The Impact of Task Granularity on Co-evolution Analyses

Keisuke Miura
Kyushu University, Japan
miura@posl.ait.kyushu-u.ac.jp

Shane McIntosh
McGill University, Canada
shane.mcintosh@mcgill.ca

Yasutaka Kamei
Kyushu University, Japan
kamei@ait.kyushu-u.ac.jp

Ahmed E. Hassan
Queen's University, Canada
ahmed@cs.queensu.ca

Naoyasu Ubayashi
Kyushu University, Japan
ubayashi@ait.kyushu-u.ac.jp

ABSTRACT

Background: Substantial research in the software evolution field aims to recover knowledge about development from the project history that is archived in repositories, such as a Version Control System (VCS). However, the data that is archived in these repositories can be analyzed at different levels of granularity. Although software evolution is a well-studied phenomenon at the revision-level, revisions may be too fine-grained to accurately represent development tasks.
Aim: In this paper, we set out to understand the impact that the revision granularity has on co-change analyses.
Method: We conduct an empirical study of 14 open source systems that are developed by the Apache Software Foundation. To understand the impact that the revision granularity may have on co-change activity, we study work items, i.e., logical groups of revisions that address a single issue.
Results: We find that work item grouping has the potential to impact co-change activity, since 29% of work items consist of 2 or more revisions in 7 of the 14 studied systems. Deeper quantitative analysis shows that, in 7 of the 14 studied systems: (1) 11% of largest work items are entirely composed of small revisions, and would be missed by traditional approaches to filter or analyze large changes, (2) 83% of revisions that co-change under a single work item cannot be grouped using the typical configuration of the sliding time window technique and (3) 48% of work items that involve multiple developers cannot be grouped at the revision-level.
Conclusions: Since the work item granularity is the natural means that practitioners use to separate development tasks, future software evolution studies, especially co-change analyses, should be conducted at the work item level.

CCS Concepts

•Software and its engineering → Software configuration management and version control systems; Collaboration in software development; •General and reference → *Empirical studies; Evaluation;*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ESEM '16, September 08-09, 2016, Ciudad Real, Spain
© 2016 ACM. ISBN 978-1-4503-4427-2/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2961111.2962607

Keywords

Task granularity, software evolution, co-change analysis

1. INTRODUCTION

Substantial research in the software evolution field aims to recover knowledge about development from the project history that is archived in repositories, such as Version Control Systems (VCSs) and Issue Tracking Systems (ITSs). For example, Neamtiu *et al.* [29] evaluate Lehman's laws of software evolution [5, 10, 24] in 705 official releases of nine open source projects. Di Penta *et al.* [8] analyze source code that is stored in VCSs to investigate the evolution of software license changes in open source projects.

There are several levels of granularity that are used in software evolution research, such as revisions (i.e., collections of changes to files that authors commit to VCSs together) [17, 19, 33], pull requests (i.e., requests to integrate collections of revisions into an upstream repository) [12, 13] and releases (i.e., milestones for delivering software to stakeholders) [6, 11, 25]. The revision-level is a popular granularity that many researchers target [18, 34, 35] because revisions can be easily extracted from a VCS [15].

However, as pointed out by our prior work [27], the revision-level is too fine of a granularity for some contexts of software evolution analysis. For example, while some issues might be addressed using a single code revision, others may require several revisions by several developers (see Section 2.1). Such a fine granularity may impact the findings that are derived from co-change analyses, which study evolutionary dependencies between entities (e.g., files) [33, 35].

Alternatively, work items, which embody development tasks like fixing a bug or adding a new feature, can be used to study software evolution. Work items may consist of multiple revisions that relate to a single development task [27]. While a few recent studies are performed at the work item level [26, 27], the impact of co-change granularity remains largely unexplored. Therefore, we set out to empirically study the characteristics of work items to understand the impact that work items can have on co-change analyses.

Through a case study of 14 systems that are maintained by the Apache Software Foundation, we find that over 29% of work items consist of 2 or more revisions in 7 of the 14 studied systems. This initial finding shows that work items hold the potential to impact to the findings of co-change analysis. Hence, we structure the remainder of our study along the following three research questions:

RQ1: How many different files are changed across the revisions of work items?
Motivation: If different files are modified across the revisions of work items, revision-level analysis may miss co-changing files by ignoring related revisions.
Exploratory analysis: Analysis of only the first revision of work items would miss 2%-43% of the co-changed files in the studied systems.
Impact analysis: We find that 11% of the largest work items are entirely composed of small revisions (i.e., the revisions that contain a small number of co-changed files) in 7 of the 14 studied systems. These collections of small revisions would not be detected by filtering or analysis techniques that focus on the revision-level.

RQ2: How much time elapses between the revisions of work items?
Motivation: If the time between revisions of a work item is too long, traditional revision-grouping techniques may not be able to discover their relationship.
Exploratory analysis: At least 1 hour elapses between the revisions of 64% of the studied work items.
Impact analysis: In 7 of the 14 studied systems, 83% of related revisions cannot be grouped using the common sliding time window of 300 seconds.

RQ3: How many developers are involved across revisions of work item?
Motivation: Developers across different teams may need to collaborate in order to address a work item. Since revision grouping techniques will miss related revisions if they are submitted by different developers [22, 36], we set out to study how often developers collaborate in order to address work items.
Exploratory analysis: 8%-41% of work items are modified by multiple developers in the studied systems.
Impact analysis: In 7 of the 14 studied systems, traditional revision-grouping would miss revisions of 48% of work items that involve multiple developers.

Implications. Our results indicate that traditional techniques for grouping revisions (i.e., co-change analyses) may miss a variety of types of co-change that are visible at the work item level. Since work items are the natural means of separating development tasks, we recommend that future software evolution studies, especially co-change analyses, should be performed at the work item level.

