Identifying Changed Source Code Lines from Version Repositories

Gerardo Canfora, Luigi Cerulo, Massimiliano Di Penta
RCOST — Research Centre on Software Technology
Department of Engineering - University of Sannio
Viale Traiano - 82100 Benevento, Italy
{canfora, lcerulo, dipenta}@unisannio.it

Abstract

Observing the evolution of software systems at different levels of granularity has been a key issue for a number of studies, aiming at predicting defects or at studying certain phenomena, such as the presence of clones or of crosscutting concerns. Versioning systems such as CVS and SVN, however, only provide information about lines added or deleted by a contributor: any change is shown as a sequence of additions and deletions. This provides an erroneous estimate of the amount of code changed.

This paper shows how the evolution of changes at source code line level can be inferred from CVS repositories, by combining information retrieval techniques and the Levenshtein edit distance. The application of the proposed approach to the ArgoUML case study indicates a high precision and recall.

1. Introduction

Versioning systems, such as Concurrent Versioning System (CVS) and Subversion (SVN), help developers to keep track of changes performed on source code during maintenance activities. When such systems are used for the purpose of collaborative working, and not merely with the purpose of sharing source code, they provide an interesting amount of information about software evolution. In recent years, a wide number of software evolution studies has been carried out by mining information from software repositories, for example studies aiming at analyzing the relationship between clones and change sets [6], between crosscutting concerns and change sets [4], or to use source code metrics to predict defects [8].

While change sets indicate the amount of source code lines added/removed by a contributor during a limited time window, they do not provide any indication about the amount of code changed. In fact, if a developer modifies one or more source code lines, this is viewed by *diff* as a sequence of additions and deletions. For some analyses, this can provide a misleading indication. For example, if one wants to study how much a system changed during a period of time, both system size (e.g., LOC) and CVS *diffs* provide wrong indications. The former only indicates the difference between code added and removed, while the latter indicates the set of source code lines affected by the change. In summary, version repositories are not able to indicate whether an existing source code line has been updated, which is different (e.g., in term of developer effort) than removing an old line and adding a new one.

This paper introduces a technique to track the evolution of source code lines, identifying whether a CVS change is due to line modifications rather than to additions and deletions. The technique compares the sets of lines added and deleted in a change set, combining the use of Information Retrieval (IR) techniques, in particular Vector Space Models, with the Levenshtein edit distance. As case study, we identified changed lines from ArgoUML change sets. Results obtained indicated that the proposed approach ensured both high precision and high recall.

The paper is organized as follows. Section 2 introduces the technique to extract evolutions tracks from version repositories. Section 3 overviews how the proposed approach can be used for some software evolution studies. Section 4 reports and discusses results from the ArgoUML case study. Section 5 reports related work and, finally, Section 6 concludes the paper and outlines directions for future work.

2. Evolution tracks

Versioning systems handle revisions of textual files by storing the difference between subsequent revisions. In particular, CVS does not support the commit of multiple files in a single transaction, while SVN does. Single transactions are useful to detect logical coupled changes performed by developers working on a bug fix or an enhancement feature. Whether the system is transaction-enabled or not, detect-

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

ing such logical changes is an open issue as developers do not explicitly associate each transaction to a well-defined logical change into source code. The literature proposed several approaches, e.g., based on time-windows [5], and time-warping [3].

Whatever is the method adopted, a software system stored in a version repository can be viewed as a sequence of source code *Snapshots* (S) generated by a sequence of *Modification Transactions* (MTs) (also known as Modification Requests or Change Sets), representing the logical changes performed by a developer in terms of added, deleted and changed source code lines. In this context we avoid to consider branches for the sake of simplicity, even if they can be considered as parallel sequences of snapshots as well. Both CVS and SVN provide mechanisms to access a version of a source code artifact. CVS uses a tagging mechanism to track snapshots that are of interest for the software production, such as alpha/beta and candidate releases, while SVN manages them explicitly as it considers revisions at repository level rather than at file level.

