Measuring Similarity of Large Software Systems Based on Source Code Correspondence

Tetsuo Yamamoto[1], Makoto Matsushita[2], Toshihiro Kamiya[3] and Katsuro Inoue[2]

[1] College of Information Science and Engineering, Ritsumeikan University,
Kusatsu, Shiga 525-8577, Japan
Phone:+81-77-561-5265,Fax:+81-77-561-5265
tetsuo@cs.ritsumei.ac.jp
[2] Graduate School of Information Science and Technology, Osaka University,
Toyonaka, Osaka 560-8531, Japan
Phone:+81-6-6850-6571,Fax:+81-6-6850-6574
matusita@ist.osaka-u.ac.jp,inoue@ist.osaka-u.ac.jp
[3] Presto,Japan Science and Technology Agency.
Current Address:Graduate School of Information Science and Technology, Osaka University,
Toyonaka, Osaka 560-8531, Japan
Phone:+81-6-6850-6571,Fax:+81-6-6850-6574
kamiya@ist.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). The resulting similarity valuations clearly revealed the evolutionary history characteristics of the BSD UNIX Operating System.

1 Introduction

Long-lived 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[1–4], 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. This was motivated by our scientific curiosity such as what is the quantitative similarity of two software systems. We would like to quantify the similarity with a

solid and objective measure. The quantitative measure for similarity is an important vehicle to observe 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[8–10]. The usual approach extracts several metric values (or attributes) characterizing the target programs and then compares those values.

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.

We have been interested in comparing all the files. 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. We are afraid that using sampled information may lose some important information. 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 does not scale so that it would be impractical to apply to large systems with thousands of files.

Instead, 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, and have performed cluster analysis of the similarity values to create a dendrogram that correctly shows evolution history of BSD UNIX.

Section 2 presents a formal definition of similarity and its metric S_{line}. Section 3 describes a practical method for computing S_{line} and shows the implementation tool *SMAT*. Section 4 shows applications of *SMAT* to versions of BSD UNIX. Results of our work and comparison with related research are given in Section 5. Concluding remarks are given in Section 6.

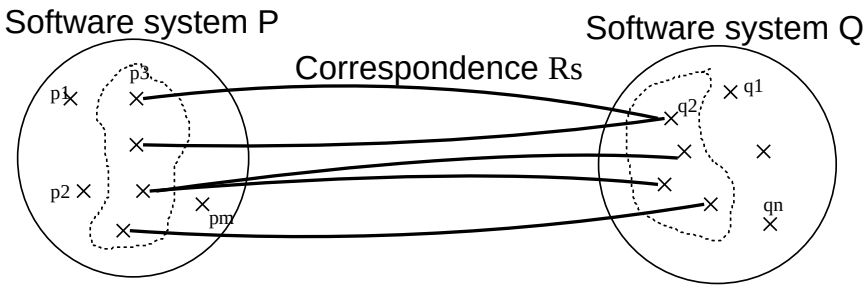

Fig. 1. Correspondence of elements Rs

2 Similarity of Software Systems

2.1 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 *Correspondence* R_s the set of matched pair (p_i, q_j), where $R_s \subseteq P \times Q$(See Figure 1). *Similarity* S of P and Q with respect to R_s is defined as follows.

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

This definition means that the similarity is the ratio of the total number of p's and q's elements composing R_s to the total number of elements of P and Q. The numerator is the total number of p_i and q_i possibly related to R_s, and the denominator is the total number of p_i and q_i. 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 $S = 1$.

2.2 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 operational 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 $x_{i1}, x_{i2}, \cdots$. Pair (x_{ij}, y_{mn}) of two lines x_{ij} and y_{mn} between 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 that will be discussed in detail later in this paper. 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 consuming process. A practical approach for this is given in Section 3.

It is possible to consider other definitions of similarity and its metrics. A comparison to other such approaches is presented in Section 5.

3 Measuring S_{line}

3.1 Approach

A key problem of S_{line} is computation of the correspondence. A straightforward approach we might consider is that first we construct appended files $x_1; x_2; \cdots$ and $y_1; y_2; \cdots$ which are concatenation of all source 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 $x_1; x_2; \cdots$ and $y_1; y_2; \cdots$ by some tool, say *diff*[13], which implements an LCS-finding algorithm[14–16]. The extracted LCS is used as the correspondence.

