On the Interplay between Structural and Logical Dependencies in Open-Source Software

Conference Paper · September 2011

DOI: 10.1109/SBES.2011.39 · Source: DBLP

Gustavo Ansaldi Oliva
Queen's University
28 PUBLICATIONS **150** CITATIONS
SEE PROFILE

Marco Aurelio Gerosa
Northern Arizona University
185 PUBLICATIONS **1,790** CITATIONS
SEE PROFILE

On the Interplay between Structural and Logical Dependencies in Open-Source Software

Gustavo Ansaldi Oliva, Marco Aurélio Gerosa
Computer Science Department, Institute of Mathematics and Statistics
University of São Paulo (USP)
São Paulo, Brazil
{goliva, gerosa}@ime.usp.br

Abstract— **Structural dependencies have long been explored in the context of software quality. More recently, software evolution researchers have investigated logical dependencies between artifacts to assess failure-proneness, detect design issues, infer code decay, and predict likely changes. However, the interplay between these two kinds of dependencies is still obscure. By mining 150 thousand commits from the Apache Software Foundation repository and employing object-oriented metrics reference values, we concluded that 91% of all established logical dependencies involve non-structurally related artifacts. Furthermore, we found some evidence that structural dependencies do not lead to logical dependencies in most situations. These results suggest that dependency management methods and tools should rely on both kinds of dependencies, since they represent different dimensions of software evolvability.**

Keywords- mining software repositories; dependency management; structural analysis; structural dependencies; structural coupling; logical dependencies; logical coupling; software evolution.

I. INTRODUCTION

Software Engineering literature has long recognized the importance of structural dependencies to software quality research [1]. In particular, this kind of dependency has been investigated in the contexts of impact analysis [2], error-proneness [3], change propagation [4, 34, 35], regression testing [36], and others.

Recent research in software evolution area has introduced a novel dependency identification approach that reveals new and more subtle relationships between software artifacts. The concept underlying this notion is known as logical dependencies, which refers to evolutionary dependencies that are established among source-code files that are frequently changed together (although not necessarily structurally related).

The relationship between logical dependencies and software quality has also been investigated. Graves *et al*. [5] showed that past changes are good predictors of future faults. Mockus and Weiss [6] found that the spread of a change over subsystems and files is a strong indicator that the change will contain a defect. Cataldo *et al.* [7] showed through a detailed empirical study that the effect of logical dependencies on failure proneness was complementary and significantly more relevant than the impact of structural dependencies for two projects from different companies. Logical dependencies have also been employed to detect design issues [8], infer code decay [9], and predict changes in software artifacts [10].

While it seems clear that both kinds of dependencies play a major role during software evolution, no large-scale empirical examination has been undertaken to investigate their interplay. Particularly, the proportion of established logical dependencies that involve structurally related elements is still unclear. In addition, classic Software Engineering literature has long stated that structural coupling should be minimized because every time a supplier class changes, its clients are also likely to change [11, 12, 13]. However, little is known about the actual proportion of structural dependencies that effectively lead to logical dependencies.

A deeper understanding of the overlapping between these two kinds of dependencies is fundamental to Software Evolution research. For instance, if such overlapping is large, then structural and logical dependencies can be used interchangeably as input to dependency management methods and tools. On the other hand, if the overlapping is actually small, then it would be necessary to conceive and develop novel dependency management methods and tools that incorporate both kinds of dependencies. This last hypothetical situation implies that structural and logical dependencies represent different dimensions of software evolvability.

In this paper, we intend to better understand the interplay between structural and logical dependencies by addressing two research questions (Figure 1).

Q1) *What is the proportion of established logical dependencies that involve non-structurally related elements?*

Q2) *What is the proportion of formed structural dependencies that involve non-logically related elements?*

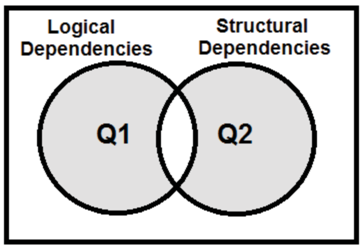

Figure 1. Venn-diagram illustrating the research questions.

We identified structural and logical dependencies from a baseline of 150 thousand revisions from the Apache Software Foundation (ASF) Subversion (SVN) repository. We used Doxygen and Doxyparse [14] to calculate structural dependencies from Java source-code files, and we used XFlow [15] to calculate logical dependencies. Once dependencies were gathered, we conducted a series of statistical analyses to answer the research questions.

Our main contributions include (i) mining ASF repository and proving a quantitative analysis of files per revision; (ii) developing an algorithmic approach to calculate and compare the degrees of structural and logical coupling; (iii) performing a segmented analysis of the relation between logical dependencies origins and structural coupling according to different support and confidence values; and (iv) analyzing the evolutionary consequences of structural dependencies by using object-oriented metrics reference values. Other contributions include improving Doxyparse metrics calculation and improving XFlow data collection performance.

The remainder of this paper is organized as follows. In Section II, we introduce both structural and logical dependencies and discuss how they can be identified. In Section III, we present the study setup by showing the rationale related to the choice of the software repository and the selection of the study supporting tools. In Section IV, we present the methodology and the dependency identification algorithms employed in this study. In Section V, we discuss our findings and in Section VI we discuss the threats to validity. In Section VII, we present related work and, in Section VIII, we state our final conclusions and future work.

II. Software Dependencies

A dependency is a semantic relationship that indicates that a client element may be affected by changes performed in a supplier element [13]. A dependency implies that the semantics of the client is not complete without its suppliers. We employ the terms *client* and *supplier* throughout this paper.

In the next subsections, we introduce structural dependencies and discuss how they can be identified in the context of object-oriented programming. We also present Coupling Between Objects (CBO) and Message Passing Coupling (MPC) metrics. Finally, we introduce the concept of logical dependencies and present the commonly associated metrics.