Paper organization. The rest of the paper is organized as follows. Section 2 provides background details about the different evolution granularities and situates this paper with respect to the related work. Section 3 describes the design of our empirical study, while Sections 4 and 5 present the results. Section 6 discloses the threats to the validity of our work. Finally, Sections 7 draws conclusions.

2. BACKGROUND & RELATED WORK

2.1 Background

Software developers modify source code files to add new functionality and fix bugs. The modified files are recorded in a VCS. A popular unit of change is the revision, i.e., a change made to a file or collection of files that were submitted to a VCS together. However, as pointed out by our prior work [27], the revision-level is often too fine-grained to accurately depict a work item, which embodies a development task, such as fixing a bug or adding a new feature [26]. Several revisions may be required to address a single work item. For example, as shown in Figure 1, a developer 'X' changes files `A` and `testA` using one revision. On the other hand, a developer 'Y' changes file `testB` under revision `R2` and file `B` under revision `R3` even though the changes are related to the same work item.

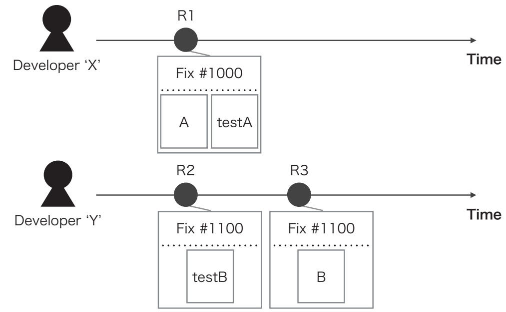


Figure 1: An example of the relation between revisions and work items.

Many software evolution studies are conducted at the revision-level. While these studies make important observations, revision-level analysis is not without limitations (e.g., granularity mismatch with development tasks). Figure 1 shows that co-change analysis at the revision-level studies would detect that file `A` and `testA` have co-changed together. However, revision-level analysis would miss the co-change relationship between file `B` and `testB` because they are changed within different revisions. In the next section, we introduce related work and explain how revision-level analysis may impact the findings of the related work.

2.2 Related Work

Co-change coupling studies [1, 9, 14] discover the groups of entities (e.g., files) that tend to change together. For example, Hassan and Holt [14] apply association rule mining to version histories in order to predict how changes propagate among software modules. Alali *et al.* [1] study the distance among co-changing entities and the age of co-change patterns in order to mitigate false positives in co-change activity. Gîrba *et al.* [9] detect groups of entities that change at the same time using concept analysis. These studies mine VCS data at the revision-level, and hence, may miss co-change relationships that are visible at the work item level.

Recently, researchers have also applied co-change analyses to study the use of software design patterns and anti-patterns. For example, Mondal *et al.* [28] apply co-change analyses to *code clones* (i.e., syntactic or semantic duplication of program logic)—a common anti-pattern in large software systems. Aversano *et al.* [4] perform an empirical study on how design patterns are changed from release to release. Kim *et al.* [21] study the role of API-level refactoring during software evolution. Similar to other co-change analyses, a work item level analysis for co-change studies that apply to software design would yield more complete results.

There are some studies on understanding the characteristics of source code changes. Hindle *et al.* [17] study the

rationale behind large commits (i.e., revisions that change a large number of files) by manually classifying large revisions in order to reveal whether or not large revisions are noise for software evolution analyses. Through analysis of 9 open source projects, they show that many large revisions perform fundamental development activities, such as merging branches, adding new features, and updating documentation. However, since this classification was performed on large revisions, it may miss large work items, which could appear as a collection of small revisions in the VCS history.

Recent work suggests that *supplementary fixes*, i.e., revisions that address an issue that was previously fixed, can impact software analyses. Through analysis of 3 open source projects, Park *et al.* [30] show that supplementary fixes impact change recommendation systems based on code clone detection analysis. An *et al.* [2] build accurate models to predict whether or not supplementary fixes will need to be re-opened. These findings further motivate our quantitative exploration of the impact that work items can have on co-change analyses.

In short, revision-level analysis may miss co-changing entities, since revisions do not depict complete development tasks. In this paper, we quantitatively explore the impact that work item grouping can have on co-change analyses.

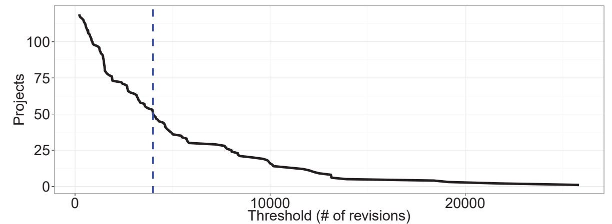


Figure 2: Threshold plot using system size.

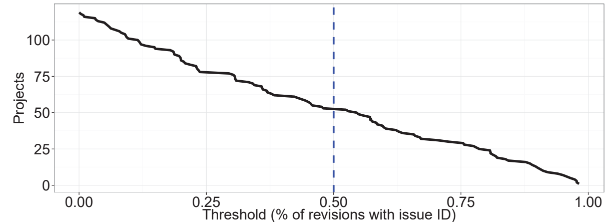


Figure 3: Threshold plot using the amount of use of an ITS.

3. CASE STUDY DESIGN

In this section, we provide our rationale for selecting our studied systems and describe how we group the revision data from VCSs into work items.

3.1 Studied Systems

We began our study with all of the systems that are maintained by the Apache Software Foundation (ASF)—a large and successful open source software community. We then identified the systems that use ITS by checking the list of systems in the ASF ITS.[1] In total, we collected data from 119 systems that we found were using the ASF ITS. However, we identified two important criteria that needed to be satisfied in order for a system to qualify for our analysis:

1. **System size.** We want to perform a case study on large open source systems (i.e., systems that contain a large number of revisions) because we need plenty of change activity to avoid unstable results. In order to select an appropriate threshold for system size, Figure 2 plots hypothetical threshold values of the number of revisions in a system against the number of surviving systems. In this study, we choose systems that have more than 4,000 revisions because we retain 42% of all systems, while removing plenty of small systems.

