Methods and Tools for the Analysis of Legacy Software Systems

Report 1. Logical dependencies extraction - impact factors.

PhD Student: Adelina Diana Stana

Department: Calculatoare şi tehnologia informaţiei
PhD Supervisor: Vladimir I. CREŢU

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of the thesis is to develop methods for analyzing legacy software systems by using historical information extracted from the versioning systems.

One of the biggest problems when it comes to legacy software systems is the lack of modernization. A legacy software system is an obsolete system that might still be in use and satisfy the organization's needs but due to the lack of modernization, it has a high maintenance cost. One solution is to modernize or refactor the system and to do that the developers must have a good understanding of the system. Here comes one other problem of the legacy software systems, the lack of up-to-date documentation or the lack of documentation. The solution for this problem is to analyze the legacy software system in order to gain more knowledge about the system and how the system works. Our approach is to analyze the legacy software system by using historical information extracted from the versioning systems.

We divided our work into two main parts, the first part focuses on historical information collection and filtering, and the second part focuses on the usage of the collected information in order to analyze the software systems.

This first report presents the experiments conducted so far and the results obtained in filtering the extracted information from the versioning system.

The information extracted from the versioning system is under the form of pairs of classes that record co-changes (co-changing pairs). We filter the co-changing pairs to increase their veridicality and to decrease the size of the extracted information. We want to decrease the size of the extracted information because other researchers identified that one possible explanation why historical information is rarely used is the size of the extracted information [25], [14].

Also, by increasing the veridicality, we mean to increase the confidence that a co-changing pair is truly related and not just an update coincidence in the versioning system. That is why at the end of the filtering process, we are calling the remaining co-changing pairs logical dependencies. The filtering process revolves around the commit size and the occurrence of the co-changing pairs in the versioning system history. The filtering based on commit size is presented in section 3.2 and filtering based on co-changing pairs occurrence is presented in sections 3.3, and 3.4. Each filtering section contains the detailed results obtained and conclusions based on the results.

In order to do the operations described above, we have developed a tool that extracts and processes the needed information from a software system. The tool workflow and technologies used are presented in section 2.3. The primary information extracted by the tool is described in sections 2.1 and 2.2.

To perform measurements based on our assumptions, we have selected a set of 27 object-oriented software systems presented in section 3.1. For each listed software system, the tool extracts, filters, and collects the information needed.

Section 3.5 focuses on the overlappings between the extracted information from the code and the filtered information from the versioning systems.

The conclusions and observations based on the performed measurements are presented in chapter 4.

Chapter 2

Extracting software dependencies

2.1 Extracting structural dependencies

A dependency is created between two elements that are in a relationship and indicates that an element of the relationship, in some manner, depends on the other element of the relationship [7], [10].

Structural dependencies can be found by analyzing the source code [24], [8], [6]. A structural dependency between two classes A and B is given by the fact that A statically depends on B, meaning that A cannot be compiled without knowing about B. In object oriented systems, this dependency can be given by many types of relationships between the two classes: A extends B, A implements B, A has attributes of type B, A has methods which have type B in their signature, A uses local variables of type B, A calls methods of B.

We use an external tool called srcML [1] to convert all source code files from the current release into XML files. All the information about classes, methods, calls to other classes are extracted by parsing the XML files and building a dependency data structure [11], [12]. We choose the srcML format because it has the same markup for different programming languages and can ease the parsing of source code written in various programming languages such as Java, C++, and C#.

2.2 Extracting co-changing pairs

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

The concepts of logical coupling and logical dependencies were first used in different analysis tasks, all related to changes: for software change impact analysis [23], for identifying the potential ripple effects caused by software changes during software maintenance and evolution [21], [20], [22], [17] or for their link to deffects [29], [32].

Software engineering practice has shown that sometimes modules which do not present structural dependencies still can be related [30]. Co-evolution represents the phenomenon when one component changes in response to a change in another component [31], [9]. Those changes can be found in the software change history from the versioning system. Gall [15], [16] identified as logical coupling between two modules the fact that these modules *repeatedly* change together during the historical evolution of the software system [4].

The versioning system contains the long-term change history of every file. Each project change made by an individual at a certain point of time is contained into a commit [19]. All the commits are stored in the versioning system chronologically and each commit has a parent. The parent commit is the baseline from which development began, the only exception to this rule is the first commit which has no parent [13].

