Preprocessing Change-Sets to Improve Logical Dependencies Identification

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

Francisco Werther Santana, Cleidson R. B. de Souza
Computing Department
Federal University of Pará (UFPA)
Belém, Brazil
wertherjr@gmail.com, cleidson.desouza@acm.org

***Abstract*— Several software evolution studies exploit the concept of logical dependencies. Such kind of dependency arises between software artifacts that have frequently changed together throughout software development. In the specific context of atomic-commit-featured repositories (e.g. Subversion), evolution studies often infer dependency dependencies based on the analysis of the committed change-sets. However, the implementation of a single change may span consecutive and closely related commits. In this paper, we propose a particular commit grouping approach that is inspired by the sliding time window algorithm. We performed a preliminary evaluation of our approach by executing it in the Apache Software Foundation code repository. Results showed that 4.6% of the produced windows contained at least two revisions and that these windows indeed grouped closely related revisions. Based on these findings, we envision that software evolution research and tools may be improved by using more accurate methods to infer logical dependencies.**

Keywords- mining software repositories; sliding time window; logical dependencies; co-changes; logical coupling; Apache Software Foundation; empirical software engineering; software evolution.

I. INTRODUCTION

Previous research in software evolution has introduced an approach for dependency identification that is able to reveal subtle relationships between software artifacts stored in version control systems (VCSs) [1]. The concept underlying this notion is known as *logical dependencies* [2] and arises from relationships established among artifacts that are frequently changed together. These artifacts are not necessarily structurally related, since they are connected from an evolutionary point of view, i.e. they have often changed together in the past, so they are likely to change together in the future. Unlike structural dependencies analysis (a.k.a., static analysis), this technique spots dependencies between any kind of artifact that composes a system, including configuration files (such as XML and property files) and documentation. The identification of logical dependencies is usually performed by parsing and analyzing the commit logs of a VCS.

The analysis of logical dependencies has supported a series of software evolution studies. For instance, Graves *et al*. [3] showed that past changes are good predictors of future faults. Mockus and Weiss [4] found that the spread of a change over subsystems and files is a strong indicator that the change will contain a defect. Cataldo *et al.* [5] reported through a detailed empirical study that the effect of logical dependencies on fault proneness was complementary and significantly more relevant than the impact of structural dependencies in two software projects from different companies. Cataldo and Nambiar [6] investigated 189 global software development (GSD) projects and showed that logical dependencies were the most significant factor impacting software quality among a series of considered factors, such as structural coupling, process maturity, developers' experience, number of regional units, and people dispersion. Logical dependencies have also been employed to detect design issues [7], infer code decay [8], and predict changes in software artifacts [9].

The reliability of all studies that employ logical dependencies is intrinsically connected to the accuracy of the approach used to identify such dependencies. In atomic-commit-featured VCSs, mutually checked-in files result in a single commit that contain all these files (change-set). In this scenario, researchers and tools developers often rely on the existence of the atomic commit feature and consider the change-set as the actual set of files that were changed together by a developer while working on a given task. However, the implementation of a change can span a series of consecutive and closely related commits. Therefore, simply inspecting the change-sets may lead to incomplete or incorrect results in terms of logical dependencies identification.

In this paper, we argue that related change-sets should be first appropriately grouped prior to the process of logical dependencies identification. As a proof of concept, we introduce a grouping approach that is inspired by the *sliding time window* algorithm [10] (presented in Section II). We also conducted a preliminary evaluation of our approach by means of an exploratory study with the Apache Software Foundation code repository (Subversion). Our main **contributions** include (i) raising the issue of using raw change-sets to identify logical dependencies in atomic-commit-featured VCSs and (ii) proposing and preliminarily evaluating our approach. An additional contribution includes implementing our approach in the XFlow tool [11].

The remainder of this paper is organized as follows. In Section II, we introduce the sliding time window algorithm. In Section III, we introduce our approach for grouping related change-sets in atomic-commit-featured VCSs. In Section IV, we present the design and results of a preliminary evaluation of our solution. In Section V, we present the related work. Finally,

in Section VI, we state our conclusions and plans for future work.