A. Structural Dependencies

Structural dependencies (a.k.a. syntactic dependencies) occur whenever a compilation unit depends on another at compilation or linkage time [16]. In this study, the term compilation unit is used to refer to an abstract class, concrete class, or an interface.

The identification of structural dependencies is conducted by recognizing clients and suppliers through code static analysis (a.k.a. code scanning), and thus it depends on the programming language in which the system was written. Occasionally, other metadata such as relationship type and dependency stereotype are also used. The static analysis is performed against source code or compiled code (the latter is the most common approach for languages such as Java and C#). A discussion about the advantages and disadvantages of analyzing each form of code can be found on [17]. In this study, we identified dependencies by analyzing source code written in Java (see Section III-B).

Several object-oriented metrics rely on the concept of structural dependencies. *Coupling Between Objects* (CBO), for example, measures the number of classes to which a class is coupled. The original CBO definition by Chidamber and Kemerer [18] considers the number of classes that a class `A` references (a.k.a. Fan-Out) plus the number of classes that references `A` (a.k.a Fan-In). If a class appears in both the referenced and the referred set of classes, it is counted only once. *Message Passing Coupling* (MPC) [19] measures the number of external operation calls, i.e. the number of calls from methods of a class to operations of other classes.

B. Logical Dependencies

Logical dependencies (a.k.a., change dependencies, evolutionary dependencies, and co-changes) are implicit dependencies that happen between software artifacts that evolved together [21, 20]. These artifacts are not necessarily structurally related, since they are connected from an evolutionary point of view, i.e. they have often changed together in the past, so they are likely to change together in the future. Hence, while structural dependencies are defined for a specific time instant, logical dependencies are defined based on a time interval.

The identification of logical dependencies is usually performed by parsing and analyzing the logs of version control systems (VCSs). Unlike structural dependencies analysis (a.k.a., static analysis), this technique is able to spot dependencies between any kind of artifact that composes a system, including configuration files (such as XML and property files) and documentation. In Section III-B, we provide details on how identified logical dependencies from projects in the ASF SVN repository.

Logical dependencies are defined for pairs of files and are commonly treated as data mining association rules [10]. Formally, an association rule is an implication of the form $X_1 \Rightarrow X_2$, meaning that when X_1 occurs, X_2 also occurs. In this notation, X_1 and X_2 are two disjoint sets of items. Furthermore, X_1 and X_2 are called the antecedent (a.k.a, left-hand-side, LHS) and the consequent (a.k.a., right-hand-side, RHS) of the rule respectively. For example, the rule $\{A, B\} \Rightarrow C$ found in the sales data of a supermarket would indicate that if a customer buys A and B together, he or she is also likely to buy C. In the context of our study, a logical dependency from a file f_2 to another file f_1 is denoted by $F_1 \Rightarrow F_2$, i.e. an association rule in which the antecedent and consequent are both singleton sets containing f_1 and f_2 respectively.

Measures of interest and significance for association rules are usually given by support and confidence thresholds. In our study, the support measure denotes the number of times two artifacts were changed together. The confidence measure defines the degree to which artifacts are logically connected, thus characterizing the strength of the relation. These concepts

Marco Gerosa receives individual grant from CNPq. This work is also partially supported by HP (Baile project) and FAPESP).

have been formalized by Zimmerman *et al.* [10] and we have adapted them for atomic-commit featured VCSs as follows:

Frequency of a set F in a set of commits C as frq(C, F) = $|\{c \mid c \in C, F \subseteq c\}|$.

Support of a rule $X_1 \Rightarrow X_2$ by a set of commits C as supp(C, $F_1 \Rightarrow F_2$) = frq(C, $F_1 \cup F_2$) = P($F_1 \cap F_2$), i.e. the probability of finding both antecedent and consequent in the set of commits C.

Confidence of a rule $F_1 \Rightarrow F_2$ as conf(C, $F_1 \Rightarrow F_2$) = frq(C,$F_1 \cup F_2$) / frq(C,F_1) = P($F_2|F_1$), i.e. the probability of finding the consequent of the rule in commits under the condition that these commits also contain the antecedent.

It should be noted that the confidence values for $F_1 \Rightarrow F_2$ and $F_2 \Rightarrow F_1$ are different. In the first case, the confidence value determines (by definition) the degree to which file f_2 is a client of file f_1. Analogously, in the second case, the confidence value determines the degree to which file f_1 is a client of file f_2. To illustrate this subtle difference, consider the example shown in Figure 2.

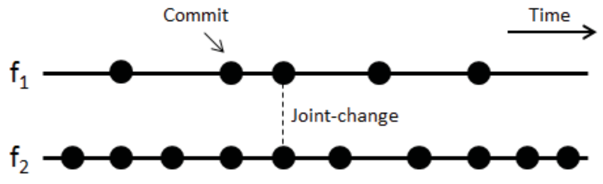

Figure 2. Association rule example.

Most of the time, when f_1 is commited, f_2 is also commited. Therefore, the rule $F_1 \Rightarrow F_2$ (which states that f_2 depends on f_1) has a high confidence value of 4/5 = 0.8 = 80%. In contrast, the rule $F_2 \Rightarrow F_1$ (which states that f_1 depends on f_2) has a much lower confidence value of 4/10 = 0.4 = 40%.

III. Study Setup

Prior to conducting the relationship study itself, we needed to address questions related to the choice of a software repository and the selection of tools for dependencies identification. In the next subsections, we show the rationale of these choices.

A. Repository choice

Apache Software Foundation (ASF) is a distinguished non-profit corporation that has developed nearly a hundred distinguishing software projects that cover a wide range of technologies and address several problems from diverse contexts. Examples of ASF projects include Apache HTTP Server, Apache Geronimo, Cassandra, Lucene, Maven, Ant, and Struts.

ASF currently owns a single giant SVN repository (with almost 1.1 million revisions) that hosts all Apache projects and subprojects. We chose to mine this repository since (i) it hosts a large number of relevant FLOSS projects; (ii) although projects are semi-autonomous, they all follow a well-defined *modus operandi* with relation to commit policies[1]; (iii) information about repository characteristics and general usage instructions are provided[2].