Currently there is no set of rules or best practices that can be applied to the extracted class co-changes and can guarantee their filtering into a set of logical dependencies. This is mainly because not all the updates made in the versioning system are code related. For example a commit that has as participants a big number of files can indicate that a merge with another branch or a folder renaming has been made. In this case, a series of irrelevant co-changing pairs of entities can be introduced. So, in order to exclude this kind of situations the information extracted from the versioning system has to be filtered first and then used. Surveys also show that historical information is rarely used due to the size of the extracted information [25], [14].

Other works have tried to filter co-changes [20], [2], [21]. One of the used co-changes filter is the commit size.The commit size is the number of code files changed in that particular commit. Ajienka and Capiluppi established a threshold of 10 for the maximum accepted size for a commit [2]. This means that all the commits that had more than 10 code files changed where discarded from the research. But setting a harcoded threshold for the commit size is debatable because in order to say that a commit is big or small you have to look first at the size of the system and at the trends from the versioning system. Even thought the best practices encourage small and often commits, the developers culture is the one that influences the most the trending size of commits from one system.

Filtering only after commit size is not enough, this type of filtering can indeed have an impact on the total number of extracted co-changes, but will only shrink the number of co-changes extracted without actually guaranteeing that the remaining ones have more relevancy and are more linked.

Although, some unrelated files can be updated by human error in small commits, for example: one file was forgot to be commited in the current commit and will be commited in the next one among some unrelated files. This kind of situation can introduce a set of co-changing pairs that are definitely not logical liked. In order to avoid this kind of situation a filter for the occurrence rate of co-changing pairs can be introduced. Co-changing pairs that occur multiple times are more prone to be logically dependent than the ones that occur only once. Currently there are no concrete examples of how the threshold for this type of filter can be calculated. In order to do that, incrementing the threshold by a certain step will be the start and then studying the impact on the remaining co-changing pairs for different systems.

Nevertheless, logical dependencies should integrate harmoniously with structural dependencies in an unitary dependency model: valid logical dependencies should not be omitted from the dependency model, but structural dependencies should not be engulfed by questionable logical dependencies generated by casual co-changes. Thus, in order to add logical dependencies besides structural dependencies in dependency models, class co-changes must be filtered until they remain only a reduced but relevant set of valid logical dependencies.

2.3 Tool for measuring software dependencies

To establish structural and logical dependencies, we developed a tool that takes as input the source code repository URL of a given system and extracts from it the software dependencies [28]. From a workflow point of view, we can identify 3 major types of activities that the tool does: downloads the required data from the git repository, extracts from the source code the structural dependencies and, extracts and filters the co-changing pairs from the repository's commit history. Figure 2-1 represents the activities mentioned above. Each block represents a different activity.

