apache.tools.ant.ProjectHelper	67	17
org.apache.tools.ant.TaskContainer	6	2
org.apache.tools.ant.Main	56	21
org.apache.tools.ant.UnknownElement	47	16
org.apache.tools.ProjectHelper2\$ElementHandler	21	14

Table 12. Hibernate key classes update overview.

Key class name	Commit count	Max occurrence with another entity
org.hibernate.Query	9	1
org.hibernate.engine.spi.SessionFactoryImplementor	26	10
org.hibernate.SessionFactory	20	3
org.hibernate.mapping.Table	39	25
org.hibernate.criterion.Projection	2	0
org.hibernate.criterion.Criterion	2	0
org.hibernate.engine.spi.SessionImplementor	16	2
org.hibernate.cfg.Configuration	88	9
org.hibernate.mapping.Column	16	3
org.hibernate.type.Type	10	0
org.hibernate.Transaction	9	0
org.hibernate.engine.ConnectionProvider	2	0
org.hibernate.Session	25	14
org.hibernate.Criteria	10	1

1 The measurements obtained are not as good as using logical and structural dependen-
2 cies combined or using only structural dependencies. But the values obtained are above
3 0.5, which means that a good part of the key classes is detected by only using logical
4 dependencies. As mentioned in section 3.4, a classifier is good if it has the ROC-AUC
5 value as close to 1 as possible.

6 One explanation for the less performing results is that the key classes may have a
7 better design than the rest of the classes, which means that are less prone to change. If the

1 key classes are less prone to change, then the associated connection strength metric is less
2 than for other entities.

3 Tables 11 and 12, provide us a better overview of the update behavior of key classes
4 in the versioning system. For both cases, some key classes change a lot in the versioning
5 system, for example, Configuration for Hibernate and ProjectHelper for Ant. Also, some
6 classes create strong connections with other entities, like IntrospectionHelper for Ant and
7 Table for Hibernate. But, in most cases, the key classes are not the entities that update the
8 most in the versioning system. So, by setting a too high connection strength threshold, we
9 risk filtering out the key classes.

10 4.5. Comparison between strength and confidence metrics

11 5. Conclusions

12 We defined the logical dependencies as filtered co-changing pairs extracted from the ver-
13 sioning system. The filters applied to the co-changing pairs are the following: the filter
14 based on commit size and the filter based on connection strength. If a co-changing pair
15 has associated metrics that surpass the threshold values settled for the filters, it is called a
16 logical dependency.

17 In section 4 we approached two scenarios to detect key classes by using logical de-
18 pendencies. In the first scenario, we used logical dependencies together with structural
19 dependencies and in the second, we used only logical dependencies to detect the key
20 classes. We modified the tool used in the baseline approach to use also logical dependen-
21 cies, and then we performed the key class identification.

22 Based on the results obtained, compared with the baseline results, we did saw a slight
23 improvement in key class detection when both logical and structural dependencies were
24 used together, the best results were obtained with a connection strength threshold of 40-
25 70. When we used only logical dependencies to detect key classes, the results were less
26 performing than using only structural or structural and logical dependencies combined.

27 As we mentioned in section 3, also other researchers tried to identify the key classes,
28 and even though the approaches are not the same, most of them have used the ROC-AUC
29 metric to evaluate the quality of the results, same as us. Osman et al. obtained in their
30 research an average Area Under the Receiver Operating Characteristic Curve (ROC-AUC)
31 score of 0.750 [18]. Thung et al. obtained an average ROC-AUC score of 0.825 [26] and
32 Şora et al. (the baseline approach) obtained an average ROC-AUC score of 0.894 [17].

33 In the current research, we obtained an average ROC-AUC score of 0.916 when us-
34 ing logical and structural dependencies combined and a score of 0.791 when using only
35 logical dependencies to detect key classes.

36 In conclusion, when using both dependencies combined, we can obtain a slightly bet-
37 ter ROC-AUC score than the one from the baseline approach. And, when using only logi-
38 cal dependencies, even though we do not obtain a better score than the baseline approach,
39 we obtain results that can be compared with results obtained by other researchers [18],
40 being almost equal.

41 The advantage of using only logical dependencies in key class detection is that it
42 only uses data extracted from the versioning system and can be generalized to various
43 programming languages.