B. Supporting Tools

XFlow. Mining repositories studies usually require extensive tool support due to large and complex data that need to be collected, processed, and analyzed [24]. XFlow is an extensible and interactive open source tool [15] whose general goal is to provide a comprehensive analysis of software projects evolution process by mining software repositories and taking into account both technical and social aspects of the developed systems. XFlow collects data from version control systems, then identifies logical dependencies and evaluates metrics over the collected data, and finally presents rich interactive visualizations (Figure 3).

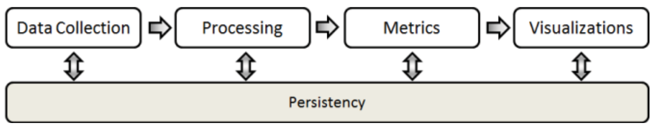

Figure 3. XFlow processing phases.

We decided that conducting the whole study using a single dependency analysis tool would be better than having separate tools for structural and logical dependencies, since it would facilitate, for instance, the grouping and analysis of the results. Therefore, we took advantage of XFlow extensible architecture to develop a structural dependencies identification module. This new developed module relies on both DoxyParse and DoxyGen tools.

Doxygen and Doxyparse. While designing the case study, we came up with the following requirements for a structural dependencies identifier tool:

(i) To identify dependencies through Java source code scan: since we were dealing with a versioning system, it would be more practical and straightforward if the identification of dependencies were done directly upon the source code of Java compilation units;

(ii) To calculate structural coupling by counting the number of operation calls from a client to a supplier: this defined our metric for structural coupling. This can be viewed as the application of the MPC metric for a single client and a single supplier. We henceforth denote it by MPC*.

Doxygen[3] is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), Fortran, VHDL, PHP, and C#. Doxygen is distributed under the GPL license and runs on a variety of operating systems. Doxyparse is a multi-language source code parser that meets our requirements and is built on top of Doxygen's internals. In theory, it supports parsing all the languages supported by Doxygen, although up to now it has only been tested with C, C++, and Java. Doxyparse's main goal is to serve as a backend

[1] http://www.apache.org/dev/new-committers-guide.html
[2] http://www.apache.org/dev/version-control.html
[3] http://www.doxygen.org/index.html

for Analizo [14], which is a source code analysis toolkit mainly aimed at calculating object-oriented metrics.

Improvements in Doxygen/Doxyparse and XFlow. Doxygen was not able to detect duplicate client method calls to the same supplier operation in the case where such calls were made inside the scope of the same client method. For example, if a method `a()` from class `A` called an operation `b()` from class `B` three times, Doxygen would only count it once. We hacked Doxygen so that it would count all operation calls (three in the example). Finally, we also hacked DoxyParse to remove useless information to this study, so that performance would be improved.

XFlow was not initially prepared to support large-scale studies (such as this one, which involves 150k revisions). Hence, during the study setup, we improved the performance of the data collection phase of the tool.

Minitab. All statistical analysis of data in this study was supported by Minitab[4]. Minitab is an easy to use and yet powerful statistical package heavily employed in both industry and statistical courses at universities worldwide.

IV. Data Collection Instruments and Methods

Once the repository was chosen and supporting tools were selected and adapted, we proceeded with the interplay study. In the next subsections, we present how and what kind of data we collected from the repository. After that, we show basic descriptive statistics related to *files per revision*. Finally, we discuss how logical and structural dependencies were identified and how coupling values were calculated.

A. Collecting data

We analyzed 150k revisions from the ASF SVN repository, which encompasses an activity time frame of approximately 2 years and 4 months (October of 2002 till February of 2005). We chose to analyze the initial revisions, so that project's growth and evolution characteristics would be preserved for the logical dependency analysis (picking up an arbitrary interval would provide inaccurate evolutionary information).

To cope with remote repository instability, we built a local mirror of the ASF SVN repository. After mirroring the repository, we executed the data collection processing phase of XFlow (Figure 3), i.e. we collected and parsed the log messages of all the 150k revisions. Since we needed to calculate structural coupling, we configured XFlow to download source code of the versioned files. All files that did not have a `.java` extension were filtered out from revisions. Furthermore, revisions having no Java files were discarded. We chose to analyze only Java source-code since structural coupling measures vary from language to language.

After collecting the data, we calculated basic descriptive statistics to gain insight and better understand the characteristics of *number of files per revision* variable (Table I and Table II). A graphical summary is given in Figure 4.

[4] http://www.minitab.com/

TABLE I. Number of Files Per Revision – Descriptive Statistics

N	Sum	Mean	StDev	Skewness	Kurtosis
40,518	251,691	6.21	36.24	33.54	1608.65

Number of revisions and number of files. XFlow data collection resulted in 40,518 revisions containing at least one java file, which accounts for 27% of our initial sample (150k revisions). This amount of revisions contained a total of 251,691 files.

Mean and Standard deviation. The mean value indicates that revisions contain approximately 6 files in average. However, standard deviation value shows that the dispersion is high. The next basic descriptive statistics measures were employed to better understand this dispersion value.

Skewness. Skewness value denotes the degree to which a data set is not symmetrical. The positive skewness value presented in Table I indicates that the data-set is right-skewed, i.e. the "tail" of the distribution points to the right. Positive skewness can be clearly noted in Figure 4A.

Kurtosis. Kurtosis value denotes the degree to which a data set is peaked. Data sets with high kurtosis tend to have a distinct peak near the mean, decline rather rapidly, and have heavy tails. The high kurtosis value presented in Table I can be clearly visualized in Figure 4A.

Test for normality. Analysis of skewness, kurtosis, and frequency histogram showed that data is not normally distributed. In fact, we conducted the Anderson-Darling test for normality and we obtained a p-value < 0005.

