Preprocessing CVS Data for Fine-Grained Analysis

Thomas Zimmermann
Saarland University, Saarbrücken, Germany
tz@acm.org

Peter Weißgerber
Cath. Univ. of Eichstätt-Ingolstadt, Germany
peter.weissgerber@ku-eichstaett.de

Abstract

All analyses of version archives have one phase in common: the preprocessing of data. Preprocessing has a direct impact on the quality of the results returned by an analysis. In this paper we discuss four essential preprocessing tasks necessary for a fine-grained analysis of CVS archives: (a) data extraction, (b) transaction recovery, (c) mapping of changes to fine-grained entities, and (d) data cleaning. We formalize the concept of sliding time windows and show how commit mails can relate revisions to transactions. We also present two approaches that map changes to the affected building blocks of a file, e.g. functions or sections.

1. Introduction

One of the first papers that analyzed version archives has the striking title "If Your Version Control System Could Talk..."[1]. In these days, many CVS [4] archives are freely available, e.g. via SourceForge.net. They all provide lots of information on the evolution of a software project: *who* changed *what* and *why*.

Such data enables many new analyses. Besides the obvious analysis of software evolution, it is also valuable input for program analysis (e.g. [2, 10, 19]), as well as for metrics (e.g. [3]). All approaches have one thing in common—they have to *preprocess* data, because direct access via CVS clients is rather slow. Additionally, some important information is not accessible via CVS: Which files have been changed in conjunction, and which methods have been affected by a change. The latter is essential for fine-grained analysis of version archives, e.g. on function-level.

In this paper, we focus on four preprocessing tasks that are performed by most analyses:

- *Data Extraction*—In Section 2 we present a lightweight and fast approach to mirror CVS information in a database.
- *Restoring Transactions*—Many analyses require the information which files have been changed in conjunction. In Section 3 we present two approaches that restore such transactions based on sliding time windows and commit mails.
- *Mapping Changes to Entities*—CVS stores only changes on files. For an analysis of functions, changes have to be examined in more detail. Section 4 presents an extensible approach that determines entities affected by a change on a file.
- *Data Cleaning*—Some transactions require special treatments by an analysis: For example, *large transactions* often result from infrastructure changes. *Merge transactions* simply reproduce changes and thus are often noise. Section 5 discusses such topics.

Preprocessing is a prerequisite for a fast access to CVS data. This data is enriched by additional information (transactions, fine-granular changes). Section 6 gives further references to related work, and Section 7 concludes the paper.

2. Data Extraction

One goal for preprocessing is to enable a fast access to the content of a CVS archive. A common solution extracts all data from the CVS repository and mirrors it in a database.

In general, it depends on the analysis what data needs to be extracted. For instance, if we analyze software evolution we are interested in everything, including deleted files. If the purpose of our analysis is to guide programmers along related changes [20], we need only existing files, because suggesting that the user should change deleted files would be awkward. In this case it suffices to extract only a subset of all files stored in the repository. But in practice, the filtering should be performed within the analysis and not within the extraction.

The extraction calls the CVS *log* command in the root directory of the project to be extracted. This returns information on all files that have ever existed in the repository. We parse this output as illustrated in Figure 1 and store the data in appropriate tables:

- Obviously, all *files* and *directories* are stored.

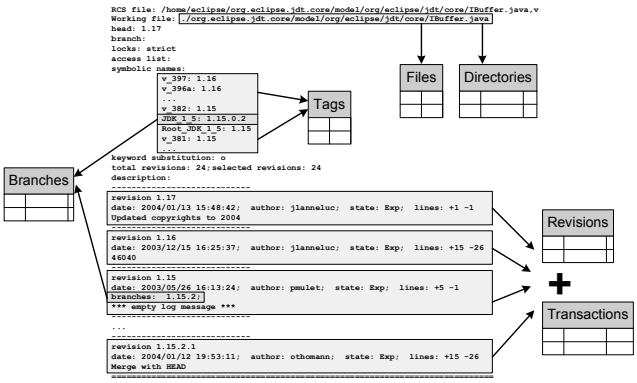
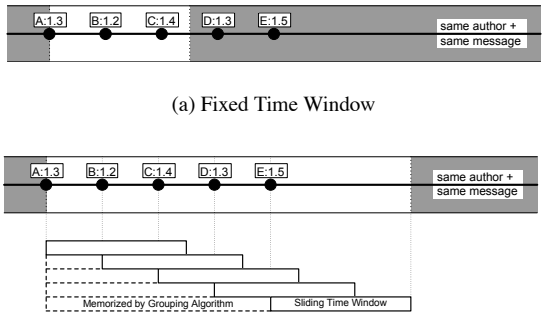


