Methods and Tools for the Analysis of Legacy Software Systems

Department of Computers and Information Technology
2021

Ph.D. student: Stana Adelina Diana

Scientific supervisor: prof.dr.ing Cretu Vladimir-Ioan

Contents

Chapter 1

Introduction

This report presents the results obtained so far on the proposed thesis. The goal of the thesis is to develop methods for analyzing legacy software systems, focusing on using historical information describing the evolution of the systems extracted from the versioning systems.

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 2.1. The primary information extracted by the tool is described in sections 2.2 and 2.3. The filtering methods selected to be applied to the extracted information are presented in sections 3.2, 3.3, and 3.4.

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.

Each filtering section (3.2, 3.3, and 3.4) contains the detailed results obtained after analyzing all the software systems and conclusions based on the results.

Section 3.5 focuses on the overlappings between the extracted information from

the code and 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 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 [30]. 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.

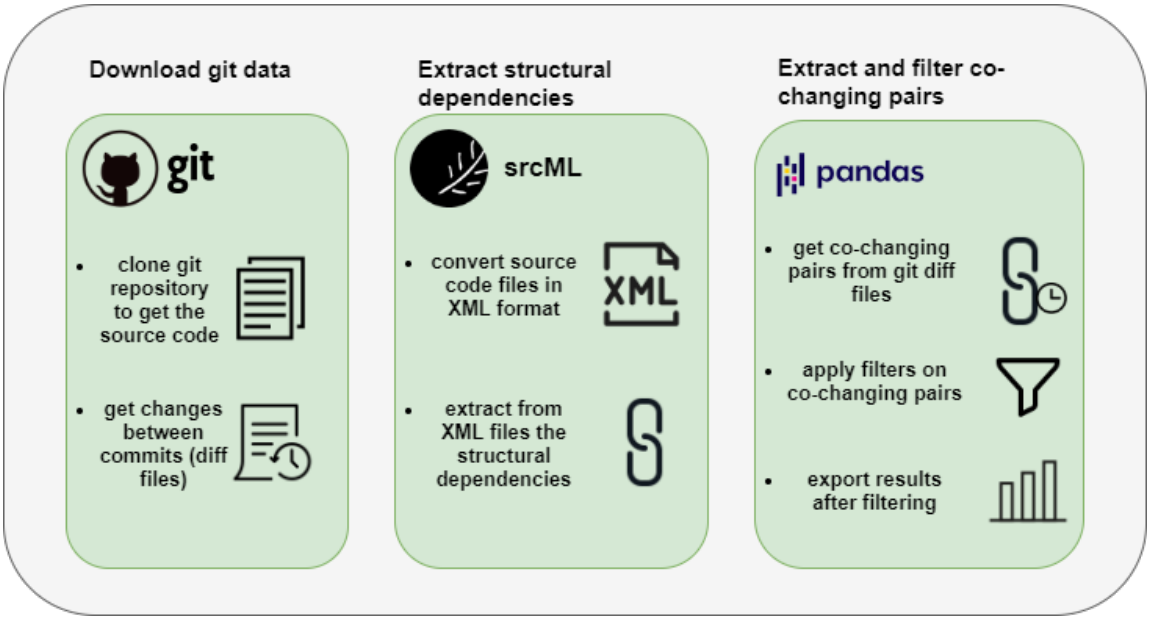

Figure 2-1: Tool workflow and major activities.

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 2.2 and the change history for co-changing pairs extraction 2.3. 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.