TABLE II. Number of Files Per Revision – Quartile Analysis

Min	Q1	Median	Q3	Max	IQR	Lower Whisker	Upper Whisker
1.0	1.0	1.0	4.0	2450.0	3.0	1.0	8.5

Quartile analysis. The boxplot gives another view on the distribution of the data-set, by showing its shape, central tendency, and variability (Figure 4B). Lower and upper whiskers reveal that "usual" revisions encompass 1 to 8 files. The largest revision included 2,450 files.

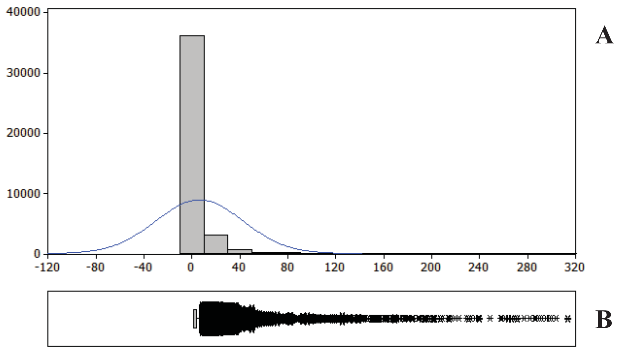

Figure 4. Graphical summary of *number of files per revision*.

In the next subsection, we describe how dependencies were identified. We also present assumptions and decisions made during this task.

B. Identifying dependencies and calculating coupling

As stated in Section III-B, we employed XFlow to detect both structural and logical dependencies. We referred to the quartile analysis to exclude revisions that contained an extreme number of files, since these revisions are statistically insignificant and would require great computational resources to be processed[5].

Logical dependencies. XFlow processing phase (Figure 2) applies the method proposed by Cataldo to detect logical dependencies between software artifacts [25]. The tool analyzes all previously parsed revisions and builds a square sparse symmetrical matrix with the following properties:

- `cell(i,i)`: number of times that file *i* changed (number of revisions that included file *i*).
- `cell(i,j), for i≠j`: number of times that file *i* and file *j* changed together (number of revisions that included both file *i* and file *j*).

Two logical dependencies are established for each non-blank cell off the matrix diagonal. The value of such cells indicates the support for both dependencies. Confidence is calculated for each one of these two dependencies by employing the support value and the corresponding value from the matrix diagonal.

Structural dependencies. While logical dependencies are defined based on a time interval, structural dependencies are defined for a specific time instant. Therefore, in order to compare logical coupling to structural coupling, we developed an algorithmic strategy. The basic idea is that, for every established logical dependency $F_1 \Rightarrow F_2$, we calculate the associated average structural coupling value. Such value corresponds to the average number of operation calls from f_2 to f_1 (average MPC*). The following pseudocode[6] describes the conceived strategy for structural coupling calculation (Listing 1). Variable names `c` and `s` stand for client and supplier respectively.

```
1.  revisions ← XFlow.getAllRevisions()
2.  for each revision r in revisions
3.     files ← r.getFiles()
4.     for each file c in files
5.        //Calc. coupling for files in rev.
6.        for each file s in files
7.           if(c ≠ s)
8.              depDegree ← Doxy.calcMPC*(c,s)
9.              dep ← XFlow.getDependency(c,s)
10.             dep.addDegree(depDegree)
11.          end-if
12.       end-for
13.       //Calc. coupling for files outside rev.
14.       old_s ← XFlow.getOldSuppliers(c)
15.       old_s ← old_s - files
16.       for each file s in old_s
17.          depDegree ← Doxy.calcMPC*(c,s)
18.          dep ← XFlow.getDependency(c,s)
19.          dep.addDegree(depDegree)
20.       end-for
21.    end-for
22. end-for
```

Listing 1. Pseudocode describing the employed strategy for structural coupling calculation

We first calculate the pair-wise structural coupling between all files in each revision (`2-12`). Given that a file `f` of a specific revision may have already been co-changed with other files in the past, we recalculate the structural coupling value between `f` and the *old suppliers* of `f` (`13-21`). We assume that `dep` is a variable that points to a Dependency object which provides the average coupling degree for a series of added degree values (`10,19`).

V. Results

Once dependencies were identified and associated coupling values were calculated, we started the analysis of the relationship between structural and logical dependencies. We first queried XFlow database and obtained a total of 270,010 dependencies, along with their properties. Based on such data, we built a spreadsheet with the following columns: antecedent (LHS), consequent (RHS), support, confidence, and MPC*. We imported the spreadsheet in Minitab and started investigating our research questions.

In the next subsections, we analyze and discuss the results obtained for each one of the questions.

A. Research question Q1

***Q1)** What is the proportion of established logical dependencies that involve non-structurally related elements?*

We calculated the proportion of logical dependencies that involved non-structurally related elements[7] (Figure 5) by applying the approaches described in Section IV-B. We called such proportion LCOP (logical-coupling-only proportion).

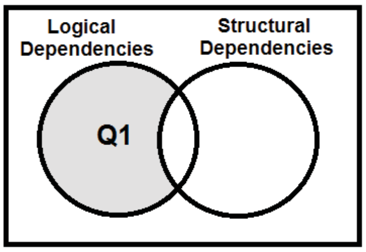

Figure 5. Venn-diagram illustrating research question Q1.

[5] In a quartile analysis, extreme values are those beyond Q3+3.0*IQR

[6] The presented pseudocode is an illustrative high-level description of how structural coupling is calculated. The real module implemented in XFlow is much more complex and optimized.

[7] Non-structurally related elements are those for which the average MPC* equals zero (see Listing 1).