2. **ITS usage.** We want to study systems that have a large number of revisions that include work item IDs and frequently use an ITS in their day-to-day development. Figure 3 plots hypothetical threshold values of the percentage of revisions that include a work item ID. Since there does not appear to be a knee in the threshold plot, we choose the systems that include work item IDs in 50% of all of their revisions.

[1]https://issues.apache.org/jira/secure/BrowseProjects.jspa#all

Surviving systems. Table 1 provides an overview of the 14 Apache systems that satisfy our criteria for analysis. All 14 systems use the Git VCS. Accumulo is a high performance data storage and retrieval system. Ambari is a system for managing and monitoring Hadoop clusters. Camel is a rule-based routing and mediation engine in a variety of domain-specific languages. Hbase is a distributed and scalable Hadoop database. Hive provides a mechanism for interactive SQL queries over big data in Hadoop. Jackrabbit is a content repository, which stores, manages and serves the digital content. OpenJPA is a Java persistence tool that is used in lightweight frameworks, such as Tomcat and Spring. Qpid develops the messaging systems for AMOP, which is an open Internet protocol for sending and receiving messages. Sling is a framework for developing REST-based web applications. Stanbol provides a set of components for semantic content management (e.g., knowledge models).

3.2 Work Item Aggregation

We group revisions into work items using the commit messages of each revision. When developers modify files to address an issue and submit the changes to a VCS, they record the addressed work item ID in the message of the revision.

Similar to a previous study [19], we identify work item IDs using the `/PROJECT_NAME.?(\d+)/i` regular expression (e.g., *Accumulo-1000*, *Accumulo:1000*). We use `/PROJECT_NAME.?(\d+)[\.\-]\d/i` to exclude the patterns that are likely to be a version number (e.g., AMBARI1.3.1). We group revisions that reference the same ID under one work item [31].

4. PRELIMINARY STUDY OF WORK ITEM DISTRIBUTION

Motivation. To understand the potential impact that work item grouping may have on co-change activity, we first investigate how often work items are composed of several revisions. To do so, we examine the distribution of the number

Table 1: An overview of the studied systems.

System	Period	# of revisions	# of revisions that include work item ID	% of revisions that include work item ID	# of work items
Accumulo	2011/10/04 - 2015/03/13	4,860	4,750	97.7	2,140
Ambari	2011/08/30 - 2015/03/15	9,936	9,756	98.2	8,529
Camel	2007/03/19 - 2015/03/16	19,157	11,209	58.5	6,287
Cassandra	2009/03/02 - 2015/03/16	10,925	6,243	57.1	4,751
Cayenne	2007/01/21 - 2015/03/12	4,297	2,561	59.6	855
Derby	2004/08/11 - 2015/03/12	8,018	6,519	81.3	3,509
Hbase	2007/04/03 - 2015/03/16	10,115	8,957	88.6	7,289
Hive	2008/09/02 - 2015/03/14	5,775	5,622	97.4	5315
Jackrabbit	2004/09/13 - 2015/03/13	8,027	5,038	62.8	2,610
Karaf	2007/11/26 - 2015/03/10	4,588	2,913	63.5	1,809
OpenJPA	2006/03/02 - 2015/03/12	4,696	3,143	66.9	1,503
Qpid	2006/09/14 - 2015/03/16	13,904	7,623	54.8	4,253
Sling	2007/09/09 - 2015/03/16	12,228	6,997	57.2	3,411
Thrift	2008/03/16 - 2015/03/15	4,003	2,338	58.4	1,871

Table 2: The percentage of work items that are composed of 2 or more revisions.

System	% of work items	System	% of work items
Accumulo	44	Hive	5
Ambari	9	Jackrabbit	27
Camel	33	Karaf	23
Cassandra	19	OpenJPA	41
Cayenne	62	Qpid	33
Derby	32	Sling	34
Hbase	15	Thrift	14
		Median	29

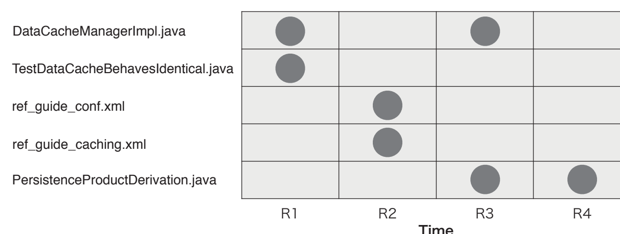


Figure 4: OpenJPA #1763 - an example of file spread.

of revisions per work item in each of the studied systems. The work items that have 2 or more revisions have the potential to impact revision-level analysis, since the co-change activity that is spread across the revisions of these work items will not be detected.

Approach. For each of the 14 studied systems, we group revisions into work items (see Section 3.2). Then, we count the number of revisions in each work item and study its distribution for each studied system.

Results. Table 2 shows the percentage of work items that consist of 2 or more revisions. The table indicates that the median value is 29%, the minimum value is 5% (Hive) and the maximum value is 62% (Cayenne). Indeed, over 29% of work items consist of 2 or more revisions in 7 of the 14 studied systems. Therefore, we suspect that we may obtain different results by conducting co-change analyses at the work item level instead of the revision-level.

A revision-level analysis would miss co-change activity in 5%-62% of work items in the studied systems. At least 29% of work items consist of 2 or more revisions in 7 of the 14 studied systems.

