Mining Version Archives for Co-changed Lines

–Extended Version–

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, fix-inducing changes in the line level, 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 [6], files [2], classes [7], and methods [15]. 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).

This paper is an extended version of a paper [16] that appeared in the proceedings of the Mining Software Repositories workshop that was held on May, 22–23, 2006 in Shanghai, China. Please cite the workshop version.

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. Additionally, they increase the precision for the recognition of fix-inducing changes (Section 4).

In recent research, data mining on co-change information was used to recommend related locations such as files [14] and methods [17] 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.

The translation of line numbers is one solution to this problem. When applied to two versions r_1 and r_2, we can use standard text differencing algorithms, like GNU *diff*. As sketched in Figure 1, a possible result might be that lines 1–9 were not changed, then lines 10–12 were inserted in r_2, thus lines 10–15 of r_1 correspond to lines 13–18 of r_2, etc. This also works for modified parts, where the differencing algorithm outputs the lines that are related: lines 16–20 of r_1 were changed into the lines 19–23 of r_2.

When analyzing more than two versions, we can compose these line number translations: As a results we get chains of lines (see Figure 1): line 15 in r_1 corresponds to line 18 in r_2, which remains line 18 in r_3 where it was changed.

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.

As an example consider Figure 2 which represents the changes of Figure 1 in an annotation graph. Edges connect lines that relate

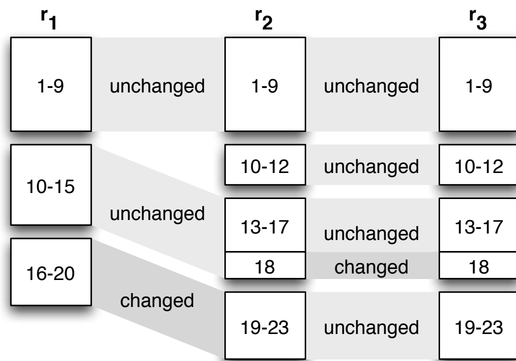

Figure 1: Tracking lines across several versions.

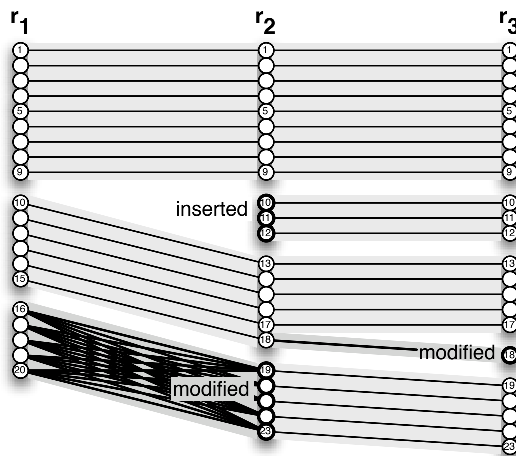

Figure 2: Tracking lines with the annotation graph.

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

$$V = \bigcup_{i=1}^{n} \{(r_i, m) \mid m \in \{1, \ldots, \textit{number_of_lines}(r_i)\}\}$$

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. To access this information, we define two functions *author(v)* and *transaction(v)* where $v = (r, l)$ represents line l in revision r.

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. Figure 3 shows a sample output for comparing two revisions 1.11 and 1.12 of the file AntUIPlugin.java. The *diff* tool returns a list of regions that differ in the two files; each region is called a *hunk*. A hunk starts with a so-called *change command* which describes the kind and the regions of the change (e.g., `8c8,9`). Next come the affected lines of both files (separated by `---`), however, for computing annotation graphs, we only need change commands.

A change command *"consists of a line number or comma-separated range of lines in the first file, a single character indicating the kind of change to make, and a line number or comma-separated range of lines in the second file"* (taken from [10]). Basically, there are three different kinds of changes, each of them results in a different change command.

Modifications `fct` — The lines in range f of the first revision were replaced with the lines in range t of the second revision. For example, in Figure 3 the change command `8c8,9` tells us that line 8 of revision 1.11 was replaced by lines 8 and 9 in revision 1.12.

In an annotation graph, modifications result in a complete bipartite subgraph. In the above example this means, that for $(1.11, 8)$ there are two outgoing edges, one to $(1.12, 8)$ and the other to $(1.12, 9)$.