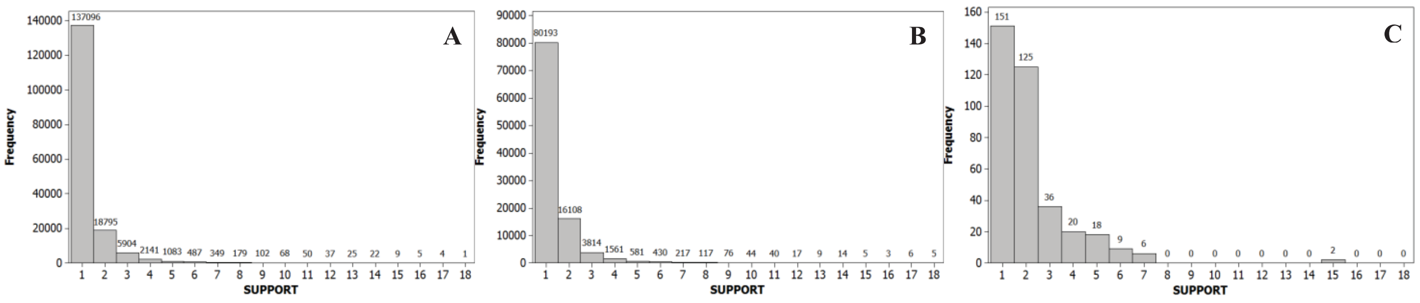

Figure 8. Support distribution for lowly (A), medially (B) and highly (C) logically coupled files.

We conducted our analysis by calculating LCOP for three different confidence intervals that we established:

- [0.00, 0.33]: Low logical coupling
-]0.33, 0.66]: Medium logical coupling
-]0.66, 1.00]: High logical coupling

Table III shows the results for all computed intervals. We can see that as confidence increased, LCOP value also increased. Interestingly enough, this revealed that highly logically coupled files were the ones that suffered the slightest influence from structural coupling (although they accounted for only 0.1% of the total number of logical dependencies). Furthermore, the last table row (which encompasses all logical dependencies) also presented a high LCOP value (91%). Therefore, since LCOP values ranged from 91% to approximately 93%, we conclude that *logical dependencies very frequently involve non-structurally related elements*. Consequently, as correlation is necessary for linear causation, we have some evidence that *logical dependencies are not directly caused by structural dependencies*. This finding corroborates the results given in [33].

TABLE III. LOGICAL DEPENDENCIES INVOLVING NON-STRUCTURALY RELATED COMPILATION UNITS

	Number of Logical dependencies	Logical deps. w/o structural counterpart	LCOP
Confidence [0.00, 0.33]	166,378 (61.6%)	151,397	91.0%
Confidence]0.33, 0.66]	103,265 (38.2%)	94,074	91.1%
Confidence]0.66, 1.00]	367 (0.1%)	341	92.9%
Total	270,010 (100%)	245,812	91.0%

To further investigate this research question, we plotted LCOP for the three confidence intervals (Figure 6). The x-axis refers to support values. Furthermore, we also computed the cumulative percentage for the support variable (Figure 7), as well as the support frequency distribution for lowly, medially, and highly logically coupled files (Figure 8).

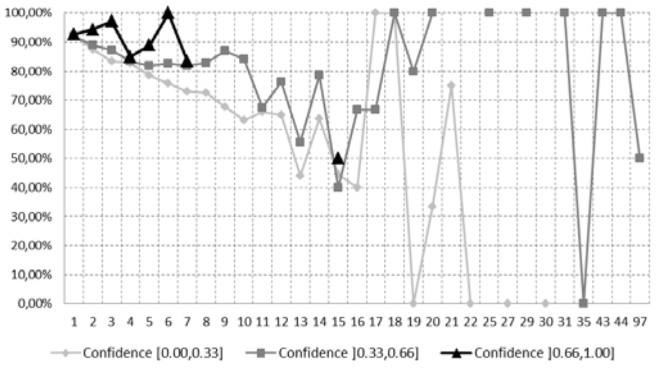

Figure 6. Logical dependencies involving non structurally related compilation units (LCOP).

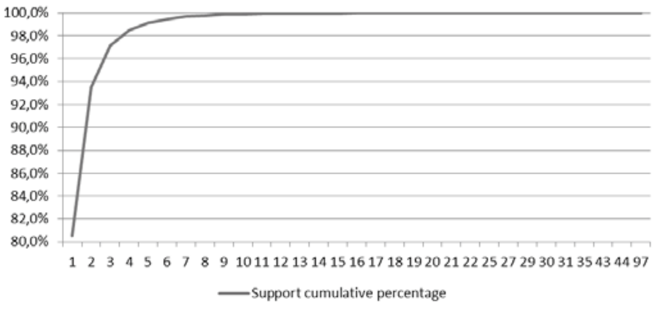

Figure 7. Support cumulative percentage.

In light of these results, we analyzed LCOP for three different support intervals:

Support interval [0, 10]: This support interval accounts for 99.9% of all identified logical dependencies (Figure 7). Lowly logically coupled files showed a decreasing curve (Figure 6), i.e. LCOP decreased and thus structural dependencies seem to play a considerable role in these cases. Medium logically coupled compilation units showed a curve with minor variations that stayed between 90% and 80% for the considered support interval. Therefore, as opposed to the previous case, structural dependencies seem to be poorly responsible for these logical dependencies.

Support interval [11, 31]: Lowly, medially, and highly logically coupled files showed curves with high variation throughout this support interval. This was expected, since such interval accounts for only 0.1% of all logical dependencies (Figure 7).

Support interval [1, 7]: Highly logically coupled files curve stayed beyond 90% most of the time, which corroborates our previous finding, i.e. these files suffered the slightest influence from structural coupling. Figure 8 also shows that support distribution for highly logically coupled files (C) is "smoother" than the ones for lowly (A) and medially (B) logically coupled files.

B. Research Question Q2

***Q2)** What is the proportion of formed structural dependencies that involve non-logically related elements*

We employed a three-step strategy to calculate the proportion of structural dependencies that involved non-logically related elements. We called this proportion SCOP (structural-coupling-only proportion). As illustrated in Figure 9, we (i) estimated the size of the structural dependencies by employing reference values for the CBO metric, (ii) calculated the size of the intersection set, and (iii) calculated the size of the structural-dependencies-only set based on the results from (i) and (ii).