However, this method is fragile due to the change of file concatenation order caused by internal reshuffling of files, since *diff* cannot follow line block movement to different positions in the files. For example, for two systems $X = a; b; c; d; e$ and $Y = d; e; a; b; c$, the LCS found by *diff* is $a; b; c$. In this case, a subsequence $d; e$ cannot be detected as a common sequence.

Another approach is that we try to greedily 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].

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[18], 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 removed, and the distinction of identifier names is disregarded. By the normalization and parameterization, code blocks with minor modification are effectively detected as clones.

Applying *CCFinder* to two sets of files $\{x_1, x_2, \cdots\}$ and $\{y_1, y_2, \cdots\}$ finds all possible clone pairs (b_x, b_y), where b_x is a code block in x_1, or $x_2, \cdots$. and b_y is that of y_1, or $y_2, \cdots$, and b_x and b_y are identical without considering difference of line breaks, white spaces, comments, user-defined identifiers, constant values, and so on. This process is performed by simply specifying two sets of file names or directory names containing $\{x_1, x_2, \cdots\}$ and $\{y_1, y_2, \cdots\}$. *CCFinder* reports all clone pairs among the files. Those clone pairs found are members of the correspondence.

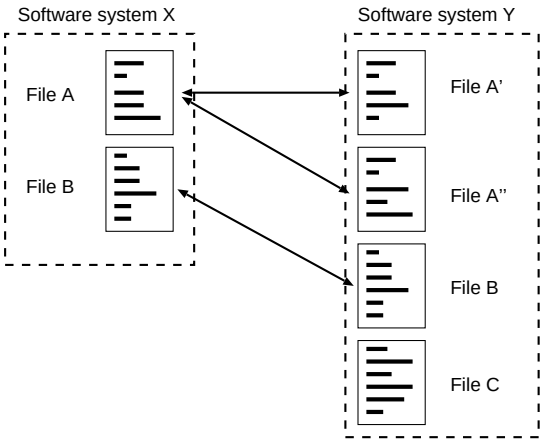

Fig. 2. How to find a correspondence

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 lengths of l_1 and l_2 are less than threshold of *CCFinder*(usually 20 tokens[4]), then *CCFinder* reports no clones at all. To reclaim such small similar blocks and similar directives undetected by *CCFinder*, *diff* is applied to all pairs of the two files 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 to increase S_{line} by about 10%, compared to using only *CCFinder*.

3.2 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 Figure 2. X is composed of two source code files A and B, while 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 file sets $\{A, B\}$ and $\{A', A'', B, C\}$. 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 line in the clones found across files is put into the correspondence.

Next, *diff* is applied to the 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

[4] The threshold 20 used here is determined by our experiences of CCFinder[17]. Based on the analyses of our experiences, a practical threshold is 20 to 30. If we set a further lower number, say 5, a lot of accidentally similar substrings (e.g., a=b+c is a clone of x=y+z) are detected as clones, and the precision of the resulting similarity value is degraded.

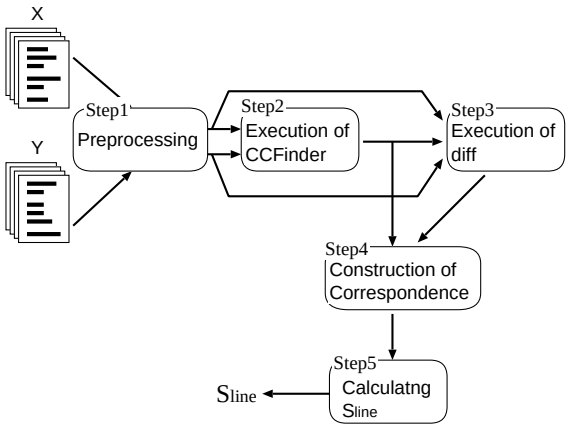

Fig. 3. Similarity measuring process

obtained by the clone detection. The correspondence finally we obtain includes all the clones found by *CCFinder* and all the common subsequences found by *diff*.