Additions `lar` — The lines in range r of the second revision were inserted after line l of the first revision. For example, in Figure 3 the change command `9a11` means that line 11 was inserted in revision 1.12 after line 9 of revision 1.11.

For the annotation graph, additions of lines do not result in any edges, only the positions of following lines have to be updated.

Deletions `rdl` — The lines in range r from the first revision were deleted; line l is the position where they would have appeared. For example, in Figure 3 the change command `17d18` means that line 17 was deleted from revision 1.11.

For the annotation graph, deletion of lines do not result in any edges, only the positions of following lines have to be updated.

When comparing two text files with GNU *diff*, we have to specify several options that are discussed in Section 2.4.

2.3 How to Compute Annotation Graphs

Once we have computed the change commands 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. Figure 4 shows an *forward-directed* algorithm that starts with the first revision.

First the algorithm creates nodes for each revision and each line with the method *createNode* (see Comment 1). Next, it iterates over all pairs (*revL*, *revR*) of subsequent revisions (Comment 2).

For each pair it computes the differences (hunks) between *revL* and *revR* (Comment 3) which then are sorted by their position *R_from* in the later revision *revR* (Comment 4). These hunks are then processed to create edges between nodes (Comment 5):

– for unchanged lines exactly one edge between the matching lines *posL* and *posR* (see Comments 6 and 10);

– for modified lines all possible edges, which means $posL \in \{L_from \ldots, L_to\}$ and $posR \in \{R_from \ldots, R_to\}$ (see Comment 7);

– for inserted and deleted lines no edges are created.

For modifications and additions, we label the nodes of the later revision *revR* with information about the change, such as author and transaction (see Comment 8). 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 Use GNU's diff

Most SCM systems can compute textual differences between two revisions. However, when we constructed the annotation graph using hunks computed by the *diff* and *rdiff* command of CVS, we observed several problems: (1) the computation of differences was rather slow, (2) the differences were not minimal, thus adding unnecessary edges to the annotation graph, (3) in some cases, line feeds were not handled correctly.

To avoid the above problems, we decided to compute the textual differences with the GNU *diff* tool. This means, first we checked out all revisions, and then called the *diff* command with the following options:

–text *"Treat all files as text."*
In the presence of special characters *diff* treats files as binary and just returns whether they differ or not. With this option, we force *diff* to treat all files as text files. (Of course, we do not compare any binary files; the annotation graph makes only sense for text files.)

–minimal *"Try hard to find a smaller set of changes."*
The *diff* tool uses optimizations; as a result, the differences are not always minimal. With this option we disable these optimization in order to always get the minimal set of differences.

–strip-trailing-cr *"Strip trailing carriage return on input."*
On Windows, lines end with both the *line feed* and *carriage return* characters, but on Unix only with the *line feed* character. With this option, trailing *carriage return* characters are ignored.

The *–strip-trailing-cr* option turned out to be very effective to address the *carriage return* problem that *diff* and CVS suffer from. For 7,131 out of the 334,518 revision pairs we investigated for ECLIPSE, the differences stored in the CVS archive were solely caused by changes in line termination. In other words, although there was no actual change by the user, there was a change stored in the CVS repository.

2.5 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 5. 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 5).

```
$ diff AntUIPlugin.java::1.11 AntUIPlugin.java::1.12

8c8,9
< import java.net.*;
---
> import java.net.MalformedURLException;
> import java.net.URL;
9a11
>
17d18
< import org.eclipse.swt.graphics.Font;
24c25,31
...
51c61,68
...
```