II. The Sliding Time Window Algorithm

Logical dependencies (a.k.a. change dependencies [12], evolutionary dependencies [13], and co-changes [14]) are implicit dependencies that connect software artifacts that have evolved together. Logical dependencies analysis has the potential to spot indirect or semantic relationships between artifacts that are not explicitly deducible from the programming language constructs [2, 5]. Logical dependencies are often detected by mining the logs of the project's VCS. In the context of VCSs that do not support atomic commits (e.g. CVS), change transactions need to be reconstructed before logical dependencies can be identified. In fact, a few algorithms have been proposed to reconstruct change transactions in such kind of VCSs.

The *fixed time window* algorithm [15, 16, 17] restricts the maximal duration of a transaction and groups commits that are made by the same author with the same message (comments), i.e. it assumes that such commits are related to a single modification derived from a task the author was working on. In this algorithm, the time interval always begins at the first check-in. An improved version of such algorithm was later proposed by Zimmerman *et al.* [10]. The authors coined it *sliding time window*, since it consists of "sliding" the time window to the last commit in the window, i.e. the beginning of the chosen time interval is shifted to the most recent commit. As such algorithm restricts the maximal gap between two subsequent commits, it may lead to the detection of transactions that take longer to complete than the specified length of the time window [10]. Formally, in a sliding time window of x seconds, the following conditions hold for all commits $\delta_1, \ldots, \delta_k$ (sorted by $time(\delta_i)$) that are part of a transaction Δ [10]:

a. $\forall \delta_i \in \Delta : author(\delta_1) = author(\delta_1)$
b. $\forall \delta_i \in \Delta : log\_message(\delta_i) = log\_message(\delta_1)$
c. $\forall_I \in \{2, ..., 3\} : |time(\delta_i) - time(\delta_{i-1})| \leq x$ seconds
d. $\forall \delta_a, \forall \delta_b \in \Delta : \delta_a \neq \delta_b \Rightarrow file(\delta_a) \neq file(\delta_b)$

Figure 1(a) illustrates how the fixed time window works in the CVS repository (commit labels are written using the pattern <name:x.y>, where *name* stands for the file name and *x.y* stands for its revision number). Right after commit A:1.3, a time window is created. Since A:1.3, B:1.2 and C:1.4 are (i) visible within the time window (drawn in white), (ii) made by the same author, and (iii) have the same log message, they are considered as being part of the same change transaction. Figure 1(b), in turn, illustrates how the sliding time window works. It shows that a sliding time window also considers commits D:1.3 and E:1.5, because the time window "slides" from commit A:1.3 to E:1.5. The window is closed after E:1.5, since no further commit exists within the time window.

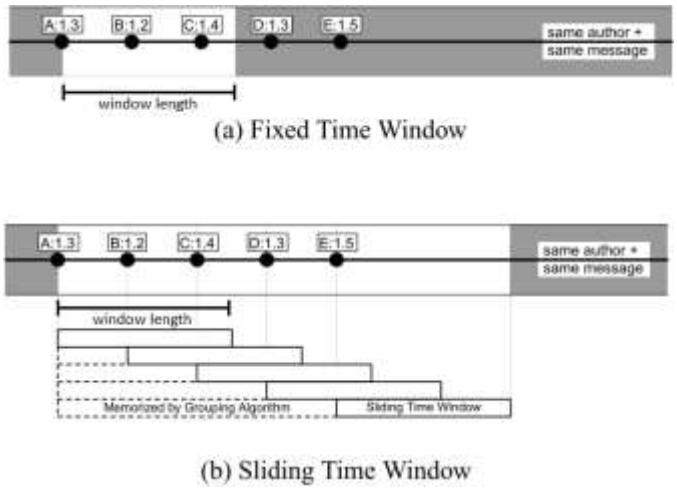

Figure 1. Fixed time window (a) and sliding time window (b) algorithms for recovering change transactions in CVS (adapted from [10])

Inspired by the sliding time window algorithm, we conceived a change-set grouping approach specially designed for atomic-commit-featured VCSs. In the following section, we present our motivation and describe the proposed approach itself.

III. Grouping Change-Sets

In the early days of logical dependencies studies, researchers conceived methods to recover change transactions in the CVS repository (e.g. fixed time window and sliding time window algorithms). At the time researchers had started mining atomic-commit-featured repositories (e.g. Subversion, Git) for logical dependencies identification, the problem of reconstructing change transactions was often neglected, since such repositories provided the required information (change-sets) out-of-the-box, i.e. which software artifacts changed together throughout the analyzed software development period. However, the implementation of a single change may span consecutive and closely related commits. It is also difficult to predict the commit habits of developers since it depends on a series of context variables, such as the software development process, the versioning system, and the task at hand. As a result, we believe that the approaches for logical dependencies identification in this kind of VCSs should take this phenomenon into account.

