When Functions Change Their Names: Automatic Detection of Origin Relationships

Sunghun Kim, Kai Pan, E. James Whitehead, Jr.
Dept. of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064 USA
{hunkim, pankai, ejw}@cs.ucsc.edu

Abstract

It is a common understanding that identifying the same entity such as module, file, and function between revisions is important for software evolution related analysis. Most software evolution researchers use entity names, such as file names and function names, as entity identifiers based on the assumption that each entity is uniquely identifiable by its name. Unfortunately names change over time. In this paper we propose an automated algorithm that identifies entity mapping at the function level across revisions even when an entity's name changes in the new revision. This algorithm is based on computing function similarities. We introduce eight similarity factors to determine if a function is renamed from a function. To find out which similarity factors are dominant, a significance analysis is performed on each factor. To validate our algorithm and for factor significance analysis, ten human judges manually identified renamed entities across revisions for two open source projects: Subversion and Apache2. Using the manually identified result set we trained weights for each similarity factor and measured the accuracy of our algorithm. We computed the accuracies among human judges. We found our algorithm's accuracy is better than the average accuracy among human judges. We also show that trained weights for similarity factors from one period in one project are reusable for other periods and/or other projects. Finally we combined all possible factor combinations and computed the accuracy of each combination. We found that adding more factors does not necessarily improve the accuracy of origin detection.

1. Introduction

When we analyze the evolution of procedural software, we frequently gather metrics about individual functions. The analysis process must associate each metric with its function and revision, usually accomplished by recording the function name and a revision identifier. Unfortunately, function names change, leading to breaks in the recorded sequence of metric values. Ideally we would like to track these function name changes, a process called *entity mapping*. If a particular entity maps across revisions, we say the successive revisions of the entity have an *origin relationship*.

We define an origin relationship as follows, based on [12].

Definition. Origin Relationship
Let ***r1*** and ***r2*** be consecutive revisions of a project history. A deleted function is one that disappears between ***r1*** and ***r2,*** while a new function is one that appears between ***r1*** and ***r2***. Let $\boldsymbol{D}=\{d_1, d_2, \dots d_n\}$ be a set of deleted functions, and $\boldsymbol{A}=\{a_1, a_2, \dots a_n\}$ a set of new functions. The origin relationship candidate set is the multiplication of the two sets: $\boldsymbol{D}\times\boldsymbol{A}$. The maximum number of origin relationships between ***r1*** and ***r2*** is the minimum of $|\boldsymbol{D}|$ and $|\boldsymbol{A}|$. A candidate set pair (d_x, a_y) has an origin relationship iff a_y is renamed and/or moved from d_x.

In this paper, we describe a technique for *automatic entity mapping* of C language software. The basic process is as follows:

1. We extract two consecutive software project revisions, r1 and r2, from the project's software configuration management repository.
2. We compute all possible mappings of deleted functions to added functions, yielding our *candidate set of origin relationships*.
3. For each function in the candidate set we gather a set of facts and metrics, including its name, incoming/outgoing calls, signature, lines of code, and a composite complexity metric. For each candidate set pair we also determine the text diff, and their similarity as computed by CCFinder [14] and MOSS [1]. This gives us the *raw data needed for computing similarity*.
4. We *compute an overall similarity between each candidate set pair*, using a weighted sum of the gathered facts, metrics, and similarities. Each pair now has a total similarity value that integrates a wide range of information about each pair.
5. We compare each *pair's total similarity to a threshold value;* exceeding the threshold indicates an origin relationship. We now know which functions have been renamed between r1 and r2.

Using this approach we are able to compute origin relationships with 87.8% accuracy for the Apache 2 web server [2] and 91.1% accuracy for the Subversion

configuration management system [4]. We asked 10 human judges to manually determine origin relationships for the same systems, resulting in 84.8% agreement across judges for Apache 2 project, and 89.1% agreement for Subversion. Hence our approach is capable of achieving human level accuracy at this task.

Our approach raises many questions:

How do you assess the accuracy of an automatic entity mapping algorithm? Typically, computed origin relationships are compared against known good ones; how do you find a known good set of origin pairs? We address this by asking human judges to manually determine origin pairs. This is more complex than it seems. Wallenstein et al. have shown that human judges never completely agree in their assessments [23].

How do you determine the weights used to combine individual facts, metrics, and similarity values? Intuitively we want to heavily weight individual factors that by themselves are good predictors of origin relationships.

Does combining a wide range of factors result in the most accurate determinination of origin relationships? If several of the factors are tightly correlated, it may be that some of the factors can be tossed out and still yield an accurate origin assessment. Since there is a computational expense to determining each factor, eliminating factors is a plus.

Does one set of factor weightings produce accurate results for multiple projects? Ideally our factor weightings will result in a similarity model that is applicable to many projects. Alternately, it might be the case that factor weightings are inherently project-specific.

We address these questions in the remainder of the paper: In Section 2, we discuss related work on origin analysis, renaming detection, and code clone detection methods. We describe each similarity factor in Section 3 and propose our automated origin relationship detection algorithm in Section 4. We describe our experimental process in Section 5 and validate our algorithm in Section 6. We briefly discuss applications for origin relationship detection in Section 7. We discuss the limitations of our analysis in Section 8, and conclude in Section 9.