Figure 3: Sample output of GNU diff

```
// 1: Create nodes
for (int i = 0; i < revisions.length; i++) {
  for (no = 1; no <= numberOfLines(revisions[i]) {
    createNode(revisions[i], no);
  }
}

// 2: Create edges
for (int i = 1; i < revisions.length; i++) {
  Revision revL = revisions[i-1];
  Revision revR = revisions[i];

  // 3: Compute difference between revisions
  Hunk[] hunks = computeDifferences(revL, revR);

  // 4: Sort hunks ascending by R_from.
  Arrays.sort(hunks);

  // 5: Iterate over all hunks
  int posL = posR = 1;
  for (int j = 0; j < hunks.length; j++) {
    Hunk hunk = hunks[j];

    // 6: Create edges for unchanged lines
    while (posL < hunk.L_from() && posR < hunk.R_from()) {
      createEdge(revL, revR, posL, posR);
      posL++; posR++
    }

    // 7: Create edges for modified lines
    if (hunk.isChange()) {
      for (int l = hunk.L_from(); l < hunk.L_to(); l++) {
        for (int r = hunk.R_from(); r < hunk.R_to(); r++) {
          createEdge(revL, revR, l, r);
        }
      }
    }

    // 8: Set labels for changed and inserted lines
    if (hunk.isChange() || hunk.isDeletion()) {
      for (int r = hunk.R_from(); r < hunk.R_to(); r++) {
        labelNode(revR, r, ...);
      }
    }

    // 9: Update positions
    if (hunk.isChange() || hunk.isDeletion()) {
      posL = hunk.L_to() + 1;
    }
    if (hunk.isChange() || hunk.isAddition()) {
      posR = hunk.R_to() + 1;
    }
  }

  // 10: Copy edges for unchanged lines
  while (posL < hunk.L_from() && posR < hunk.R_from()) {
    createEdge(posL, posR);
    posL++; posR++
  }
}
```