Figure 1. Data Extraction



(a) Fixed Time Window

(b) Sliding Time Window

Figure 2. Fixed vs. Sliding Time Window

- Information about single *revisions* is stored in the table *Revisions*. The author and the log message are stored in the table *Transactions*, because in this step, we treat each revision as one *transaction*—Section 3 groups several revisions/transactions together.
- With CVS the user can set symbolic names for revisions. These symbolic names are called *tags* and are frequently used to mark releases or other events.
- The table *Branches* records the branch points and branch names. This information has to be gathered from two different sections of the CVS *log* output (see Figure 1). Branch names are symbolic names for revision numbers that contain a zero, e.g. *JDK_1_5* for revision number 1.15.0.2. The branch prefix is constructed by removing the zero—in our example it is 1.15.2. The link between the section "symbolic names" and the branch point is established by a hash map using the branch prefix as keys.

Note that all preprocessing steps can also be done *incrementally*—it is only necessary to preprocess the data for new revisions instead of working on the whole repository again. To determine new revisions several approaches exist: Many open-source projects send an email to a mailing list for each commit. This approach is based on the *commitinfo* and *loginfo* files that can be used to track commits on the server-side. A possibility to get recently changed files on the client-side is the CVS *rdiff* operation (with option *-s* for *summary*) or the CVS *status* operation.

3. Restoring Transactions

CVS does not keep track of which files have been changed in conjunction in one commit operation. Often this information is required for an analysis, e.g. for determination of logical coupling [10, 19]. An obvious solution is to consider all changes by the same developer, with the same log message, made at the same time as one *transaction*. The term "same time" is inaccurate in this context, because usually commit operations take several seconds or minutes—especially if many files are involved. In practice, many approaches consider not only checkins at the same time as candidates, but also checkins during a time interval:

Fixed Time Windows restrict the maximal duration of a transaction. The time interval always begins at the *first checkin*. This approach has been used by [15, 10] for the analysis of CVS archives.

Sliding Time Windows restrict the maximal gap between two subsequent checkins of a transaction. The begin of the time interval is shifted to the *most recent checkin*. Thus, this approach can recognize transactions that take longer to complete than the duration of the time window. This approach originates from *ChangeLog* programs like *cvs2cl* [9] or *CVSps* [13].

Figure 2(a) illustrates fixed time windows: After the checkin of A:1.3 both checkins B:1.2 and C:1.4 are part of the same transaction, because they are visible within the time window (drawn in white). Figure 2(b) shows that a sliding time window additionally considers D:1.3 and E:1.5, because the time window "slides" from checkins A:1.3 to finally E:1.5. The transaction is closed after E:1.5 as no further checkins are visible within the time window.

Formally, using a sliding time window of 200 seconds, for all checkins $\delta_1, \ldots, \delta_k$ (sorted by $time(\delta_i)$) that are part of a transaction Δ, the following conditions hold:

$$\forall \delta_i \in \Delta : author(\delta_i) = author(\delta_1)$$
$$\forall \delta_i \in \Delta : log\_message(\delta_i) = log\_message(\delta_1)$$
$$\forall i \in \{2, \ldots, k\} : |time(\delta_i) - time(\delta_{i-1})| \leq \langle 200 \text{ sec} \rangle$$

Additionally, each file can only be part of a single transaction once, because CVS does not allow to commit two revisions of a file at the same time:

$$\forall \delta_a, \delta_b \in \Delta : \ \delta_a \neq \delta_b \Rightarrow file(\delta_a) \neq file(\delta_b)$$

The algorithm for grouping checkins to transactions is straightforward: Simply sort checkins by author, checkin time, and log message. Iterate over checkins in this order: Each time the author or log message differs to the ones of the previous checkin or the time window is exceeded start a new transaction.