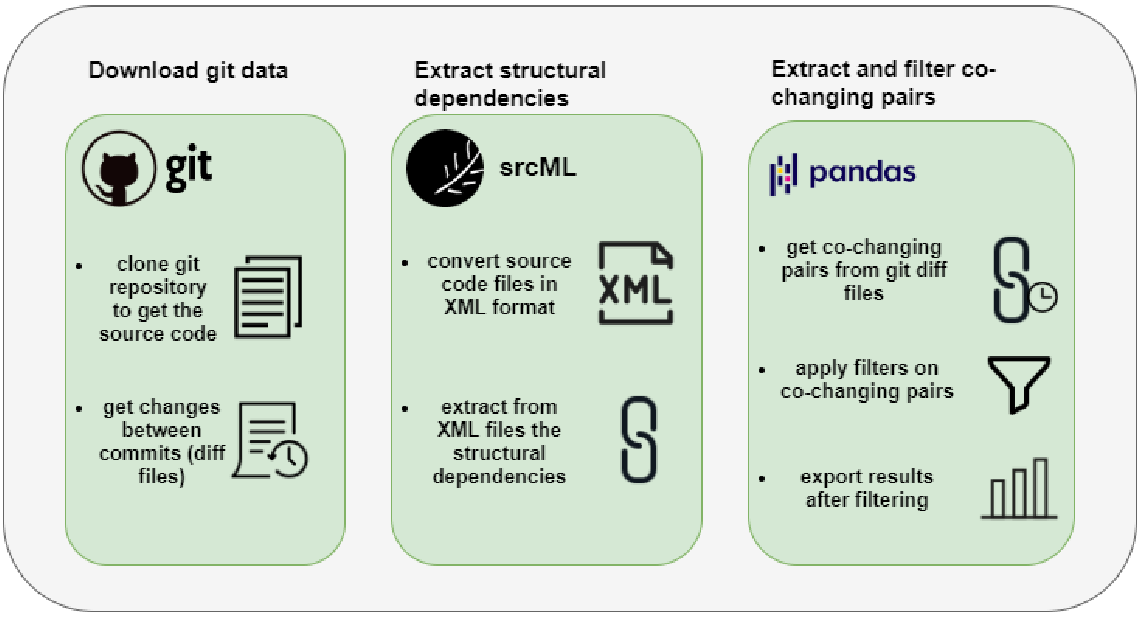

Figure 2-1: Tool workflow and major activities.

Download git data.

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

As we can see in figure 2-2, the *"clone"* command will download a Git repository to your local computer, including the source code files. The *"diff"* command will get the differences between two existing commits in the Git repository. The tool gets the Git repository and the source code files by executing the "clone" command. Afterward, it gets all the existing commits within the Git repository. The commits are ordered by date, beginning with the oldest one and ending with the most recent one. The tool executes the "diff" command between each commit and its parent (the previous commit). The "diff" command generates a text file that contains the differences between the two commits: code differences, the number of files changed and changed file names.

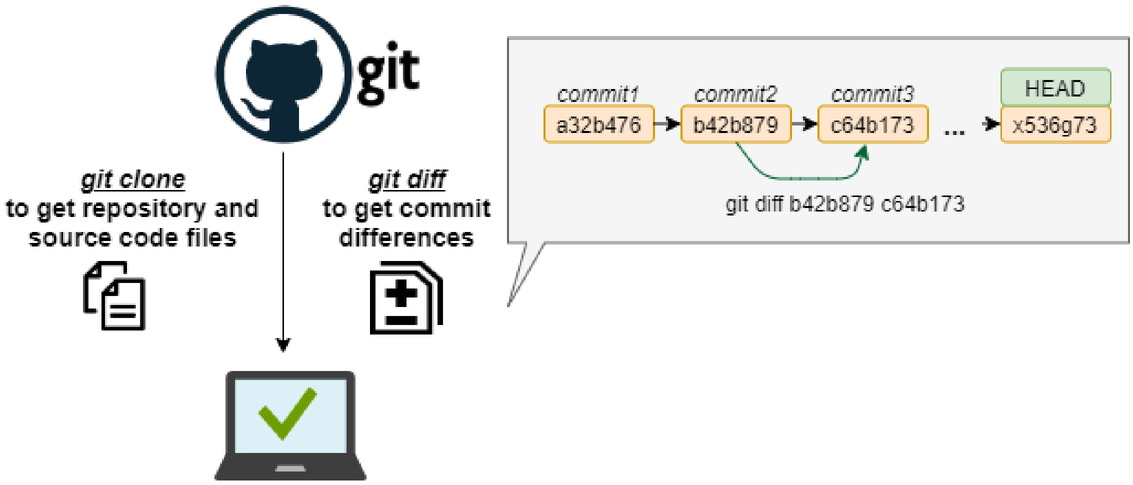

Figure 2-2: Commands used to download the required data from GitHub.

Extract structural dependencies.

To extract the structural dependencies from the source code files the tool converts each source code file into srcML format using an open-source tool called srcML. The srcML format is an XML representation for source code. Each markup tag identifies elements of the abstract syntax for the language [1]. After conversion, the tool parses each file and identifies all the defined entities (class, interface, enum, struct) within the file. It also identifies all the entities that are used by the entities defined. The connection between both types of entities mentioned above constitutes a structural dependency.

Extract and filter co-changing pairs.

The process of extracting and filtering the co-changing pairs is represented in figure 2-3. For co-changing pairs extraction, the tool parses each generated diff file. For each file, the tool gets the number of changed files and the name of the files. After structural dependencies extraction, the tool knows all the software entities contained in a file. Two entities from two changed files form a co-changing pair. After all the co-changing pairs of one diff file are extracted, the tool moves to the next diff file and extracts the set of co-changing pairs.

As will be presented in more details in sections 3.2, 3.3, and 3.4, not every co-changing pair extracted is a logical dependency. For a co-changing pair to be labeled as a logical dependency, it has to meet some criteria. Each criterion constitutes a filter that a co-changing pair has to pass in order to be called logical dependency. The filters are implemented in the tool and can be combined. The input for each filter is the set of co-changing pairs extracted, and the output is the remaining co-changing pairs that respect the filter criterion.