5. CASE STUDY RESULTS

In this section, we describe the results of our case study, which aims to better understand the impact that revision-level analysis can have on software evolution studies with respect to our research questions on file spread (RQ1, i.e., the files that are changed across the revisions of work items), time spread (RQ2, i.e., the time that elapses between the revisions of work items) and developer spread (RQ3, i.e., the number of developers who submitted revisions to address each work item).

For each research question, we describe the motivation, our approach to addressing the question, empirical observations about the distributions of work items and the impact that that type of spread can have on co-change analyses. We focus our study on the work items that consist of 2 or more revisions because they may be impacted by analysis at the work item level.

(RQ1) File spread: How many different files are changed across the revisions of work items?

Motivation. If developers modify different files after the first revision of a work item, co-change analysis at revision-level will miss these related files.

Approach. We use the number of changed files within a work item as the *file spread*. To understand if the same files are changed, we split the modified files into those that are changed in the first revision of the work item and those that are changed after the first revision of the work item.

To understand file spread across the revisions of work items, we compute the percentage of files that are changed after the first revision of a work item. Specifically, we divide the number of the unique files that are changed after the first revision by the total number of unique files of the work item. For example, Figure 4 illustrates the work item OPENJPA-

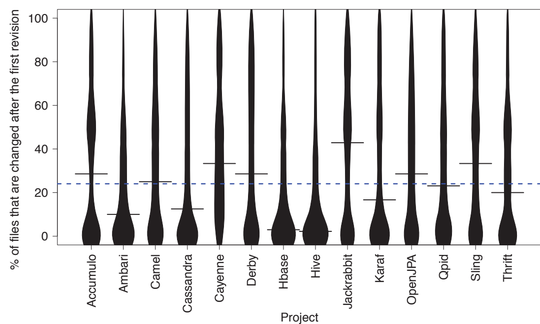


Figure 5: The percentage of the files that are changed after the first revision in a work item.

1763, which is composed of four revisions.[2] As shown in Figure 4, while file `DataCacheManagerImpl.java` and `TestDataCacheBehavesIdentical.java` are modified by the first revision, `ref_guide_conf.xml` and `ref_guide_caching.xml` are modified by the second revision, `DataCacheManagerImpl.java` and `PersistenceProductDerivation.java` are modified by the third revision and `PersistenceProductDerivation.java` is modified by the fourth revision. In this case, 60% ($\frac{3}{5}$) of the files are changed after the first revision.

Observations. Figure 5 shows the percentage of files that are changed after the first revision of the work items of the studied systems using *beanplots*. Beanplots are boxplots in which the vertical curves summarize the distribution of the dataset [20]. The solid horizontal lines indicate the median values of each system. The blue dashed line shows the median value across the 14 studied systems. The figure indicates that the smallest system-specific median value is 2% (Hive) and the largest system-specific median value is 43% (Jackrabbit). We find that the median value across the 14 studied systems is 24%. This *file spread* can impact the findings of co-change activities. In short, revision-level analyses will miss detecting co-change activities in the files that are changed after the first revision.

To gain a richer perspective about file spread, we manually inspect the work items that satisfy the following condition: the number of revisions is at least 5 and the percentage of files that are changed after the first revision is over 24% (i.e., larger than the overall median value).

For example, work item QPID-4575[3] adds support for Visual Studio 2012. A developer changes `cpp` and `h` files in the first revision,[4] then changes `csproj` files (XML-based build configuration files) in the fifth revision.[5] Revision-level co-change analysis of production code and the build system would miss this co-change activity.

Furthermore, in the work item HIVE-1081,[6] developers use 7 revisions to correct formatting and style issues in Java files. The number of changed files ranges between 21 and 480 in each revision and the total number of changed files is 757. Some revisions are not large, but are involved in the larger work item. These types of work items introduce noise in studies of large revisions (e.g., Hindle *et al.* [17])—focusing on the revision-level may miss large work items, which appear as a collection of small revisions.

Impact. To quantitatively investigate the impact that work items can have on large change detection and analyses, we show how many work items that have a large number of changed files are entirely composed of small revisions. Hindle *et al.* [17] define that a large revision as the top 1% of revisions that contain the largest number of files in a system. Through analysis of 9 open source systems, Hindle *et al.* [17] show that many large revisions perform fundamental development activities, such as merging branches, adding new features, and updating documentation. However, since this classification was performed on large revisions, it may miss large work items, which could appear as a collection of small revisions in the VCS history.

We count the number of changed files within each work item, and then sort work items by the number of changed files. Similar to Hindle *et al.* [17], we select the top 1% of work items that modify the largest number of files in each studied system (i.e., large work items). Then, we compute the percentage of these large work items that are entirely composed of small revisions (i.e. revisions that do not appear in the top 1% of the largest revisions).

Table 3: The percentage of work items that are entirely composed of small revisions.

System	% of work items	System	% of work items
Accumulo	9	Hive	13
Ambari	1	Jackrabbit	22
Camel	42	Karaf	26
Cassandra	6	OpenJPA	6
Cayenne	20	Qpid	5
Derby	33	Sling	14
Hbase	5	Thrift	5
		Median	11

Table 3 shows that 1%-42% of work items are entirely composed of small revisions (11% median). This indicate that an analysis of large changes that focuses on the revision-level will miss 11% of the largest changes in 7 of the 14 studied systems.