Based on our experience, sliding time windows are superior to fixed time windows, because they deal with transactions of any duration. The selection of the length of a time window (fixed or sliding) depends on the analyzed project and the analysis itself. The time window should be chosen based on the assumption on how long it takes to check in the largest file with high network latency. Up to now, most lengths of time windows are arbitrary: They range from two to four minutes.

In our approach we chose 200 seconds which is three minutes plus a buffer of 20 seconds. Without this buffer the end of the time window can clash with the release of a CVS *lock*. In this case the continuation of an interrupted transaction is considered as a new transaction. Using such a time window for the GNU Compiler Collection (GCC), the average duration of a transaction is 6.2 seconds and the maximal duration 1 hour 32 minutes[1].

Time windows are a good approximation for restoring transactions from CVS. A more precise solution is based on *commit mails*—that are mails sent to developer mailing lists after a commit. Such a mail contains the committer, the timestamp, the modified files, and the log message. With this information it is straightforward to relate files to revisions and then to transactions. Commit mails are available for many open-source projects, e.g. GCC.

4. Mapping Changes to Entities

CVS provides only information on files and differences, but not which function has been changed. For an analysis of such fine-grained entities, another preprocessing step is required: Each revision is compared with its predecessor and the changes are mapped to syntactic components of files. If a revision is a merge of multiple predecessors, it should get a special treatment (see Section 5). A revision with no predecessors is compared against an empty file.

Fine-grained changes can be computed using a *diff*-tool and a light-weight analysis that creates the building block of files. This approach is open to everything: source code, documentation, XML files and even diagrams. For a change from revision r_1 to r_2 we compute the entities as follows:

1. Create mappings E_i : *int* $\rightarrow$ *entities* from source code lines to entities using a light-weight analysis (e.g. counting brackets). The mapping for revision r_1 is called E_1 and for r_2 it is E_2.

[1]Transaction "dummy import to prevent merge lossage" (*4081 files*)

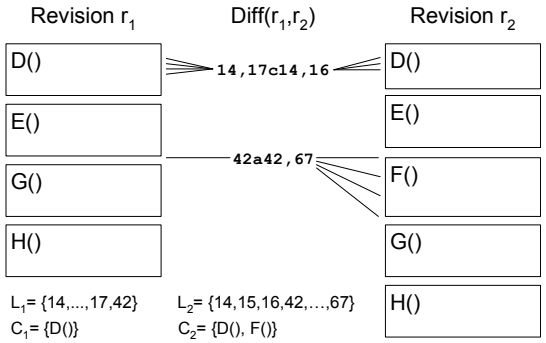


Figure 3. Map Changes to Entities

2. Perform a *diff* between r_1 and r_2. The results are the lines affected by the change: lines L_1 for revision r_1 and L_2 for r_2.

3. The entities (functions, sections) in r_1 affected by the change are C_1 (respectively C_2 for r_2):

$$C_1 = \bigcup_{l \in L_1} E_1(l) \qquad C_2 = \bigcup_{l \in L_2} E_2(l)$$

4. Thus the change from revision r_1 to r_2...
   - actually *changed* entities $C_1 \cap C_2$
   - *added* entities $C_2 \setminus C_1$
   - *removed* entities $C_1 \setminus C_2$

Figure 3 shows an example for the above algorithm. First each revision is decomposed into its building blocks—in our example functions. Then a diff between the two revisions r_1 and r_2 is calculated. The result is used to create the sets $L_1 = \{14, \ldots, 17, 42\}$ and $L_2 = \{14, 15, 16, 42, \ldots, 67\}$. Next, each line is mapped to its enclosing function and the sets $C_1 = \{\mathsf{D()}\}$ and $C_2 = \{\mathsf{D()}, \mathsf{F()}\}$ are created. Now we know that the function D() has been modified ($C_1 \cap C_2$) and F() has been inserted ($C_2 \setminus C_1$).

This approach has two weaknesses: First, its quality depends largely on the precision of the used diff tool, and second, it determines changes only based on lines, rather than on exact source code positions. Thus, in some rare cases this approach recognizes too many changed entities.

A more precise but more expensive approach first determines all entities that occur in both revisions. Then it compares the source codes of each of those entities. In other words, the diff operation is pushed from file-level to entity-level:

1. Determine all entities $\mathcal{E}_1$ of revision r_1 and all entities $\mathcal{E}_2$ of revision r_2.

2. The *added* entities are $\mathcal{E}_2 \setminus \mathcal{E}_1$, and the *removed* entities are $\mathcal{E}_1 \setminus \mathcal{E}_2$.