Inspired by the core ideas embedded in the sliding time window algorithm, we developed an approach to group closely related change sets in atomic-commit-featured VCSs. Nevertheless, since the sliding time window and our approach were conceived for different purposes, there are also some fundamental differences between them. For instance, the sliding time window algorithm creates a new empty window every time the same file reappears during the current window scope (see *condition d* in Section II). The main reason behind it is that the algorithm was conceived to recreate change-sets [10] and, per definition, a change-set cannot contain the same file twice. However, in the context of atomic-commit-featured VCSs, our ultimate goal is to better reconstruct change transactions by grouping closely related change-sets. Hence, in

our proposed approach, we *do not* recreate the window when the same file reappears. Instead, we keep the window running and add the file. Such design decision poses a series of challenges to be faced, such as handling files that are added, modified, and removed in the different change-sets included a single window. Consequently, after the window is closed, an algorithmic procedure is run in order to create consolidated versions of the files.

In the following, we present a detailed description of our approach. We also show algorithms written in pseudo-code notation and explain its main steps.

A. *Algorithm Description*

The algorithm shown in Listing 1 describes our proposed approach. For each author in the project (1-2), we obtain the list of his/her commits in chronological order (4). Afterwards, we investigate timely adjacent commits and decide whether they should belong to the same window (7-9). Adjacent commits are then grouped until one of following criteria is **not** satisfied: (i) revisions are `seconds` apart (input parameter to the algorithm) and (ii) revisions have the same author comments. When the grouping ends, the window is created and saved (10-12).

```
Algorithm: Change-set grouper
Parameters: seconds
1.  authors ← Project.getAuthors(project);
2.  for each author a in authors
3.    windowRevsList ← new List()
4.    allRevsList ← Project.getRevsSortedByDate(a)
5.    allRevsList.add(Project.createDummyRev())
6.    prevRev ← allRevsList.removeFirst()
7.    for each revision rev in allRevsList
8.      windowRevsList.add(prevRev)
9.      if (!revsOnSameWindow(prevRev,rev,secs))
10.         window ← createWindow(a,windowRevsList)
12.         saveWindow(window)
13.         windowRevsList.clear()
14.     end-if
15.     prevRev ← rev
16.   end-for
17. end-for
```

Figure 2. Change-set grouper algorithm

The subroutine shown in Figure 3 describes the process of window creation. First, basic window properties are set, including its timestamp (inherited from last commit) and length (3-9). After that, the files from all of the window revisions are obtained (11). Then, for each file, a list of its versions in chronological order is obtained (13). We consider that a file has more than one version when it is committed in different commits of the same window.

As previously described, our proposed solution does not create a new window whenever the same file reappears in the window scope. For instance, consider a window that groups revisions 1, 2, 3 and 4. Suppose that a developer adds the file `Foo.java` in revision 1, modifies it in revision 2, modifies it again in revision 3, and then decides to delete it in revision 4. In this scenario, the file Foo.java would have 4 versions.

As the grouping of commits should necessarily result in a new commit, all versions of the same file should be consolidated into a new file. As a matter of fact, in case a file has two or more versions (14), we obtain its first and last versions (15, 16), create a consolidated file (17), and add it to the window (18). In the simple case in which only a single version exists, then we just add it straight to the window (20). After processing all files, the time window is returned (23).

```
Subroutine: Window builder
Parameters: author, windowRevsList
1.  firstRev ← windowRevsList.getFirst()
2.  lastRev ← windowRevsList.getLast()
3.  //Creating the window
4.  window ← new Window()
5.  window.setAuthor(author)
6.  window.setComment(lastRev.getComments())
7.  window.setTimestamp(lastRev.getTimestamp())
8.  length ← getLength(firstRev,lastRev)
9.  window.setLength(length)
10. //Adding the files to the window
11. files ← getAllFiles(windowRevsList)
12. for each file f in files
13.   versionsList ← f.getVersionsSortedByDate()
14.   if (versionsList.size() >= 2)
15.     firstVer ← versionsList.getFirst();
16.     lastVer ← versionsList.getLast();
17.     file ← handleVersions(firstVer, lastVer);
18.     window.addFile(file)
19.   else
20.     window.addFile(versions.getFirst())
21.   end-if-else
22. end-for
23. return window
```

