Measuring Similarity of Large Software Systems Based on Source Code Correspondence

Tetsuo Yamamoto, Makoto Matsushita, Toshihiro Kamiya, and Katsuro Inoue

The authors are with the Software Engineering Research Group (C/O Katsuro Inoue), Division of Software Science, Graduate School of Engineering Science, Osaka University, 1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, JAPAN. E-mail: {t-yamamt, matusita}@ics.es.osaka-u.ac.jp, kamiya@is.aist-nara.ac.jp, inoue@ics.es.osaka-u.ac.jp.

Abstract

It is an important and intriguing issue to know the quantitative similarity of large software systems. In this paper, a similarity metric between two sets of source code files based on the correspondence of overall source code lines is proposed. A Software similarity MeAsurement Tool *SMAT* was developed and applied to various versions of an operating system(BSD UNIX OS). The resulting similarity valuations clearly revealed the evolutionary history characteristics of the BSD UNIX Operating System. Also, as an extension of *SMAT*, a system-wide difference extraction tool was developed, which effectively compressed a set of source code files relative to a base set.

Keywords

BSD UNIX, Code Clones, Plagiarism, Software Metrics

I. Introduction

Long-life software systems evolve through multiple modifications. Many different versions are created and delivered. The evolution is not simple and straightforward. It is common that one original system creates several distinct successor branches during evolution. Several distinct versions may be unified later and merged into another version. To manage the many versions correctly and efficiently, it is very important to know objectively their relationships. There has been various kinds of research on software evolution[1], [2], [3], [4], most of which focused on changes of metric values for size, quality, delivery time or process, etc.

Closely related software systems usually are identified as product lines, so development and management of product lines are actively discussed[5]. Knowing development relations and architectural similarity among such systems is a key to efficient development of new systems and to well-organized maintenance of existing systems[6].

We have been interested in measuring the similarity between two large software systems. A quantitative and objective measure for similarity is an important vehicle for knowing the evolution of software systems, as is done in the Bioinformatics field. In Bioinformatics, distance metrics are based on the alignment of DNA sequences. Phylogenetic trees using this distance are built to illustrate relations among species[7]. There are huge numbers of software systems already developed in the world and it should be possible to identify the evolution history of software assets in a manner like that done in Bioinformatics.

Various research on finding software similarities has been performed, most of which focused on detecting program plagiarism[8], [9], [10]. The usual approach extracts several metric values (or attributes) characterizing the target programs and then compares those values, usually in the educational environment with limited applicability elsewhere.

There also has been some research on identifying similarity in large collections of plain-text or HTML documents[11], [12]. These works use sampled information such as keyword sequences or "fingerprints". Similarity is determined by comparing the sampled information.

It is important that the software similarity metric is not based on sampled information as the attribute value (or fingerprint), but rather reflect the overall system characteristics. A collection of all source code files used to build a system contains all the essential information of the system. Thus, we analyze and compare overall source code files of the system. This approach requires more computation power and memory space than using sampled information, but the current computing hardware environment allows this overall source code comparison approach.

In this paper, a similarity metric called S_{line}, is used, which is defined as the ratio of shared source code lines to the total source code lines of two software systems being evaluated.

S_{line} requires computing matches between source code lines in the two systems, beyond the boundaries of files and directories. A naive approach for this would be to compare all source file pairs in both systems, with a file matching program such as *diff*[13], but the comparison of all file pairs would be impractical to apply to large systems with thousands of files.

Insted, an approach is proposed that improves efficiency and precision. First, a fast, code clone (duplicated code portion) detection algorithm is applied to all files in the two systems and then *diff* is applied to the file pairs where code clones are found.

Using this concept, a similarity metric evaluation tool called *SMAT*(Software similarity MeAsurement Tool) was developed and applied to various software system targets. We have evaluated the similarity between various versions of BSD UNIX OS, and have performed cluster analysis of the similarity values to create a dendrogram that correctly

shows evolution history of BSD OS. Also, the similarity evaluations of student compiler system projects have confirmed the ability for plagiarism detection.

Further, using a method similar to measuring the similarity, a tool called *DET*(Difference Extraction Tool) was developed for extracting the difference between two systems. Application of *DET* it to several versions of FreeBSD OS has confirmed its effectiveness for compressing one software version to another and identifying system-wide distinctions, rather than simply using *diff*.

Section II presents a formal definition of similarity and its metric S_{line}. Section III describes a practical method for computing S_{line} and shows the implementation tool *SMAT*. Section IV shows applications of *SMAT* to versions of BSD UNIX OS and a student project. Section V shows an extension to a difference extraction tool *DET*. Results of our work and comparison with related research are given in Section VI. Concluding remarks are given in Section VII.

II. Similarity of Software Systems

A. Definitions

First we will give a general definition of software system similarity and then a concrete similarity metric.

A software system P is composed of elements $p_1, p_2, \cdots, p_m$, and P is represented as a set $\{p_1, p_2, \cdots, p_m\}$. In the same way, another software system Q is denoted by $\{q_1, q_2, \cdots, q_n\}$. We will choose the type of elements, such as files and lines, based on the definitions of the similarity metrics described later.

Suppose that we are able to determine matching between p_i and q_j $(1 \leq i \leq m, 1 \leq j \leq n)$, and we call the set of all matched pair (p_i, q_j) *Correspondence* R_s, where $R_s \subseteq P \times Q$. *Similarity* S of P and Q with respect to R_s is defined as follows.

$$S(P,Q) \equiv \frac{|\{p_i|(p_i,q_j) \in R_s\}| + |\{q_j|(p_i,q_j) \in R_s\}|}{|P| + |Q|}$$

As shown in Fig. 1, this definition means that the similarity is the ratio of the number of elements in R_s to the total number of elements in P and Q. If R_s becomes smaller, S will decrease, and if $R_s = \phi$ then $S = 0$. Moreover, when P and Q are exactly the same systems, $\forall i(p_i, q_i) \in R_s$ and then $S = 1$.