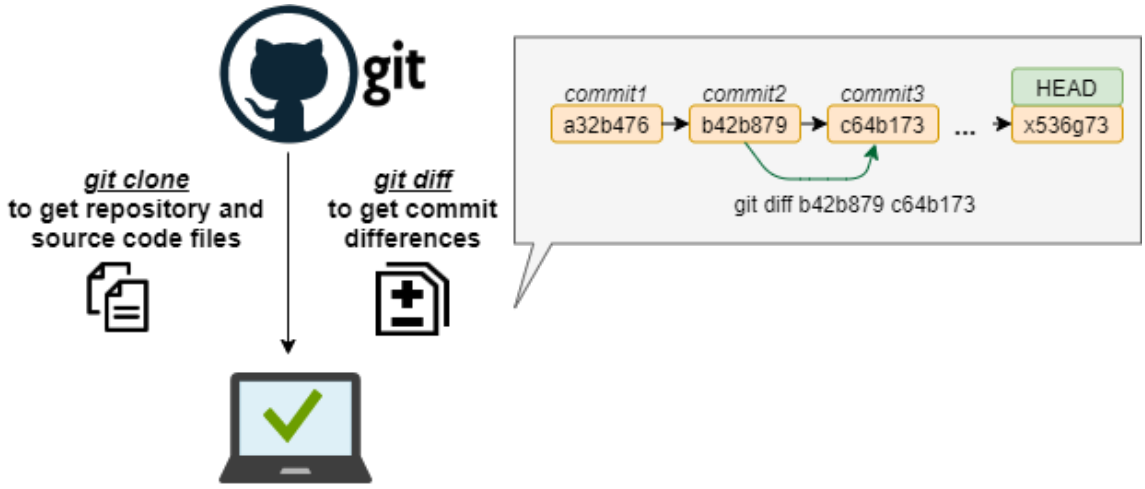

Figure 2-2: Commands used to download the required data from GitHub.

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 presented 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.

