Mining Version Archives for Co-changed Lines

Thomas Zimmermann[1] Sunghun Kim[2] Andreas Zeller[1] E. James Whitehead Jr.[2]

[1] Department of Computer Science
Saarland University
Saarbrücken, Germany
{tz, zeller}@acm.org

[2] Department of Computer Science
University of California
Santa Cruz, CA, USA
{hunkim, ejw}@cs.ucsc.edu

ABSTRACT

Files, classes, or methods have frequently been investigated in recent research on co-change. In this paper, we present a first study at the level of lines. To identify line changes across several versions, we define the annotation graph which captures how lines evolve over time. The annotation graph provides more fine-grained software evolution information such as life cycles of each line and related changes: "Whenever a developer changed line 1 of version.txt she also changed line 25 of Library.java."

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*corrections, version control*; D.2.9 [**Management**]: Software configuration management

General Terms

Management, Measurement

1. INTRODUCTION

One of the most frequently used techniques for mining version archives is *co-change*. The basic idea is that items that are changed together, are related to each other. These items can be of any granularity; in the past co-change has been applied to changes in modules [7], files [2], classes [8], and methods [14]. All these approaches stopped at the granularity of methods. Applying them to more fine-grained items such as blocks or lines seemed infeasible, in particular since they are difficult to identify across versions.

Typically lines are identified by their line number. However, since lines may be moved within files, e.g., when other lines are inserted or deleted before, line numbers are not fixed across versions and thus not suitable as identifiers for co-change analysis. We abstract line evolution from line numbers by representing each line as several nodes in a graph (one node for each revision); edges connect lines (nodes) that evolved from each other. We call this graph an *annotation graph* (Section 2).

Today, many SCM systems such as CVS and Subversion come with an annotation feature that returns for each line the last modification. Such information is not enough to track lines across revisions. In contrast, using the annotation graph we can build more general annotation algorithms that return *all* past modifications instead of just the last one (Section 3). Such annotations provide information about the life cycle of lines (Section 4).

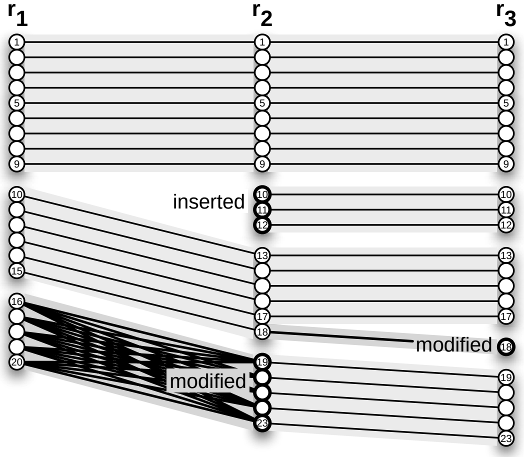

Figure 1: Tracking lines with the annotation graph.

In recent research, data mining on co-change information was used to recommend related locations such as files [13] and methods [16] after one initial change. In Section 5 we show that this is also possible for lines: "Whenever a developer changed line 1 of version.txt she also changed line 25 of Library.java." In Section 6 we discuss related work and Section 7 closes the paper with an outlook on future work.

2. TRACKING LINES

Tracking how lines evolve over time requires the identification of lines *across several versions* of a file. Within one single version, lines are typically identified by line numbers or in some cases by their contents. However both cases do not work when applied to several versions: line numbers may change when other lines are deleted or inserted, and the content of lines may be modified.

2.1 What are Annotation Graphs?

To capture how lines evolve over time, we introduce the annotation graph. The annotation graph is a multipartite graph where each part corresponds to one version of a file. Within each part/version every line is represented by a single node; edges between node indicate that a line originates from another: either by modification or by movement. Whether a line was changed in a revision is captured by labels, e.g., bold nodes indicate changes lines.

MSR'06, May 22–23, 2006, Shanghai, China.

As an example consider Figure 1 which represents several changes in an annotation graph. Edges connect lines that relate to each other across revisions, e.g., line 1 in revisions r_1, r_2, and r_3. Modifications such as from lines 16–20 in r_1 to lines 19–23 in r_2 result in a complete bipartite subgraph for that area. In other words, every node from 16 to 20 in r_1 is connected with every node from 19 to 23 in r_2.

