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, 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. This means that software systems that use dynamically typed languages such as JavaScript, Objective-C, Python or Ruby, for which the code dependencies are harder to obtain, can use the dependencies extracted from the version control to obtain 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 [31], 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 and can mean that the entities from the source code files that are changing 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

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

relationships found in the version control system, and suggest that these two types of relationships can be used together [14], [1]. But, in practice, dependencies extracted from the version management system are rarely used due to the size of the information extracted [21]. A relatively small source code repository with roundabout one thousand commits can lead to millions of connections. In this paper, 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 by applying a set of filters with different thresholds to the information extracted. 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 had 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 [11]. A logical dependency between two software entities (classes, modules, interfaces, etc.) signals that those entities in some way depend on one another. They defined the logical dependency between two modules as the fact that the modules repeatedly change together during the historical evolution of the software system.

Structural dependencies do not co-evolve, and vice-versa [14], [15], [1]

Lanza et all consider that LD might be useful because can reveal dependencies that are not visible via code analysis [6]

Previous research uses the frequency and the confidence measures [31], [1], [28]. Zimmermann et al use Apriori algorithm to filter the co-changes, but it took several days to complete the experiments.

2.2. Current approach

To define the logical dependencies, we have first to define the co-changing pairs. A *co-changing pair* are two software entities that update together in the same commit. For example, a commit that contains seven entities will generate 21 co-changing pairs ($C_k^n = \frac{n!}{k!(n-k)!} = \frac{7!}{2!(5)!} = 21$).

After extraction, some co-changing pairs are removed by passing them through various filters (filtering phase). The *logical dependencies* are those co-changing pairs that remain after the filtering phase. Another reason for the filtering is to decrease the size of processed data; a commit with 1030 entities will produce over half a million co-changing pairs. These kinds of commits are, in most cases, not code-related, so the co-changing pairs extracted from this kind of commit have nothing to do with the code.

Previously, we tried to obtain reliable logical dependencies from co-changing pairs by applying different types of filters like the occurrence filter and commit size filter [23], [24]. The main disadvantage of the occurrence filter is that it does not work well for systems with few commits.

Currently, we aim to refine the filtering method with a new filter that can be applied for all sorts of commit history sizes. This new filter will be used together with a filter previously explored, the commit size filter [24].

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 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

Ant number of classes: 458, and Hibernate 3671.

This filter is new for our research regarding logical dependencies and is based on the experience with the occurrence filter. One important conclusion drawn from the measurements 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. This made us realize that we need a filter that is computed according to the system's specifications.

We will call this filter, the connection strength filter and, it is applied after the commit size filter. To determine the connection strength, we first need to calculate the connection factors of both entities that form a co-changing pair.

Assuming that we have a co-changing pair formed by entities A and B, the connection factor of entity A with entity B is the percentage from the total commits involving A that contains entity B. The connection factor of entity B with entity A is the percentage from the total commits involving B also containing entity A.

$$connection\ factor\ for\ A = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ A} \tag{1}$$

$$connection\ factor\ for\ B = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ B} \tag{2}$$

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

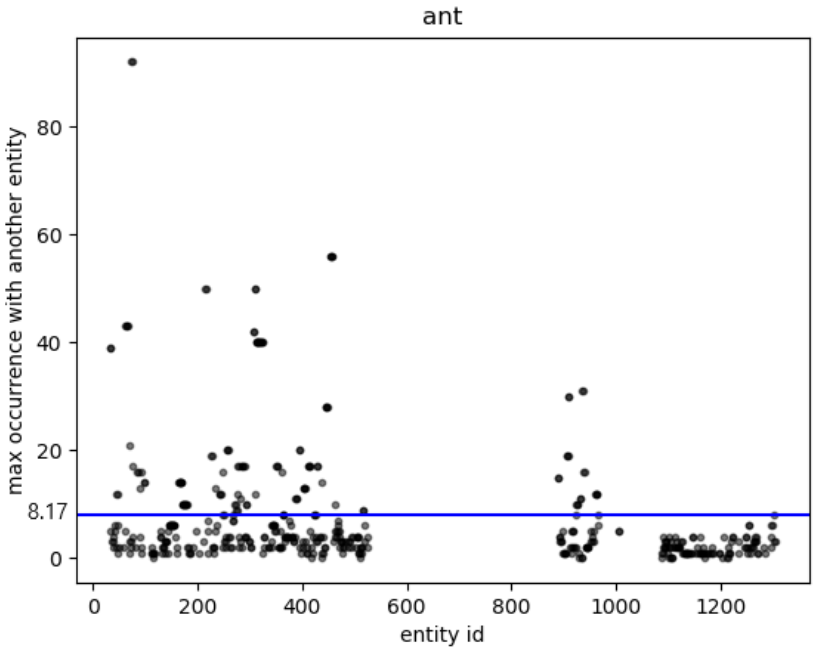

Fig. 1. Overview of the number of occurences in Ant.