3. All entities in $\mathcal{E}_1 \cap \mathcal{E}_2$ *may* have been changed. Whether an entity e has been actually changed is decided by performing a diff between the source-code of e in r_1 and its source-code in r_2.

For the example of Figure 3 the above algorithm first determines that the function F() is new, because it appears only in revision r_2. Next, it compares for each function the respective parts and recognizes that D() has been changed.

The ECLIPSE platform [16] provides a powerful and extensible framework for comparing files. Both approaches described above can be realized using this framework:

- *Range Differencer*—The RangeDifferencer[2] class compares two versions based on *tokens*. This approach is based on the traditional *diff* algorithm [14]. The tokens are created using classes implementing the interface ITokenComparator[3], e.g. for lines the class DocLineComparator[4]. The calculated differences are returned in a list.

- *Structure Merge Viewer*—The Differencer[5] class compares two versions of any given *hierarchical structure* and returns a delta tree describing each change in detail. The structure is created with an own implementation of the interface IStructureCreator[6]. The fearless can use existing *internal* classes[7], e.g. the JavaStructureCreator[8].

Furthermore, ECLIPSE provides easy access to JAVA abstract syntax trees and facilitates further analysis of source code. The only drawback is that many of those features cannot be executed from the command line.

5. Data Cleaning

The previous sections described the extraction of data that is needed for fine-grained analysis. However, several issues call for identifying noise and appropriate cleaning (i.e. a special treatment). *Large transactions* which often result from infrastructure changes and *merge transactions* which simply reproduce changes are such noise.

Large Transactions

Large transactions are very frequent in real-life. Here are two examples from OPENSSL:

[2] `compare.rangedifferencer.RangeDifferencer`
[3] `compare.contentmergeviewer.ITokenComparator`
[4] `compare.internal.DocLineComparator`
[5] `compare.structuremergeviewer.Differencer`
[6] `compare.structuremergeviewer.IStructureCreator`
[7] Read [17] before you decide to use internal classes.
[8] `jdt.internal.ui.compare.JavaStructureCreator`

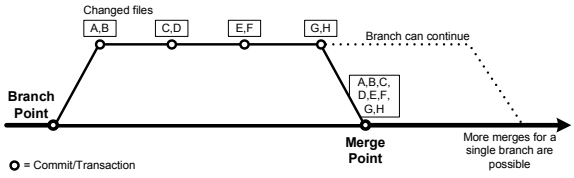


Figure 4. Merges Considered Harmful

- "Change #include filenames from <foo.h> *[sigh]* to <openssl.h>." *(552 files)*

- "Change functions to ANSI C." *(491 files)*

As the log messages indicate, the files contained in these transactions have been changed because of infrastructure changes and not because of logical relations. We refer to such transactions as noise, as it is likely that we will get incorrect results if we use them for any analysis.

A solution is to filter out transactions of size greater N in the analysis. The upper bound N depends on the examined software project.

Merge Transactions

Another more sophisticated kind of noise are merges of branches. CVS simply reproduces all changes made to one branch to the other—in one large transaction. One real-life example taken from GCC is the following:

"mainline merge as of 2003-05-04" *(5874 files)*

Figure 4 shows a smaller example: On the branch four transaction have been committed: $\{A, B\}$, $\{C, D\}$, $\{E, F\}$, and $\{G, H\}$. These files are now again changed at the merge point within a transaction that contains all changes made on the branch: $\{A, B, C, D, E, F, G, H\}$.

Merge transactions are noise for two reasons: First, they contain unrelated changes (e.g. B and C), and second they rank changes on branches higher (because they are duplicated, e.g. A and B). Taking such transactions into account has a significant influence on the results. Thus transactions that resulted from merges have to be identified. Depending on the analysis they should be ignored or at least get some special treatment.

Unfortunately, CVS does not keep track of which revisions resulted from a merge. Michael Fischer et al. proposed a heuristic to detect these revisions [8]. Their approach is restricted to merges to the main branch, but it is straightforward to apply it to other branches. Additionally, they work only on revisions instead of analyzing complete transactions. Analyzing transactions simplifies the detection of merges, because if a merge is detected for a single file, the whole transaction is probably a merge. Nonetheless, automatic merge detection is difficult to realize, because of the large number of existing merge policies. For

example, as Figure 4 indicates the development can continue on both branches after a merge, creating additional complexity for all heuristics.