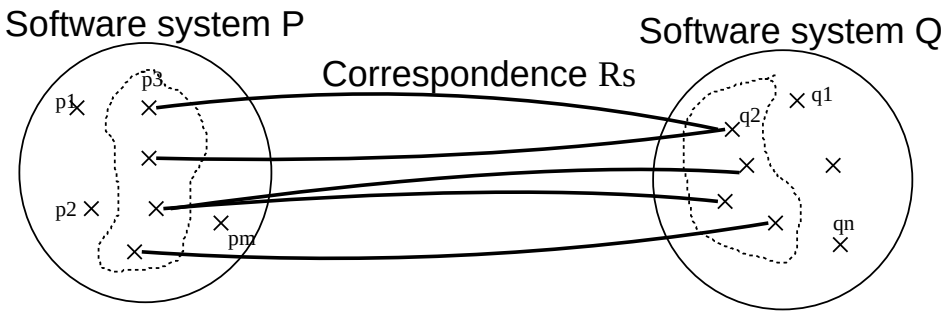

Fig. 1. Correspondence of elements Rs.

B. Similarity Metrics

The above definition of the similarity leaves room for implementing different concrete similarity metrics by choosing the element types or correspondences. Here, we show a concrete similarity metric which will be used in subsequent discussions.

Similarity Metric S_{line} using equivalent line matching:
Each element of a software system is a single line of each source file composing the system. For example, if a software system X consists of source code files $x_1, x_2, \cdots$ and each source code file x_i is made up of lines $x_{i1}, x_{i2}, \cdots$, then $x_{11}, x_{12}, \cdots, x_{21}, x_{22}, \cdots, x_{i1}, x_{i2}, \cdots, x_{(i+1)1}$, $x_{(i+2)2}, \cdots$ are the elements. Pair (x_{ij}, y_{mn}) of two lines x_{ij} and y_{mn} in system X and system Y is in correspondence when x_{ij} and y_{mn} match as equivalent lines. The equivalency is determined by the duplicated code detection method and file comparison method discussed in detail later. Intuitively, two lines with minor distinction such as space/comment modification and identifier rename are recognized as equivalent.
S_{line} is not affected by file renaming or path changes. Modification of a small part in a large file does not give great impact to the resulting value. On the other hand, finding equivalent lines generally would be a time and space consuming process. A practical approach for this is given in Section III.

It is possible to consider other definitions of similarity and its metrics. Comparison to other such approaches are presented in Section VI.

III. Measuring S_{line}

A. Approach

The key of S_{line} is computing the correspondence. A straightforward approach we might consider is that first we construct appended files $x_1 \parallel x_2 \parallel \cdots$ and $y_1 \parallel y_2 \parallel \cdots$ which are concatenation of all source code files $x_1, x_2, \cdots$ and $y_1, y_2, \cdots$ for systems X and Y, respectively. Then we extract the longest common subsequence (LCS) between $x_1 \parallel x_2 \parallel \cdots$ and $y_1 \parallel y_2 \parallel \cdots$ by some tool, say *diff* [13], which implements an LCS-finding algorithm[14], [15], [16]. The extracted LCS is used as the correspondence.

However, this method is fragile to the change of file concatenation order caused by renaming files and reorganizing file structures, since *diff* cannot follow line block movement to different positions in the files. For example, for two files $x_1 \parallel x_2$ and $x_2 \parallel x_1$, the LCS found by *diff* is either x_1 or x_2 (longer one of them).

Another approach is that we try to apply *diff* to all combination of files between two systems. This approach might work, but the scalability would be an issue. The performance applied to huge systems with thousands of files would be doubtful.

Here, an approach is proposed that effectively uses both *diff* and a clone detection tool named *CCFinder*[17], [18].

CCFinder is a tool used to detect duplicated code blocks (called *clones*) in source code written in C, C++, Java, and COBOL. It effectively performs lexical analysis, transformation of tokens, computing duplicated token sequences by a suffix tree algorithm[19], and then reports the results. The clone detection is made along with normalization and parameterization, that is, the location of white spaces and lines breaks are ignored, comments are removes, and the distinction of identifier names are disregarded. By the normalization and parameterization, code blocks with minor modification are effectively detected as clones.

Appling *CCFinder* to two concatenated files $x_1 \parallel x_2 \parallel \cdots$ and $y_1 \parallel y_2 \parallel \cdots$ finds all clone pairs (b_x, b_y) where b_x is a subsequence of $x_1 \parallel x_2 \parallel \cdots$. and b_y is that of $y_1 \parallel y_2 \parallel \cdots$. Those clone pairs found are members of the correspondence.

Code clones are only non-gapped ones. Closely similar code blocks with a gap block(unmatching to them) such as $l_1 l_2$ and $l_1 l_x l_2$ are not detected as a larger clone $l_1 * l_2$ but identified as two smaller clones l_1 and l_2. When the lenghts of l_1 and l_2 are less than threshold of

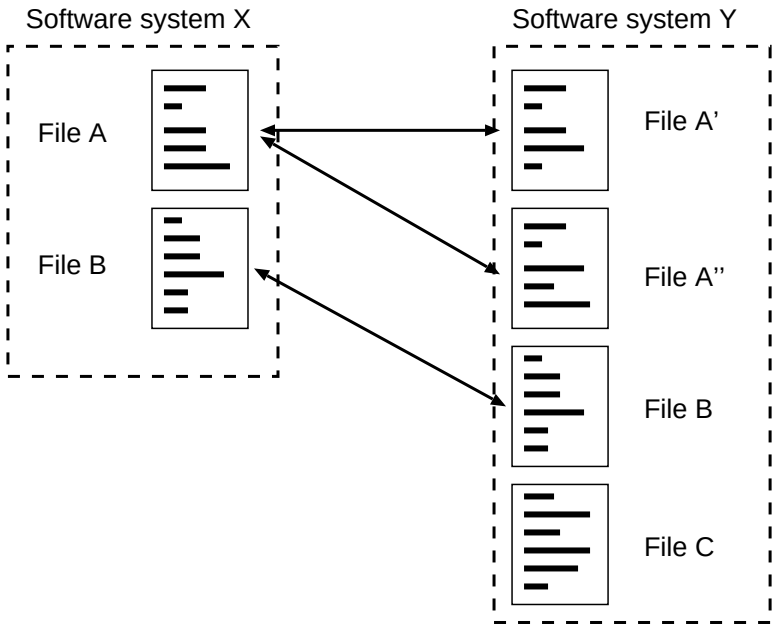

Fig. 2. How to find a correspondence.