A new snapshot is generated from its previous one by applying a *patch*: the minimum set of source code lines to be added to and deleted from the previous snapshot to obtain the new one. Formally, if $S(i)$ and $S(j)$ are respectively the set of lines of code belonging to the $i-th$ and $j-th$ snapshots, and $\Delta(i,j) = S(i) \backslash S(j)$ is the set difference between $S(i)$ and $S(j)$, then, $\Delta(j,i) \cup \Delta(i,j)$ is the patch applied to $S(j)$ to obtain $S(i) = (S(j) \backslash \Delta(j,i)) \cup \Delta(i,j)$. The first, $\Delta(j,i)$, is the set of lines deleted from $S(j)$, while the second, $\Delta(i,j)$ is the set of lines added to $S(j)$. From a semantic point of view, not all the lines belonging to $\Delta(j,i)$ and $\Delta(i,j)$ are pure deletions and additions respectively.

It is realistic to assume that some lines belonging to $\Delta(j,i)$ are a changed version of lines belonging to $\Delta(i,j)$. We model this with a binary relation, $\mathbf{C}(i,j) \subseteq \Delta(j,i) \times \Delta(i,j)$. It is not obvious, however, to compute an approximation of $\mathbf{C}(i,j)$. Some *diff* tools can compute the change relation for the purpose of showing the difference between files by considering sequences of different lines interspersed with sequences of matching lines. A change relation computed in this way does not necessarily satisfy the triangular equality:

$$\mathbf{C}(i,j) = \mathbf{C}(i,k) \circ \mathbf{C}(k,j)$$

where $\circ$ is the composition operation of a binary relation, and $i < k < j$. In other words, the strict matching of lines between $S(i)$ and $S(j)$ does not necessarily imply their matching between $S(i)$, $S(k)$, and $S(j)$, as they can be changed twice with a result in $S(j)$ that have the same content in $S(i)$. To preserve the triangular equality, we use a heuristic to compute $\mathbf{C}(i, i+1)$ for two subsequent snapshots, $S(i)$ and $S(i+1)$, based on the idea that $\mathbf{C}(i, i+1)$ models the changes performed by a developer

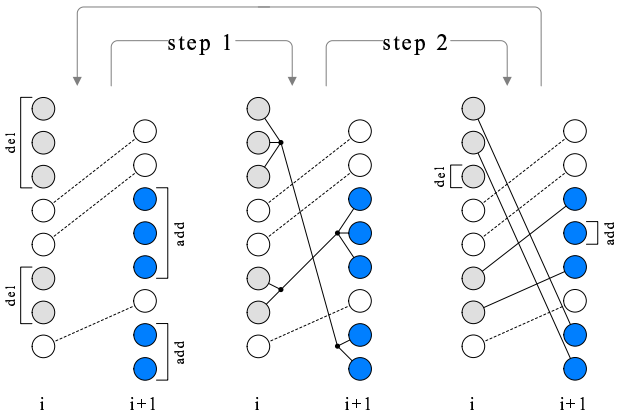

Figure 1. Computing $\mathbf{C}(i, i+1)$

when a new snapshot is generated, and then we derive the others $\mathbf{C}(i, i+k)$, with $k > 1$, by using a redefined composition operator:

$$\mathbf{C}(i,i{+}k) = \mathbf{C}(i,i{+}1) \circ \mathbf{C}(i{+}1,i{+}2) \circ \cdots \circ \mathbf{C}(i{+}k{-}1,i{+}k)$$

where the composition operator, $\mathbf{C}(i,k) \circ \mathbf{C}(k,j)$, is defined as the set of (x_i, x_j), such that $\exists x_k \in S(k)$ that satisfy one the following conditions:

$$(x_i, x_k) \in \mathbf{C}(i,k) \wedge (x_k, x_j) \in \mathbf{C}(k,j)$$

$$x_k \equiv x_j \in S(j) \wedge (x_i, x_k) \in \mathbf{C}(i,k)$$

$$x_k \equiv x_i \in S(i) \wedge (x_k, x_j) \in \mathbf{C}(k,j)$$

As $C(i,j) = C^{-1}(j,i)$, $\mathbf{C}(i, i+k)$, can also be computed for $k < 1$.