Formally, an annotation graph $G = (V, E)$ for a file with n revisions $r_1, \ldots, r_n$ (sorted by their creation time) consists of nodes

$$V = \bigcup_{i=1}^{n} \{(r_i, m) \mid m \in \{1, \ldots, \mathit{number\_of\_lines}(r_i)\}\}$$

and edges $e = ((r_a, l_a), (r_b, l_b)) \in E$ for which

1. r_b is a direct successor of r_a and
2. l_b originates from l_a—either by modification (contents differ) or by movement (contents and relative position are equal)

Additionally, when lines were changed, we label the corresponding nodes with a description of the change such as the author who changed the lines, or the transaction in which the lines were changed.

2.2 How to Read GNU's diff

In order to construct an annotation graph, we need to compare all subsequent revisions of a file. For computing textual differences, we use the GNU *diff* tool. The *diff* tool returns a list of regions that differ in the two files; each region is called a *hunk.* Basically, there are three different kinds of changes:

Modifications. In an annotation graph, modifications result in a complete bipartite subgraph like in Figure 1 between lines 16–20 in revision r_1 and lines 19–23 in revision r_2.

Additions. For the annotation graph, additions do not result in any edges, only the positions of following lines are updated. In Figure 1, the lines 10–12 are inserted in revision r_2, thus line 10 of revision r_1 corresponds to line 13 in r_2.

Deletions. For the annotation graph, deletions do not result in any edges, only the positions of following lines are updated.

When comparing two text files with *diff*, we specified the *–text*, *–minimal*, and *–strip-trailing-cr* options. These options turned out to be very effective to return a small set of differences and to address the *carriage return* problem that *diff* and CVS suffer from.

2.3 How to Compute Annotation Graphs

Once we have computed the changeregions for all subsequent revisions, we can use this information to build an annotation graph for a file. When computing an annotation graph, one can either start from the first revision computing forward (to the last revision), or start from the last revision computing backward. We now describe a *forward-directed* algorithm that starts with the first revision; fore more details we refer to the extended version of this paper [15].

First the algorithm creates nodes for each revision and each line with the method *createNode*. Next, it iterates over all pairs $(\mathit{revL}, \mathit{revR})$ of subsequent revisions. For each pair it computes the differences (hunks) between *revL* and *revR* which then are sorted by their position *R_from* in the later revision *revR*. These hunks are then processed to create edges between nodes:

– for unchanged lines exactly one edge between the matching lines *posL* and *posR*;

– for modified lines all possible edges, which means $\mathit{posL} \in \{\mathit{L\_from} \ldots, \mathit{L\_to}\}$ and $\mathit{posR} \in \{\mathit{R\_from} \ldots, \mathit{R\_to}\}$;

– for inserted and deleted lines no edges are created.

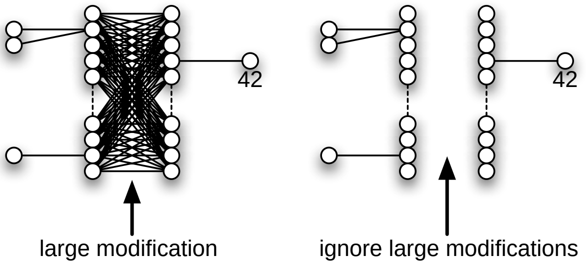

Figure 2: Ignoring large modifications for annotation graphs.

For modifications and additions, we label the nodes of the later revision *revR* with information about the change, such as author and transaction. These labels are later used to compute annotations that are more general than the ones provided by existing SCM systems (see Section 3).

2.4 How to Recognize Large Modifications

One problem for annotation graphs are changes that *modify large* parts of a file, since they results in a large number of edges. As an example consider the left part of Figure 2. When we investigate the evolution of line 42 and go back in time, we come across a large modification. If we take this modification into account, line 42 originates from every modified line. Such a result is not reasonable for evolution analysis.

In order to reduce noise, we treat large modifications not as a modifications but as combined deletions and additions. This means that for large modifications, we do not create any edges in the annotation graph (see the sketch in the right part of Figure 2).