CCFinder(usually 20 tokens), then *CCFinder* reports no clones at all. Therefore, to reclaim those tokens *diff* is applied to all pairs of two file x_i and y_j, where *CCFinder* detects a clone pair (b_x, b_y) and b_x is in x_i and b_y is in y_j, respectively. The result of *diff* is the longest common subsequences, which also are considered members of the correspondence. The combined results of *CCFinder* and *diff* is increases S_{line} by about 10%, compared to using only *CCFinder*.

B. Example of Measurement

A simple example of computing S_{line} with *CCFinder* and *diff* is given here. Consider a software system X and its extended system Y as shown in Fig. 2. X is composed of two source code files A and B, and Y is composed of four files A', A'', B, and C. Here, A' and A'' are evolved versions of A, and C is a newly created file.

At first, *CCFinder* is applied to detect clones between two concatenated files $A \parallel B$ and $A' \parallel A'' \parallel B' \parallel C$. This finds clones between A and A', A and A'', and B and B'. Assume that no clones are detected between other combination of files. Each lines in the clones found are put into the correspondence.

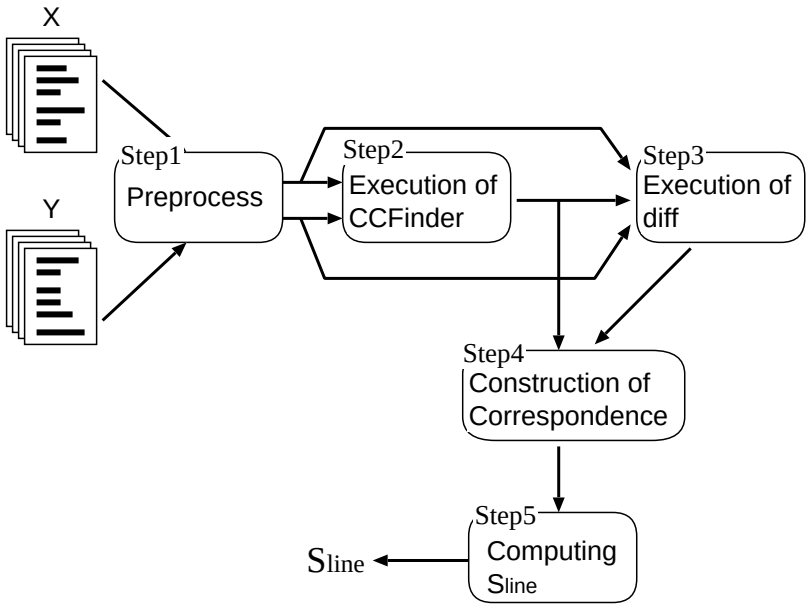

Fig. 3. Similarity measuring process.

Next, *diff* is applied to file pairs A and A', A and A'', and B and B'. Then, the lines in the resulting common subsequences by *diff* are added to the correspondence obtained by the clone detection.

This approach has benefits in the sense that we do not need to perform *diff* on all the file pair combinations. Also, we can chase movement of lines inside or outside of the files, which cannot be detected by *diff* only. Also, this approach can identify and count the directives and macros not detected by *CCFinder*.

C. SMAT

Based on this approach, we have developed a similarity evaluation tool *SMAT* which effectively computes S_{line} for two systems. The following is the detailed process of the system. An overview is illustrated in Fig. 3.

INPUTS: File paths of two systems X and Y, each of which represents the subdirectories containing all source codes

OUTPUTS: S_{line} of X and Y $(0 \leq S_{line} \leq 1)$

Step 1 Preprocessing:

All comments, white spaces, and empty lines are removed, which do not affect the execution

of the programs. This step helps to improve the precision of the following steps, especially Step 3.

Step 2 Execution of *CCFinder*:

We execute *CCFinder* between two concatenated files of X and Y. *CCFinder* has an option for the minimum number of tokens of clones to be detected, and which is set at 20. This number is obtained through experiences. Smaller numbers generate many meaningless clones and larger numbers increase the chance of overlooking useful clones.

Step 3 Execution of *diff*:

Execute *diff* on any file pair x_i and y_j in X and Y respectively, where at least one clone is detected between x_i and y_j.

Step 4 Construction of Correspondence:

The lines appearing in the clones detected by Step 2 and in the common subsequences found in Step 3 are merged to determin the correspondence between X and Y.

Step 5 Computing S_{line}:

S_{line} is calculated using its definition; i.e., the ratio of lines in the correspondence to those in whole systems. Note that the number of lines in the whole systems is one after Step 1 where all comments and white spaces are removed.

SMAT works on Windows 2000 for the source code files written in C, C++, Java, and COBOL.

For two systems,each of which has m files of n lines, the worst case time complexity is as follows. *CCFinder* requires $O(mn \log(mn))$[17]. *diff* requires $O(n^2 \log n)$[13] for a single file pair and we have to perform $O(m^2)$ execution of *diff* for all file pairs. So in total, $O(m^2 n^2 \log n)$ is the worst case time complexity.

However, in practice, the execution of *diff* is not performed for all file pairs. In many cases, code clones are not detected between all file pairs, but only a few file pairs.

Practically, the execution speed of *SMAT* is fairly efficient, since it grows super-linearly. For example, it took 329 seconds to compute S_{line} of about 500K line C source code files in total on Pentium III 1GHz CPU system with 2G Bytes memory, and 980 seconds for 1M line files. On the other hand, in the case of using only *diff* for all file pairs, it took about 6 hours to compute S_{line} for 500K line files.