Figure 3. Window builder subroutine

The subroutine *handleVersions* (17) creates a consolidated version of a file based on its initial and final versions. In fact, the consolidated file directly inherits all properties from the final version (including the code itself). However, the consolidated file status (*added*, *modified*, or *deleted*) must be carefully chosen. In the following, we describe the set of rules that we conceived in order to decide upon the ultimate status of a file within a specific window (Table I). We use the template `<Status of Initial File Version> → <Status of Last File Version>` to refer to the rules.

TABLE I. RULES FOR HANDLING DIFFERENT VERSIONS OF A FILE

Initial Version / Last Version	Deleted	Modified	Added
Deleted	Deleted	Modified	Added
Added	Deleted	Added	Added
Modified	Deleted	Modified	Modified

- `{Deleted, Added, Modified} → Deleted`

  If the last file version has the status *deleted*, then no matter what the status of the initial version is, the ultimate status will be **deleted**. In fact, if a file is *deleted*, there is no point in calculating its logical dependencies to other files in the system.

If the last version of a file has the status *deleted* and the initial version has the status *added*, then we treat the file as if it had never existed within the window. In fact, if such file only exists in this window, then the file will be completely will not take part in the logical dependencies identification process (noise reduction).

- `{Deleted, Modified} → Modified`

  If the last version of a file has the status *modified*, then the ultimate status will be **modified** in the cases where the initial version status is either *deleted* or *modified*.

  If the first version has the status *deleted* and the last version of a file has the status *modified*, then we simply ignore all intermediate versions of the file and pick up only the last version. As illustrated in Figure 4, we treat the file as if it had never been deleted.

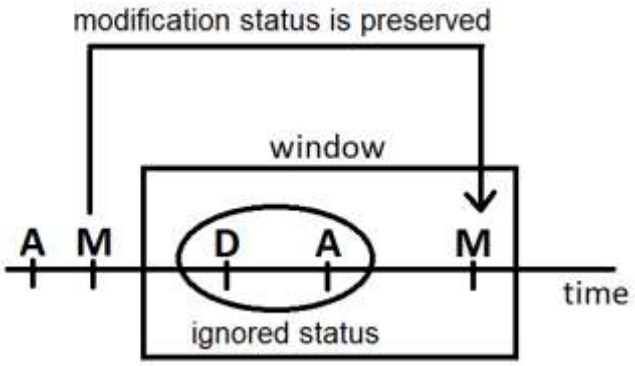

Figure 4. Preserving the file status (ignoring intermediate statuses)

- `Added → Modified`

  If the initial version of a file has the status *added* and the last version has the status *modified*, then the ultimate status will be considered as **added** (as if all subsequent modifications to the file had been done when it was first added).

- `{Added, Deleted} → Added`

  If the last version of a file has the status *added*, then the ultimate status will be **added** in the cases where the initial version status is either *added* or *deleted*.

- `Modified → Added`

  If the initial version of a file has the status *added* and the last version has the status *modified*, then the ultimate status will be considered as **modified**. In this case, we treat the addition as if it were a subsequent change to the initial version of the file.

This aforementioned set of rules represents the main contribution of the proposed algorithm.

IV. PRELIMINARY EVALUATION

In this section, we describe a preliminary evaluation of our proposed algorithm. We present the supporting tools, the data collection procedures, and the results of our quantitative and qualitative empirical studies.

A. Supporting tools

In the following, we describe the supporting tools that we used to conduct the evaluation:

XFlow. Mining repositories studies usually require extensive tool support due to large and complex data that need to be collected, processed, and analyzed [18]. XFlow is an extensible and interactive open source tool [11] whose general goal is to provide a comprehensive analysis of software projects evolution process by mining software repositories and taking into account both technical and social aspects of the developed systems. XFlow collects data from version control systems, identifies logical dependencies, evaluates metrics over project's artifacts, and presents interactive visualizations.

Minitab. All statistical analysis of data in this study was supported by Minitab[1]. Minitab is an easy to use and yet powerful statistical package heavily employed in both industry and statistical courses at universities worldwide.