This approach has benefits in both computation complexity and precision of the results. We do not need to perform *diff* on all the file pair combinations. Also, we can chase movement of lines inside or outside the files, which cannot be detected by *diff* alone.

3.3 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 Figure 3.
INPUTS: File paths of two systems X and Y, each of which represents the subdirectory containing all source code.
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 file sets X and Y. *CCFinder* has an option for the minimum number of tokens of clones to be detected, and whose default is set to 20.

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 determine the correspondence between X and Y.

Step 5 Calculating 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 after Step 1 is used hereafter, where all comments and white spaces are removed.

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

For given 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, $O(m^2n^2\log n)$ 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 performance 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 a 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.

4 Applications of SMAT

4.1 BSD UNIX Evolution

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[19], FreeBSD[5], NetBSD[6], OpenBSD[7]. The evolution histories of these versions are shown in Figure 4[8]. 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 Figure 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).

Results Table 1 shows the number of files and total source code lines of each version after the preprocessing of Step 1. Table 2 shows part of the resulting values S_{line} for pairs of each version. Note that Table 2 is symmetric, and the values on the main diagonal line are always 1 by the nature of our similarity.

[5] http://www.freebsd.org/
[6] http://www.netbsd.org/
[7] http://www.openbsd.org/
[8] http://www.tribug.org/img/bsd-family-tree.gif

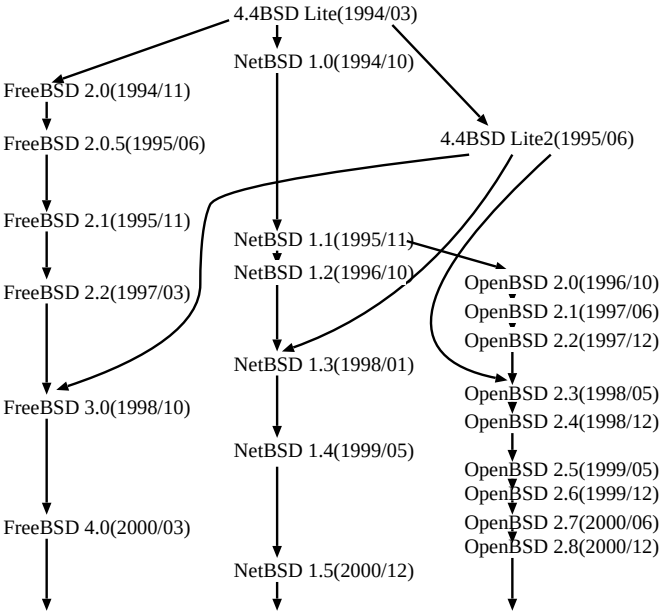

Fig. 4. BSD UNIX evolutional history

Table 1. The number of files and LOC of BSD UNIX

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 2. Part of S_{line} values between BSD UNIX kernel files

	F 2.0	F 2.0.5	F 2.1	F 2.2	F 3.0	F 4.0	Lite	Lite2	N 1.0	N 1.1	N 1.2	N 1.3
FreeBSD 2.0	1.000											
FreeBSD 2.0.5	0.833	1.000										
FreeBSD 2.1	0.794	0.943	1.000									
FreeBSD 2.2	0.550	0.665	0.706	1.000								
FreeBSD 3.0	0.315	0.392	0.421	0.603	1.000							
FreeBSD 4.0	0.212	0.264	0.286	0.405	0.639	1.000						
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	1.000				
NetBSD 1.0	0.440	0.429	0.411	0.291	0.220	0.140	0.540	0.450	1.000			
NetBSD 1.1	0.334	0.348	0.336	0.254	0.193	0.152	0.421	0.431	0.691	1.000		
NetBSD 1.2	0.255	0.269	0.265	0.225	0.190	0.158	0.331	0.436	0.553	0.783	1.000	
NetBSD 1.3	0.205	0.227	0.225	0.201	0.208	0.179	0.259	0.366	0.445	0.622	0.769	1.000

F 2.0 means FreeBSD 2.0, Lite means 4.4BSD-Lite, N 1.0 means NetBSD 1.0.

As a general tendency, S_{line} values between a version and its immediate ancestor/descendant version are higher than the values for non-immediate ancestor/descendant versions. Figure 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.