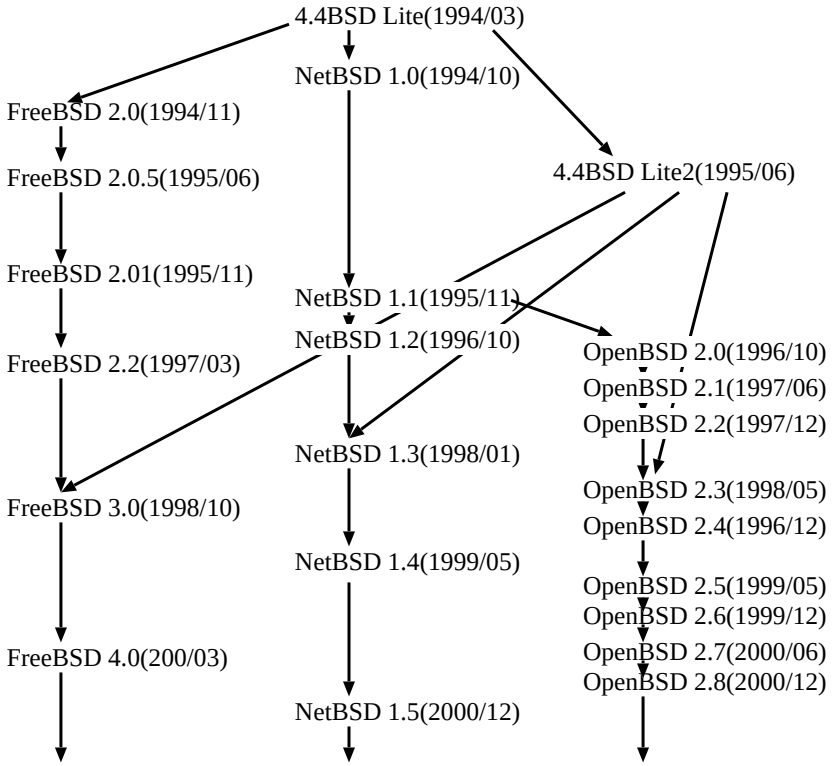

Fig. 4. BSD UNIX evolutional history.

IV. Applications of SMAT

A. BSD UNIX OS Evolution

A.1 Target systems

To explore the applicability of S_{line} and *SMAT*, we have used many versions of open-source BSD UNIX operating systems, namely 4.4-BSD Lite, 4.4-BSD Lite2[20], FreeBSD[21], NetBSD[22], OpenBSD[23]. The evolution histories of these versions is shown in Fig. 4[24]. As shown in this figure, 4.4-BSD Lite is the origination of the other versions. New versions of FreeBSD, NetBSD, and OpenBSD are currently being developed in open source development style. 23 major-release versions, as listed in Fig. 4, were chosen for computing S_{line} of all pair combinations. The evaluation was performed only on source code files related to the OS kernels written in C(i.e., *.c or *.h files).

TABLE I

THE NUMBER OF FILES AND LOC OF OS.

FreeBSD

Version	2.0	2.0.5	2.1	2.2	3.0	4.0
No. of files	891	1018	1062	1196	2142	2569
LOC	228868	275016	297208	369256	636005	878590

NetBSD

Version	1.0	1.1	1.2	1.3	1.4	1.5
No. of files	2317	3091	4082	5386	7002	7394
LOC	453026	605790	822312	1029147	1378274	1518371

OpenBSD

Version	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
No. of files	4200	4987	5245	5314	5507	5815	6074	6298	6414
LOC	898942	1007525	1066355	1079163	1129371	1232858	1329293	1438496	1478035

4.4BSD

Version	Lite	Lite2
No. of files	1676	1931
LOC	317594	411373

TABLE II

PART OF S_{line} VALUES BETWEEN BSD OS KERNEL FILES.

	FreeBSD 2.0	FreeBSD 2.0.5	FreeBSD 2.1	FreeBSD 2.2	FreeBSD 3.0	FreeBSD 4.0	4.4BSD-Lite
FreeBSD 2.0	1.000	0.833	0.794	0.550	0.315	0.212	0.419
FreeBSD 2.0.5	0.833	1.000	0.943	0.665	0.392	0.264	0.377
FreeBSD 2.1	0.794	0.943	1.000	0.706	0.421	0.286	0.362
FreeBSD 2.2	0.550	0.665	0.706	1.000	0.603	0.405	0.226
FreeBSD 3.0	0.315	0.392	0.421	0.603	1.000	0.639	0.138
FreeBSD 4.0	0.212	0.264	0.286	0.405	0.639	1.000	0.101
4.4BSD-Lite	0.419	0.377	0.362	0.226	0.138	0.101	1.000
4.4BSD-Lite2	0.290	0.266	0.258	0.179	0.133	0.100	0.651
NetBSD 1.0	0.440	0.429	0.411	0.291	0.220	0.140	0.540
NetBSD 1.1	0.334	0.348	0.336	0.254	0.193	0.152	0.421
NetBSD 1.2	0.255	0.269	0.265	0.225	0.190	0.158	0.331
NetBSD 1.3	0.205	0.227	0.225	0.201	0.208	0.179	0.259

A.2 Results

TABLE I shows the number of files and total source code lines of each version after the preprocessing of Step 1. TABLE II shows part of the resulting values S_{line} for pairs of each version. Note that TABLE II is symmetric, and the values on the main diagonal line are always 1 by the nature of our similarity.

S_{line} values between a version and its immediate ancestor/descendant version are higher

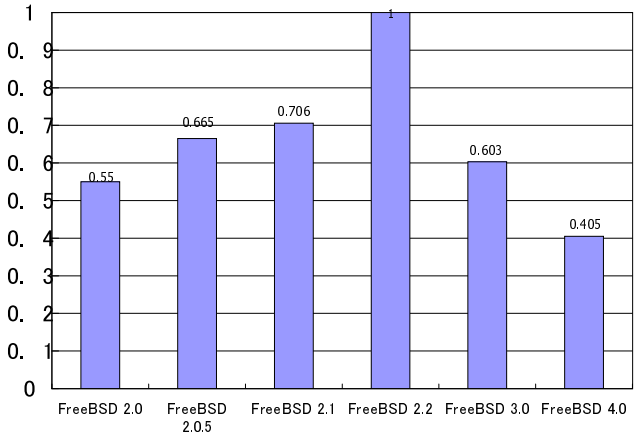

Fig. 5. S_{line} of FreeBSD 2.2 and other versions.

than the values for non-immediate ancestor/descendant versions. Fig. 5 shows S_{line} evolution between FreeBSD 2.2 and other FreeBSD versions. The values monotonically decline with increasing version distance. This indicates that the similarity metric S_{line} properly captures ordinary characteristics of software systems evolution.