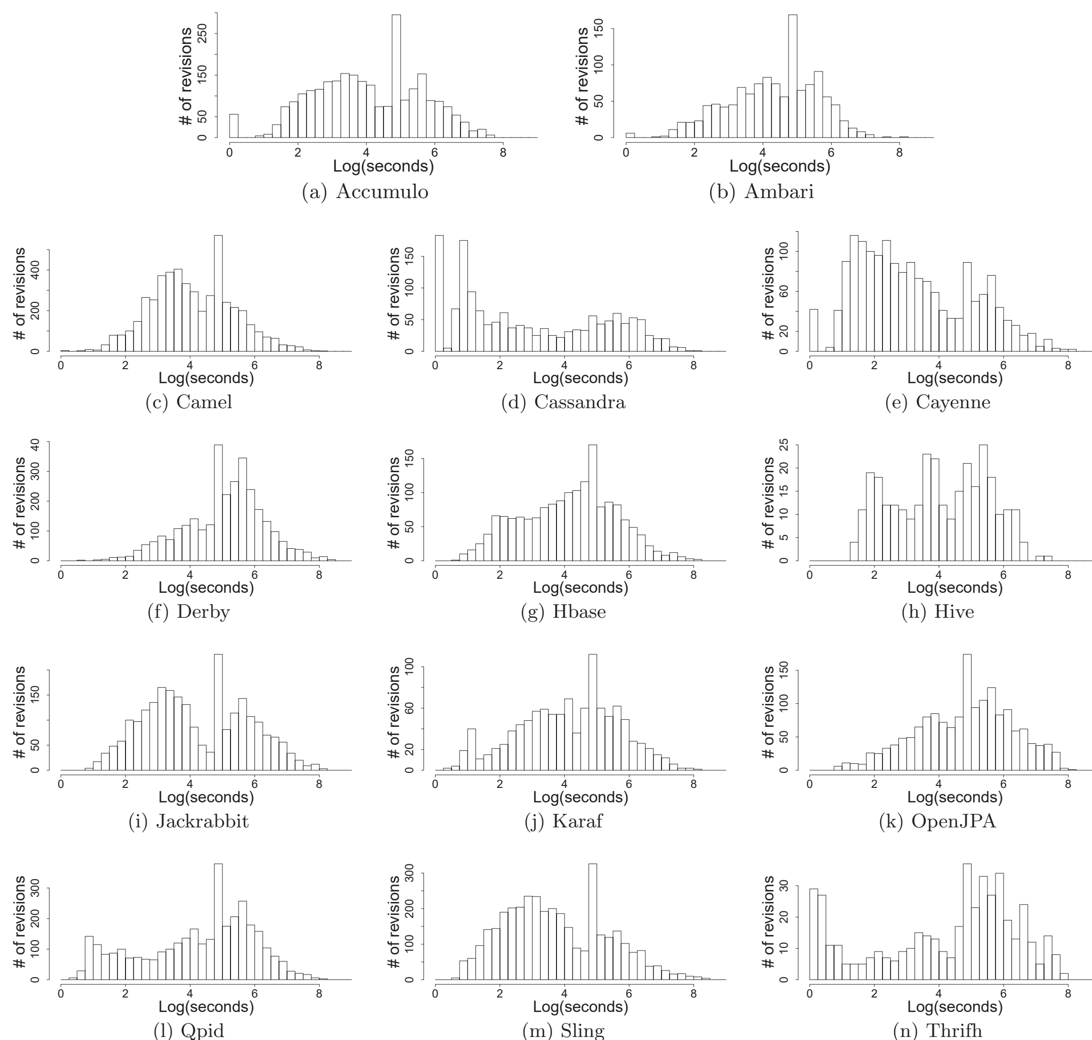
Analysis of only the first revision of work items (i.e., revision-level analysis) would miss 2%-43% of the co-changed files. Furthermore, at least 11% of large work items are entirely consist of small revisions in 7 of the 14 studied systems.

(RQ2) Time spread: How much time elapses between the revisions of work items?

Motivation. If the time between a revision and the next revision is too long, we may not be able to detect that these revisions correspond to a single task when using revision-centric techniques like the sliding time window.

Approach. We compute the *time spread* $T_{w,r}$ of each pair of subsequent revisions r_{x-1} and r_x in a work item w by computing the elapsed time (in seconds) between r_{x-1} and r_x.

[2]https://issues.apache.org/jira/browse/OPENJPA-1763
[3]https://issues.apache.org/jira/browse/QPID-4575
[4]https://github.com/apache/qpid/commit/b111ea9e3690b34b47289a8f78cbaa0428f45442
[5]https://github.com/apache/qpid/commit/157eef15ba4ce70c506ed9e7d50e1ffd1364d384
[6]https://issues.apache.org/jira/browse/HIVE-1081



(a) Accumulo (b) Ambari

(c) Camel (d) Cassandra (e) Cayenne

(f) Derby (g) Hbase (h) Hive

(i) Jackrabbit (j) Karaf (k) OpenJPA

(l) Qpid (m) Sling (n) Thrifh

Figure 6: The time spread among related revisions (i.e., revisions that appear in the same work item).

Observations. Figure 6 shows the distribution of time spread values on a logarithmic scale (base 10). We find that 64% of time spread values are over 4 in log scale (i.e., about one hour). Furthermore, 14% of time spread values are over 6 in log scale (i.e., about one week).

Impact. To study the impact that *time spread* has on co-change analyses, we show how well the sliding time window technique approximates the revision grouping of work items. Some studies [22, 36] and VCS migration tools (e.g., cvs2svn[7]) use the sliding time window technique that groups related revisions into work items. If this popular technique cannot detect the majority of the revisions that belong to larger work items, it may impact co-change analyses, as well as the results that are produced by VCS migration tools.

We compute the *time spread* $T_{w,r}$ of each revision r in a work item w using the number of seconds between r and the previous revision $r-1$. We set the sliding time window to 300 seconds to match the prior studies (200 seconds used by Zimmermann *et al.* [36] + 100 seconds of buffer time). Sliding time windows are frequently used to mitigate noise in software repositories [3]. If the time spread $T_{w,r} \leq 300$ seconds, we can detect that revision r is part of the work item w using the sliding time window.