2. Related Work

In this section we discuss previous work on renaming detection, origin analysis, and code clone detection algorithms.

2.1. Renaming Detection

Malpohl et al. proposed an algorithm that automatically detects renamed identifiers between pairs of program modules [18]. Their rename detection algorithm focuses on variable names rather than function names. Similar to our algorithm, they use context information such as the relation of the renamed variable to other variables, declaration similarity, implementation similarity, and reference similarity. However, the granularity of the renaming detection target is different. Here we are focusing on function name changes to map entities over multiple revisions, while Malpohl et al. focus on variable name changes to help merge and diff tools.

2.2. Origin Analysis

Godfrey et al. proposed an origin analysis algorithm to find renamed entities over revisions or releases, and our work is inspired by their origin analysis [12, 22]. They were the first to identify the importance of this problem for evolution analysis. Their origin analysis algorithm uses four matchers: name matcher, declaration matcher, metrics matcher, and call relation matcher. Based on the matchers, the algorithm provides a list of the top five origin candidates and requires a human to pick the best one of the five. We reused the matchers for our algorithm with a different similarity measurement.

Godfrey et al. provide expression matchers that enable users to decide combinations or threshold values for each matcher. However, they did not perform a significance analysis on each matcher to figure out which matchers are more or less important, and determine better combinations or threshold values. They also did not provide the accuracy of their algorithm, and their analysis results rely on only one human judge. Our work extends their work by focusing on accuracy comparison, significance analysis on similarity factors, and comparison of renaming behaviors of different projects and periods.

2.3. Code Clone Detection

The goal of code clone detection is to find similar or identical code fragments in program source code. The clone detection techniques can be used in software maintenance, software refactoring, and software evolution analysis. In our research, code clone detection is used to assist origin relationship detection across revisions, i.e. identifying a particular function in a prior revision. The code clone detection problem has been intensively studied in recent years, and many clone detection techniques have been introduced.

String matching is the most straightforward way to compare two text fragments. Linux/UNIX diff [10] is a tool usually used to compute the text difference of two files based on line string matching. RCS and CVS use diff to compute the delta between two revisions of a file. Diff calculates the longest common subsequence (LCS) of two text files at the line level. The LCS indicates the similarity of the two files. Some text transformations can be applied in diff before text matching is performed, such as removal of unnecessary white spaces, ignoring of tab expansion,

and ignoring blank lines. Duploc [8] is another clone detection approach based on string matching, in which diff's transformation technique is used, and a hash technique on text lines is leveraged to improve the performance of string matching.

While the clone detection techniques based on straightforward string matching are language independent, some techniques involve code transformation based on specific lexical or parsing analysis for a programming language in detecting code clones [3, 14]. CCFinder [14] transforms source code using different transformation rules for different languages (C, C++ and Java) and performs parameter replacement by replacing all identifier tokens with '$p' in the program. After that, CCFinder performs substring matching on the transformed token sequence for clone detection. The transformation and parameter replacement in CCFinder allows it to find code clones with identifier-renaming and code clones with similar syntax structures. The approach in CloneDr [3] involves syntactic analysis. For code clone detection, CloneDr parses program source code and generates its abstract syntax tree. Then it performs sub-tree matching in the abstract syntax tree to locate code clones.

Beagle [22] detects code clones using statistical data for procedures, including lines of code, code nesting, fan-out, number of global variables accessed and updated, number of parameters, number of local variables, and the number of input/output statements. Based on these statistics, five metrics of the measurement vector are computed for each procedure: S-Complexity, D-Complexity, Cyclomatic complexity, Albrecht, and Kafura [17]. The similarity of two procedures is indicated by the distance of the Bertillonage measurement vectors of these two procedures.

Instead of using software metrics, MOSS (Measure Of Software Similarity) [19] uses text sampling to characterize text content. The algorithm in MOSS is based on selecting fingerprints of k-grams, which are contiguous substrings of length k. Each k-gram is hashed and MOSS selects some of these hash values as a document's fingerprints. The more fingerprints shared by detection two documents, the more similar they are.

One of the big differences between code clone detection techniques and our work is that code clone detection techniques focus on identifying similar code areas in one revision without any boundaries of each entity while we are trying to find origins of functions based on function similarities across revisions.

3. Selecting Similarity Factors

Our algorithm computes the similarities of two-function pairs in the origin relationship candidate set to determine if they have an origin relationship. We introduce eight similarity factors which can be used to measure function similarities, shown in Table 1. Each of the factors will be discussed in the following sections. The requirements of the ideal factor are low computation complexity and high accuracy.