Figure 4: Algorithm for computing an annotation graph

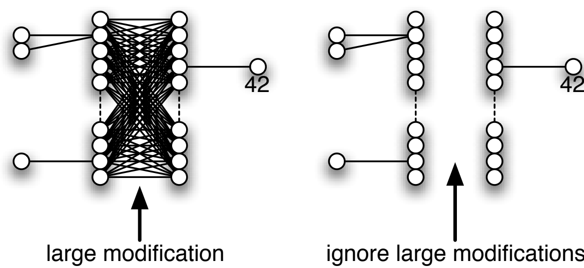

Figure 5: Ignoring huge modifications for annotation graphs.

```
    $ 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 6: CVS annotations for Foo.java

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 $file_length_L$ and $file_length_R$ 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*

$$length_L > max(\alpha \cdot file_length_L; \beta) \\ \vee\, length_R > max(\alpha \cdot file_length_R; \beta)$$

– *Ratio of region lengths exceeds a threshold*

$$\frac{length_L}{length_R} < \frac{1}{\gamma} \;\vee\; \gamma < \frac{length_L}{length_R}$$

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 6 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.

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. FIRST APPLICATIONS

In this section, we present first applications for the annotation graph. We show how to investigate the *life cycles of lines* and how to *improve the localization of fix-inducing changes*.

4.1 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 are able to 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 7 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 source code.

How many developers change a line?. We repeated the above experiment, but instead of counting lines, we counted how many different developers change a line. Figure 8 shows the results, once again broken down to file extensions. For most file extensions, we observe that more than 90% of all lines are changed by only one author. The only exceptions are `.htm` (85%), `.java` (86%), and `.properties` (67%).

What are the most frequently changed lines?. Figure 9 shows most frequently changed lines of ECLIPSE. We observe that most of these lines store version numbers. The line at the third position (that contains the copyright notice) has obviously been counted too often. This is because it was once changed together with the line containing the version (see position 1 of the list) in the same hunk. We will address such problems by implementing origin analysis for lines (see our future work in Section 7).

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

file	revision	line	count	line contents
.../jdt/internal/compiler/batch/messages.properties	1.474	17	347	compiler.version = 0.624, pre-3.2.0 milestone-4
org.eclipse.swt/Eclipse SWT/common/version.txt	1.199	1	196	version 3.215
.../jdt/internal/compiler/batch/messages.properties	1.474	18	188	compiler.copyright = Copyright IBM Corp 2000, 2005. All rights reserved.
.../common_**j2me**/org/eclipse/swt/internal/Library.java	1.188	25	180	static int MINOR_VERSION = 215;
.../common_**j2se**/org/eclipse/swt/internal/Library.java	1.192	25	180	static int MINOR_VERSION = 215;
org.eclipse.jdt.doc.isv/jdtOptions.txt	1.57	4	29	-classpath @rt@;../org.apache.ant/lib/ant.jar;../org.eclipse.debug.core/@dot; *[... many other classpath entries follow]*

Figure 9: Most frequently changed lines in ECLIPSE.

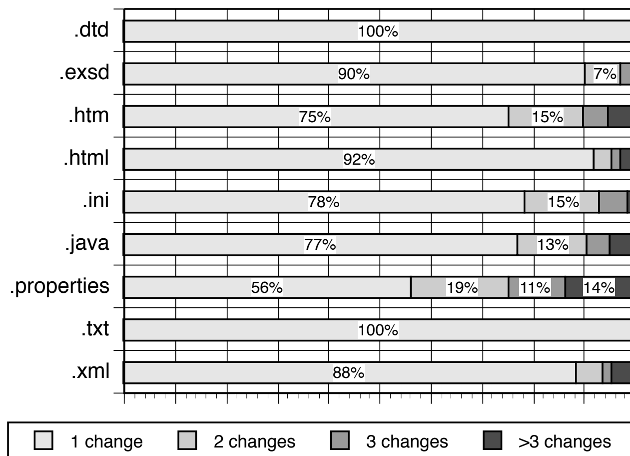

Figure 7: How frequently are lines changed?

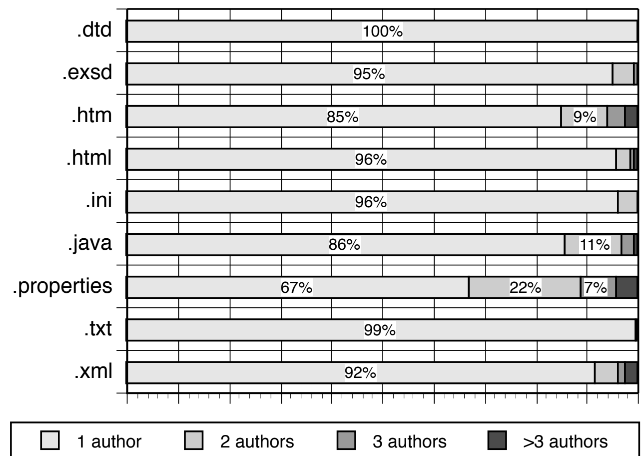

Figure 8: How many developers change a line?

4.2 Fix-Inducing Changes

Fix-inducing changes indicate potential bug introductions [13]. While modification requests can give only the *location* of a bug, fix-inducing changes provide the *time* when a bug was introduced. Fix-inducing changes can be used to compute bug occurrence statistics, classify buggy changes, and mine bug introduction patterns.

Locating fix-inducing changes.. We locate fix-inducing changes by mining change histories in SCM systems. First, we identify bug-fixes based on the change log messages that are supplied with a change. For example, we can identify bug-fix changes by looking for *keywords* like “Fixed” or “Bug” as introduced by Mockus and Votta [11]. Once we know that a revision is a bug-fix, we annotate each line of the preceding revision with the most recent author and revision that changed this line to identify fix-inducing changes.

For example, suppose the change log at revision 21 states “Fixed bug #355”, which indicates that it is a bug-fix. One file was changed in revision 21 (between revision 20 to revision 21) as shown in Figure 10. There are three kinds of changes: deletion, modification, and addition. To locate fix-inducing changes we need the lines of revision 20, since by deleting or modifying those lines a problem was fixed.

Assume a bug was fixed by deleting three lines in revision 20 (see Figure 10). Since they were deleted, the lines likely have introduced the bug. Using SCM *annotate*, we get the revisions in which

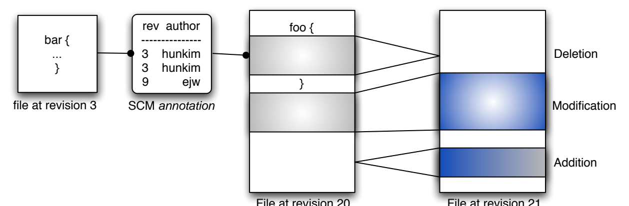

Figure 10: Finding fix-inducing changes in the file level using textual differences and annotations.

these lines were initially added. The first two lines were added in revision 3, and the third line was added in the revision 9. Thus we identify the file changes between revision 2 and 3 and between revision 8 and 9 as fix-inducing changes [13].

Problems for fine-grained fix-inducing changes.. A problem occurs when we try to locate fix-inducing changes on entity level (such as function or methods). Suppose the deleted source code in revision 20 was part of the ’foo’ function (see Figure 10). Note that *annotations* of SCM systems such as CVS or Subversion includes only revision and author information. This means, we know the first two lines in Figure 10 were added in revision 3 by ’hunkim’, but we do not know the actual line numbers in revision 3. So in past research, it was assumed that the lines in revision 3 are

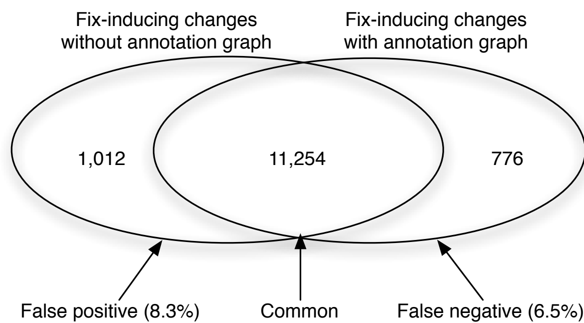

Figure 11: Fix-inducing changes identified in the method level with and without annotation graphs. We used 5,000 revisions (from 06/2001 to 07/2004) of the ECLIPSE (org.eclipse.jdt.core) project. Without the annotation graph, identified fix-inducing changes have 8.3% false positive and 6.5% false negatives (total 14.8% errors).

part of the 'foo' function which was marked as a fix-inducing, although it is not guaranteed that it existed in revision 3.

Suppose that at revision 3 the 'foo' function does not exist, and only the 'bar' function exists shown in Figure 10. Then our assumption is wrong and the 'foo' function in revision 3 is not a fix-inducing change (false positive). Since the SCM annotation does not provide the line number of the annotated lines, it is not feasible to identify the right function that includes the changed lines.

Improving precision with annotation graphs.. The annotation graph can address the problem by providing line level evolution information including line numbers in each revision. We can then simple identify the function that includes the annotated lines using line numbers provided by the annotation graph.

To demonstrate the usefulness of annotation graphs for locating fix-inducing changes, we identified method level fix-inducing changes of the ECLIPSE (org.eclipse.jdt.core) project *with* and *without* using annotation graphs. The left circle in Figure 11 shows the count of method level fix-inducing changes identified without using the annotation graph; the right circle shows the sane count when using the annotation graphs. Without the annotation graph we have about 8.3% false positive and 6.5% false negative (total 14.8% errors) fix-inducing changes. Thus annotation graphs provide information for accurate fix-inducing change identification.

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.

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 different files: a text file called version.txt, and two Java files, both named Library.java, but within different directories. The lines contained information about the minor version of an SWT component and were change 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;
```