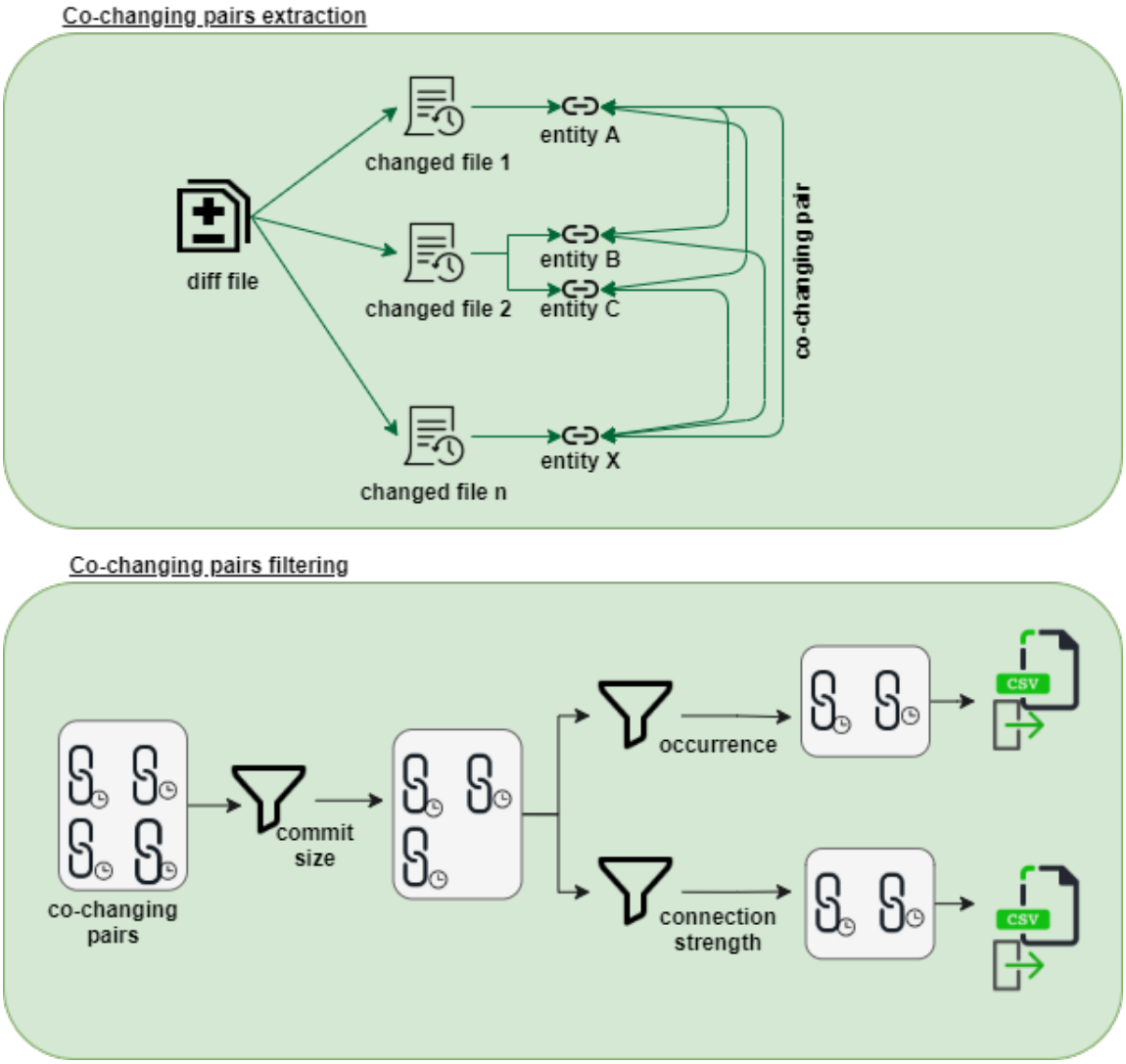

Figure 2-3: Co-changing pairs extraction and filtering.

2.2 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 [27], [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#.

2.3 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 [26], for identifying the potential ripple effects caused by software changes during software maintenance and evolution [24], [23], [25], [20] or for their link to deffects [31], [34].

Software engineering practice has shown that sometimes modules which do not present structural dependencies still can be related [32]. Co-evolution represents

the phenomenon when one component changes in response to a change in another component [33], [9]. Those changes can be found in the software change history from the versioning system. Gall [18], [19] 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 [22]. 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 [28], [17].

Other works have tried to filter co-changes [23], [2], [24]. 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.

Chapter 3

Filtering extracted co-changing pairs

3.1 Data set used

We have analyzed a set of open-source projects found on GitHub[1] [21] 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 size of commit transactions

As presented in section 2.3, 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 [23] 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 [30].

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

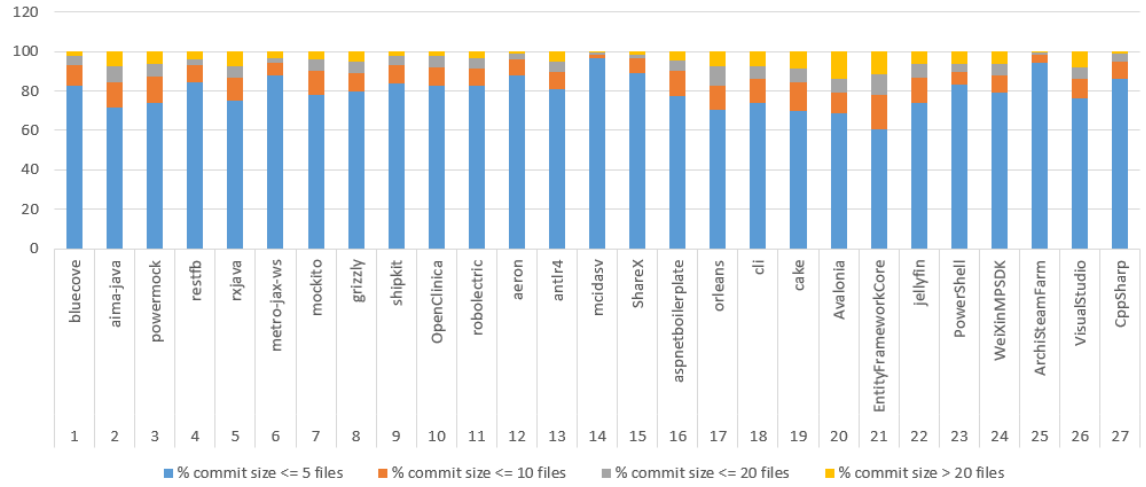

Figure 3-1: Commit transaction size(cs) trend in percentages.

