Computer Science and Information Systems 00(0):0000–0000 https://doi.org/10.2298/CSIS123456789X

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

Adelina Diana Stana[1] and Ioana Şora[2]

[1] Stana Adelina Diana
Politehnica University, Piaţa Victoriei Nr. 2, ; 300006 Timişoara, jud. Timiş, România
stana.adelina.diana@gmail.com
[2] Şora Ioana
Politehnica University, Piaţa Victoriei Nr. 2, ; 300006 Timişoara, jud. Timiş, 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, and 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 are not on pair with structural dependencies results, 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 new. Previous works have used version control information to detect design issues [27], predict fault incidence among modules [12], [3] or guide software changes [9], [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 and can

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

mean that the entities from the source code files that change together, evolve together, and might depend on one another. Studies show that dependency relationships found in the source code overlap only in a small percentage with dependency relationships found in the version control system, and suggest that these two types of relationships can be used together [13], [1]. But, in practice, dependencies extracted from the version management system are rarely used because of the size of the information extracted [19]. A relatively small source code repository with roundabout one thousand commits can lead to millions of connections. In this paper, by applying a set of filters with different thresholds to the information extracted, we intend to speed up the processing time, reduce the size of connections extracted from the version control and increase the confidence that the connections obtained might be related. In order to validate the results obtained, and to see if the filtering methods had or not had a favorable effect on the final result, we want to identify the key classes of different systems. The identification of key classes has been previously performed by using structural dependencies, so we intend to use the results obtained together with structural dependencies, and also separate, and see how the final results fluctuate.

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

2. The concept of logical dependencies

2.1. State of the art

The concept of logical coupling (dependency) was first introduced by Gall et al. [11]. They defined the logical dependency between two software entities (classes, modules, interfaces, etc.) as the fact that the entities repeatedly change together during the historical evolution of a software system. Since then, logical dependencies have been used in multiple areas of software engineering, most commonly in fault and change prediction. Besides the studies on how logical dependencies can help gain knowledge about software systems, some studies also focused on the interplay between logical and structural dependencies. Ajienka et al. and Olivia et al. studied the interplay between structural and logical dependencies, and they concluded that, in most cases, structural dependencies do not lead to logical dependencies [13], [14], [1]. The above affirmation is also supported by Lanza et al., who consider that logical dependencies are important because they can reveal dependencies that are not visible via code analysis [8].

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

By appling different thresholds to the metrics presented above, the logical dependencies to further use were selected [14], [1], [27].

2.2. Current approach

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

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

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

Previously, we tried to filter logical dependencies from co-changing pairs by applying filters like the occurrence filter and commit size filter [21], [22]. The commit size filter, presented in more detail in section 2.3, and used by other authors [1], proved to be helpful, and it will be also used for this paper. But we cannot say the same for the occurrence filter. The filter consisted of different thresholds applied to the support metric and proved to not work well for systems with few commits.

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

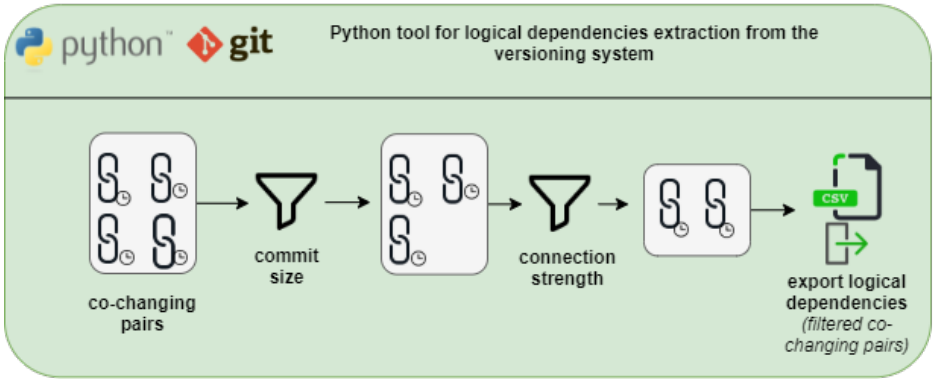

Fig. 1. Workflow for logical dependencies extraction.

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 serval git open-source repositories, and we concluded that most of the commits contain less than 10 files. On average, only 10% of the total commits have more than 10 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 [22], [21].