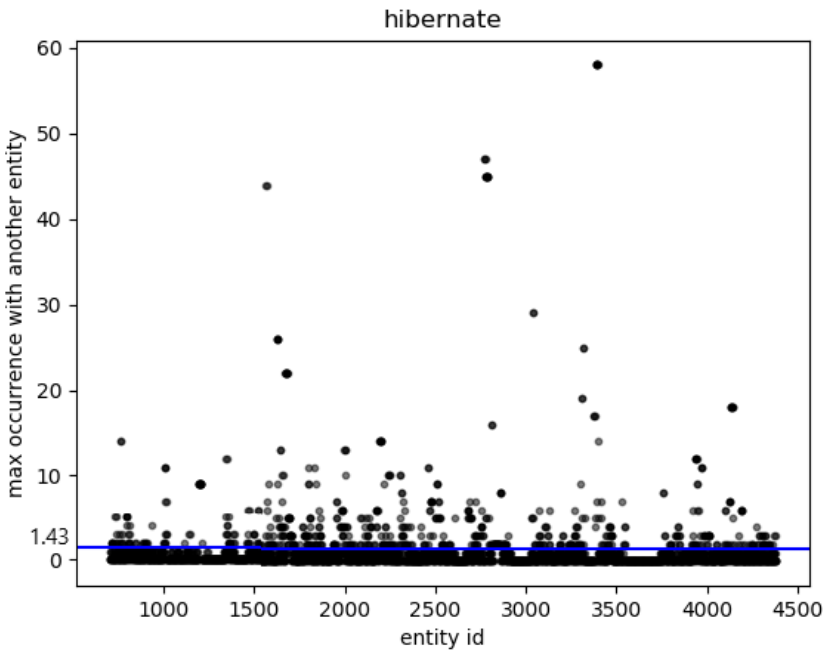

Fig. 2. Overview of the number of occurences in Ant.

As a practical example, if the pair formed by A and B update together 7 times and the total number of commits involving A is 20 and involving B is 7, then the factor for A is 35 and for B is 100. If the connection strength threshold is set to 50, that means that this co-changing pair will be filtered out because the factor of A is smaller than the threshold.

Since the factors can vary from 0 to 100, for this filter, we started with a threshold value of 10 and incremented it by 10 until we reached 100. And we want to see how the threshold value affects the number of co-changing pairs left.

In figure 3 we plot the number of structural dependencies, co-changing pairs before filtering, and co-changing pairs after filtering with different threshold values for two systems, one small-sized and one medium-sized. As can be observed, with this 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 [21], [10], the main reason why logical dependencies (a.k.a filtered co-changes) are not used together with structural dependencies is because of their size. So, it is essential to get an overview of the comparison between the co-changing pairs number and the structural dependencies number at each filtering step.

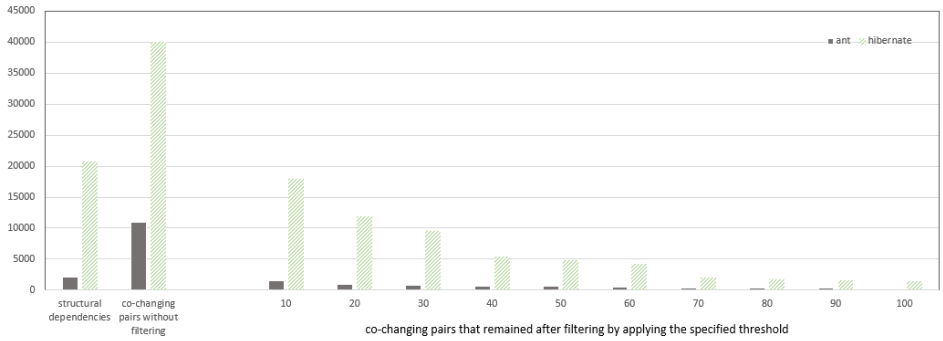

Fig. 3. Overview of the impact of connection strength filtering on the number of co-changing pairs.

The co-changing pairs that remain after filtering, can be called logical dependencies. The entire process of extracting co-changing pairs from the versioning system, filter them, and export the remaining ones is done with a tool written in Python. The tool's workflow is presented in figure 4.

After this step, we will continue our research by using 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 logical dependencies and decided if a filter is good or not. But now, we can also look at the results obtained by using the logical dependencies and decide.