B. Study Setup and Data collection

Apache Software Foundation (ASF) is a non-profit organization that has developed nearly a hundred distinguishing software projects that cover a wide range of technologies and address several problems from diverse contexts. Examples of ASF projects include Apache HTTP Server, Apache Geronimo, Cassandra, Lucene, Maven, Ant, and Struts. ASF currently owns a single Subversion repository that hosts all Apache projects and subprojects.

The context of our study comprehends the whole ASF Subversion repository. This repository encompasses 1,120,394 revisions, nearly 100 top-level projects (some of which have a few subprojects), 2421 developers, and an activity time frame of approximately 16 years and 9 months (August of 1994 till May of 2011). Working with large remote repositories poses a series of challenges. Firstly, to cope with repository instability, we built a local mirror of the whole ASF Subversion repository. This task was rather time-consuming due to the inefficiency of the Subversion protocol and available network bandwidth. After mirroring the repository, we executed the data collection processing phase of XFlow, in which the tool collects and parses the log messages of all considered revisions. Due to practical constraints, we only considered Java files, i.e. we filtered out the files that did not have the `.java` extension. Furthermore, revisions having no Java files were discarded. This data collection process was rather complex, since we had to deal with inconsistent data found on some revisions, such as files being deleted without ever being added. Such data collection process took approximately 6 hours in a dedicated Dell XPS L502X notebook (Core i7 2820QM, 6GB RAM DDR3, 500GB HD 7200RPM) and resulted in 479,794 revisions, which corresponds to approximately 43% of all ASF Subversion revisions.

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

Finally, we applied our algorithm with a window size of 200 seconds. This is a default value that has been extensively used in literature for the original sliding time windows algorithm [10, 9, 19, 20, 21, 7, 22].

C. Quantitative evaluation

We calculated basic descriptive statistics for the *number of revisions per window* variable in order to investigate to which extent the revisions were actually grouped. As depicted in Table II, the algorithm resulted in 453,865 windows. Most important, it produced 20,812 (4.6%) windows that grouped two or more revisions. The largest window grouped 106 revisions. Overall, there was little dispersion among the values, as evidenced by the mean and standard deviation values.

TABLE II. NUMBER REVISIONS PER WINDOW - DESCRIPTIVE STATISTICS

N	Sum	Mean	Max	StDev
453,865	479,790	1.06	106	0.41

We also computed basic descriptive statistics for the *length per window* variable to better understand the results of the grouping. According to Table III, there was little dispersion among the values, as evidenced by mean and standard deviation values. The algorithm produced 12,103 (2.7%) windows whose length was larger than or equal to 6 seconds. In fact, as depicted in Figure 3, there are few windows that have a long length. The largest window length was of 34 minutes and 10 seconds. This shows that even very time-distant revisions can be change coupled.

TABLE III. LENGTH PER WINDOW - DESCRIPTIVE STATISTICS

N	Max	Mean	StDev
453,865	34.17	0.038 (2.28s)	0.34

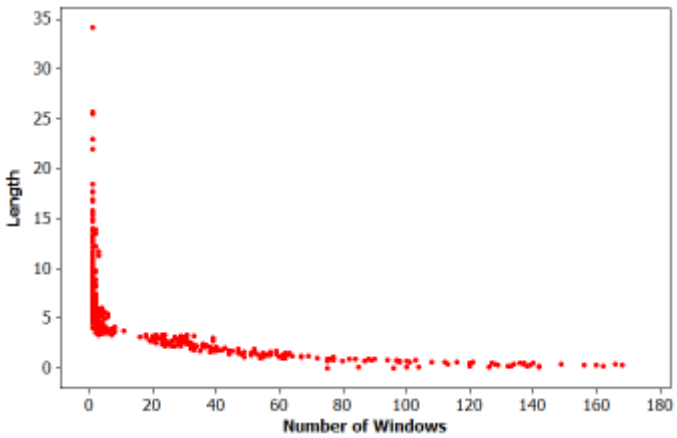

Figure 5. Scatterplot of length versus number of windows

Finally, we calculated basic descriptive statistics for the *number of files per revision* variable (Table IV). This calculation was done prior to and after the application of the algorithm, so that we could compare the results and check how many files would disappear.