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, it can reduce the co-changes filtered to 0. To avoid this kind 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 metric derive from the support and the confidence metrics.

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

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

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

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

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 take into account the big picture of the system, we defined a new metric for a co-changing pair (association rule), called the *connection strength metric*.

The strength metric is computing the same ratio as the confidence metric, a.k.a the ratio between the support metric and the frequency of the antecedent. And additionally, it multiplies it with a system factor and with 100. The *system factor* calculates the ratio between the support metric and the system mean value for updates. The *system mean* is the mean value of all the support values for all the association rules from the system. We multiply with 100 because we want to scale the metric values to structural dependencies metric values that have, in most cases, supraunitary values. The values obtained are clipped between 0 and 100, where 100 is the best value and 0 is the worst.

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

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

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 the entities A and B update only once, and in that one update, they update together, the strength metric value is 10. For the scenario in which entity A updates 100 times in the entire history from which 80 times updates together with B, the strength metric value is 100.

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

In figure 4 we plotted for two systems (one small-sized and one medium-sized) the number of structural dependencies, co-changing pairs before filtering, and co-changing pairs after filtering. With the connection strength filter, the small-sized system didn't lose all the co-changing pairs once with the filtering. We compare the number of remaining co-changing pairs with the number of structural dependencies because, according to surveys [19], [10], the main reason why logical dependencies (filtered co-changes) are not used together with structural dependencies is their size. So, it is essential to get an overview of the comparison between the number of co-changing pairs and the number of structural dependencies at each filtering step.

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

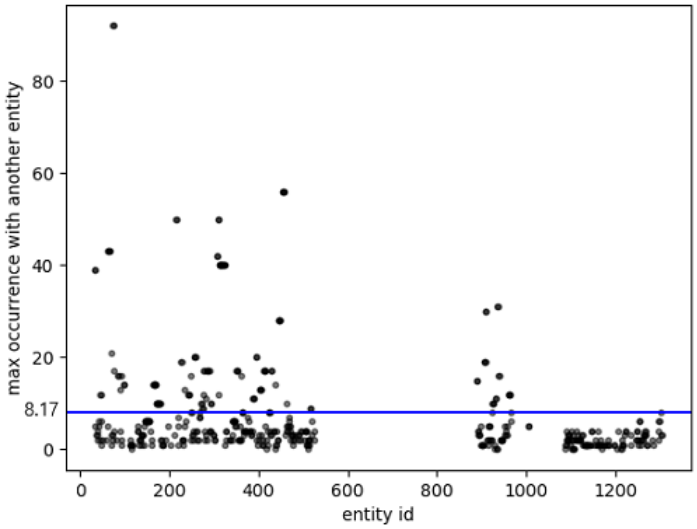

Fig. 2. Overview of the number of occurences in Ant.

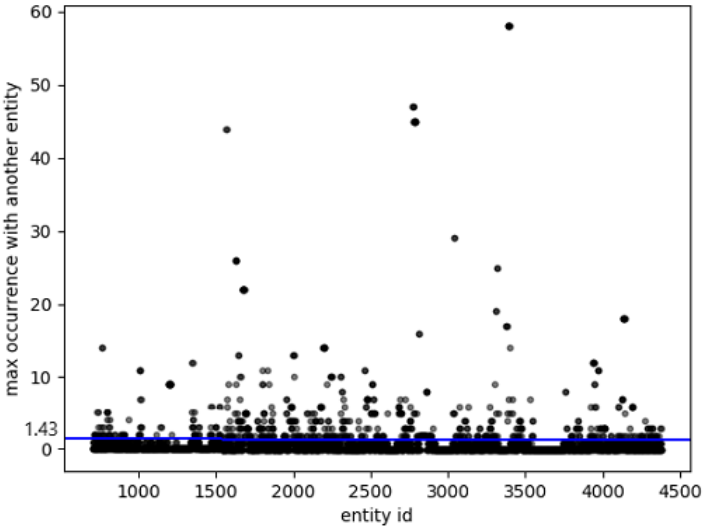

Fig. 3. Overview of the number of occurences in Hibernate.