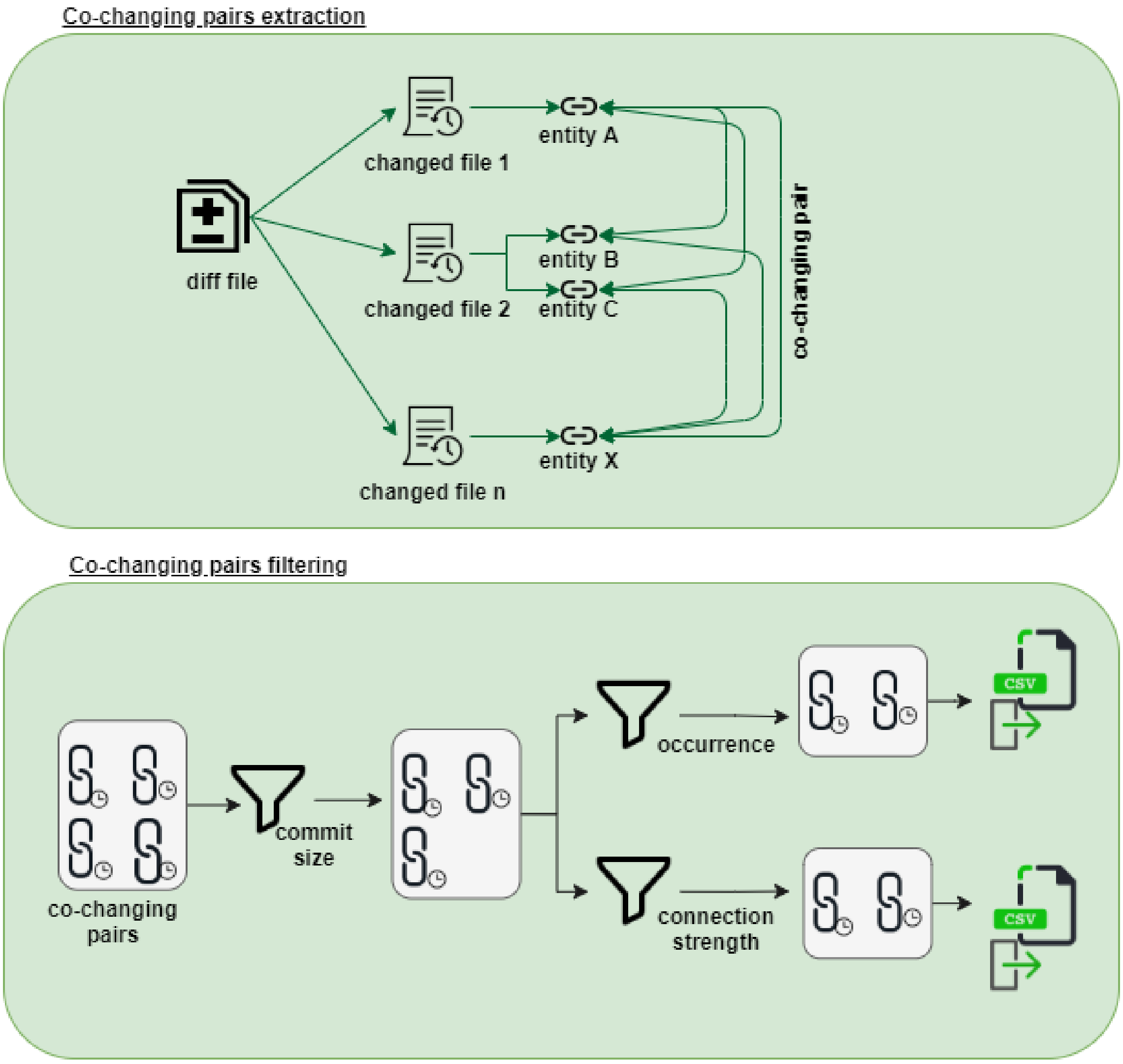

Figure 2-3: Co-changing pairs extraction and filtering.