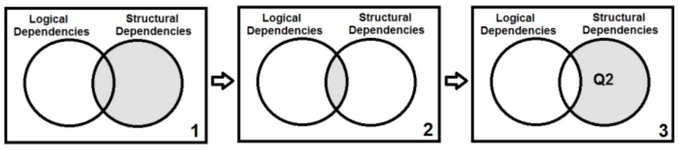

Figure 9. Venn-diagram illustrating the employed strategy to answer research question Q2.

i) Estimating the size of the structural dependencies set. Barkmann *et al.* developed a set of tools to perform a large-scale metrics analysis that involved 160 Java FLOSS projects (70 thousand compilation units and 11 million lines of code) in order to provide reference values and thresholds [26]. Figure 10 shows a histogram for CBO, in which each grid line corresponds to a thousand classes. Values below the histogram refer to minimum, maximum, average, median, and modus respectively. The author computed CBO by only considering the Fan-Out of a compilation unit (native API was discarded and class constructor did not count as a method) [27]. In fact, CBO and Fan-Out have been treated as synonyms in a number of studies, e.g. [28, 29].

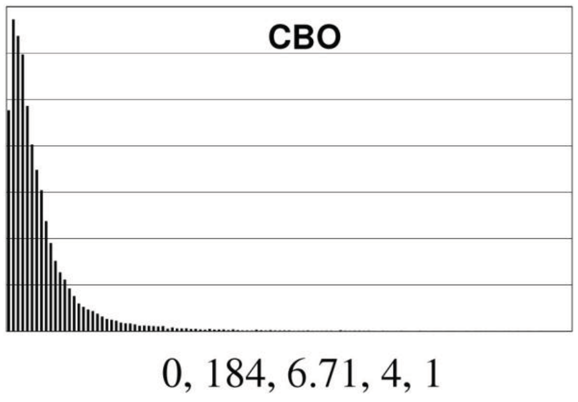

Figure 10. Reference values for the CBO metric (minimum, maximum, average, median, modus) [26].

XFlow data collection phase identified 72,523 distinct compilation units out of the 150k revisions from ASF SVN repository. Given that the reference average value for CBO is 6.71 (Figure 10), we estimated that the total number of structural dependencies is 72,523 * 6.71 = 486,630.

ii) Calculating the size of the intersection set. From the data provided in the last row of Table III, we calculated the size of the intersection set: 270,010 - 245,812 = 24,198.

iii) Calculating the size of the structural-dependencies-only set. By subtracting (ii) from (i), we calculated the size of the structural-dependencies-only set: 486,630 – 24,198 = 462,432. As a result, SCOP corresponded to 95% of the total number of estimated structural dependencies. Hence, we conclude that *structural dependencies very frequently involve non-logically related elements*. Consequently, as correlation is necessary for linear causation, we have some evidence that *structural dependencies do not lead to logical dependencies*.

VI. THREATS TO VALIDITY

There are some factors that may have influenced the validity of the study.

Internal validity. In relation to logical dependencies detection in SVN repositories, we employed a simple and recurrently used strategy. Even though Pirklbauer recently surveyed and evaluated different strategies to detect logical dependencies [30], his results only apply for two industrial projects and thus cannot be generalized to our context. Some constraints on logical coupling detection were also imposed by XFlow design decisions. One key decision is that replaced files are considered added files for the sake of logical dependencies detection. This may break the evolutionary relation between pairs of files under rare circumstances. Finally, although we used the ASF SVN repository to minimize bias due to different commit policies, individual projects may not strictly follow such policies. Also, these projects may have other particular characteristics that can affect the results of the employed logical dependencies detection strategy.

To address the first research question, we divided logical dependencies in three different confidence intervals of the same length. It could be that a deeper statistical investigation would prove that smaller (or even different) intervals would provide better input to the performed analysis. Finally, we acknowledge that support and confidence measures can provide misleading results under rare circumstances. Consider the example in which a file f_1 joint-changed 5 times with f_2 and then, after that, f_1 changed alone for other 5 times (f_2 did not change anymore). Although the confidence for the logical dependency $F_1 \Rightarrow F_2$ would be 0.5, it may be the case that f_2 does not actually depend on f_1 anymore (e.g. a refactoring occurred right after f_1 and f_2 changed together for the last time). Finally, even though the investigated repository hosts all ASF projects, we found a negligible amount of commits that encompassed files from different projects. Besides that, revisions regarding large repository operations that involved a high number of files were discarded by means of the performed quartile analysis (Section IV-B). Therefore, the results presented in Table III were not hindered by the fact that we analyzed ASF SVN "as a whole".

While we identified structural dependencies by only counting external method calls (Listing 1), CBO reference values used to address the second research question additionally rely on external instance references. However, external instance references that receive no operation calls commonly represent situations of delegation. Thus, we imagine that these situations occur rather rarely, since object-oriented systems are made of collaborating objects. In fact, even if the size of the intersection set was twice as large, SCOP would still correspond to approximately 90% of the total number of estimated structural dependencies. Since we made assumptions based on reference values, we acknowledge that our evidences for the second research question are less reliable than those obtained in the investigation of the first research question.

External validity. In relation to external validity, we may have introduced some bias in the generalization of our results by having analyzed projects from ASF only. Although this decision helped us in other dimensions (Section III-A), a broader analysis should also consider projects from other FLOSS repositories (such as SourceForge) and communities. Our study results are applicable for open-source software only, since industry software projects may have distinct characteristics that were not considered in this empirical investigation.

VII. Related Work

Although logical dependencies have been extensively investigated to different purposes, such as to support change prediction [10] and establish coordination requirements among developers [25], few studies have explored the interplay between logical and structural dependencies.