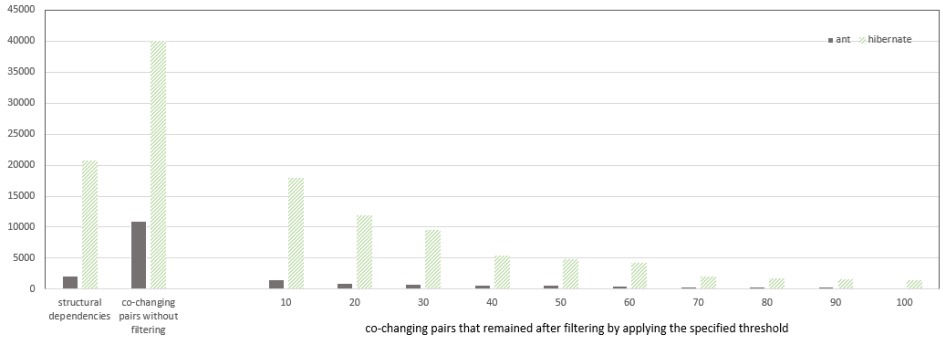

Fig. 4. Overview of the impact of connection strength filtering on the number of co-changing pairs.

3. Key classes: baseline versus current approach

3.1. State of the art

Zaidman et al. [26] 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" [23].

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 [17]. Thung et al. built on top of Osman et al.'s approach and added network metrics and optimistic classification to detect key classes [24].

Zaidman et al. used a web mining algorithm and dynamic analysis of the source code to identify the key classes [26].

3.2. Baseline approach

We use the research of I. Şora et al. [16] 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 [5], [15], [6], [20],[16]. The page ranking algorithm is a customization of PageRank, the algorithm used to rank web pages [18], and it works based on a recommendation system. If one node has a connection with another node, then it recommends the second node. In previous research, 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, according to their importance. To identify the important classes from the rest, a threshold for the top classes from the top of the ranking is set. We call this TOP threshold, and its value can range from 1 to the total number of classes found in the system.

3.3. Current approach

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. We modified the tool used by the baseline approach to take also the logical dependencies as input; the rest of the workflow is the same as in the baseline approach (figure 5).

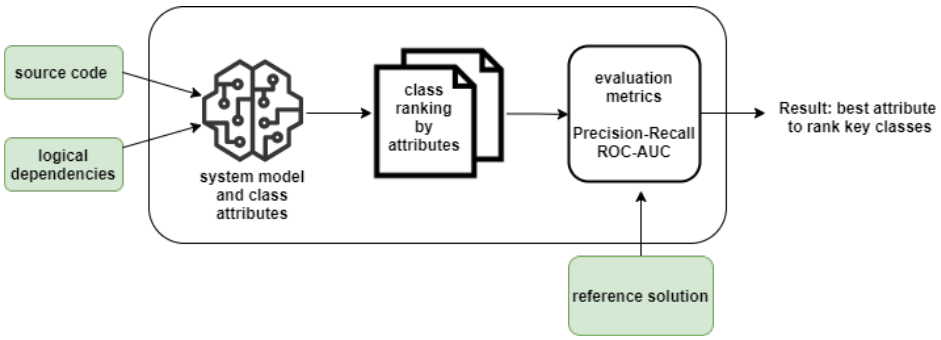

Fig. 5. Overview of the current approach.

Below are some of the class metrics used in the baseline approach and in our current research to rank the classes according to their importance.

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

Previous research used PageRank values computed on both directed and undirected, weighted and unweighted graphs. In the current research, we use the PR, which is the PageRank value computed on the directed and unweighted graph, the PR-U, which is the value computed on the undirected and unweighted graph, and the PR-U2-W, the value computed on the weighted graph with back-recommendations [5], [15], [16], [20].

3.4. 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. The classes mentioned in the documentation are considered key classes and form the reference solution (ground truth) used for validation [25].

For the comparison between both solutions, a classification model is used. 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:

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

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:

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

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 are not found in the TOP ranked classes.

In related research, 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 ROC-AUC score of 0.750 [17]. Thung et al. obtained an average ROC-AUC score of 0.825 [24] and Şora et al. (the baseline) obtained an average ROC-AUC score of 0.894 [16].

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 use the same tool used in the baseline approach presented in section 3, and we 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. takes into consideration structural dependencies that were extracted using static analysis techniques, and were performed on the object-oriented systems presented in table 1 [16].