References

1. Ajenka, N., Capiluppi, A.: Understanding the interplay between the logical and structural coupling of software classes. *Journal of Systems and Software* 134, 120–137 (2017), <https://doi.org/10.1016/j.jss.2017.08.042>
2. Ajenka, N., Capiluppi, A., Counsell, S.: An empirical study on the interplay between semantic coupling and co-change of software classes. *Empirical Software Engineering* 23(3), 1791–1825 (2018), <https://doi.org/10.1007/s10664-017-9569-2>
3. Cataldo, M., Mockus, A., Roberts, J.A., Herbsleb, J.D.: Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering* 35, 864–878 (2009)
4. Şora, I.: 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*. pp. 122–140. Springer International Publishing (2015)
5. Şora, I.: Helping program comprehension of large software systems by identifying their most important classes. In: Maciaszek, L.A., Filipe, J. (eds.) *Evaluation of Novel Approaches to Software Engineering*. pp. 122–140. Springer International Publishing, Cham (2016)
6. D’Ambros, M., Lanza, M., Lungu, M.: The evolution radar: Visualizing integrated logical coupling information. pp. 26–32 (01 2006)
7. D’Ambros, M., Lanza, M., Lungu, M.: Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering* 35(5), 720–735 (2009)
8. Ding, Y., Li, B., He, P.: An improved approach to identifying key classes in weighted software network. *Mathematical Problems in Engineering* 2016, 1–9 (2016)
9. do Nascimento Vale, L., de A. Maia, M.: Keele: Mining key architecturally relevant classes using dynamic analysis. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. pp. 566–570 (2015)
10. Ducasse, S., Pollet, D.: Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering* 35(4), 573–591 (July 2009)
11. Gall, H., Hajek, K., Jazayeri, M.: Detection of logical coupling based on product release history. In: *Proceedings of the International Conference on Software Maintenance*. pp. 190–. *ICSM ’98*, IEEE Computer Society, Washington, DC, USA (1998), <http://dl.acm.org/citation.cfm?id=850947.853338>
12. Graves, T., Karr, A., Marron, J., Siy, H.: Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering* 26(7), 653–661 (2000)
13. Kamran, M., Ali, M., Akbar, B.: Identification of core architecture classes for object-oriented software systems. *Journal of Applied Computer Science & Mathematics* 10, 21–25 (2016)
14. Oliva, G.A., Gerosa, M.A.: On the interplay between structural and logical dependencies in open-source software. In: *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering*. pp. 144–153. *SBES ’11*, IEEE Computer Society, Washington, DC, USA (2011), <https://doi.org/10.1109/SBES.2011.39>
15. Oliva, G.A., Gerosa, M.A.: 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*. pp. 250–260 (2015), <https://doi.org/10.1109/ISSRE.2015.7381818>
16. Şora, I.: 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.* pp. 129–140. *INSTICC, SciTePress* (2015)
17. Şora, I., Chirila, C.B.: Finding key classes in object-oriented software systems by techniques based on static analysis. *Information and Software Technology* 116, 106176 (2019), <https://www.sciencedirect.com/science/article/pii/S0950584919301727>

- 1 18. Osman, M.H., Chaudron, M.R.V., v. d. Putten, P.: An analysis of machine learning algorithms
2 for condensing reverse engineered class diagrams. In: 2013 IEEE International Conference on
3 Software Maintenance. pp. 140–149 (2013)
- 4 19. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to
5 the web. Technical Report 1999-66, Stanford InfoLab (November 1999), <http://ilpubs.stanford.edu:8090/422/>, previous number = SIDL-WP-1999-0120
- 6 20. Pan, W., Song, B., Li, K., Zhang, K.: Identifying key classes in object-oriented soft-
7 ware using generalized k-core decomposition. *Future Generation Computer Systems* 81,
8 188–202 (2018), [https://www.sciencedirect.com/science/article/pii/](https://www.sciencedirect.com/science/article/pii/S0167739X17302492)
9 [S0167739X17302492](https://www.sciencedirect.com/science/article/pii/S0167739X17302492)
- 10 21. Shtern, M., Tzerpos, V.: Clustering methodologies for software engineering. *Adv. Soft. Eng.*
11 2012, 1:1–1:1 (Jan 2012), <http://dx.doi.org/10.1155/2012/792024>
- 12 22. Şora, I.: A PageRank based recommender system for identifying key classes in software sys-
13 tems. In: 2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelli-
14 gence and Informatics (SACI). pp. 495–500 (May 2015)
- 15 23. Stana, A.D., Şora, I.: Analyzing information from versioning systems to detect logical depen-
16 dencies in software systems. In: 2019 IEEE 13th International Symposium on Applied Com-
17 putational Intelligence and Informatics (SACI). pp. 000015–000020 (2019)
- 18 24. Stana, A.D., Şora, I.: Identifying logical dependencies from co-changing classes. In: Pro-
19 ceedings of the 14th International Conference on Evaluation of Novel Approaches to Software
20 Engineering - Volume 1: ENASE, pp. 486–493. INSTICC, SciTePress (2019)
- 21 25. Tahvildari, L., Kontogiannis, K.: Improving design quality using meta-pattern transformations:
22 a metric-based approach. *J. Softw. Maintenance Res. Pract.* 16, 331–361 (2004)
- 23 26. Thung, F., Lo, D., Osman, M.H., Chaudron, M.R.V.: Condensing class diagrams by analyz-
24 ing design and network metrics using optimistic classification. In: Proceedings of the 22nd
25 International Conference on Program Comprehension. p. 110–121. ICPC 2014, Association
26 for Computing Machinery, New York, NY, USA (2014), [https://doi.org/10.1145/](https://doi.org/10.1145/2597008.2597157)
27 [2597008.2597157](https://doi.org/10.1145/2597008.2597157)
- 28 27. Yang, X., Lo, D., Xia, X., Sun, J.: Condensing class diagrams with minimal manual labeling
29 cost. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMP-
30 SAC). vol. 1, pp. 22–31 (2016)
- 31 28. Zaidman, A., Calders, T., Demeyer, S., Paredaens, J.: Applying webmining techniques to exe-
32 cution traces to support the program comprehension process. In: Ninth European Conference
33 on Software Maintenance and Reengineering. pp. 134–142 (2005)
- 34 29. Zaidman, A., Demeyer, S.: Automatic identification of key classes in a software system using
35 webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*
36 20(6), 387–417 (2008)
- 37 30. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide
38 software changes. In: Proceedings of the 26th International Conference on Software Engi-
39 neering. pp. 563–572. ICSE '04, IEEE Computer Society, Washington, DC, USA (2004),
40 <http://dl.acm.org/citation.cfm?id=998675.999460>
- 41