Figure 7, which depicts the three revisions of CAMEL-7525,[8] provides an illustrative example of the time spread computation. Since the second revision is committed 258 seconds after the first revision, it can be grouped because the time spread is less than 300 seconds. Since the third revision is committed 232 seconds after the second revision, it can also be grouped. Note that the time spread is 232 seconds, and not $258 + 232 = 490$ seconds.

Table 4 shows the percentage of related revisions that cannot be grouped using the 300-second sliding time window. We exclude the first revisions in work items to calculate

[7]cvs2svn is a tool for migrating a CVS repository to Subversion and Git. It sets 5 minutes as a time window to group revisions in the time window as a work item.

[8]https://issues.apache.org/jira/browse/CAMEL-7525

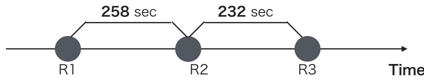


Figure 7: An illustration of the time spread computation for the three revisions of CAMEL-7525.

Table 4: The percentage of revisions that cannot be detected using a 300-second sliding time window.

System	% of revisions	System	% of revisions
Accumulo	82	Hive	79
Ambari	90	Jackrabbit	86
Camel	91	Karaf	84
Cassandra	48	OpenJPA	93
Cayenne	59	Qpid	79
Derby	97	Sling	76
Hbase	84	Thrift	76
		Median	83

the percentage, since there is no prior change from which we can compute the elapsed time. Surprisingly, the results show that 48%-97% of related revisions cannot be grouped using the sliding time window (median 83%).

We manually inspect the messages of the work items that cannot be grouped using the 300-second sliding time window. For example, developers may forget to add some related files in the first revision. We observe that there are cases where the sliding time window cannot detect related revisions that add files that were missing in the first revision. One example is ACCUMULO-1890, which describes how Accumulo recovers from a failure due to limited resources.[9] In this work item, a developer modifies a test file across two revisions. 11 minutes elapse between the first and second revisions. The commit message of the second revision[10] states that the developer: *"Forgot to re-add changes before commit."*

AMBARI-5504[11] is another example of an unintentionally omitted change. In this work item, developers update the layout of a web application using three revisions. The first revision updates 13 source code files, while the second revision, which appears 7 minutes of the first revision, adds 707 new lines of code in three new source code files. Although the commit message of the second revision does not explicitly mention that the revision is fixing a mistake in the prior revision, we believe that the files are missed in the first revision because it is unrealistic to implement 707 lines of code in 7 minutes.

The elapsed time between revisions of a work item is over an hour in 64% of cases. Furthermore, in 7 of the 14 studied systems, 83% of related revisions that belong cannot be grouped using a 300-second sliding time window.

[9]https://issues.apache.org/jira/browse/ACCUMULO-1890
[10]https://git-wip-us.apache.org/repos/asf?p=accumulo.git;h=a40a6d4
[11]https://issues.apache.org/jira/browse/AMBARI-5504

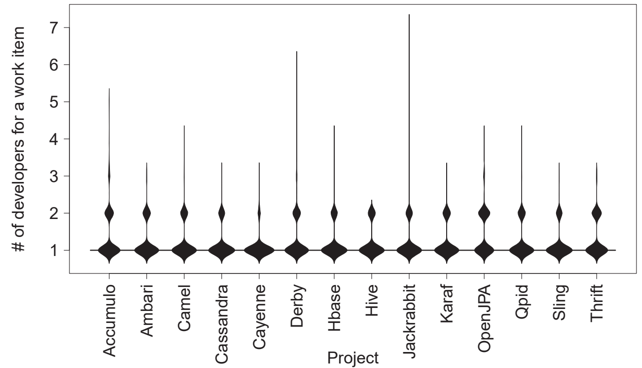


Figure 8: The number of developers per work item.

Table 5: The percentage (and count, shown in brackets) of work items modified by multiple developers.

System	% of work items modified by multiple devs	
Accumulo	33	(309)
Ambari	25	(200)
Camel	24	(503)
Cassandra	19	(172)
Cayenne	8	(42)
Derby	27	(297)
Hbase	21	(230)
Hive	22	(54)
Jackrabbit	23	(167)
Karaf	25	(104)
OpenJPA	41	(250)
Qpid	25	(344)
Sling	19	(232)
Thrift	32	(81)
median	25	(215)

(RQ3) Developer spread: How many developers are involved across revisions of a work item?

Motivation. The sliding time window technique assumes that all revisions not only appear in the same time window, but also that the same developer commits all revisions [22, 36]. However, Xuan and Filkov [32] find that synchronous development (i.e., different developers modifying the same files during a certain period) plays an important role in the development of open source systems. If several developers collaborate on a work item across revisions, the conventional techniques like sliding time window will not detect that these revisions correspond to a single task. In this question, we investigate how often the commits that address a work item are distributed among multiple developers.

Approach. We call the number of developers that are involved in a work item its *developer spread*. We compute the developer spread for each work item of the studied systems, and then investigate the distribution of developer spread. If the developer spread of a work item is more than one, we flag the work item as collaborative.

Observations. Figure 8 shows a beanplot of the number of developers that are involved in the work items of each

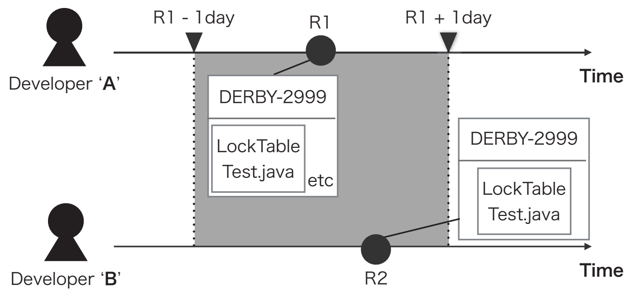


Figure 9: An illustration of the detection of synchronous development at the revision level using DERBY-2999.