Fig. 6 shows S_{line} between each version of FreeBSD and some of NetBSD. These two version streams have the same origin, 4.4-BSD Lite, and it is naturally assumed that older versions between the two streams have higher S_{line} values, since younger versions have a lot of independently added codes. This assumption is true for FreeBSD 2.0 through 2.2. However, for FreeBSD 3.0 and 4.0, the youngest version NetBSD 1.3 has higher values than other NetBSD versions (Fig. 6 A and B). This is because that both FreeBSD 3.0 and NetBSD 1.3 imported a lot of codes from 4.4-BSD Lite2 as shown Fig. 4. *SMAT* clearly spotted such an irregular nature of the evolution.

A.3 Cluster Analysis

Classifications were made of OS versions using a cluster analysis technique[25] with respect to S_{line} values shown above. The distance used for the analysis was $1 - S_{line}$, and the average value was the distance between clusters. The dendrogram from this cluster analysis is shown in Fig. 7. The horizontal axis represents the distance. OS versions categorized on the left-hand side are closer ones with high similarity values to each other.

This dendrogram reflects very well the evolution history of BSD OS versions depicted

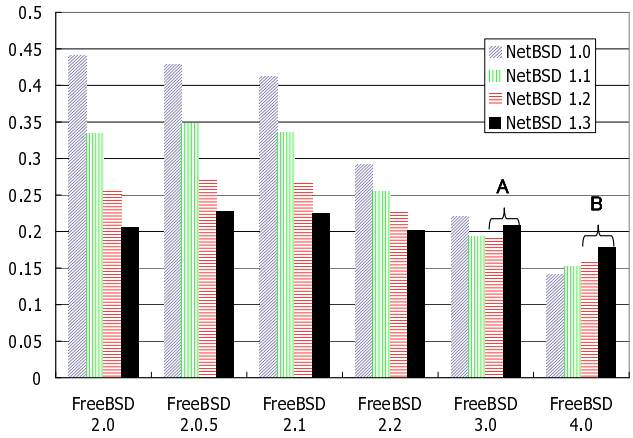

Fig. 6. S_{line} between FreeBSD and NetBSD.

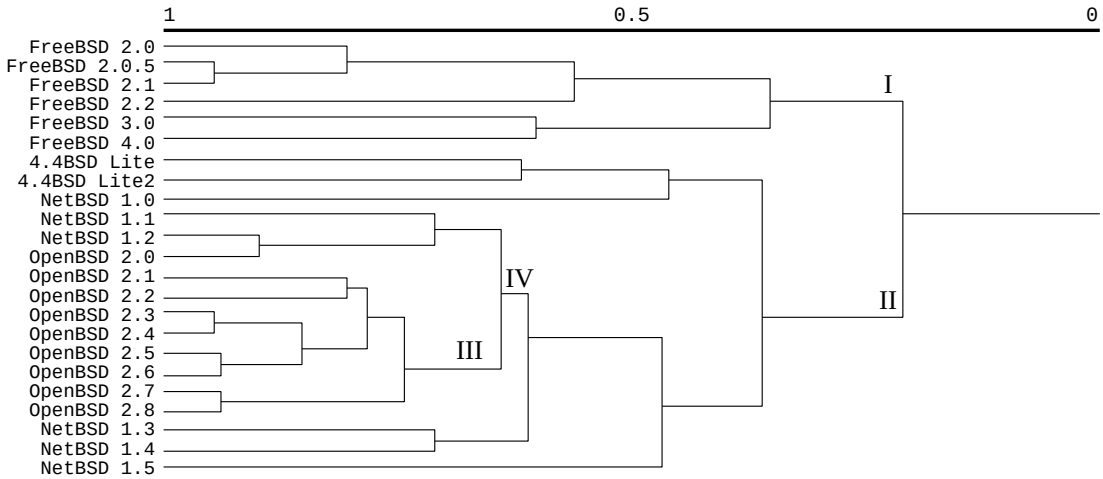

Fig. 7. Dendrogram using similarity S_{line}.

previously by Fig. 4. Further, as shown in Fig. 7, all FreeBSD versions are contained in Cluster I and all OpenBSD are in Cluster II. FreeBSD and OpenBSD are distinct genealogical systems that diverged at a very early stage of their evolution, as shown in Fig. 4. The dendrogram using S_{line} objectively discloses it.

Also, we can see the classification of NetBSD and OpenBSD. All versions of OpenBSD except for 2.0 are in the same cluster III, and this cluster is combined with NetBSD 1.1 in cluster IV together with OpenBSD 2.0. This suggests that all OpenBSD versions were derived from NetBSD 1.1. This is confirmed by their evolution history.

TABLE III

S_{line} OF STUDENT EXPERIMENT.

	A	B	C	D	E	F	G	H
A	1	0.009	0.024	0.038	0.034	0	0.054	0.047
B	0.009	1	0.040	0.001	0	0	0	0.023
C	**0.024**	0.040	1	0.060	0.042	0.088	0.118	0.170
D	0.038	0.001	0.060	1	0.010	0.040	0.069	0.039
E	0.034	0	0.042	0.010	1	0.022	0.172	0.237
F	0	0	0.088	0.040	0.022	1	0	0
G	0.054	0	0.118	0.069	0.172	0	1	0.797
H	0.047	0.023	0.170	0.039	0.237	0	**0.797**	1

A.4 Similarity with Linux

The similarity between FreeBSD 4.0 and Linux 2.2.1[26] was evaluated. These two UNIX Operating Systems were released almost at the same time, but they are considered to share no common ancestors. The resulting S_{line} value is 0.031, which is a relatively very low value (most of the lines in the correspondence are for device-dependent codes). This result indicates that S_{line} is very effective in distinguishing different systems with little shared code.