TABLE IV. NUMBER OF FILES PER REVISION - DESCRIPTIVE STATISTICS

	N	Sum	Mean	StDev	Skewness	Kurtosis
Before	479,794	3,206,900	6.68	37.84	33.80	1,844.00
After	453,865	3,174,051	6.99	40.79	39.56	2,829.94

As we stated earlier, the application of the sliding time window resulted in 453,865 revisions. These revisions encompassed a total of 3,174,051 Java files (including all distinct versions of the file), which represent a reduction of approximately 1% in the original number of Java files. The mean value indicates that revisions contain 7 files in average. However, standard deviation value shows that the dispersion is very high. Both mean and standard deviation did not change much when compared to original values. As a final point, we calculated skewness and kurtosis values to further investigate such dispersion. The positive skewness value indicates that the dataset is right-skewed, i.e. the "tail" of the distribution points to the right. Also, the high kurtosis indicates that the data set has a distinct peak near the mean, declines rather rapidly, and has heavy tails. Although these values increased when compared to original ones, they did not change the shape of the distribution curve. In fact, we also performed a quartile analysis of the variable in question and we confirmed that there were no changes in the quartiles, including the median. Furthermore, as a result of the quartile analysis, we noticed that "usual" revisions (or windows) encompassed 1 to 8 files.

D. Qualitative evaluation

We qualitatively evaluated the algorithm by manually inspecting a random sample [2] of the 20,812 windows that contained at least two revisions. In this preliminary evaluation, we inspected a total of 20 windows and analyzed whether the revisions should be really grouped, i.e. if they were related to a single change purpose. Such inspection enabled us to generalize the results with a margin of error of 15% and confidence level of 82% [3]. The results of our inspection are given in Table V.

By checking the revision files, comments, and diff code, we were able to judge whether the grouping made sense. Except for the last grouping (which we classified as inconclusive), all of the others seemed perfectly appropriate. As depicted in the aforementioned table, such groupings involved fixing a specific bug, implementing a specific change, or adjusting code style and documentation.

[2] The sampling was obtained directly from the MySQL database by means of the `RAND()` function.

[3] http://www.vsai.pt/amostragem.php

TABLE V. QUALITATIVE EVALUATION OF REVISION GROUPINGS

Window	Revisions	Makes sense?	Notes
1	1116762, 1116764	Yes	Deleting Files
2	135701, 145726	Yes	Bug fix in two stages (same set of modified files)
3	768654, 768655	Yes	Running checkstyle on files from same folder
4	431981, 431983	Yes	Applying a patch to different versions of same file
5	1017168, 1017169	Yes	Fixing a bug in CollectionFieldMethodsFacetFactory.java
6	987268, 987269, 987270	Yes	Fixing a bug in different versions of same file
7	549264, 549265, 549266	Yes	Removing blank lines (adjusting style) from files in same folder
8	504397, 504398	Yes	Fixing a bug in different versions of same file
9	1100057, 1100058	Yes	Merge of a change in different branches
10	246361, 253433	Yes	Fixing a bug (same diff in both revisions)
11	225245, 225246	Yes	Fixing a bug in different versions of same file
12	1014042, 1014043	Yes	Implementing a change
13	245235, 252307	Yes	Fixing a bug (same diff in both revisions)
14	1118611, 1118612	Yes	Implementing a change
15	334583, 334584	Yes	Change made in two steps (fixing/adding info to Javadoc in class)
16	73016, 73991	Yes	Fixing a bug (same diff in both revisions)
17	235594, 236846	Yes	Fixing style (same diff in both revisions)
18	914457, 914458	Yes	Implementing a change
19	1106641, 1114539	Yes	Implementing a change
20	1118631, 1118632	?	Inconclusive

We believe that the absence of false-positives can be explained by the conservative nature of the algorithm, i.e. a window is created only when revisions have the same author, same comments, and are close in time.