The change relation **C**(i,j) is complemented by $\mathbf{U}(i,j)$, the set of lines that developers left unchanged during the evolution from $S(i)$ to $S(j)$. It is defined as:

$$\mathbf{U}(i,i{+}k) = \begin{cases} S(i) \cap S(i+k) & for\ k = \pm 1 \\ \mathbf{U}(i,i+1) \circ \mathbf{U}(i+1,i+2) & \\ \circ \cdots \circ \mathbf{U}(i+k-1,i+k) & for\ k > 1 \end{cases}$$

where the composition operator, $\mathbf{U}(i,k) \circ \mathbf{U}(k,j)$, is defined as the set of x, such that:

$$x \in S(i) \wedge x \in S(k) \wedge x \in S(j).$$

As $\mathbf{U}(i,j) = \mathbf{U}(j,i)$, $\mathbf{U}(i, i+k)$ can be computed also for $k < 1$.

Algorithm 1: Change relation thinning

Data: $[l_{start}, l_{end}]$ left side line numbers range
$[r_{start}, r_{end}]$ right side line numbers range
Result: $\mathbf{C}[l_{end}-l_{start}][r_{end}-r_{start}]$ thinned relation
begin
  for $i \in [l_{start}, l_{end}]$ **do**
    for $j \in [r_{start}, r_{end}]$ **do**
      $\mathbf{C}[i,j] \longleftarrow 0$;
  $sl \longleftarrow l_{start}$;
  $sr \longleftarrow r_{start}$;
  while $sl < l_{end}$ **and** $sr < r_{end}$ **do**
    find (l, r) such that the Normalized Levenshtein Distance $NLD(l, r)$ between l and r is the minimum for $l \in [sl, l_{end}]$, and $r \in [sr, r_{end}]$;
    if $NLD(l, r) < Threshold$ **then**
      $\mathbf{C}[l, r] \longleftarrow 1$;
    else
      break;
    $sl \longleftarrow (l + 1)$;
    $sr \longleftarrow (r + 1)$;
end

2.1. Computing $\mathbf{C}(i, i+1)$

$\mathbf{C}(i, i+1)$ is computed by iterating the two steps shown in Figure 1:

1. line moving detection;
2. change relation thinning;
3. **if** (result is not satisfactory) **goto** step 1.

The computation starts from the output of a *CVS/SVN diff* command applied to each file belonging to $S(i)$ and $S(i+1)$. If the file does not exist, an empty file is considered. When comparing two files, *CVS/SVN diff* finds sequences of lines common to both files, interspersed with groups of differing lines called *hunks* [16]. Each hunk is considered as a deletion if it belongs to the left side, otherwise it is an addition. There are many ways to match up lines between two files. The *CVS/SVN diff* attempts to minimize the total hunk size by searching for large sequences of common lines interspersed with small hunks of differing lines. The main problem with the *CVS/SVN diff* command is that it cannot detect semantical changes, moves, splits, and merges of line ranges [9].

The approach proposed in this paper overcomes the *diff* limitation by iterating two steps. The first step compares ranges of deleted source code lines with ranges of added source code lines. We assume that when a range is changed it does not differ so much from the previous version. This certainly can be true when small changes are performed for bug fixing, and minor maintenance purposes, while it is partially true when large changes are performed during code restructuring and re-factoring. Ranges with a similarity greater than a given threshold are assumed to be in change relation. This permits the detection of line range moves or composition of moves and changes also between different files, otherwise not detectable by using a *diff* tool. The similarity measure adopted in this paper is the IR cosine similarity. In our case the cosine similarity is the cosine of the angle between two weighted term vectors, representing deletion and addition line sets. If $T = \{t_1, t_2, \ldots, t_n\}$ denotes the set of tokens extracted from line ranges a range is represented by the weighted term vector, $v =< v_1, v_2, \ldots, v_{|T|} >$, where the $i-th$ element is given by:

$$v_i = tf_j(t_i) log(idf_j(t_i))$$

where $tf_j(t_i)$ is the frequency of term t_i in the $j-th$ line range, and $idf_j(t_i)$, known as inverse document frequency, is the ratio between the total number of ranges and the number of ranges containing t_i. The similarity between a deletion and an addition range is computed as the cosine of the angle between the corresponding vectors, D and A, according to the equation:

$$\sigma(D, A) = \frac{\sum_k D_k A_k}{\sqrt{\sum_k (D_k)^2}\sqrt{\sum_k (A_k)^2}}$$

Ranges with a cosine similarity greater than a given threshold are considered in change relation. The cosine similarity cannot give any indication about lines inside each range, then, we assume that each line belonging to a range is in change relation with each line belonging to the other range (i.e. the change relation is assumed to be the cartesian product of similar ranges).

As an alternative to the use of cosine similarity, groups of lines can be compared by using a clone detection technique. In particular, a token-based approach [10] can be particularly suited for this purpose. We plan to perform experiments on such techniques in our future work.

The second step (*change relation thinning*) further reduces the change relation, with the aim of improving the precision, by computing line-by-line differences. This is done by using the *Levenshtein* edit distance [15]. The underlying assumption is that changed lines differ for a limited number of edits, while a higher edit distance indicates that most probably the two lines are different. To permit comparisons, we used the *Normalized Levenshtein Distance* $NLD(l, r)$, which ranges in the interval $[0, 1]$, where 0 means that lines match, while 1 means that lines are strictly

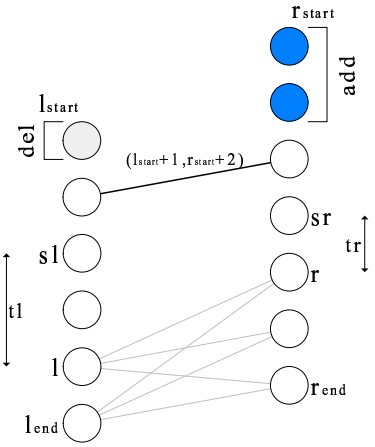

Figure 2. Change relation thinning

different. For two non-empty strings, S_1 and S_2, it is defined as:

$$NLD(S_1, S_2) = \frac{LD(S_1, S_2)}{max(S_1, S_2)}$$

where $LD(S_1, S_2)$ is the *Levenshtein Distance*, and $max(S_1, S_2)$ is the length of the longer string. The expression is obtained by considering that $LD(S_1, S_2) = 0$ if the two string are equal and $LD(S_1, S_2) = max(S_1, S_2)$ if the two string are strictly different.

The Algorithm 1 shows how the change relation is thinned for a pair of ranges detected in the previous step. The minimum, below a given threshold, is searched for all pairs, (l, r), such that, $sl \leq l \leq l_{end}$ and $sr \leq r \leq r_{end}$ (Figure 2). If more than one minimum is found the one with the lower, $(tl+tr)/2$, is preferred (i.e., the average distance from the top is low). When a minimum if found, (l, r), the search proceeds on the subsequent ranges, $[l+1, l_{end}]$ and $[r+1, r_{end}]$. The left and right line ranges between the last two found minima are considered respectively deleted and added ranges. The search stops when no more minima below the given threshold are found. This step, as shown in Figure 1, may generate new addition and deletion ranges, and therefore the process can be repeated until the result is satisfactory. As shown in Section 4, we obtained good results by stopping the process after two iterations.