B. Student Project

SMAT was also applied to the results from an undergraduate student project. Students developed compilers written in C for a subset of PASCAL, under a lecture of theory and practice of compiler construction. The students turned in all of the object files and source code files after all 15 test cases had been passed.

We have randomly chosen 8 student results (named A to H). The total source code sizes were between 3427 and 6866 lines. The results of S_{line} between any two compilers are shown in TABLE III.

As you can see, most of the similarity values were very low. For example, between A and C the value is 0.024. It is considered that A and C wrote many distinct codes independently and that they created accidentally a few similar codes, as shown in Fig. 8. These code portions were detected as shared clones since they both have an "if ... then ... else" structure with function calls having two parameters.

```
else if (s==SMOD) generate_code(C_CALL, L_MOD);
else generate_code(C_CALL, L_MULT);
```

```
if(nenum == STRUE){
fprintf(ofp," LEA GR1,1\n");
} else {
    fprintf(ofp," LEA GR1,0\n");
}
```

Fig. 8. Similar codes between A and C.

```
outputfilename[len+1]=NULL;
infile = fopen(argv[1], "r");
outfile = fopen(outputfilename, "w");
if ((!infile)——(!outfile))
{
    fprintf(stderr, "could not open file \n");
    exit(1);
}
```

```
outputfilename[len+1]='\0';
fp = fopen(argv[1], "r");
outfile = fopen(outputfilename, "w");
if ((!fp)——(!outfile)) {
    fprintf(stderr, "could not open file\n");
    exit(1);
}
```

Fig. 9. Similar codes between G and H.

The highest value, 0.797, was obtained between G and H. As shown in Fig. 9, the corresponding lines have different line breaks and variable names, but they have the same system structures. This case was considered plagiarism and *SMAT* was very effective in detecting it.

V. Extension to Difference Extraction Tool

As discussed the above sections, *SMAT* very effectively computes similarity metric S_{line} of two software systems. However, the result of *SMAT* is simply a similarity value, and there is no report or observation of the detail of the difference of the systems.

S_{line} computes the correspondence of lines. Using this information, the detailed difference can be reported and then one system can be compactly represented (or *compressed*) by another system.

A difference extraction tool (*DET*) of two software systems was developed. An overview of *DET* follows.

INPUTS:

Directory of source code files for the target system

Directory of source code files for the base system

OUTPUTS:

Difference file between the target system and the base system

Summary report of file correspondence

Step 1: We execute *CCFinder* on all concatenated files of the target system and the base system.

Step 2: For each target file f_t, a base file f_b is picked which contains more clones against f_t than any other files. Then we perform *diff* between f_t and f_b. If the result of *diff* is smaller than f_t itself then we consider that f_t and f_b match and the *diff* result is output. Otherwise raw f_t is output. If no clones are found in f_t, we also output f_t itself.

Step 3: Report a summary which contains the list of matched file names, their original sizes, and *diff* result sizes.

Underline concept of *DET* is very similar to *SMAT*. It performs clone detection by *CCFinder* between two systems, and then extracts difference by *diff* between file pairs with clones. *SMAT* explores all possible file pairs where any shared clone exists, but *DET* tries only one file pair where the most shared clones are found.

DET also restores the target system from the given difference file together with the base system.

DET was applied to various BSD UNIX versions. For example, it was applied to FreeBSD 2.0 and 2.0.5 where the total sizes of all kernel source code files were about 10.8M Bytes and 12.8M Bytes, respectively. The size of the *DET* difference file from 2.0 to 2.0.5 is 4.8M Bytes, and from 2.0.5 to 2.0 is 2.8M Bytes. The reason why the former is larger is that the newer version 2.0.5 contains many newly added files which are straightforwardly included in the difference file.

The difference extracted by *DET* was smaller than simply using *diff*. To check this, we executed "diff -nN" recursively into directories and measured the output sizes, which were about 5.8M Bytes either from 2.0 to 2.0.5 or from 2.0.5 to 2.0. The outputs of *DET* were about 1.2 to 5 times smaller than *diff* for the cases of other versions.

This suggested that *DET* could be useful to archive older versions based on the current version of a system. Considering the dramatic increase of disc capacity and rapid decrease of its price, it may be feasible to store all versions of a system without the compression.

However, *DET* generates a summary report containing correspondence of old files and new files, which gives very important clues about file name changes and directory structure modifications.

DET is considered a suitable tool to trace evolution of large software systems. Also, *DET* could be useful for managing large product lines where many versions with minor modifications exist.

VI. Discussion and Related Work

As presented in previous sections, our similarity definition, similarity metric S_{line}, and metric evaluation *SMAT* worked well. Our approach provides a practical, meaningful and useful measure for maintaining and managing large software systems.

A. Similarity Definition

The definition of similarity used symmetric in the sense that the similarity for X and Y and that for Y and X are the same. This is because the similarity is defined as $(|x| + |y|)/(|X| + |Y|)$ where x is the set of X's elements in the correspondence and y is that of Y's elements.

Another definition of the similarity is such that $|x|/|X|$ where the correspondence is determined with respect to Y, but Y and y do not appear explicitly in the similarity formula[11]. This similarity definition gives a single side view of the system difference, which would make it suitable for investigating characteristics of individual systems. However, an overview of system evolution is difficult to archive with the single side definition. For example, to make dendrograms as shown in Fig. 7, it is necessary to define the distance of two systems X and Y. With our approach, the similarity is used as the distance. In the case of the single side definition, an average of two similarity values might be used such that $(|x|/|X| + |y|/|Y|)/2$, but this average value has less rational meaning.

B. Metric S_{line}

The correspondence which determines S_{line} is a many-to-many matching between source lines located within files and directories. The reasons of the many-to-many matching is that we would like to trace the movement of any source code block within files and directories

as much as possible, and obtain the ratio of succeeded and revised codes to overall codes.

It is possible to use one-to-one matching in the correspondence, but it characterizes the similarity metric too naively to copied codes. Assume that a system X is composed of a file x_1, and a new system X' is composed of two files x'_1 and x'_2 where both x'_1 and x'_2 are the same copies of x_1. In our definition using the many-to-many matching, the similarity is 1.0, but using the one-to-one matching gives 0.5 which does not reflect development efforts properly.