6. Related Work

Data extraction from CVS is very well covered and many tools are available for free: Daniel German and Audris Mockus created *SoftChange*[9] — a tool that extracts and summarizes information from CVS and bug tracking databases [11]. Dirk Draheim and Lukasz Pekacki developed *Bloof*[10] which extracts CVS log data into a database and visualizes the software evolution using metrics [6].

Michael Fischer et al. demonstrated how to populate a *release history database* linking data from CVS and BUGZILLA [8]. In [7] they also combined their approach with features. Another project that considers additional data sources is *Hipikat* by Davor Čubranić and Gail Murphy [5]. They link information from CVS, BUGZILLA and developer mailing lists using text similarity.

To our knowledge, transaction recovery has been used by many approaches but has nowhere been covered in detail: Harald Gall, Daniel German, and Audris Mockus used fixed time windows in the past [10, 11, 15], and we used sliding time windows in our previous work [19, 20]. Commit mails have not been used in recent work to restore transactions.

Up to now, only a few approaches have considered fine-grained changes: Harald Gall et al. [10] and James Bieman et al. [3] both analyzed relations between classes. In our previous work we applied the approach presented in Section 4 and mined for relations [19] and association rules [20] between functions, sections and other fine-grained building blocks.

Michael Fischer et al. also proposed an algorithm for detecting merges of revisions in their release history database paper [8]. Lijie Zou and Michael Godfrey showed how to use origin analysis to detect merging and splitting of functions in [21]. Nonetheless, data cleaning is often neglected and there is still much room for improvement.

7. Conclusion

CVS archives contain lots of information — which is usually accessible via clients. This data provides a basis for analyses that mine additional knowledge. But CVS has some weaknesses: it is slow and loses information on transactions, fine-grained changes and merges. Thus, a preprocessing step is required.

This paper is a first attempt to collect and formalize preprocessing tasks that are used by analyses of version archives. We hope that it facilitates upcoming research in this area and provides a fruitful base for further discussions.

Acknowledgments. This project is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Stephan Diehl, Richard Kuntschke, Andreas Zeller and the anonymous MSR reviewers gave helpful comments on earlier revisions of this paper.

[9] http://sourcechange.sourceforge.net

[10] http://bloof.sourceforge.net

References

[1] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk.... In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[2] J. Bevan and J. Whitehead. Identification of software instabilities. In WCRE 2003 [18], pages 134–143.

[3] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in OO software through visualization. In *Proc. 11th International Workshop on Program Comprehension*, pages 44–53, Portland, Oregon, May 2003.

[4] P. Cederqvist. *Version Management with CVS*, Dec. 2003. www.cvshome.org/docs/manual/.

[5] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, May 2003.

[6] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In IWPSE 2003 [12].

[7] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In WCRE 2003 [18].

[8] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE.

[9] K. Fogel and M. O'Neill. *cvs2cl.pl: CVS-log-message-to-ChangeLog conversion script*, Sept. 2002. http://www.red-bean.com/cvs2cl/.

[10] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In IWPSE 2003 [12], pages 13–23.

[11] D. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of ICSE '03 Workshop on Open Source Software Engineering*, Portland, Oregon, USA, May 2003.

[12] *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, Helsinki, Finland, Sept. 2003. IEEE Press.

[13] D. Mansfield. *CVSps – Patchsets for CVS*, Feb. 2004. http://www.cobite.com/cvsps/.

[14] W. Miller and E. W. Myers. A file comparison program. *Software—Practice and Experience*, 15(11):1025–1040, Nov. 1985.

[15] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[16] Object Technology International. *Eclipse Platform Technical Overview*, Feb. 2003. Available at www.eclipse.org.

[17] J. des Rivières. *How to use the Eclipse API*, May 2001. http://eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html.

[18] *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, Nov. 2003. IEEE.

[19] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In IWPSE 2003 [12], pages 73–83.

[20] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004.

[21] L. Zou and M. W. Godfrey. Detecting merging and splitting using origin analysis. In WCRE 2003 [18].