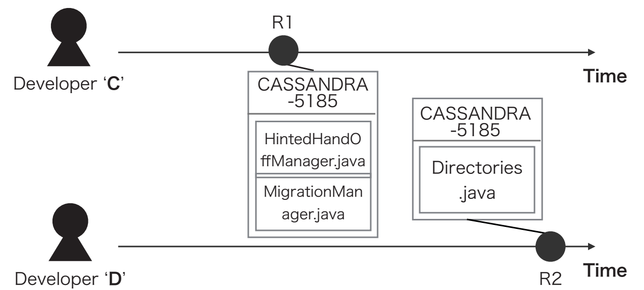


Figure 10: The example of CASSANDRA-5185—collaborative activity that can only be detected at the work item level.

Table 6: The percentage of work items that have synchronous development that cannot be detected using revision-level analysis with a variety of time window configurations.

System	% of work items		
	1 day	5 days	10 days
Accumulo	67	42	32
Ambari	33	15	14
Camel	53	28	20
Cassandra	77	63	49
Cayenne	75	68	57
Derby	77	55	44
Hbase	62	41	32
Hive	66	32	30
Jackrabbit	72	55	48
Karaf	71	57	48
OpenJPA	78	61	55
Qpid	76	57	48
Sling	76	62	55
Thrift	88	67	56
median	74	56	48

studied system. Although the majority of work items involve one developer, there are still many collaborative work items. Interestingly, there are work items that involve 5 or more developers in Accumulo, Derby and Jackrabbit.

Table 5 shows the percentage and number of work items that are modified by multiple developers. We find that 8%-41% of work items involve multiple developers. The median across the 14 studied systems is 25%.

Impact. To study the impact that *developer spread* has on co-change analyses, we study how often work items have collaborative development that cannot be detected by the method proposed by Xuan and Filkov [32], which is depicted in Figure 9. Xuan and Filkov [32] define synchronous development as a set of revisions where one file is modified by multiple developer within a time window. In DERBY-2999,[12] developers use two revisions to convert an `sql` file to a `java` test file. In Figure 9, the developer 'A' changes `LockTableTest.java` and other 6 files, then the developer 'B' changes `LockTableTest.java` within 1 day of developer 'B's commit. In this example, we can flag the collaborative work item as having synchronous development using Xuan and Filkov's method. However, synchronous development will miss collaborative work items that stretch beyond the sliding time window.

Table 6 shows the percentage of work items having synchronous development that cannot be detected using Xuan and Filkov's method. Indeed, we find that 33%-88% of collaborative work items have synchronous development that cannot be detected with a 1-day time window (74% median). Even with a 10-day time window, 14%-57% of collaborative work items have synchronous development that cannot be detected (48% median).

We also investigate collaborative work items that cannot be detected at the revision-level, i.e., revisions that are submitted by different developers that modify different files, but appear under the same work item. Figure 10 shows how we detect this work item level collaboration. In CASSANDRA-5185,[13] developers use two revisions to fix links to data directories in Cassandra 1.2.0. In Figure 10 developer 'C' changes `HintedHandOffManager.java` and `MigrationManager.java`, and developer 'D' changes `Directories.java`. We find that this type of collaboration is not rare—27%-83% of collaborative work items involve developers modifying different files, and hence, cannot be detected at the revision-level.

Multiple developers collaborate by submitting different revisions of 8%-41% of work items. Moreover, in 7 of the 14 studied systems, 48% of work items that involve multiple developers cannot be grouped using conventional techniques at the revision-level.

6. THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study and its conclusions.

6.1 Construct Validity

We assume that developers generally record the work item ID in the commit message of related revisions. However, in reality, developers may omit the work item ID [7]. To combat this bias, we focus our study on systems that we can link at least 50% of the revisions to work items. Nonetheless, better linkage between work items and revisions may yield more accurate results.

[12] https://issues.apache.org/jira/browse/DERBY-2999

[13] https://issues.apache.org/jira/browse/CASSANDRA-5185

6.2 Internal Validity

We group revisions that have the same work item ID as a work item using regular expressions. These regular expressions may miss some patterns that detect work item IDs. However, we manually inspected several commit messages to check whether or not there are false positives (i.e., the number is identified as a work item ID when in fact it is actually not). Again, better linkage between work items and revisions may yield more accurate results.

Some studies report that there are tangled changes, which are unrelated or loosely related code changes in a single revision [16, 23]. Tangled changes may introduce noise into our datasets. For example, if a developer changes file A to fix a bug and file B to add a new feature, but commits the changes using one revision or work item, our analysis will still group the two changes together. In future work, we plan to study tradeoffs between work item grouping and tangled change splitting.

6.3 External Validity

We use datasets that we collected from the Apache Software Foundation. Our findings may not generalize to all software systems. However, we set two important criteria that are need to be satisfied and study 14 systems of different sizes and domains.

We set two thresholds to select appropriate systems for our case study. Although we chose the threshold values based on the distribution of the surviving systems using threshold plots, selecting different threshold values may provide different results.

7. CONCLUSIONS

Substantial research in the software evolution field aims to recover knowledge about development from system history that is archived in repositories. Although the data that is archived in these repositories can be analyzed at several different granularities, there are many studies on software evolution at the revision-level. However, the revision-level is often too fine-grained to accurately depict development tasks. For example, while some developers may use one revision per task, others may use several revisions. Such a fine granularity may impact the findings of co-change analyses.