Chapter 3

Filtering extracted co-changing pairs in order to obtain logical dependencies

3.1 Data set used

We have analyzed a set of open-source projects found on GitHub[1] [18] in order to extract the structural and logical dependencies between classes. Table 3.1 enumerates all the systems studied. The 1st column assigns the projects IDs; 2nd column shows the project name; 3rd column shows the number of entities(classes and interfaces) extracted; 4th column shows the number of most recent commits analyzed from the active branch of each project and the 5th shows the language in which the project was developed.

[1]http://github.com/

Table 3.1: Summary of open source projects studied.

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

3.2 Filtering based on the size of commit transactions

As presented in section 2.2, according to surveys, co-changing pairs are not used because of their size. One system can have millions of co-changing pairs. Whit this filtering type, we not only want to decrease the total size of the extracted co-changing pairs. But also to be one step closer to the identification of the logical dependencies among the co-changing pairs. In this step, we want to filter the co-changing pairs extracted after commit size (cs). This means that the co-changing pairs are extracted only from commits that involve fewer files than an established threshold number.

Different works have chosen fixed threshold values for the maximum number of files accepted in a commit. Cappiluppi and Ajienka, in their works [2], [3] only take into consideration commits with less then 10 source code files changed in building the logical dependencies.

The research of Beck et al [5] only takes in consideration transactions with up to 25 files. The research [20] provided also a quantitative analysis of the number of files per revision; Based on the analysis of 40,518 revisions, the mean value obtained for the number of files in a revision is 6 files. However, standard deviation value shows that the dispersion is high.

We analyzed the overall transaction size trend for 27 open-source csharp and java systems with a total of 74 332 commits. The results are presented in Figure 3-1 and in table 3.2, based on them we can say that 90% of the total commit transactions made are with less than 10 source code files changed. This percent allows us to say that setting a threshold of 10 files for the maximum size of the commit transactions will not affect so much the total number of commit transactions from the systems since it will still remain 90% of the commit transactions from where we can extract co-changing pairs [28].

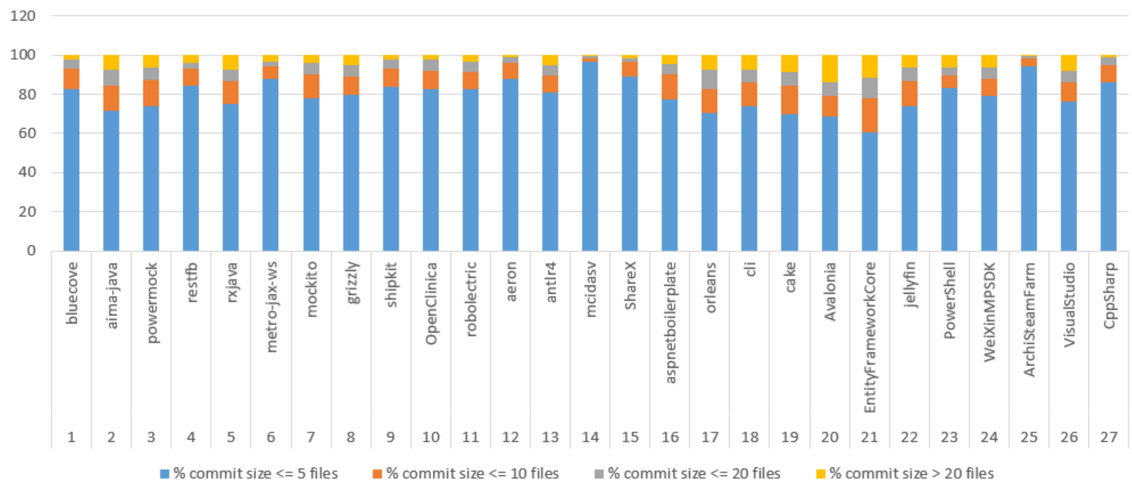

Figure 3-1: Commit transaction size(cs) trend in percentages.