Figure 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 (Figure 6 A and B). This is because both FreeBSD 3.0 and NetBSD 1.3 imported a lot of code base from 4.4-BSD Lite2 as shown Figure 4. *SMAT* clearly spotted such an irregular nature of the evolution.

Cluster Analysis Classifications were made of OS versions using a cluster analysis technique[20] with respect to S_{line} values shown above. For the cluster analysis, we need to define the distance of two OS versions. We defined it by $1 - S_{line}$. A cluster is a non-empty collection of OS versions, and the distance of two clusters are the average of the pairwise distances of the numbers of each cluster. To construct a dendrogram, we start with clusters having exactly one version, and merge the nearest two clusters into one cluster. The merging process is repeated until we get only one cluster. The dendrogram from this cluster analysis is shown in Figure 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 UNIX versions depicted previously by Figure 4. Further, as shown in Figure 7, all FreeBSD versions are contained in Cluster I and all OpenBSD are in Cluster II. FreeBSD and OpenBSD

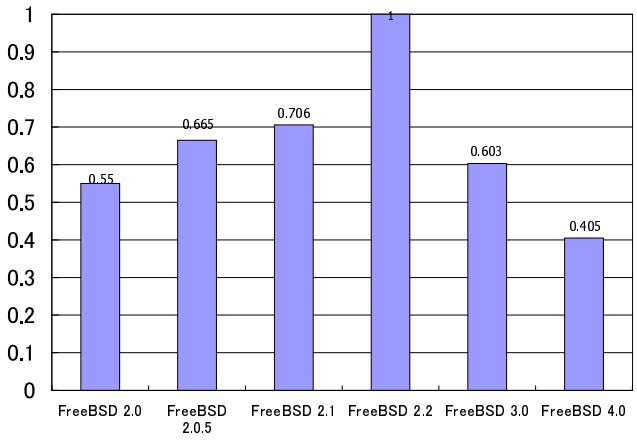

Fig. 5. S_{line} of FreeBSD 2.2 and other versions

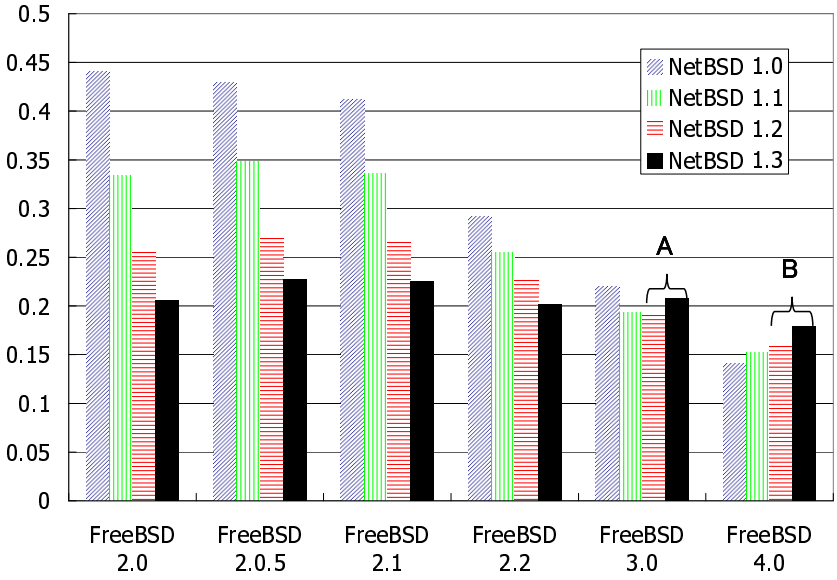

Fig. 6. S_{line} between FreeBSD and NetBSD

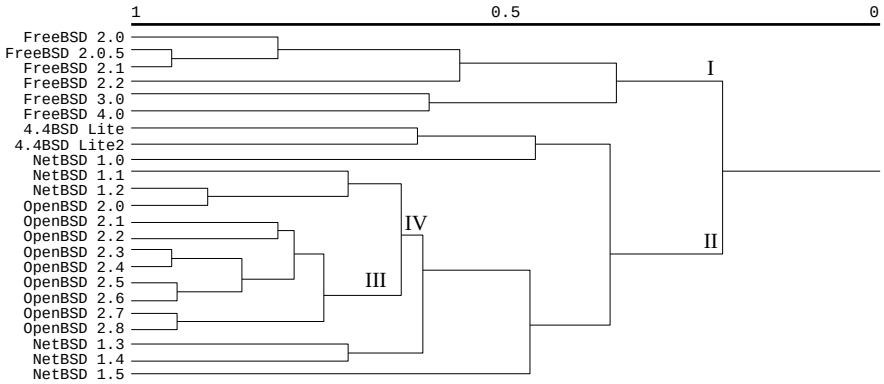

Fig. 7. Dendrogram of BSD UNIX

are distinct genealogical systems that diverged at a very early stage of their evolution, as shown in Figure 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.

5 Discussion and Related Work

As presented in previous sections, our similarity definition, similarity metric S_{line}, and the similarity measurement tool *SMAT* worked fine to various software systems.

5.1 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 duplicated source code. 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 0.67, since x_1 matches x'_1 (or x'_2) by the one-to-one matching so that the similarity is $(|\{x_1\}| + |\{x'_1\}|)/(|\{x_1\}| + |\{x'_1, x'_2\}|) = 2/3 \cong 0.67$. Therefore, we think that the one-to-one matching does not reflect development efforts properly.

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 possibilities, which generally is not a simple, straightforward process.