For recognizing large modifications we use a heuristic. Let length_L and length_R be the lengths of the left (L) and right (R) region of a hunk `fct`, and $\mathit{file\_length}_L$ and $\mathit{file\_length}_R$ be the lengths of the corresponding files. A hunk is a large modification if one of the following conditions hold:

– *Region lengths exceed a threshold*

$$\begin{aligned} &\mathit{length}_L > \mathit{max}(\alpha \cdot \mathit{file\_length}_L; \beta) \\ \vee\; &\mathit{length}_R > \mathit{max}(\alpha \cdot \mathit{file\_length}_R; \beta) \end{aligned}$$

– *Ratio of region lengths exceeds a threshold*

$$\frac{\mathit{length}_L}{\mathit{length}_R} < \frac{1}{\gamma} \;\vee\; \gamma < \frac{\mathit{length}_L}{\mathit{length}_R}$$

The first condition recognizes changes that affect large parts of a file, in contrast, the second one recognizes changes that insert or delete large portions to or from a region. For our experiments, we used $\alpha = 0.10$ and $\beta = \gamma = 4$.

3. ANNOTATING LINES

Most SCM systems come with an annotation feature that returns for each line when it was inserted and by whom. For instance, the CVS annotations in Figure 3 for revision 1.17 of file Foo.java, tell us that line 39 was inserted by Mary in revision 1.10 and line 40 was inserted by Kate in revision 1.14. In this section, we briefly show how to compute such annotations using the annotation graph. While SCM systems typically return only information about the *last* change, the annotation graph can provide more general annotations that collect information about *all* past changes.

```
    $ cvs annotate -r 1.17 Foo.java
    ...
19: 1.11 (john 12-Feb-03): public int a() {
20: 1.11 (john 12-Feb-03):     return i/0;
    ...
39: 1.10 (mary 12-Jan-03): public int b() {
40: 1.14 (kate 23-May-03):     return 42;
    ...
59: 1.10 (mary 17-Jan-03): public void c() {
60: 1.16 (mary 10-Jun-03):     int i=0;
    ...
```

Figure 3: CVS annotations for Foo.java

Annotating with the last change. When computing annotations for a revision r_s, we perform for each line l_s a backward-directed breadth-first search in the annotation graph, starting from node (r_s, l_s). The search stops when we visit a node (r_x, l_x) that is labeled as a change (either the line was added or modified). We then annotate the line l_s with information from revision r_x, such as the revision identifier, the author, or the time of the change. Note that for a line l_s the last change is unique, thus l_x and r_x are unique too. It may also hold that $r_s = r_x$ in case (r_s, l_s) is already labeled as a change.

Annotating with all changes. When annotating a revision r_s with all changes, we also perform for each line l_s a backward-directed breadth-first search in the annotation graph, starting from node (r_s, l_s). However, we do not stop when visiting a changed node; instead we collect for every visited node that is labeled as a change, its information in (multi)sets. Once the breadth-first search is completed, we annotate the line l_s with these sets.

4. LIFE CYCLE OF LINES

In order to investigate the life cycle of lines for the complete ECLIPSE project (snapshot 2005-11-23) we annotated all text files with information about *all* past changes. In particular, we collected the revision identifiers and the authors. Additionally, we ignored lines containing whitespace or single curly braces. Computing the annotations took approximately 10 hours for 31,950 files.[1]Using these annotations we provide answers to the following questions.

How frequently are lines changed? We computed for each line the *change count*, that is the number of distinct revisions in its annotation. Note that we also counted the addition of a line as a change. Figure 4 shows the distribution of the change count broken down to different file extensions. We observe that most lines are changed only one time, in other words, they are inserted to a file and never touched again. This is the case for almost every line in `.dtd` and `.txt` files. In contrast, lines in `.properties` files are more frequently modified (44% at least once). Such files are used to separate properties (e.g., text messages) from the actual ECLIPSE code.

How many developers change a line? see [15].

What are the most frequently changed lines? see [15].

5. FINDING RELATED LINES

In this section, we show how to compute related lines using frequent pattern mining. In order to create the input for data mining, we annotated all lines of ECLIPSE (snapshot 2005-11-23) with all past changes. However, instead of revision ids that are only unique per file, we used the corresponding transaction ids. As a result, we get for every line the set of transactions that changed this line is the past. By using transactions instead of revisions, we are able to recognize patterns that are spread across several files.