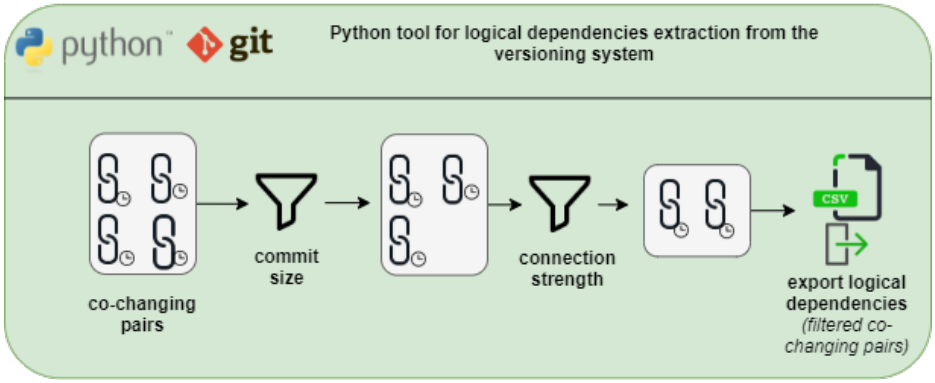

Fig. 4. Workflow for logical dependencies extraction.

3. Key classes: baseline versus current approach

3.1. State of the art

Zaidman et al [30] were the first to introduce the concept of key classes and it refers to classes that can be found in documents written to provide an architectural overview of the system or an introduction to the system structure. Tahvildari and Kontogiannis have a more detailed definition regarding key classes concept: "Usually, the most important concepts of a system are implemented by very few key classes which can be characterized by the specific properties. These classes, which we refer to as key classes, manage many other classes or use them in order to implement their functionality. The key classes are tightly coupled with other parts of the system. Additionally, they tend to be rather complex, since they implement much of the legacy system's functionality" [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. builds on top of Osman et al.'s approach and adds network metrics and optimistic classification in order to detect key classes [26].

Zaidman et al. use a webmining algorithm and dynamic analysis of the source code to identify the key classes [30].

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.

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

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).

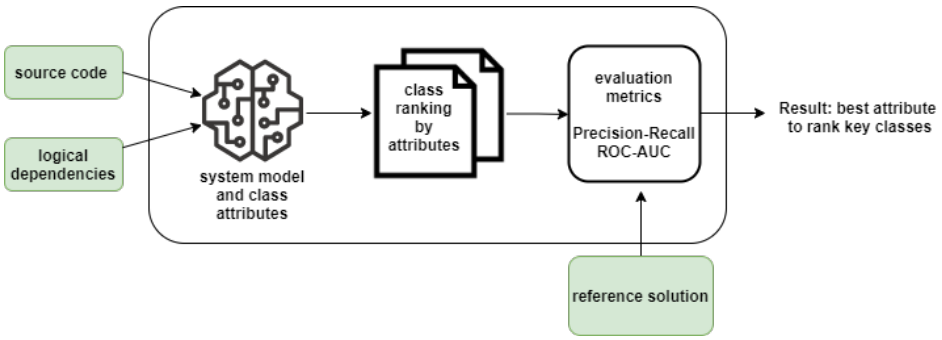

Fig. 5. Overview of the current approach.

In order to rank the classes according to their importance, different class metrics are used [8], [30], [20]. Below are some of the class metrics used in the baseline approach and our current research to rank the classes according to their importance. We use only a subset of the metrics used in previous research because the extracted logical dependencies are undirected.

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 used by a class [17].

Previous research used PageRank values computed on both directed and undirected, weighted and unweighted graphs. In the current research, we used 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 PR-U2-W, the value computed on the weighted graph with back-recommendations [4], [16], [17], [22].

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. Classes mentioned in the documentation are considered key classes and form the reference solution (ground truth) used for validation [27].

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:

$$True\ positive\ rate(TPR) = \frac{TP}{TP + FN} \tag{3}$$

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

$$False\ positive\ rate(FPR) = \frac{FP}{FP + TN} \tag{4}$$

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

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

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

4.1. Data set used

In this section, we will look over all the systems studied in the baseline research presented in section 3, and we will try to identify the systems that could be used also in our current research involving logical dependencies.

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

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 paper [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], [29], [13].

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

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

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. The graph used contains the structural dependencies extracted from static source code analysis. Each edge in the graph represents a dependency, the entities that form a structural dependency are represented as vertices in the graph. As mentioned in section 3, we modified the tool to read also logical dependencies and add them to the graph. In this section, we add in the graph the logical dependencies together with the structural dependencies.

In tables 3, 4, and 5, on each line, we have the metric that is calculated and on each column, we have the connection strength threshold that was applied to the logical dependencies used in identifying the key classes. We started with logical dependencies that have a connection strength greater than 10%, which means that in at least 10% of the commits involving A or B, A and B update together. Then we increased the threshold value by 10 until we remained only with entities that update in all the commits together. The last column contains the results obtained previously by the tool by only using structural dependencies.