Actually, metric S_{line} showed very high correlation with release durations of FreeBSD. The release durations are calculated from the difference of OS release dates presented in Figure 4. The Pearson's correlation coefficient between S_{line} values and release durations of FreeBSD versions is -0.973. On the other hand, the increases of the size or the release durations are not highly correlated. The Pearson's correlation coefficient between the size increases(Table 1) and the release durations is 0.528. Therefore, we think that S_{line} is a reasonable measures of release durations in this case.

5.2 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 3.3. 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[21–27].We could have used those tools instead of *CCFinder*.

5.3 Related Work

There has been a lot of work on finding plagiarism in programs. Ottenstein used Halstead metric valuations[28] of target program files for comparison[29]. There are other approaches which use a set of metric values to characterize source programs[30–32]. Also, structural information has been employed to increase precision of comparison[33, 34]. In order to improve both precision and efficiency, abstracted text sequences (token sequences) can be employed for comparison[8–10, 35]. 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 environment. The similarity metric values computed by comparison of metrics values do not show the ratio of similar codes to non-similar codes. Also, scalability of those evaluation methods to large software system such as BSD UNIX 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 maintainability and understandability[36–38]. 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.

There are many literatures for detection of code clones and patterns[17, 21–27]. Some of those proposed metrics for the clones; however they have not been extended to the similarity of two large software systems.

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. 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.

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

6 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.

Further applications of *SMAT* to various software systems and product lines will be made to investigate their evolution. From a macro level analysis viewpoint, 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. Antoniol, G., Villano, U., Merlo, E., Penta, M.D.: Analyzing cloning evolution in the linux kernel. Information and Software Technology 44 (2002) 755–765
2. Basili, V.R., Briand, L.C., Condon, S.E., Kim, Y.M., Melo, W.L., Valett, J.D.: Understanding and predicting the process of software maintenance release. In: 18th International Conference on Software Engineering, Berlin (1996) 464–474