3. Applications

The approach described in Section 2 can be exploited to perform, in a more precise way, evolution studies for software artifacts at different levels. Let us consider the evolution of source code entities, such as methods, classes, clones, or concerns. The evolution of a source code entity, $E(i)$, belonging to a snapshot $S(i)$, consists to find the set of $E(j)$, $(j < i)$ it originates from. Formally, given a source code entity represented as a set of source code lines, $E(i) \subseteq S(i)$, its evolution is given by the set of $E(j) \subseteq S(j)$, with $j \neq i$ that satisfies the following equation:

$$E(j) = E(i) \backslash \Delta(j, i) \cup E(j) \cap \Delta(i, j)$$

The solution for this equation is not unique and can range between $E(i)\backslash\Delta(j, i)$ and $E(i)\backslash\Delta(j, i) \cup \Delta(i, j)$. By using the change and the un-change relations, $\mathbf{C}$ and $\mathbf{U}$, a lower bound approximation can be defined as the set, $E(j) \simeq \{x_j \in S(j)\}$, such that $\exists x_i \in E(i)$, with one of the following conditions:

$$(x_i, x_j) \in \mathbf{C}(i, j)$$

or

$$x_i \equiv x_j \in \mathbf{U}(i, j)$$

It contains at least the changed and unchanged elements, while not those added. Evaluating the effectiveness of such approximation is part of our future work.

The following metrics can be evaluated with the elements defined in this paper:

Entity size. It measures the size of a source code entity and it is defined as: $|E(i)|$ (i.e. NLOC).

Patch size. It measures the size of the patch applied to the snapshot $S(i-1)$ to obtain $S(i)$ and it is defined as: $|\Delta(i-1, i)|+|\Delta(i, i-1)|$ (i.e. number of added and deleted to/from $S(i-1)$ lines of code).

Entity patch size. It measures the size of the patch fraction related to source code entity $E(i-1)$ to obtain $E(i)$ and it is defined as: $|\Delta(i-1, i) \cap E(i-1)|+|\Delta(i, i-1) \cap E(i)|$ (i.e. number of added and deleted to/from $E(i-1)$ lines of code).

Finally, from the above defined quantities, the following two ratios can be computed:

$$P_P(E(i)) = \frac{|\Delta(i-1, i) \cap E(i-1)| + |\Delta(i, i-1) \cap E(i)|}{|\Delta(i-1, i)| + |\Delta(i, i-1)|}$$

$$E_C(E(i)) = \frac{|\Delta(i-1, i) \cap E(i-1)| + |\Delta(i, i-1) \cap E(i)|}{|E(i-1) \cup E(i)|}$$

They are the *Patch Precision* $P_P(E(i))$ and Entity Coverage $E_C(E(i))$ respectively. The first measures the fraction of a patch P applied to an entity E, while the second measures the fraction of an entity E affected by a patch P.

Once the above entities have been defined, they can be exploited for different applications, for example:

1. **Software evolution studies:** studies aiming at analyzing the evolution of software systems mainly focus on values of dimensional or structural metrics for different releases. In the authors' knowledge, no study attempted to analyze to what extent a system was modified between two releases, if not using the mechanisms that CVS and *diff* currently provide;

2. **Effort estimation:** the amount of change can constitute a useful indicator for predicting efforts in open source projects, even though this needs to be complemented with other information, e.g., time between the date when a bug was reported and the date when the patch was posted on the bug tracking, or the change committed in the CVS;

3. **Crosscutting concerns evolution:**. if $CC(i)$ is a crosscutting concern belonging to the snapshot $S(i)$, it is straightforward answering to the following questions for $CC(j) \in S(j)$:

   - How much of the concern has been changed? ($E_C(CC(j))$ measure)
   - How much of the patch deals with the concern ? ($P_P(CC(j))$ measure)

4. **Clones evolution:** although some studies on clone evolution based on the analysis of change sets were able to answer to a number of research questions [2, 11, 13], the availability of $\mathbf{C}(i, j)$ can help to perform finer-grained analyses. If $Clone_1(i)$, and $Clone_2(i)$ are two clones belonging to the snapshot $S(i)$, it is straightforward answering to the following questions for $Clone_1(j), Clone_2(j) \in S(j)$:

   - Do $Clone_1(j)$, and $Clone_2(j)$ still exist in snapshot $S(i)$? ($Clone_1(j), Clone_2(j) \neq \emptyset$)
   - Were $Clone_1(j)$, and $Clone_2(j)$ changed together in $S(i)$? ($E_C(Clone_1(j)) > 0$ and $E_C(Clone_2(j)) > 0$)
   - Are $Clone_1(j)$, and $Clone_2(j)$ still clones in $S(i)$?

4. Case study

The proposed approach has been applied to identify changed lines between snapshots extracted from the ArgoUML[1] CVS repository. ArgoUML is an open source UML modeling tool with advanced software design features, such as reverse engineering and code generation. The project started in September 2000 and is still active. We considered an interval of observation ranging from September, 2000 (release 0.9.0) to December, 2005 (release 0.20 ALPHA 4) where a number of 58 releases have been produced including alpha, beta, and release candidates. The number of active developers was initially 5 and has grown rapidly in the beginning reaching a peak of 23 active developers in September, 2002. At the end of December 2005 the total number of developers involved since the beginning was 32, some of which were very active (about 11). The number of classes grew almost linearly from 446, in September, 2000, to 1538, in December, 2005, except in the interval between releases 0.11.4 and 0.13.1 where an almost exponential increment can be observed. The number of non-commented lines of code (NLOC) has grown from 45000 to 200000 in the same interval. Also the average NLOC densities per class (NLOC/class) has similarly grown from 95 to 130. We extracted 5525 snapshots from the ArgoUML CVS repository, considering only the HEAD development trunk, by using the time-window heuristic proposed in [5]. The exclusion of all branches has not affected our results as they are used very rarely. The number of snapshots between two subsequent releases are about 100 in average. Table 1 summarizes the main characteristics of the open source project considered.

Table 1. ArgoUML statistics

Classes	446 – 1538
NLOC	45000 – 200000
Snapshots	5525
Releases	58

The thresholds for the two steps have been calibrated so to achieve the best results. For the first step we considered a zero–threshold for the cosine similarity. In other words, all lines having a cosine similarity different than zero were selected, permitting to achieve a high recall, while increasing the time necessary to perform the analyses. For the second step, we considered a threshold of 0.4 for the *Normalized Levenshtein Distance* to distinguish line changes from additions and deletions.

We applied the approach over the 5525 snapshots extracted from the ArgoUML CVS repository. Figure 3 shows the number of additions, deletions and changes identified for each snapshot after the first iteration. To assess the approach performances, we manually inspected a random sample of 100 snapshots, with the aim of identifying the number of correct positives (C), false positives (FP) and false negatives (FN). Such a sample size ensures an estimate with 95% confidence level and a confidence interval of $\pm$ 10% on the estimated precision and recall percentages. For each sample we manually inspected the results by identifying Precision and Recall, computed by using the

[1] *http://argouml.tigris.org*

Table 2. Example of false positive: two similar imports clause detected as a change

Classification	Before the change file: org/argouml/model/Model.java	After the change file: org/argouml/application/Main.java
CHANGE	`26: import org.argouml.model.uml .DefaultModelImplementation;`	`60: import org.argouml.model.Model;`

Table 3. Example of false negative: a source code line split into more lines and only partially classified as a change

Classification	Before the change	After the change
CHANGE	`if (tv != null && tv.length() > 0) sb .append(INDENT).append(tv).append('\n');`	`if (tv != null && tv.length() > 0) {`
ADD		`sb.append (INDENT)`
ADD		`.append (tv)`
ADD		`.append ('\n');`
ADD		`}`