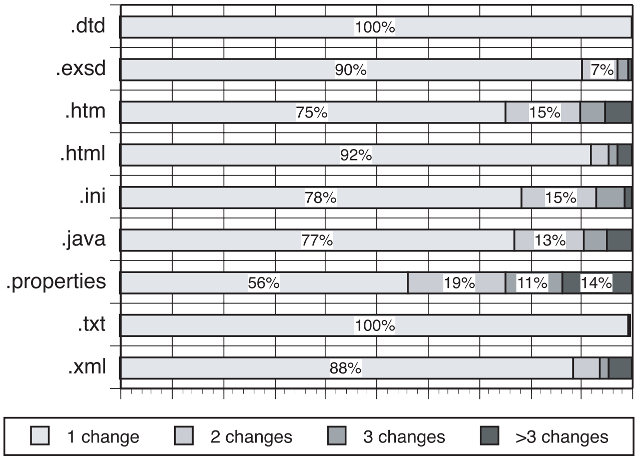

Figure 4: How frequently are lines changed?

For our experiments with frequent pattern mining, we used the Apriori algorithm [1]. In order to keep the complexity low, we applied the following optimizations:

- ignore lines containing whitespace or just a single curly brace
- investigate only modifications (not additions)
- combine lines with exactly the same change history to blocks and use blocks instead of lines as input for mining

Using the above optimizations, we could reduce the size of the input for data mining from 4,493,244 changes on lines to 255,778 changes on blocks and the calculation time to 19 seconds. On the new input we mined for all patterns that had a minimum support count of 23. The support count tells us how frequently lines that are part of a pattern have been changed together in the past. For lower support thresholds the computation did either not finish or ran out of memory (more than 16G). Improving the mining performance will remain future work.

Because of the high support count threshold we found only 29 patterns and only two them were interesting. The first pattern was found in file plugin.xml where several lines defining icons. These lines were changed together 23 times.

```
line 666:  icon=”$nl$/icons/full/obj16/package_ obj.gif”
     676:  icon=”$nl$/icons/full/elcl16/static_ co.gif”
     686:  icon=”$nl$/icons/full/elcl16/constant_ co.gif”
     717:  icon=”$nl$/icons/full/obj16/package_ obj.gif”
     727:  icon=”$nl$/icons/full/elcl16/static_ co.gif”
     737:  icon=”$nl$/icons/full/elcl16/constant_ co.gif”
     750:  hoverIcon=”$nl$/icons/full/elcl16/exc_ catch.gif”
     752:  disabledIcon=”$nl$/icons/full/dlcl16/exc_ catch.gif”
     753:  icon=”$nl$/icons/full/elcl16/exc_ catch.gif”
     762:  icon=”$nl$/icons/full/obj16/package_ obj.gif”
     776:  icon=”$nl$/icons/full/obj16/package_ obj.gif”
     808:  hoverIcon=”$nl$/icons/full/etool16/run_ sbook.gif”
     810:  disabledIcon=”$nl$/icons/full/dtool16/run_ sbook.gif”
     812:  icon=”$nl$/icons/full/etool16/run_ sbook.gif”
```

The second pattern was spread across three files: a text file version.txt, and two Java files, both named Library.java, but within different directories. The lines contain the minor version of an SWT component and were changed 171 times together.

```
version.txt              line 1:  version 3.215
j2me/.../Library.java,   line 25: static int MINOR_VERSION = 215;
j2se/.../Library.java,   line 25: static int MINOR_VERSION = 215;
```

[1] All experiments were run on an Opteron cluster using eight processors, each with 2 Mhz and 2 GB memory.

Using the above pattern, we can infer association rules such as: *"Whenever a developer changed line 1 of version.txt she also changed line 25 of Library.java."* Such a rule holds with a high confidence of 87% (171 out of 196 changes).

6. RELATED WORK

In this section we discuss work that is related to annotation graphs.

Annotating revisions. Chen et al. developed the CVSSearch tool that annotates source code with the log messages from the last code change and uses this information to guide programmers using *textual similarity* [5]. Hassan and Holt annotated static dependency graphs with *sticky notes*. A sticky note for a dependency contains the developer who created it, including the time when it was created and the log message that was provided with that change. In contrast to the work by Chen et al. and Hassan and Holt, the annotation graph considers *all* changes and not only the last ones.