In this paper, we set out to better understand the impact that task granularity can have on co-change analyses. Through a case study of 14 systems that are maintained by the Apache Software Foundation, we observe that:

- Analysis of only the first revision of work items would miss 2%-43% of the co-changed files.
- At least an hour elapses between the related revisions of 64% of the studied work items. A 300-second sliding time window (a common setting in software evolution studies) will miss this co-change activity.
- Multiple developers are involved with the revisions of 8%-41% of the studied work items. Since the sliding time window technique assumes that revisions are committed by the same author, it may miss this collaborative co-change activity.

Through additional impact analyses, we observe that in 7 of the 14 studied systems:

- 11% of large work items entirely consist of small revisions, which will cause large revision filters and analyses to miss important changes.
- 83% of co-change revisions cannot be detected using a 300-second sliding time window.
- 48% of work items that involve multiple developers have synchronous development that cannot be detected by revision-level analysis.

Given the impact that work item grouping, we recommend that future software evolution studies (especially co-change analyses) be performed at the work item level.

8. ACKNOWLEDGMENTS

This research was partially supported by JSPS KAKENHI Grant Number 15H05306 and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

9. REFERENCES

[1] A. Alali, B. Bartman, C. D. Newman, and J. I. Maletic. A preliminary investigation of using age and distance measures in the detection of evolutionary couplings. In *Proc. Working Conf. on Mining Software Repositories (MSR'13)*, pages 169–172, 2013.

[2] L. An, F. Khomh, and B. Adams. Supplementary bug fixes vs. re-opened bugs. In *Proc. Working Conf. on Source Code Analysis and Manipulation (SCAM'14)*, pages 205–214, 2014.

[3] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proc. Int'l Conf. on Software Engineering (ICSE'09)*, pages 298–308, 2009.

[4] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. In *Proc. European Software Engineering Conf. and Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, pages 385–394, 2007.

[5] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Syst. J.*, 15(3):225–252, 1976.

[6] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at the release level. *Sci. Comput. Program.*, 77(6):760–776, 2012.

[7] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proc. European Software Engineering Conf. and Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pages 121–130, 2009.

[8] M. Di Penta, D. M. German, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of the evolution of software licensing. In *Proc. Int'l Conf. on Software Engineering (ICSE'10)*, pages 145–154, 2010.

[9] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel. Using concept analysis to detect co-change patterns. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE'07)*, pages 83–89, 2007.

[10] M. W. Godfrey and D. M. German. The past, present, and future of software evolution. In *Proc. Int'l Conf on Software Maintenance (ICSM'08), Frontiers of Software Maintenance (FoSM) Track*, pages 129–138, 2008.

[11] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. Int'l Conf. on Software Maintenance (ICSM'00)*, pages 131–142, 2000.

[12] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proc. Int'l Conf. on Software Engineering (ICSE'14)*, pages 345–355, 2014.

[13] G. Gousios and A. Zaidman. A dataset for pull-based development research. In *Proc. Working Conf. on Mining Software Repositories (MSR'14)*, pages 368–371, 2014.

[14] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *Proc. Int'l Conf. on Software Maintenance (ICSM'04)*, pages 284–293, 2004.

[15] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proc. Int'l Conf. on Software Engineering (ICSE'12)*, pages 200–210, 2012.

[16] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proc. Working Conf. on Mining Software Repositories (MSR'13)*, pages 121–130, 2013.

[17] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proc. Working Conf. on Mining Software Repositories (MSR'08)*, pages 99–108, 2008.

[18] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. Int'l Conf. on Software Engineering (ICSE'09)*, pages 485–495, 2009.

[19] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of Just-in-Time quality assurance. *IEEE Trans. Software Engineering*, 39(6):757–773, 2013.

[20] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, 2008.

[21] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of API-level refactorings during software evolution. In *Proc. Int'l Conf. on Software Engineering (ICSE'11)*, pages 151–160, 2011.

[22] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Software Engineering*, 34(2):181–196, 2008.

[23] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. Hey! are you committing tangled changes? In *Proc. Int'l Conf. on Program Comprehension (ICPC'14)*, pages 262–265, 2014.

[24] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolution metrics on software maintenance. In *Proc. Int'l Conf. on Software Maintenance (ICSM'98)*, pages 208–217, 1998.

[25] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of java build systems. *Empirical Software Engineering*, 17(4-5):578–608, 2012.

[26] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan. Mining co-change information to understand when build changes are necessary. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME'14)*, pages 241–250, 2014.

[27] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan. An empirical study of build maintenance effort. In *Proc. Int'l Conf. on Software Engineering (ICSE'11)*, pages 141–150, 2011.

[28] M. Mondal, C. K. Roy, and K. A. Schneider. Prediction and ranking of co-change candidates for clones. In *Proc. Working Conf. on Mining Software Repositories (MSR'14)*, pages 32–41, 2014.

[29] I. Neamtiu, G. Xie, and J. Chen. Towards a better understanding of software evolution: an empirical study on open-source software. *Journal of Software: Evolution and Process*, 25(3):193–218, 2013.

[30] J. Park, M. Kim, B. Ray, and D.-H. Bae. An emprical study of supplementary bug fixes. In *Proc. Working Conf. on Mining Software Repositories (MSR'12)*, pages 40–49, 2012.

[31] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int'l Workshop on Mining Software Repositories (MSR'05)*, pages 1–5, 2005.

[32] Q. Xuan and V. Filkov. Building it together: Synchronous development in OSS. In *Proc. Int'l Conf. on Software Engineering (ICSE'14)*, pages 222–233, 2014.

[33] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Engineering*, 30(9):574–586, 2004.

[34] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proc. Int'l Conf. on Software Testing, Verification, and Validation (ICST'08)*, pages 220–229, 2008.

[35] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. Int'l Conf. on Software Engineering (ICSE'04)*, pages 563–572, 2004.

[36] T. Zimmermann and P. Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proc. Int'l Workshop on Mining Software Repositories (MSR'04)*, pages 2–6, 2004.