Actually, metrics S_{line} have very high correlation to with development efforts. The correlation coefficient between S_{line} values and release durations of FreeBSD versions was -0.973. On the other hand the correlation coefficient between the size increases and the release durations was 0.528. Therefore, S_{line} should be a reasonable measures of development efforts.

Another reason for using many-to-many matching is performance. The one-to-one approach needs some mechanism to choose the best matching pair from many possiblities, which generally is not a simple, straight forward process.

C. Similarity Metric S_{fn} using file name matching

An alternative and much simpler metric S_{fn} for can be employed for similarity.

Consider a software systems composed of source code files. The correspondence between two such systems is the set of all file pairs having the same file(path) names. That is, if a file p_i in one system and a file q_j in another system have the same file names, including file paths, then pair (p_i, q_j) is included in the correspondence.

It is very easy to compute S_{fn}, by checking each file path. However, S_{fn} is very fragile to renames and restructures of source code files. Also, it cannot detect changes of file contents. Furthermore, since S_{fn} does not account for the sizes of each source code files, it might produce values far from reality.

For example, when S_{fn} was applied to the student project described in Section IV-B, the similarity values were as shown in TABLE IV. Using S_{line} and *SMAT*, we were able to detect a possible plagiarism between G and H(S_{line}=0.797). However, the S_{fn} between G and H was 0.190, which was too low to suspect plagiarism. The correlation between S_{line} and S_{fn} was -0.004, meaning that there was no relation between them.

TABLE IV

S_{fn} OF COMPILER SYSTEMS.

	A	B	C	D	E	F	G	H
A	1	0	0	0.113	0	0	0	0
B	0	1	0.666	0.125	0.600	0.666	0.105	0.428
C	0	0.666	1	0.137	0.857	1	0.125	0.545
D	0.113	0.125	0.137	1	0.133	0.137	0.102	0.117
E	0	0.600	0.857	0.133	1	0.857	0.235	0.500
F	0	0.666	1	0.137	0.857	1	0.125	0.545
G	0	0.105	0.125	0.102	0.235	0.125	1	0.190
H	0	0.428	0.545	0.117	0.500	0.545	**0.190**	1

D. SMAT

SMAT worked very efficiently for large software systems. To compute S_{line}, execution of *diff* for all possible file pairs would have been a simple approach. However, the execution speed would have become unacceptably slow as mentioned in III-C. Combining *CCFinder* and *diff* boosted the performance of *SMAT*. Also, as mentioned before, the movement and modification of source code lines can be traced better by *CCFinder*, which effectively detects clones with different white spaces, comments, identifier names, and so on. The matching computation using only *diff* cannot chase those changes.

There are a lot of researches on clone detection and many tools have been developed[27], [28], [29]. We would be able to use those tools in stead of CCFinder.

E. Related Work

There has been a lot of work on finding plagiarism in programs. Ottenstein used Halstead metric valuations[30] of target program files for comparison[31]. There are other approaches which use a set of metric values to characterize source programs[32], [33], [34]. Also, structural information has been employed to increase precision of comparison[35], [36]. In order to improve both precision and efficiency, abstracted text sequences (token sequences) can be employed for comparison[8], [9], [37], [10]. Source code texts are translated into token sequences representing programs structures, and the longest common subsequence algorithm is applied to obtain matching.

These systems are aimed mainly at finding similar software code in the education envi-

ronment. The similarity metric values computed by comparison of metrics values do not show the ratio of similar codes to non-similar codes, and thus would be less intuitively accurate. Also, scalability of those evaluation methods to large software system such as UNIX OS is not known.

In reverse engineering field, there has been research on measuring similarity of components and restructuring modules in a software system, to improve its maintenanceability and understandability[38], [39], [40]. Such similarity measures are based on several metric values such as shared identifier names and function invocation relations. Although these approaches involve important views of similarity, their objectives are to identify components and modules inside a single system, and cannot be applied directly to inter-system similarity measurement.

A study on the similarity between documents is presented by Broder[11]. In this approach, a set of fixed-length token sequences are extracted from documents. Then two sets X and Y are obtained for each document to compute their intersection of them. The similarity is defined as $(|X| \cap |Y|)/(|X| \cup |Y|)$.

This approach is very suitable for efficiently computing the resemblance of a large collection of documents such as world-wide web documents. However, choosing token sequences greatly affects the resulting values. Tokens with minor modification would not be detected. Therefore, this is probably an inappropriate approach for computing subjective similarity metric for source code files.

Manber[12] developed a tool to identify similar files in large systems. This tool uses a set of keywords and extracts subsequences starting with those keywords as fingerprints. A fingerprint set X of a target file is encoded and compared to a fingerprint set Y of a query file. The similarity is defined as $|X \cap Y|/|X|$.

This approach works very efficiently for both source program files and document files and would fit exploration of similar files in a large system. However, it is fragile to the selection of keywords. Also, it would be too sensitive to minor modifications of source program files such as identifier changes and comment insertions.

These methods are all quite different from those developed and presented herein, since they do not perform comparison on raw and overall text sequences, but rather on sam-

pled text sequences. Sampling approaches would get high performance, but the resulting similarity values would be less significant than our whole text comparison approach.

VII. Conclusion

A proposed definition of similarity between two software systems with respect to correspondence of source code lines was formulated as a similarity metric called S_{line}. An S_{line}-based evaluation tool *SMAT* was developed and applied to various software systems. The results showed that S_{line} and *SMAT* are very useful for identifying the origin of the systems and to characterize their evolution. Furthermore, using the computation process of *SMAT*, a difference extraction tool, *DET*, was developed which compresses a target software system relative to a base system and reports the difference.

Further applications of *SMAT* to various software systems and product lines will be made to investigate their evolution. From a macro level analysis view point, categorization and taxonomy of software systems analogous to molecular phylogeny should be an intriguing issue to pursue. From a micro level analysis view point, chasing specific code blocks through system evolution will be interesting to perform.