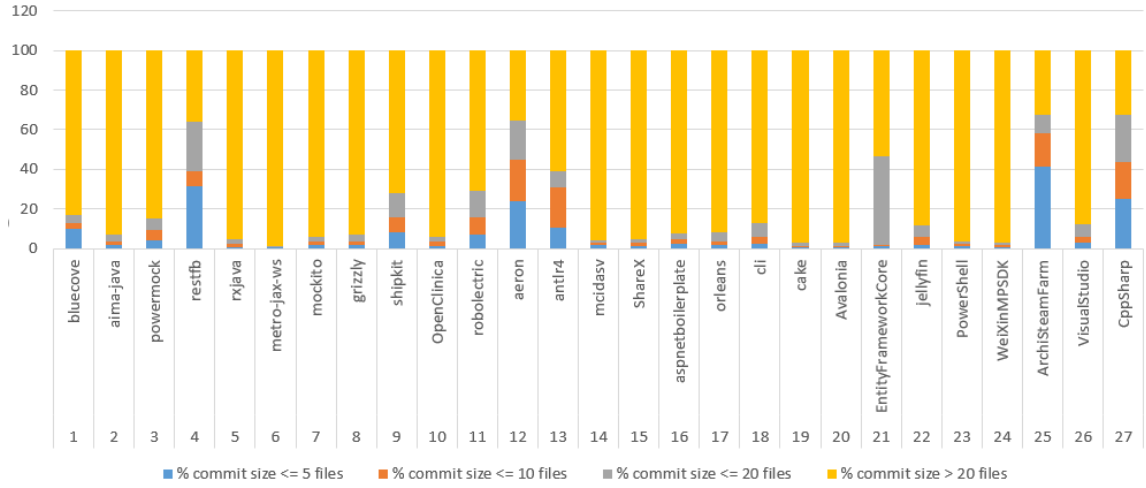

Figure 3-2: Percentages of LD extracted from each commit transaction size(cs) group.

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 sig-

nificant 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 [30].

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

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 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).

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

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

number of co-changing pairs from the systems decreases with the minimum number 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.

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

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 strenght

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.

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

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

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 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 number of co-changing pairs that resulted after filtering out pairs that have both factors below the specified threshold in the column header.

Table 3.5: Commit strength AND

Project	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 100%
bluecove	3561	1768	1163	1077	656	535	523	58	30	30
aima-java	1392	716	365	231	190	102	24	20	18	18
powermock	1415	680	411	240	172	88	87	87	87	87
restfb	2755	546	152	58	36	7	3	3	3	3
rxjava	4940	2510	1145	774	718	370	265	228	154	154
metro-jax-ws	1474	1007	657	501	455	231	118	112	106	106
mockito	8252	4171	1853	1496	1116	458	269	189	168	168
grizzly	22158	3136	1815	1189	996	537	416	360	229	76
shipkit	955	306	173	91	69	38	30	7	5	5
OpenClinica	2444	1306	894	749	600	405	232	185	182	167
robolectric	1018	771	575	330	242	220	10	4	4	4
aeron	1153	566	352	287	219	189	164	61	28	15
antlr4	53939	3424	145	47	31	17	1	1	1	1
mcidasv	10553	2635	2159	1744	1614	1484	1264	1165	16	16
ShareX	26150	3109	2843	2417	2143	1038	755	728	4	4
aspnetboilerplate	12648	3232	2129	1422	917	441	133	78	64	64
orleans	13850	10852	9348	8527	8410	3324	2693	2577	2404	2404
cli	3458	480	329	243	210	128	119	60	53	53
cake	28621	9227	7526	4132	922	263	80	46	45	45
Avalonia	14147	1952	1648	304	212	38	14	11	11	11
EntityFrameworkCore	169462	84850	80670	79479	79087	65752	41	35	32	32
jellyfin	1160	53	25	20	14	2	1	1	1	1
PowerShell	4166	3124	381	128	17	3	0	0	0	0
WeiXinMPSDK	23312	2364	1327	431	120	39	20	4	1	1
ArchiSteamFarm	630	336	301	297	289	283	238	6	1	0
VisualStudio	6235	436	259	137	95	64	27	4	3	3
CppSharp	1697954	17399	16929	16721	16579	15812	1329	1283	1249	1221