Hanakawa studied the relation between sets of *highly structurally coupled elements* (M) and sets of *highly logically coupled elements* (L) throughout time [22]. The hypothesis stated by the author is that the average intersection between M and L tends to decrease throughout time due to an increase in "copy & paste" actions (leading to logical coupling only), and developers forgetting to commit structurally related classes at once (leading to structural coupling only). Hanakawa proposes a complexity metric based on the size of such intersection (Figure 11). Although the obtained results confirms that the intersection between M and L does decrease (i.e. complexity increases) in most situations, there is poor evidence that such results derive from the author assumptions. This is particularly evident in the conducted JUnit complexity analysis.

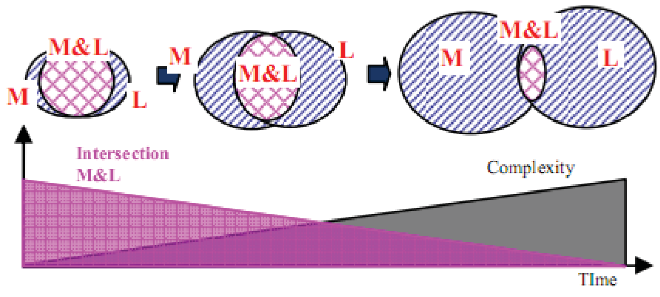

Figure 11. Hanakawa hyphotesis regarding the relation between M&L *intersection* and *complexity* throughout time.

Cataldo *et al.* examined the relative impact that structural, logical, and work dependencies have on the failure proneness of a software system [7]. During this study, the authors analyzed two projects from different companies and verified that there were low levels of correlation between structural and logical dependency measures in the latest version of both projects. Although details on how dependencies were gathered are not provided, their result is consistent with ours.

Oliva *et al.* investigated the origins of logical dependencies by means of a case study involving a Java FLOSS project [33]. The authors conducted a manual inspection of logical dependencies origins by reading revision comments, looking at code diffs, and holding informal interviews with the project developers. Preliminary results showed that there was no distinct underlying reason behind the establishment of the analyzed logical dependencies, since they involved pairs of files that changed together ("joint-changed") for different reasons. The authors noticed that structural dependencies were responsible for less than 20% of all joint-changes.

Cataldo and Nambiar investigated 189 global software development (GSD) projects and showed that logical coupling was the most significant factor impacting software quality among all considered factors (such as structural coupling, process maturity, developers' experience, number of regional units, and people dispersion) [31]. Pirklbauer *et al.* developed a change impact analysis framework (process and tool) that incorporates both structural and logical dependencies [32].

VIII. Conclusion and Future Work

In this paper, we conducted a large scale empirical study to investigate the interplay between structural and logical dependencies in FLOSS projects. We analyzed all Java files of the first 150k commits of the Apache Software Foundation Subversion repository in order to quantify (i) the proportion of established logical dependencies that involve non-structurally related elements and (ii) the proportion of formed structural dependencies that involve non-logically related elements. In relation to (i), we concluded that in at least 91% of the cases, logical dependencies involve files that are not structurally related, i.e. we have some evidence that *logical dependencies are not directly caused by structural dependencies*. Regarding (ii), we concluded that structural dependencies very frequently involve files that are not logically related, i.e. we have some evidence that *structural dependencies do not usually lead to logical dependencies*. Hence, we conclude that there is a very small intersection between the sets of structural and logical dependencies.

Although static analysis mechanisms and frameworks have proven their value in software maintainability and evolvability throughout the years [1], our findings support the idea that *such kind of analysis is still necessary but not sufficient*. As a result, we believe that dependency management methods and tools should rely on both kinds of dependencies, since they represent different dimensions of software evolvability. An integrated view of these two kinds of dependencies shall improve the effectiveness of both software change and maintenance activities. Finally, we also believe that software visualization tools that comprise both kinds of dependencies (e.g. [15, 32])

should be a fertile research field with strong implications in software quality related areas.

As future work, we plan to conduct a deeper analysis of the interplay between structural and logical dependencies by (i) considering the whole ASF SVN repository (~1.1 million revisions), (ii) employing the adapted sliding time window algorithm presented in [33] to group related revisions (so that they are treated as a unit), and (iii) developing a more efficient structural dependencies identifier module in order to answer research question Q2 based on the actual data-set (instead of applying metric reference values). As a final point, we believe that investigating the interplay between structural dependencies and other kinds of dependency (e.g. data-flow/hidden [37], conceptual [38]) should be another fertile research topic.

ACKNOWLEDGMENT

We thank Mauricio F. Aniche, Mauricio de Diana, and Cleidson R. B. de Souza for insightful contribution to the design of this study. We thank Francisco W. Santana for wisely helping in the XFlow performance improvement process. We thank Antonio Terceiro, Paulo Meirelles, and Carlos Denner for providing DoxyParse and giving hints on where to change its code in order to detect all operation calls between classes. We also thank Francisco W. Santana (again), Igor Steinmacher, and Rafael Szot de Lima for reviewing a previous version of this paper. Finally, we thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] D.Binkley, "Source code analysis: a road map," Proc. International Conference on Software Engineering (ICSE '07), May 19-27 2007, pp.104-119.

[2] L.C. Briand, J. Wust, J.W. Daly, and D.V. Porter, "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems," J. Systems and Software, vol. 51, pp. 245- 273, 2000.

[3] R.W. Selby and V.R. Basili, "Analyzing Error-Prone System Structure," IEEE Trans. Software Eng., vol. 17, no. 2, pp. 141-152, Feb. 1991.

[4] S.S. Yau, J.S. Collofello, and T. MacGregor, "Ripple effect analysis of software maintenance," Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International, pp. 60- 65, 1978.

[5] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," IEEE Trans. Software Eng., vol. 26, no. 7, pp. 653-661, July 2000.

[6] A. Mockus and D. Weiss, "Predicting Risk of Software Changes," Bell Labs Technical J., vol. 5, pp. 169-180, 2000.

[7] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software Dependencies, Work Dependencies, and Their Impact on Failures," IEEE Trans. Software Eng., vol.35, no. 6, pp. 864-878, 2009.