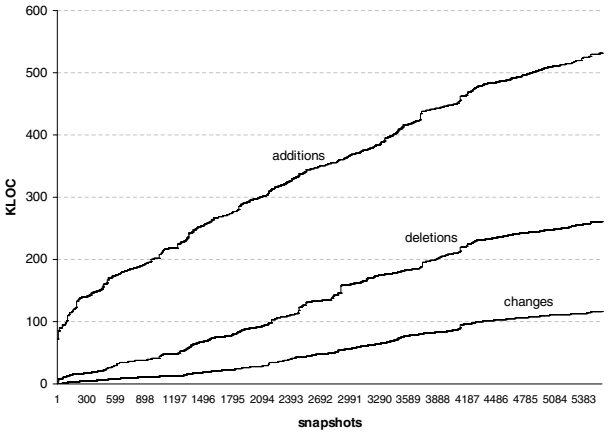

Figure 3. Additions, deletions, and changes

following equations:

$$precision = \frac{C}{C + FP}; \quad recall = \frac{C}{C + FN}$$

Table 4 reports the average precision and recall obtained with 1 and 2 iterations. For a small number of samples (about 10%) the number of false negatives detected could be higher, since we only considered those visible in subsequent sets of additions and deletions, i.e., the one that were possible to identify through a manual browsing of addition/deletion sequences. This means that the computed *Recall* has to be considered as an upper bound.

As shown in Table 4, both precision and recall are high

Table 4. Average precision and recall of $\mathbf{C}(i, i+1)$

	1	2
Precision	0.96	0.96
Recall	0.94	0.95

and, in particular, the recall increases across the two iterations. This because, when iterating, the additions/deletions not yet classified as changes were indexed again, and the new cosine values can potentially lead to the identification of changed lines not identified before.

Apart for computing precision and recall, the manual inspection permitted to understand the causes of false positives and negatives. It is rare to find false positives when the *Normalized Levenshtein Distance* is near to the threshold. This means that the threshold is not the only discriminant factor for distinguishing changed lines from additions/deletions; more heuristics are necessary to improve such detection. We found false positives when the approach identified small, similar line ranges (1 to 3 lines) belonging to different files. This happens, for example, when an *import* Java clause is deleted from one file and a similar one is added to another (see the example in Table 2). Also for false negatives, the threshold is not the only discriminant. The negative distance from the threshold does not necessarily imply the presence of a false negative. This happens, for example, when a long source code line is split into more lines in the next revision. In such case a change relation is detected for the first pair of lines, while it is missed (i.e., lines are classified as additions) for the remaining ones, where

0-7695-2950-X/07 $20.00

the *Normalized Levenshtein Distance* is lower the threshold (see the example in Table 3). The approach works well when a class is moved in the package hierarchy (see the example in Table 5). This is not trivial with CVS repositories as a file moving is stored first as a deletion and then as a new addition.

5. Related Work

The matching of source code elements among multiple software versions for the purpose of tracking the evolution of such elements has been performed with different heuristics and for different purposes [12]. Zimmermann *et al.* [21] and, similarly, Ying *et al.* [19] computed the differences between classes and methods by matching their names for the purpose of identifying fault-prone modules. Tichy [17] used a relaxed version of the *diff* algorithm for detecting code block moving. Yang [18] developed an Abstract Syntax Tree (AST) differencing algorithm to detect version merging. Laski and Szermer [14] computed the difference between control flow graphs for software maintenance support. Zimmermann *et al.* [20] introduced the notion of annotation graph, which is a data structure that represents the line-by-line evolution of software project over its evolution. Our approach improves such data structure by handling changed lines.

In the context of origin analysis methods based on vector algebra [1] and software metrics [7] have been used to infer refactoring events such as splitting, merging, renaming, and moving. However, while they focus on detecting class renaming, merging and splitting, we work at a finer-grain level to identify whether the difference between two source code line ranges is due to changes. Similarly to Antoniol *et al.* [1], we use Vector Space Models to compute the similarity between different artifacts. While at class-level the cosine similarity is precise enough, in our context it is necessarily to perform a second step, based on the *Levenshtein* edit distance, to improve the precision. Finally, a tangible advantage of the proposed approach is its language independence.