3.5 Overlaps between structural and logical dependencies

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

Table 3.6: Commit strength OR

Project	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 100%
bluecove	3524	3170	1880	1608	1125	631	587	124	120	120
aima-java	2251	1466	920	617	541	292	117	107	103	103
powermock	1422	918	656	500	421	257	255	254	254	254
restfb	2219	1124	410	223	196	53	49	49	49	49
rxjava	5887	4353	3056	2103	2089	1001	925	818	724	724
metro-jax-ws	1754	1691	1322	1118	1038	687	530	524	518	518
mockito	12873	7895	4691	3231	2135	1034	662	557	524	524
grizzly	14237	8953	5503	3421	3077	1519	1246	1136	961	808
shipkit	879	692	463	329	271	122	95	67	60	60
OpenClinica	8012	4190	3033	2474	2094	1251	902	788	773	698
robolectric	3262	1092	781	410	280	241	25	19	19	19
aeron	3248	1866	1102	791	664	399	258	127	88	74
antlr4	53939	3789	261	103	53	31	11	10	10	10
mcidasv	6320	3935	2807	2230	1893	1549	1296	1193	44	44
ShareX	11499	5541	3930	3129	2544	1229	899	860	73	72
aspnetboilerplate	10242	7367	4790	3534	2647	1267	646	486	446	445
orleans	19006	16322	14409	12728	12615	9482	8173	8074	7901	7901
cli	3458	1573	1156	896	773	554	489	307	293	293
cake	23081	14674	12270	6629	2274	1257	237	165	159	159
Avalonia	8538	3752	2310	624	470	179	95	52	48	48
EntityFrameworkCore	132251	94750	85042	81458	80668	66064	296	287	284	284
jellyfin	1160	263	139	93	74	24	19	19	16	16
PowerShell	8308	3963	558	195	50	10	7	7	7	7
WeiXinMPSDK	9498	4264	2316	1016	568	332	105	59	53	53
ArchiSteamFarm	3841	994	583	475	404	316	248	16	11	10
VisualStudio	6235	1148	693	447	290	193	149	97	93	93
CppSharp	938951	22909	18871	17810	17568	16765	7668	7556	7519	7489

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 [29].

In the main series of experiments, for each system, we extracted the structural dependencies and the logical dependencies 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 towards a logical dependency, even if the change is inside comments in all files
- without comments: commits that changed source code files only by editing comments are ignored as logical dependencies

In all cases, we varied the following threshold values:

- commit size (*cs*): the maximum size of commit transactions which are accepted to generate logical dependencies. 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 logical dependencies (LD) to the number of structural dependencies (SD)
- the mean percentage of structural dependencies that are also logical dependencies (calculated from the number of overlaps divided to the number of structural dependencies)
- the mean percentage of logical dependencies that are also structural dependencies (calculated from the number of overlaps divided to the number of logical dependencies)

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 LD 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 LD 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 LD, 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 LD, 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 LD 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 LD 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

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

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

From Tables 3.11 and 3.12 we notice that the percentage of logical dependencies

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 [29].

Chapter 4

Conclusions

Different applications based on dependency analysis could be improved if, beyond structural dependencies, they also take into account the hidden non-structural dependencies. For example, works which investigate different methods for architectural reconstruction [16], [14], [15], all of them based on the information provided by structural dependencies, could enrich their dependency models by taking into account also logical dependencies. However, a thorough survey [17] shows that historical information has been rarely used in architectural reconstruction. The software architecture is important in order to understand and maintain a system. Often code updates are made without checking or updating the architecture. This kind of updates cause the architecture to drift from the reality of the code over time [17]. So reconstructing the architecture and verifying if still matches the reality is important [21].

Surveys also show that architectural reconstruction is mainly made based on structural dependencies [28], [17], the main reason why historical information is rarely used in architectural reconstruction is the size of the extracted information.

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] Ioana Şora. Software architecture reconstruction through clustering: Finding the right similarity factors. In *Proceedings of the 1st International Workshop in Software Evolution and Modernization - Volume 1: SEM, (ENASE 2013)*, pages 45–54. INSTICC, SciTePress, 2013.

[15] Ioana Şora. Helping program comprehension of large software systems by identifying their most important classes. In *Evaluation of Novel Approaches to Software Engineering - 10th International Conference, ENASE 2015, Barcelona, Spain, April 29-30, 2015, Revised Selected Papers*, pages 122–140. Springer International Publishing, 2015.

[16] Ioana Şora, Gabriel Glodean, and Mihai Gligor. Software architecture reconstruction: An approach based on combining graph clustering and partitioning. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 259–264, May 2010.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.