V. RELATED WORK

Gall *et al.* [2] used information from the release history of a telecommunications system to discover logical dependencies and change patterns among modules and subsystems. Gall *et al.* [16] later on proposed the *Relation Analysis* (RA) technique to identify logical dependencies. The technique consists in an investigation of the classes that frequently changed together. More precisely, the authors mined logs in the CVS repository and identified classes that were changed together by a same author. A fixed time window of four minutes was considered, since large commits usually take some time to be completed. Zimmermann *et al.* [17] applied the same technique, but they considered a fixed time window of three minutes instead. Besides that, the authors introduced the idea of *support* and *confidence* as measures of significance for logical dependencies. Canfora et *al.* [22] propose the use of a multivariate time series approach (based on the Granger causality test) to address the issue that we raise in this paper, i.e. capturing logical dependencies between artifacts that are modified in subsequent change-sets. Pirklbauer [23] empirically evaluated a series of logical dependencies identification approaches, including those that are only suitable when change #id information is provided in commit log messages [10, 24, 25, 26]. Although not clear, it seems that the author does not group commits when there is an intersection between their sets of files. Fluri and Gall [28] argue that logical dependencies that arise from code styling and minor adjustments (such as the inclusion of code comments) are not significantly relevant. Based on an empirical evaluation, the authors conceived a classification of changes in order to filter out irrelevant logical dependencies. Hence, logical dependencies analysis can be improved by not only grouping change-sets prior to the dependencies identification (as in our proposal), but also after all dependencies are actually discovered.

In the following, we present less related but yet relevant work. Canfora *et al.* [29] proposed the concept of line co-change to identify cross-cutting concerns in code hosted in CVS. In the context of Model Driven Development (MDD), Wenzel *et al.* [30] used a differencing algorithm called SiDiff to identify logical dependencies between elements of a model. Wang *et al.* [31] proposed a method to identify existing fine grained logical dependencies between functions.

VI. CONCLUSION

In this paper, we discussed about the importance of grouping related change-sets in atomic-commit-featured VCSs before performing the identification of logical dependencies between software artifacts. We argued that changes in the system may span a series of timely-close and semantically-related commits. In fact, grouping related change-sets refines and improves the process of logical dependencies identification, since evolutionary links between artifacts are more accurately recovered.

As a proof of concept, we presented a rather conservative approach that is inspired by the sliding time window algorithm (whose purpose is to reconstruct change transactions in VCSs that do not support atomic commits) [10]. The preliminary evaluation that we conducted showed that our approach is feasible and produced relevant results. Our analysis indicated that approximately 4.6% of the produced windows contained at least two revisions and that these windows indeed grouped closely related revisions. We also leverage the external validity of our study, since we applied it to a large repository hosting more than a million of revisions. Based on these outcomes, we claim that software evolution research and tools can definitely be improved by using more accurate methods to infer logical dependencies.

Some factors, however, may have influenced our results. In terms of the approach input, we chose the default time window value found in the literature (200 seconds). We believe that trying other values (especially large ones, like a week) may

provide completely different results. This is thus a future research path that could be explored. Furthermore, we acknowledge that analyzing a larger number of windows would make our evidence stronger. Finally, because of practical constraints of processing time, we may have introduced some bias by considering only the Java files in the revisions.

As future work, we plan to apply the algorithm to a single software project and empirically evaluate its effectiveness in terms of precision and recall. We will also investigate whether some minor variations and heuristics can improve the results, such as considering only non-blank revision comments. Finally, we also envision empirically comparing the algorithm to other proposals found in the literature.

ACKNOWLEDGMENT

This work is also partially supported by HP (Baile project), CHOReOS EC FP7 project, and FAPESP. Marco Gerosa receives individual grant from CNPq.

REFERENCES

[1] T. Ball, J.-M. K. Adam, A. P. Harvey, and P. Siy, "If your version control system could talk..." in *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, Mar. 1997.

[2] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 190–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853338

[3] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, pp. 653–661, July 2000. [Online]. Available: http://dl.acm.org/-citation.cfm?id=347489.347496

[4] A. Mockus and D. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, Apr. 2000.

[5] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Trans. Softw. Eng.*, vol. 35, pp. 864–878, November 2009. [Online]. Available: http://dx.doi.org/10.1109/TSE.2009.42

[6] M. Cataldo and S. Nambiar, "The impact of geographic distribution and the nature of technical coupling on the quality of global software development projects," *Journal of Software Maintenance and Evolution: Research and Practice*, 2010. [Online]. Available: http://dx.doi.org/-10.1002/smr.477

[7] M. D'Ambros, M. Lanza, and M. Lungu, "Visualizing co-change information with the evolution radar," *IEEE Trans. Software Eng*, vol. 35, no. 5, pp. 720–735, 2009. [Online]. Available: http://-doi.ieeecomputersociety.org/10.1109/TSE.2009.17