6. Conclusions and Work–in–progress

This paper proposed an approach that combines Vector Space Models and the *Levenshtein* edit distance to determine if *CVS/SVN diffs* are due to line additions/deletions or if they are due to line modifications. Such a classification can be useful to improve a number of evolution studies where knowing the amount of change is relevant.

A manual inspection performed on a random sample of ArgoUML snapshots indicated that the approach exhibits a high precision (96%) and a high recall as well (95%). An important advantage of the proposed approach is that it does not need a parser, but just a tokenizer to extract symbols and then compute the cosine similarity. This eases its application to analyze source code written in other programming languages than Java.

Work-in-progress aims to improve the proposed approach, for example experimenting the use of token-based clone detection instead of Vector Space Models, and by considering the edit distance on tokens instead than on characters. An alternative to having two steps one using Vector Space Models and one using the *Levenshtein* edit distance, would be to combine them in a unique similarity measure. Further empirical evidence on the approach performances need to be gained through further case study. Finally, we plan to apply the proposed approach to improve several evolution studies, such as the ones related to the evolution of clones [2].

References

[1] G. Antoniol, M. Di Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE '04: Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 31–40. IEEE Computer Society, 2004.

[2] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *European Conference on Software Maintenance and Reengineering*, Amsterdam, The Netherlands, October 2007 (to appear).

[3] S. Bouktif, Y.-G. Guéhéneuc, and G. Antoniol. Extracting change-patterns from cvs repositories. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 221–230, 2006.

[4] G. Canfora, L. Cerulo, and M. Di Penta. On the use of line co-change for identifying crosscutting concern code. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 213–222, 2006.

[5] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of 19th IEEE International Conference on Software Maintenance*, Amsterdam, Netherlands, Sept. 2003.

[6] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE)*, number 3922 in Lecture Notes in Computer Science, pages 411–425, Vienna, Austria, March 2006. Springer.

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

[8] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, 2005.

Classification	Before the change file: org/argouml/util/osdep/OsUtil.java (rev. 1.9)	After the change file: org/argouml/model/OsUtil.java (rev 1.1)
	...	...
CHANGE	25: package org.argouml.util.osdep;	25: package org.argouml.model;
	...	...
CHANGE	33: * @author Thierry Lach	33: * @author Thierry Lach
CHANGE	34: * @since ARGO0.9.8	34: * @since ARGO0.9.8
CHANGE	35: */	35: */
CHANGE	36: public class OsUtil {	36: public class OsUtil {
CHANGE	37: /*	37: /*
CHANGE	38: * Do not allow this class to be instantiated.	38: * Do not allow this class to be instantiated.
CHANGE	39: */	39: */
CHANGE	40: private OsUtil() {	40: private OsUtil() {
CHANGE	41: }	41: }
	...	...

Table 5. Example of correct positive: a class that has been moved from one package to another

[9] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990.

[10] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[11] C. Kapser and M. W. Godfrey. 'cloning considered harmful' considered harmful. In *Proceedings of the 2006 Working Conference on Reverse Engineering*, Benevento, Italy, October 2006.

[12] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64. ACM Press, 2006.

[13] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, pages 187–196, Lisbon, Portogal, September 2005.

[14] J. Laski and S. Szermer. Identification of program modifications and its applications in software maintenance. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 10–13. IEEE Computer Society, 1992.

[15] V. I. Levenshtein. Binary codes capable of correcting deletions,insertions, and reversals. *Cybernetics and Control Theory*, (10):707–710, 1966.

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

[17] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[18] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 7(21):739–755, 1991.

[19] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining revision history. 30:574–586, sep 2004.

[20] T. Zimmermann, S. Kim, A. Zeller, and J. E. James Whitehead. Mining version archives for co-changed lines. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75, New York, NY, USA, 2006. ACM Press.

[21] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.