As we can see in Figure 3-2 even though only 5% of the commit transactions

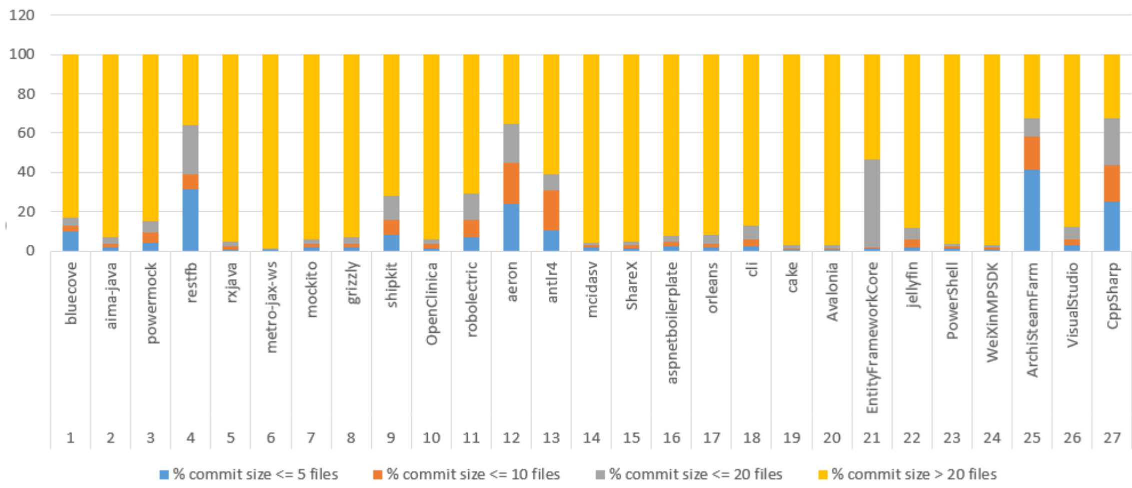

Figure 3-2: Percentages of LD extracted from each commit transaction size(cs) group.

have more than 20 files changed ($20 < cs < inf$) they generate in average 80% of the total amount of co-changing pairs extracted from the systems. The high number of co-changing pairs extracted from such a small number of commit transactions is caused by the number of files involved in those commit transactions.

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

So filtering 10% of the total amount of commit transactions can lead to a significant decrease of the amount of co-changing pairs and that is why we choose the value of 10 files as our fixed threshold for the maximum size of a commit transaction [28].

3.3 Filtering based on number of occurrences

In the previous section, we filtered the co-changing pairs based on the commit size. Even though the number of extracted co-changing pairs was reduced, this type of filtering will not guarantee that the remaining co-changing pairs can pass as logical

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

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

dependencies. One occurrence of a co-change pair can be a valid logical dependency, but can also be a coincidence.

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

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

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

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

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

Based on Table 3.3 we can say that only a small percentage of the extracted co-changing pairs are also structural dependencies. This is consistent with the findings of related works [2], [3]. The percentage of co-changing pairs that are also structural dependencies increases with the minimum number of occurrences because the number of co-changing pairs from the systems decreases with the minimum number

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

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

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

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

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

3.4 Filtering based on connection strength

In section 3.2 we filtered the co-changing pairs extracted from the versioning system history based on the commit size. Based on the results obtained, we decided to filter out all co-changing pairs extracted from commits with more than 10 files changed.

In section 3.3, we added a new filtering rule based on the occurrence of a co-changing pair. The new filter is applied to the co-changing pairs resulted after commit size filtering. In this case, the filtering method proved insufficient due to the size diversity of the systems. One important conclusion drawn from the occurrence number filtering is that setting a hard threshold for a filter is not always a good idea. One threshold value can be too much for a small-sized system and too little for a medium-sized system.

To avoid the above problem, we decided to introduce another filter complementary to the commit size filter described in section 3.2. This filter focuses on the connection strength of a co-changing pair. In this section, we will filter out all the co-changing pairs that are not strongly connected.

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

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

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

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

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

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

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

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

We calculate the ratio number between the co-changing pairs and the structural dependencies because we want to evaluate the size of the extracted co-changing pairs compared to the size of the structural dependencies from the system. According to surveys [25], [14], the main reason why logical dependencies (a.k.a filtered co-changes) are not used together with structural dependencies is because of their size. So, it is important to us to get at each filtering step an overview regarding the ratio between co-changes size and structural dependencies size.

From the results presented in tables 3.5 and 3.6 we conclude that the number of co-changing pairs is drastically reduced. In most cases, the number of structural

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

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

dependencies surpasses the number of co-changing pairs that remain after filtering. But, we do the filtering not only to reduce the size of the co-changing pairs extracted. We do the filtering of co-changing pairs extracted to make sure that the remaining co-changing pairs are indeed logically dependent.