[8] M. D'Ambros, M. Lanza, and M. Lungu, "Visualizing Co-Change Information with the Evolution Radar," IEEE Transactions on Software Engineering, vol. 99, no. RapidPosts, pp. 720-735, 2009.

[9] S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster, "Visualizing Software Changes," IEEE Trans. Software Eng., vol. 28, no. 4, pp. 396-412, Apr. 2002.

[10] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," IEEE Transactions on Software Engineering, pp. 429-445, June, 2005.

[11] D. L. Parnas "On the criteria to be used in decomposing systems into modules," Communications of the ACM. 1972;15(12):1053-1058..

[12] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd ed.: Prent ice Hall, 2004.

[13] G. Booch, Object-Oriented Analysis and Design with Applications, 3rd ed.: Addison-Wesley, 2007.

[14] A. Terceiro, J. Costa, J. Miranda, P. Meirelles, L. R. Rios, L. Almeida, C. Chavez, and F. Kon, "Analizo: an Extensible Multi-Language Source Code Analysis and Visualization Toolkit" in Tools Session of the 1st Brazilian Conference on Software (CBSoft), September 2010.

[15] F. Satana, G. Oliva, C. R. B. de Souza, and M. Gerosa "XFlow: An Extensible Tool for Empirical Analysis of Software Systems Evolution", VIII Experimental Software Engineering Latin American Workshop (ESELAW 2011)

[16] J. Lakos, Large-Scale C++ Software Design, 1st ed.: Addison-Wesley Professional, 1996.

[17] L. A. Barowski and J. H. Cross, "Extraction and use of class dependency information for Java," in Proceedings of the Ninth Working Conference on Reverse Engineering, 2002, p. 309.

[18] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design". IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476–493, 1994.

[19] W. Li and S. Henry, "Maintenance metrics for the object oriented paradigm," Software Metrics Symposium, 1993. Proceedings., First International, pp.52-60, 21-22 May 1993.

[20] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," Software Maintenance, 1998. Proceedings. International Conference on, pp.190-198, 16-20 Nov 1998.

[21] T. Ball , J.-M. Kim, A.A. Porter, and H.P. Siy , "If Your Version Control System Could Talk," Proc. ICSE Workshop Process Modelling and Empirical Studies of Software Eng., 1997.

[22] N. Hanakawa, "Visualization for Software Evolution Based on Logical Coupling and Module Coupling," Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific, pp. 214-221, 4-7 Dec. 2007 doi: 10.1109/ASPEC.2007.36 .

[23] J. Costa, R. Feitosa, and C. R. B. Souza, "RaisAware: uma ferramenta de auxílio à Engenharia de Software," *Scientia*, pp. 12-24, 2009.

[24] J. Bevan, J. E. Whitehead, S. Kim, and M. Godfrey, "Facilitating Software Evolution Research With Kenyon" In: SIGSOFT Softw. Eng. Notes 177-186, 2005.

[25] M. Cataldo, P.A. Wagstrom, J.D. Herbsleb, and K. Carley, "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools", In Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work, ACM, New York, NY, pp. 353-362, 2006.

[26] H. Barkmann, R. Lincke, W. Löwe, "Quantitative Evaluation of Software Quality Metrics in Open-Source Projects," International Conference on Advanced Information Networking and Applications Workshops, pp.1067-1072, 2009.

[27] R. Lincke, J. Lundberg, and W. Löwe, "Comparing Software Metrics Tools" International Symposium on Software Testing and Analysis Seattle, WA, July 20-24 2008.

[28] R. Kollmann and M. Gogolla, "Metric-based selective representation of UML diagrams," Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on, pp.89-98, 2002 doi: 10.1109/CSMR.2002.995793

[29] A. Chatzigeorgiou, "Mathematical assessment of object-oriented design quality," IEEE Trans. Software Eng., vol.29, no.11, pp. 1050- 1053, Nov. 2003 doi: 10.1109/TSE.2003.1245306

[30] G. Pirklbauer, "Empirical Evaluation of Strategies to Detect Logical Change Dependencies". SOFSEM 2010: Theory and Practice of Computer Science. pp. 651–662, 2010.

[31] M. Cataldo and S. Nambiar, "The impact of geographic distribution and the nature of technical coupling on the quality of global software development projects". Journal of Software Maintenance and Evolution: Research and Practice, 2011. doi: 10.1002/smr.477

[32] G. Pirklbauer, C. Fasching, and W. Kurschl, "Improving Change Impact Analysis with a Tight Integrated Process and Tool," Information Technology: New Generations (ITNG), 2010 Seventh International

Conference on, pp. 956-961, 12-14 April 2010 doi: 10.1109/ITNG.2010.100.

[33] G. Oliva, F. Santana, M. A. Gerosa, and Cleidson R. B. de Souza. “Towards a Classification of Logical Dependencies Origins: A Case Study,” In Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-EVOL 2011), Szeged, Hungary. [Accepted for publication]

[34] V.Rajlich, "A model for change propagation based on graph rewritting", Proc.International Conference on Software Maintenance (ICSM'97), October 1-3 1997, pp. 84-91.

[35] V.Rajlich, "Modeling software evolution by evolving interoperation graphs", Annals of Software Engineering, vol.9, May'00, pp. 235-248

[36] L.White, K. Jaber, B. Robinson, and V.Rajlich, "Extended firewall for regression testing:an experience report", Journal of Software Maintenance and Evolution: Research and Practice, vol.20, July 2008, pp. 419-433.

[37] R. Vanciu and V. Rajlich “Hidden Dependencies in Software Systems”. Proceedings of the IEEE International Conference on Software Maintenance (ICSM2010), Timisoara, Romania.

[38] H. Kagdi, M. Gethers, D. Poshyvanyk, M. L. Collard. “Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code”. Proceedings of the 17th Working Conference on Reverse Engineering (WCRE2010), 2010, pp. 119–128.