3. Cook, S., Ji, H., Harrison, R.: Dynamic and static views of software evolution. In: the IEEE International Conference On Software Maintenance (ICSM 2001), Florence, Italy (2001) 592–601
4. Kemerer, C.F., Slaughter, S.: An empirical approach to studying software evolution. IEEE Transactions on Software Engineering 25 (1999) 493–509
5. The First Software Product Line Conference (SPLC1): The First Software Product Line Conference (SPLC1), http://www.sei.cmu.edu/plp/conf/SPLC.html (2000) Denver, Colorado (2000)
6. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley (2001)
7. Baxevanis, A., Ouellette, F., eds. In: Bioinformatics 2nd edition. John Wiley and Sons, Ltd., England (2001) 323–358
8. Schleimer, S., Wilkerson, D., Aiken, A.: Winnowing: Local algorithms for document fingerprinting. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. (2003) 76–85
9. Prechelt, L., Malpohl, G., Philippsen, M.: Jplag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Fakultat fur Informatik, Universitat Karlsruhe, Germany (2000)
10. Wise, M.J.: YAP3: Improved detection of similarities in computer program and other texts. SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education) 28 (1996)
11. Broder, A.Z.: On the resemblance and containment of documents. In: Proceedings of Compression and Complexity of Sequences. (1998) 21–29
12. Manber, U.: Finding similar files in a large file system. In: Proceedings of the USENIX Winter 1994 Technical Conference, San Fransisco, CA, USA (1994) 1–10
13. Hunt, J.W., McIlroy, M.D.: An algorithm for differential file comparison. Technical Report 41, Computing Science, Bell Laboratories, Murray Hill, New Jersey (1976)
14. Miller, W., Myers, E.W.: A file comparison program. Software- Practice and Experience 15 (1985) 1025–1040
15. Myers, E.W.: An $O(ND)$ difference algorithm and its variations. Algorithmica 1 (1986) 251–256
16. Ukkonen, E.: Algorithms for approximate string matching. INFCTRL: Information and Computation (formerly Information and Control) 64 (1985) 100–118
17. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering 28 (2002) 654–670
18. Gusfield, D.: Algorithms on strings, trees, and sequences. Computer Science and Computational Biology. Cambridge University Press (1997)
19. McKusick, M., Bostic, K., karels, M., Quarterman, J.: The Design and Implementation of the 4.4BSD UNIX Operating System. Addison-Wesley (1996)
20. Everitt, B.S.: Cluster Analysis. Edward Arnold, 3rd edition, London (1993)
21. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Second Working Conference on Reverse Engineering, Toronto, Canada (1995) 86–95
22. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance, Bethesda, Maryland (1998) 368–378
23. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: Proceedings of the International Conference on Software Maintenance, Oxford, England, UK (1999) 109–119
24. Johnson, J.H.: Identifying redundancy in source code using fingerprints. In: Proceedings of CASCON '93, Toronto, Ontario (1993) 171–183

25. Johnson, J.H.: Substring matching for clone detection and change tracking. In: Proceedings of the International Conference on Software Maintenance, Victoria, British Columbia (1994) 120–126
26. Kontogiannis, K.: Evaluation experiments on the detection of programming patterns using software metrics. In: Proceedings of Fourth Working Conference on Reverse Engineering, Amsterdam, Netherlands (1997) 44–54
27. Mayrand, J., Leblanc, C., Merlo, E.: 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 (1996) 244–253
28. Halstead, M.H.: Elements of Software Science. Elsevier, New York (1977)
29. Ottenstein, K.J.: An algorithmic approach to the detection and prevention of plagiarism. ACM SIGCSE Bulletin 8 (1976) 30–41
30. Berghel, H.L., Sallach, D.L.: Measurements of program similarity in identical task environments. ACM SIGPLAN Notices 19 (1984) 65–76
31. Donaldson, J.L., Lancaster, A.M., Sposato, P.H.: A plagiarism detection system. ACM SIGCSE Bulletin(Proc. of 12th SIGSCE Technical Symp.) 13 (1981) 21–25
32. Grier, S.: A tool that detects plagiarism in pascal programs. ACM SIGCSE Bulletin(Proc. of 12th SIGSCE Technical Symp.) 13 (1981) 15–20
33. Jankowitz, H.T.: Detecting plagiarism in student Pascal programs. The Computer Journal 31 (1988) 1–8
34. Verco, K.L., Wise, M.J.: Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In Rosenberg, J., ed.: Proc. of 1st Ausutralian Conference on Computer Science Education, Sydney, Australia (1996) 86–95
35. Whale, G.: Identification of program similarity in large populations. The Computer Journal 33 (1990) 140–146
36. Choi, S.C., Scacchi, W.: Extracting and restructuring the design of large systems. IEEE Software 7 (1990) 66–71
37. Schwanke, R.W.: An intelligent for re-engineering software modularity. In: Proceedings of theThirteenthInternational Conference on Software Engineering, Austin, Texas, USA (1991) 83–92
38. Schwanke, R.W., Platoff, M.A.: Cross references are features. In: Proceedings of the 2nd International Workshop on Software Configuration Management. (1989) 86–95