As for the new results obtained by combining structural and logical dependencies, the values close to the previously registered values, but did not surpass them, are underlined with one line. And, with two lines, are underlined the values that are better than the previously registered values. At this step, we can also observe that for all three systems measured in tables 3, 4, and 5, the best values obtained are for connection strength between 40-70%.

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

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

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

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

Table 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 tables 3, 4, and 5 we can also observe that the measurements at the beginning are smaller than the rest. Once with the increasing of the threshold value also the measurements begin to increase. Meaning that better results for key class detection are found. The best measurements are when the threshold value is between 40 and 60, after that, the measurements tend to decrease a little bit and stay at that fixed value.

A possible explanation of the results fluctuation and then capping is that if we are looking at table 7 we can see that at the beginning, the total number of logical dependencies used is close to the number of existing structural dependencies. The high volume of logical dependencies introduced might cause an erroneous detection of the key classes, in consequence, smaller measurements. When the threshold begins to be more restrictive and the total number of logical dependencies used begins to decrease, the key classes detection starts to improve. This improvement stops after the threshold value reaches 60%. If we look again at table 7 we can see that after 60% the number of structural dependencies outnumbers the number of logical dependencies up to 124 times in some cases. In addition, if we look at table 6 we can see that the remaining logical dependencies overlap a lot with the structural dependencies, so we are not introducing too much new information.

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

4.4. Measurements using only logical dependencies

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

In tables 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 using logical and structural dependencies combined or using only structural dependencies. But, all the values obtained are above 0.5, which means that a good part of the key classes is detected by only using logical dependencies. As mentioned in section 3.4, a classifier is good if it has the ROC-AUC value as close to 1 as possible.

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

Initially, we expected to see a Gaussian curve, but instead, we see a bell curve. We think that in the beginning, we use a high number of logical dependencies in key class detection, among those logical dependencies is an important number of key classes and also an important number of other classes. But the number of other classes does not influence the key classes detection. When we start to increase the value of the threshold and filter more the logical dependencies, we also filter some of the initial detected key classes and remain with a significant number of other classes. In this case, the other classes that remain influence the measurements, causing the worst-performing solutions. Some of the key classes are strongly connected in the versioning system, and even for higher threshold values don't get filtered out. Meanwhile, the rest of the classes that are not key classes get filtered out for higher threshold values which leads to better performing measurements when the threshold value are above 60%.

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

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

5. Conclusions

The logical dependencies are filtered co-changing pairs extracted from the versioning system history. The filters applied to the co-changing pairs are the following: the filter based on commit size and the filter based on connection strength.

The filter based on connection strength had a variable threshold, starting with 10% and ending with 100%. We used a variable threshold for connection strength because we wanted to observe how this threshold will impact the key classes detection.

In section 4 we approached two scenarios to detect key classes by using logical dependencies. In the first scenario, we used logical dependencies together with structural dependencies and in the second, 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 using that tool. The quality of the results obtained was evaluated with the same tool, the metric used to evaluate the results is Area Under the Receiver Operating Characteristic Curve (ROC-AUC). We then compared the evaluation results with the results obtained by the baseline approach.

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 are not the same, most of them have used the ROC-AUC metric to evaluate the quality of the results. 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 approach) obtained an average ROC-AUC score of 0.894 [17].

In the current research, we obtained an average ROC-AUC score of 0.926 when using logical and structural dependencies combined and a score of 0.747 when using only logical dependencies to detect key classes.

In conclusion, by using both dependencies combined, we can obtain a slightly better ROC-AUC score than the one obtained by the baseline approach. And, by using only logical dependencies we don't obtain a better score than the baseline approach but compared with the results obtained by other researchers [18], the score obtained is almost equal. The 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.

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.: 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.: Keecle: 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
18. 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)
19. 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
20. Pan, W., Song, B., Li, K., Zhang, K.: Identifying key classes in object-oriented software using generalized k-core decomposition. Future Generation Computer Systems 81, 188–202 (2018), https://www.sciencedirect.com/science/article/pii/S0167739X17302492
21. 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
22. 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)
23. 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)
24. 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)
25. Tahvildari, L., Kontogiannis, K.: Improving design quality using meta-pattern transformations: a metric-based approach. J. Softw. Maintenance Res. Pract. 16, 331–361 (2004)
26. 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
27. 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)

28. Yu, L.: Understanding component co-evolution with a study on linux. Empirical Software Engineering 12(2), 123–141 (Apr 2007), https://doi.org/10.1007/s10664-006-9000-x
29. Zaidman, A., Calders, T., Demeyer, S., Paredaens, J.: Applying webmining techniques to execution traces to support the program comprehension process. In: Ninth European Conference on Software Maintenance and Reengineering. pp. 134–142 (2005)
30. 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)
31. 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