The requirements for a system to qualify as suited for investigations using logical dependencies are: has to be version controlled by Git, has to have releases for different code versions (previous research was done only on specific versions), and also has to have a significant number of commits. From the total of 14 object-oriented systems listed in the baseline [16], 13 of them have repositories in git 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 (more than 5000 commits), so we concluded to use only 3 systems from the initial candidates for key classes detection using logical dependencies: Ant, Hibernate, and Tomcat Catalina.

Table 1. Systems and versions of the systems found in Git.

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

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.

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

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

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 take sturcural and logical dependencies as input. For this subsection's measurements, we add the logical dependencies in the graph that contains all structural 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 weight computed for the structural dependency and the connection strength metric associated with the logical dependency.

In tables 3, 4, and 5, on each line, we have the computed the key class metric generated with logical dependencies extracted with the connection strength threshold that is specified in the colums header.

We started with logical dependencies that have a connection strength metric greater than 10, then we repeteadly increased the value by 10 until we reached 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 observe that for all 3 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	0.955	0.932	0.936	0.936	0.880	0.884	0.887	0.889	0.888	0.890	0.855
PR_U	0.933	0.937	0.936	0.939	0.940	0.939	0.941	0.943	0.942	0.940	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. 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.

In table 7 are the ratio numbers between structural dependencies and logical dependencies. We added this table to highlight how different the numbers of both dependencies are.

Table 4. Measurements for Tomcat Catalina 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

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

In most cases, for all systems, the results tend to become better once with increasing the value of the connection strength threshold up until one point, after which the results obtained begin to drop. If we look at table 6, we can observe that the bigger the threshold for the connection strength filter, the smaller the number of total logical dependencies becomes. For example, in Hibernate, the value 70 for the connection strength threshold makes the structural dependencies outnumber 10 times the logical dependencies.

We can identify 3 scenarios based on tables 3, 4, 5 and 7. In the 1^{st} scenario, the connection strength threshold is too small, and we remain with a lot of logical dependencies 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 2^{nd} 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 and less performing results. This leads us to the 3^{rd} scenario, in which the connection strength threshold is 'just right'. Not too small, because it will introduce too many logical dependencies in the graph and produce less performing results. And not too high, because it 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

In the previous subsection, we added the logical and structural dependencies in the graph based on which the ranking algorithm works. Currently, we 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.

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

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 Catalina 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

One 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

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

key classes are less prone to change, then the associated connection strength metric has a lower value than for other entities.

Tables 11 and 12, provide us a better overview of the update behavior of key classes in the versioning system. The commit count column presents the number of commits in which the entity was involved. The column 'Max occurrence with another entity' contains the maximum number of updates with another entity from the system (the strongest connection with another entity).

It can be observed that some key classes change a lot in the versioning system, for example, Configuration for Hibernate and ProjectHelper for Ant. Also, some classes create strong connections with other entities, like IntrospectionHelper for Ant and Table for Hibernate. But, in most cases, the key classes are not the entities that update the most in the versioning system. So, by setting too high the connection strength threshold, we risk filtering out the key classes.

Table 11. Ant key classes update overview.

Key class name	Commit count	Max occurence 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

4.5. Comparison between results obtained with strength versus confidence metric

As mentioned in section 2.4, we did not use the confidence metric because it does not consider the big picture of the system. A co-changing pair $A \rightarrow B$, where A updates only once in the entire history, and when it updates, it updates together with B, will have the best confidence value that we can get. This is why we introduced the strength metric, to balance the metric in the favor of those which update more frequently. Since both metrics require the same inputs and only the calculation method is different, we computed with

Table 12. Hibernate key classes update overview.

Key class name	Commit count	Max occurence 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

our tool the confidence metric and applied the same threshold to it as to the strength metric. The only difference from how other authors computed the metric is that we multiplied its value by 100. So, the confidence values can fluctuate between 0 and 100. In the graph used by the key classes detection tool, the structural dependencies weights are supraunitary values. So, we multiplied with 100 the confidence value to scale it to the structural dependencies weights. Otherwise, if we add a subunitary value (confidence value) to a high value (the structural weight), it will not make a difference, so we will not be able to see the impact of the logical dependencies in the graph.

Table 13. Average results obtained with strength versus confidence metric.