[8] S. G. Eick, T. L. Graves, A. F. Karr, A. Mockus, and P. Schuster, "Visualizing software changes," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 396–412, April 2002. [Online]. Available: http://dx.doi.org/10.1109/-TSE.2002.995435

[9] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, pp. 429–445, June 2005. [Online]. Available: http://dx.doi.org/-10.1109/TSE.2005.72

[10] T. Zimmermann and P. Weißgerber, "Preprocessing CVS data for fine-grained analysis," in *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*. Los Alamitos CA: IEEE Computer Society Press, 2004, pp. 2–6.

[11] F. Santana, G. Oliva, C. R. B. de Souza, and M. A. Gerosa, "Xflow: An extensible tool for empirical analysis of software systems evolution," in *Proceedings of the VIII Experimental Software Engineering Latin American Workshop*, ser. ESELAW '11, 2011.

[12] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger, "Analysing software repositories to understand software evolution," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 37–67. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-76440-3

[13] M. Burch, S. Diehl, and P. Wei, "Visual data mining in software archives," in *Proceedings of the 2005 ACM symposium on Software visualization*, ser. SoftVis '05. New York, NY, USA: ACM, 2005, pp. 37–46. [Online]. Available: http://doi.acm.org/10.1145/-1056018.1056024

[14] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *Proceedings of the 13th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 259–268. [Online]. Available: http://dl.acm.org/-citation.cfm?id=1058432.1059363

[15] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 309–346, July 2002. [Online]. Available: http://doi.acm.org/10.1145/567793.567795

[16] H. Gall, M. Jazayeri, and J. Krajewski, "Cvs release history data for detecting logical couplings," in *Proceedings of the 6th International Workshop on Principles of Software Evolution*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 13–. [Online]. Available: http://-dl.acm.org/citation.cfm?id=942803.943741

[17] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, sept. 2003, pp. 73 – 83.

[18] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating software evolution research with kenyon," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 177–186, September 2005. [Online]. Available: http://-doi.acm.org/10.1145/1095430.1081736

[19] S. Breu, T. Zimmermann, and C. Lindig, "Mining eclipse for cross-cutting concerns," in *Proceedings of the 2006 international workshop on Mining software repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 94–97. [Online]. Available: http://doi.acm.org/-10.1145/1137983.1138006

[20] L. Aversano, L. Cerulo, and M. D. Penta, "Relating the evolution of design patterns and crosscutting concerns," in *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 180–192. [Online]. Available: http://dl.acm.org/-citation.cfm?id=1306878.1307347

[21] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 81–90. [Online]. Available: http://-dl.acm.org/citation.cfm?id=1251979.1252776

[22] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "Using multivariate time series and association rules to detect logical change coupling: An empirical study," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, sept. 2010, pp. 1 –10.

[23] G. Pirklbauer, "Empirical evaluation of strategies to detect logical change dependencies," in *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, ser. SOFSEM '10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 651–662. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11266-9_54

[24] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 international workshop on Mining software repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/-1082983.1083147

[25] R. Robbes, "Mining a change-based software repository," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 15–. [Online]. Available: http://dx.doi.org/10.1109/-MSR.2007.18

[26] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Proceedings of the 11th IEEE International Software Metrics Symposium*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 29–. [Online]. Available: http://-dl.acm.org/citation.cfm?id=1090955.1092169

[27] G. Antoniol, V. Rollo, and G. Venturi, "Detecting groups of co-changing files in cvs repositories," in *Principles of Software Evolution, Eighth International Workshop on*, sept. 2005, pp. 23 – 32.

[28] B. Fluri and H. Gall, "Classifying change types for qualifying change couplings," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 0-0 2006, pp. 35 –45.

[29] G. Canfora, L. Cerulo, and M. Di Penta, "On the use of line co-change for identifying crosscutting concern code," in *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, sept. 2006, pp. 213 –222.

[30] S. Wenzel, H. Hutter, and U. Kelter, "Tracing model elements," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, oct. 2007, pp. 104 –113.

[31] X. Wang, H. Wang, and C. Liu, "Predicting co-changed software entities in the context of software evolution," in *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on*, dec. 2009, pp. 1 –5.