Related changes. Ying et al. [13] and Zimmermann et al. [16] applied data mining on co-change information in order to recommend related locations such as files or methods. We applied the same data mining techniques, however, our focus was on lines and not on coarse-grained items such as methods or files.

Origin analysis. Godfrey et al. [9] and Kim et al. [10] proposed algorithms called origin analysis, which identify the same entities over revisions by computing entity similarities—even when entity name changes. Origin analysis is similar to our work in that origin analysis tries to map entities over revisions, while the annotation graph maps lines over revisions.

Small changes. Sliwerski et al. showed how to locate fix-inducing changes in version archives [12]. A subset of fix-inducing changes has been investigated under the name *dependencies* by Purushothaman and Perry [11] to measure the likelihood that small changes introduce errors. Their dependency concept is similar to the annotation graph, however our work focuses on the annotation of line evolution in order to compute related changes.

7. CONCLUSION

In this paper we presented the annotation graph which captures the evolution of lines. With this graph we carried out a first investigation of the life cycle of lines and pointed out that it is possible to find related lines with co-change analysis. However, data mining on co-changed lines is still expensive. Thus our future work will focus on improving the mining performance and exploring other mining techniques.

Origin analysis on lines. Modifications result in a complete bipartite subgraph, since we cannot figure out which lines are changed to which lines (see Section 2.2). We will apply origin analysis [9, 10] in the line level to identify the origin of each line. This will lead to more precise annotation graphs.

Large modifications. The parameters for recognizing large modifications (see Section 2.4) were selected after a manual inspection of several code changes. We are planning a sensitivity analysis to determine how our results depend on the selection of these parameters.

Increase mining performance. Frequent pattern mining on line level turned out to be too extensive. As a first optimization we combined lines that shared the same history to blocks. This yielded first results, however only for patterns with high support count values. Currently, we investigate other optimizations to find interesting patterns that have a low support.

Visualize evolution of lines. Using the models and layout algorithms implemented in EpoSee [4] and CCVisu [3] and fractal figures [6], we plan to visualize line level co-changes to identify related lines and to detect abnormalities.

Build tool support. We are currently developing plug-ins that will integrate annotation graphs into the ECLIPSE development environment. The user will be able to explore the evolution of lines with an *annotation graph browser* and related lines will be automatically displayed with tool tips.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of 20th International Conference on Very Large Data Bases (VLDB 1994)*, pages 487–499. Morgan Kaufmann, September 1994.

[2] J. Bevan and E. J. Whitehead Jr. Identification of software instabilities. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 134–145, Victoria, Canada, 2003. IEEE Computer Society.

[3] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*, pages 259–268. IEEE Computer Society Press, Los Alamitos (CA), 2005.

[4] M. Burch, S. Diehl, and P. Weißgerber. Visual data mining in software archives. In *Proceedings of the 2005 ACM symposium on Software visualization (SoftVis 2005)*, pages 37–46, New York, NY, USA, 2005. ACM Press.

[5] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through source code using CVS comments. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 364–373, Florence, Italy, 2001. IEEE Computer Society.

[6] M. D'Ambros, M. Lanza, and H. Gall. Fractal figures: Visualizing development effort for cvs entities. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 46–51. IEEE Computer Society, Sept. 2005.

[7] 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 (ICSM 1998)*, pages 190–197, Bethesda, Maryland, USA, 1998. IEEE Computer Society.

[8] 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 (IWPSE 2003)*, pages 13–23, Helsinki, Finland, 2003. IEEE Computer Society.

[9] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[10] S. Kim, K. Pan, and E. J. Whitehead Jr. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005)*, pages 143–152, Pittsburgh, Pennsylvania, USA, 2005. IEEE Computer Society.

[11] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[12] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR 2005)*, St. Louis, Missouri, USA, 2005. ACM Press.

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

[14] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 73–84, Helsinki, Finland, 2003. IEEE Computer Society.

[15] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr. Mining version archives for co-changed lines. Technical report, Saarland University, Saarbrücken, Germany, March 2006. Available at http://www.st.cs.uni-sb.de/softevo/.

[16] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.