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

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

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

3.5 Overlaps between structural and co-changing pairs

A logical dependency can be also a structural dependency and vice-versa, so studying the overlapping between logical and structural dependencies while filtering is important since the intention is to introduce those logical dependencies among with structural dependencies in architectural reconstruction systems. Current studies have shown a relatively small percentage of overlapping between them with and without any kind of filtering [2]. This means that a lot of non related entities update together in the versioning system, the goal here is to establish the factors that determine such a small percentage of overlapping [27].

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

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

- with comments: a change in source code files is counted as a co-changing pair, even if the change is inside comments in all files
- without comments: commits that changed source code files only by editing comments are ignored

In all cases, we varied the following threshold values:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

From Tables 3.11, 3.12, and 3.14 we notice that the percentage of co-changes which are also structural is always low to very low. This means that most co-changes are recorded between classes that have no structural dependencies to each other [27].

Chapter 4

Conclusions

In order to obtain logical dependencies, we have to filter the co-changes extracted from the versioning system. The filtering will increase the confidence that the remaining co-changing pairs are indeed logically coupled. And also will help to reduce the size of the co-changes extracted.

We defined 3 methods for filtering the co-changing pairs into logical dependencies. For each method, we analyzed the extracted information from the history of the systems defined in section 3.1, and we drew the conclusions presented below.

Filtering based on the size of commit transactions. For this, we filter out each commit transaction that has more files changed than an established threshold. This type of filtering will significantly reduce the amount of co-changing pairs extracted. Big commit transactions (more than 10 files) are rather related to refactoring of names, spellchecks, or file reformating and not to actual code changes. Less than 10% of the total commits are commits with more than 10 files. So filtering out every commit that has more than 10 files changed will not impact so much the size of the studied information from the versioning system. After filtering we are remaining with 90% of commits from which we can extract co-changing pairs.

Filtering based on the number of occurrences. In this step, we filter out each co-changing pair that does not occur in the versioning system more than an established threshold. This type of filtering is meant to increase the confidence that the extracted co-changing pairs are logically coupled. We decided not to go any further with this type of filtering. Mainly because in our experiments, we observed that if we try to go higher with the occurrences threshold, we risk filtering all the existing co-changing pairs for some small systems. From this filtering method, we concluded that we need to consider the system size and the occurrence trends for each system in particular, rather than setting a "hard" threshold for all the studied systems.

Filtering based on connection strength. The connection strength is calculated based on the connection factors of both entities that form a co-changing pair. For 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 same applies for B with A. This filter type is meant to establish how important, to one another, the entities that form a co-changing pair are. If we filter out all the co-changing pairs that do not update at least half of the time together (factor A and factor B $\geq 50\%$) we remain with a decent quantity of co-changing pairs. And at this point, we can say that the remaining co-changing pairs are logical dependencies.

Based on all the conclusions described above, further in our research, we will use the filter based on commit transaction size (filter out commits that have more than 10 files changed) and the filter based on connection strength.

The research presented in this report was also published in two articles [27], [26]. As we mentioned at the beginning, we divided our work into two main parts: the historical information collection and filtering and the usage of the collected information in order to analyze legacy software systems. This report covers the first part, further, we will focus on the second part and we will use the knowledge gained and the filtering methods presented to develop methods to analyze legacy software systems using historical information.

Chapter 5

Bibliography

[1] srcml; www.srcml.org.

[2] Nemitari Ajienka and Andrea Capiluppi. Understanding the interplay between the logical and structural coupling of software classes. *Journal of Systems and Software*, 134:120–137, 2017.

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

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

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

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

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

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

[9] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, Nov 2009.

[10] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35:864–878, 2009.

[11] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based lightweight C++ fact extractor. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 134–, Washington, DC, USA, 2003. IEEE Computer Society.

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

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

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

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

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

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

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

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

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

[21] Gustavo Ansaldi Oliva and Marco Aurélio Gerosa. Experience report: How do structural dependencies influence change propagation? an empirical study. In *26th IEEE International Symposium on Software Reliability Engineering,*

ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015, pages 250–260, 2015.

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

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

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

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

[26] Adelina Diana Stana and Ioana Şora. Identifying logical dependencies from co-changing classes. In *Submitted to The 7th International Workshop on Software Mining (SoftwareMining) at ASE 2018*, 2018.

[27] Adelina Diana Stana. and Ioana Şora. Identifying logical dependencies from co-changing classes. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 486–493. INSTICC, SciTePress, 2019.

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

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

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

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

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