Metric used	Using	
	Only logical dependencies	Structural and logical dependencies
Average values obtained for all systems		
strength	0.791	0.916
confidence	0.731	0.893
Average values obtained for Ant		
strength	0.826	0.903
confidence	0.741	0.873
Average values obtained for Tomcat Catalina		
strength	0.816	0.910
confidence	0.752	0.878
Average values obtained for Hibernate		
strength	0.732	0.935
confidence	0.699	0.929

The comparison between the average values obtained by using the confidence metric and the strength metric, can be found in table 13. As we expected, based on the results,

we can say that the connection strength metric is more suited for logical dependencies detection. So, by considering the mean update frequency of the entire system in the filtering process, we improve the detection of logical dependencies.

5. Conclusions

We defined the logical dependencies as filtered co-changing pairs extracted from the versioning system.

The filters applied to the co-changing pairs are the filter based on commit size and the filter based on connection strength. If a co-changing pair has associated metrics that surpass the threshold values settled for the filters mentioned, then it is called a logical dependency.

In section 4 we approached two scenarios to detect key classes by using logical dependencies. In the 1^{st} scenario, we used logical dependencies together with structural dependencies and in the 2^{nd}, we used only logical dependencies to detect the key classes. We modified the tool used in the baseline approach to use also logical dependencies, and then we performed the key class identification.

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

As we mentioned in section 3, also other researchers tried to identify the key classes, and even though the approaches were not the same, most of them used the ROC-AUC metric to evaluate the quality of the results, same as us. Osman et al. obtained in their research an average ROC-AUC score of 0.750 [17]. Thung et al. obtained an average ROC-AUC score of 0.825 [24] and Şora et al. (our baseline approach) obtained an average ROC-AUC score of 0.894 [16].

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

So, when using both dependencies combined, we can obtain a slightly better ROC-AUC score than the one from the baseline approach. And, when using only logical dependencies, even though we do not obtain a better score than the baseline approach, we obtain results that can be compared with results obtained by other researchers [17], being almost equal.

To sum up briefly the findings of this paper, we consider that the big advantage of using only logical dependencies in key class detection is that it only uses data extracted from the versioning system, and can be generalized to various programming languages. In the future, we want to check if there are also other areas that can be improved by using logical dependencies, like software clustering [7], [19], [4].

References

1. Ajienka, 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. Ajienka, 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.: 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). pp. 45–54. INSTICC, SciTePress (2013)
5. 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)
6. 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)
7. Şora, I., Glodean, G., Gligor, M.: 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. pp. 259–264 (May 2010)
8. D'Ambros, M., Lanza, M., Lungu, M.: The evolution radar: Visualizing integrated logical coupling information. pp. 26–32 (01 2006)
9. 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)
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. 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
14. 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
15. 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)
16. 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
17. Osman, M.H., Chaudron, M.R.V., v. d. Putten, P.: An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In: 2013 IEEE International Conference on Software Maintenance. pp. 140–149 (2013)
18. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab (November 1999), http://ilpubs.stanford.edu:8090/422/, previous number = SIDL-WP-1999-0120

19. Shtern, M., Tzerpos, V.: Clustering methodologies for software engineering. Adv. Soft. Eng. 2012, 1:1–1:1 (Jan 2012), http://dx.doi.org/10.1155/2012/792024
20. Şora, I.: 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). pp. 495–500 (May 2015)
21. Stana, A.D., Şora, I.: Analyzing information from versioning systems to detect logical dependencies in software systems. In: 2019 IEEE 13th International Symposium on Applied Computational Intelligence and Informatics (SACI). pp. 000015–000020 (2019)
22. Stana., A.D., Şora., I.: 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,. pp. 486–493. INSTICC, SciTePress (2019)
23. Tahvildari, L., Kontogiannis, K.: Improving design quality using meta-pattern transformations: a metric-based approach. J. Softw. Maintenance Res. Pract. 16, 331–361 (2004)
24. Thung, F., Lo, D., Osman, M.H., Chaudron, M.R.V.: Condensing class diagrams by analyzing design and network metrics using optimistic classification. In: Proceedings of the 22nd International Conference on Program Comprehension. p. 110–121. ICPC 2014, Association for Computing Machinery, New York, NY, USA (2014), https://doi.org/10.1145/2597008.2597157
25. Yang, X., Lo, D., Xia, X., Sun, J.: Condensing class diagrams with minimal manual labeling cost. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC). vol. 1, pp. 22–31 (2016)
26. Zaidman, A., Demeyer, S.: 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)
27. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering. pp. 563–572. ICSE '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=998675.999460