References

[1] V. R. Basili, L. C. Briand, S. E. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett, "Understanding and predicting the process of software maintenance release," in *18th International Conference on Software Engineering*, berlin, 1996, pp. 464–474.

[2] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[3] S. Cook, H. Ji, and R. Harrison, "Dynamic and static views of software evolution," in *the IEEE International Conference On Software Maintenance (ICSM 2001)*, Florence, Italy, Nov. 2001, pp. 592–601.

[4] C. F. Kemerer and S. Slaughter, "An empirical approach to studying software evolution," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 493–509, 1999.

[5] The First Software Product Line Conference (SPLC1), ," http://www.sei.cmu.edu/plp/conf/SPLC.html, Denver, Colorado, August 2000.

[6] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison Wesley, 2001.

[7] A. Baxevanis and F. Ouellette, Eds., *Bioinformatics 2nd edition*, pp. 323–358, John Wiley and Sons, Ltd., England, 2001.

[8] A. Aiken, "Moss (measure of software similarity) plagiarism detection system," http://www.cs.berkeley.edu/ moss/.

[9] L. Prechelt, G. Malpohl, and M. Philippsen, "Jplag: Finding plagiarisms among a set of programs," Technical Report 2000-1, Fakultat fur Informatik, Universitat Karlsruhe, Germany, 2000.

[10] M. J. Wise, "YAP3: Improved detection of similarities in computer program and other texts," *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, vol. 28, 1996.

[11] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings of Compression and Complexity of Sequences*, 1998, pp. 21–29.

[12] U. Manber, "Finding similar files in a large file system," in *Proceedings of the USENIX Winter 1994 Technical Conference*, San Fransisco, CA, USA, January 17–21 1994, pp. 1–10.

[13] J. W. Hunt and M. D. McIlroy, "An algorithm for differential file comparison," Tech. Rep. 41, Computing Science, Bell Laboratories, Murray Hill, New Jersey, 1976.

[14] W. Miller and E. W. Myers, "A file comparison program," *Software- Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

[15] E. W. Myers, "An $O(ND)$ difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–256, 1986.

[16] E. Ukkonen, "Algorithms for approximate string matching," *INFCTRL: Information and Computation (formerly Information and Control)*, vol. 64, pp. 100–118, 1985.

[17] T. Kamiya, S. Kusumoto, , and K. Inoue, "A token-based code clone detection tool - ccfinder and its empirical evaluation," Techinal report, Osaka University, Department of Information and Computer Scineces, Inoue Laboratory, 2000.

[18] T. Kamiya, S. Kusumoto, , and K. Inoue, "A token-based code clone detection technique and its evaluation," *Technical report of IEICE, SS2000-42-52*, vol. 100, no. 570, pp. 41–48, 2001.

[19] D. Gusfield, "Algorithms on strings, trees, and sequences," Computer Science and Computational Biology. Cambridge University Press, 1997.

[20] M.K. McKusick, K. Bostic, M.J. karels, and J.S. Quarterman, *The Design and Implementation of the 4.4BSD UNIX Operating System*, Addison-Wesley, 1996.

[21] The FreeBSD Project, "The FreeBSD Project," http://www.freebsd.org/.

[22] The NetBSD Foundation Inc., "The NetBSD Project," http://www.netbsd.org/.

[23] OpenBSD, "OpenBSD," http://www.openbsd.org/.

[24] W. Schneider, "The unix system family tree: Research and bsd," ftp://ftp.freebsd.org/pub/FreeBSD/branches/-current/src/share/misc/bsd-family-tree.

[25] B. S. Everitt, *Cluster Analysis*, Edward Arnold, 3rd edition, London, 1993.

[26] Linux Online, "The Linux Home Page at Linux Online," http://www.linux.org/.

[27] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Second Working Conference on Reverse Engineering*, Toronto, Canada, July 1995, pp. 86–95.

[28] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, Bethesda, Maryland, Nov. 1998, pp. 368–378.

[29] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of the International Conference on Software Maintenance*, Monterey, California, Nov. 1996, pp. 244–253.

[30] M. H. Halstead, *Elements of Software Science*, Elsevier, New York, 1977.

[31] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," *ACM SIGCSE Bulletin*, vol. 8, no. 4, pp. 30–41, 1976.

[32] H. L. Berghel and D. L. Sallach, "Measurements of program similarity in identical task environments," *ACM SIGPLAN Notices*, vol. 19, no. 8, pp. 65–76, 1984.

[33] J. L. Donaldson, A. M. Lancaster, and P. H. Sposato, "A plagiarism detection system," *ACM SIGCSE Bulletin(Proc. of 12th SIGSCE Technical Symp.)*, vol. 13, no. 1, pp. 21–25, 1981.

[34] S. Grier, "A tool that detects plagiarism in pascal programs," *ACM SIGCSE Bulletin(Proc. of 12th SIGSCE Technical Symp.)*, vol. 13, no. 1, pp. 15–20, 1981.

[35] H. T. Jankowitz, "Detecting plagiarism in student Pascal programs," *The Computer Journal*, vol. 31, no. 1, pp. 1–8, 1988.

[36] K. L. Verco and M. J. Wise, "Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems," in *Porc. of 1st Ausutralian Conference on Computer Science Education*, John Rosenberg, Ed., Sydney, Australia, july 1996, pp. 86–95.

[37] G. Whale, "Identification of program similarity in large populations," *The Computer Journal*, vol. 33, no. 2, pp. 140–146, 1990.

[38] S. C. Choi and W. Scacchi, "Extracting and restructuring the design of large systems," *IEEE Software*, vol. 7, no. 1, pp. 66–71, Jan. 1990.

[39] R. W. Schwanke, "An intelligent for re-engineering software modularity," in *Proceedings of theThirteenthInternational Conference on Software Engineering*, Austin, Texas, USA, May 1991, pp. 83–92.

[40] R. W. Schwanke and M. A. Platoff, "Cross references are features," in *Proceedings of the 2nd International Workshop on Software Configuration Management*, Oct. 1989, pp. 86–95.

English is reviewed and revised by Edit Science, Inc.