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. [14] and Zimmermann et al. [17] 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. It is a common understanding that identifying the same entity such as module, file, method, and function between revisions is important for software evolution related analysis. Most software evolution researchers use entity names (such as file names and function names) as entity identifiers based on the assumption

that each entity is uniquely identifiable by its name over revisions. Unfortunately names change over time. Godfrey et al. [8] and Kim et al. [9] 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. However, origin analysis is very coarse-grained entity mapping compared to the annotation graph. Origin analysis can benefit from annotation graphs, since observing mapped lines over revisions can provide a simple way to track entities such as functions and methods and detect entity name changes.

Small changes. Sliwerski et al. showed how to locate fix-inducing changes in version archives [13]. A subset of fix-inducing changes has been investigated under the name *dependencies* by Purushothaman and Perry [12] 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.

Visualization. CCVisu provides a visualization to show a clustering layout for co-changed entities [3]. Nodes represent entities, and energy models are used to layout and cluster nodes. The basic idea of energy models is making entity nodes bigger and closer if they changed together. CCVisu reveals related groups of entities and allows developers to detect abnormal co-changes. For example, we can identify related entities in cluster groups, which are gathered together in the visualization. Suppose each node color represents modules of the corresponding entity. Then clusters with many different colored nodes indicate a violation of modularization—entities of many different modules are changing together too often. Using the energy models and layout algorithms of CCVisu, we can visualize line level co-changes in order to identify related lines and detect abnormalities.

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 improved the localization of fix-inducing changes for fine-grained entities such as classes or methods. Additionally, we pointed out that it is possible to find related lines with co-change analysis using the annotation graph. However, data mining on co-change 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 [8, 9] 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 2.5) 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, such as the ones implemented in EpoSee [4] or CCVisu [3], 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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] D. MacKenzie, P. Eggert, and R. Stallman. Comparing and merging files. http://www.gnu.org/software/diffutils/manual/, 2002.

[11] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 120–130, San Jose, California, USA, 2000. IEEE Computer Society.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr. Mining version archives for co-changed lines. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, Shanghai, China, may 2006.

[17] 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.