Table 1. Similarity Factors

Factor Name	Description
Function name (name)	Function name
Incoming call set (in)	Incoming calls
Outgoing call set (out)	Outgoing calls
Signature (sig)	Return type and arguments
Function body diff (body)	Text difference of function body
Complexity metrics (com)	Distance of complexity metrics value
MOSS (moss)	Similarity from MOSS
CCFinder (ccf)	Similarity from CCFinder

3.1. Function Name

When developers change function names, it is often the case that the new name is similar to the old one. The similarity of function names can be used to check the similarity of two functions. This factor's computation complexity is relatively low. Function name similarity can be computed using the intersection set count (ISC) or longest common sequence [13] count (LCSC) similarity metrics, described in Section 4.

3.2. Incoming Call Set

It is often the case that even if the function name and body change across revisions, the incoming call pattern remains the same. This makes the incoming call set a good similarity factor for determining the similarity between two functions. Computing the incoming call set requires parsing the project source code to statically determine the call relationships.

3.3. Outgoing call set

Similar to the incoming call set, a function's outgoing call set can also be used as a similarity measure.

3.4. Signature

Since most of a function's signature does not change frequently, signature patterns can be a similarity factor in deciding the similarity of two functions [16]. Getting and comparing signature patterns of functions is relatively simple, as a complete parser is not necessary to identify function signatures.

3.5. Text diff

Text diff can be used to check the similarity of two functions. In our text diff, we ignore white space differences between two lines. Note that text diff compares line by line. If there is a single character change in a line, it assumes the whole line is different.

3.6. Complexity metrics

Rather than comparing function bodies or names, comparing complexity metrics or the volume of two functions can indicate similarity of the two functions, acting somewhat as a "fingerprint" for the function [22].

We used five complexity metrics from 'Understand for C++' [20]: LOC, cyclomatic complexity, count of incoming calls, count of outgoing calls, and maximum nesting. Following [22], we compute distance of the two functions' five metrics values. The distance computation is simple, but computing the actual metrics is expensive.

3.7. CCFinder

CCFinder is a language-dependent code clone detection tool for C, C++, COBOL, Java and Lisp [14]. CCFinder detects code clones within a file and across files. In Burd's evaluation [6], CCFinder had the best recall rate of five code clone detection tools and fine clone detection precision. To compare the similarity of two functions, we store each function as a file and CCFinder reads two files to compute the similarity of the files. CCFinder has an option for the minimum token length. If the token length of a clone candidate is smaller than the minimum token length, CCFinder ignores this candidate. The default minimum token length is 30. Since there are many functions whose token length is less then 30, we compute the token length of two functions in advance and use the minimum of the two function's token length as CCFinder's minimum token length option.

3.8. MOSS

MOSS [1] is a web-based service for plagiarism detection. Generally, MOSS is language independent, but some language options can be turned on to increase the detection accuracy for programs of a specific language. Burd el al. [6] evaluated MOSS and four other code clone detection tools. It turned out that MOSS had the lowest recall rate in finding the code clones, but had a decent detection precision.

3.9. Other Possible Similarity Factors

There are other possible similarity factors we can use to determine origin relationships. One is Abstract Syntax Tree (AST) shape comparison to determine similarity. While we use a line-based diff in our algorithm, AST diff provides a software structure comparison, rather than detecting lexical differences. Another possible similarity factor is suggested by the observation that when a function name changes, sometimes the physical location of the function in the source file remains unchanged or changes minutely. Based on that assumption, we could use the physical location of a function such as line number or block number in a source code file.

4. Proposed Algorithms

First we need to compute all of the similarity factors for each two function-pair in the origin relationship candidate set. Our algorithm evaluates the presence of an origin relationship by combining values of similarity factors. Some similarity factors, such as MOSS and CCFinder, provide their own similarity algorithm. For the complexity metrics factor (Section 3.6), we compute the distance of the two vectors combining the five metrics values. We use the longest common sequence [13] count (LCSC) or the intersection set count (ISC) to compute similarities for other factors.

After computing similarities of each factor, we compute the total similarity using a weighted sum of each factor. The weights of each factor are also automatically computed.

4.1. ISC Similarity Metric

Factors such as function name, incoming calls, outgoing calls, and signatures can be expressed as a set. For example, a function name, '*hello_hunkim*' can be a set called an element set:

{h, e, l, l, o, _, h, u, n, k, i, m}.

If we have two element sets for a two-function pair in the origin relationship candidate set, the similarity of the two element sets can be computed using the ISC similarity metric. The basic task is to count of the intersection of the two sets. The ratio of the intersection count decides the similarity of the two sets. The ISC similarity metrics is shown in Equation 1.

Equation 1. ISC similarity measurement of two factors, A and B

$$ISC\ Similarity\ S_{AB} = \frac{\frac{|A \cap B|}{|A|} + \frac{|A \cap B|}{|B|}}{2}$$

ISC Similarity ranges from 0 to 1, where 1 indicates two factors are exactly the same. Note that the ISC similarity metrics doesn't consider the order of two sets. Suppose we have two element sets: $\boldsymbol{A}$={h, e, l, l, o} and $\boldsymbol{B}$={l, e, o, h, 1}. The ICS Similarity is 1 while the two function names, 'hello' and 'leohl' are different.

4.2. LCSC Similarity Metric

In some cases, we need to consider the order of the common part of two sets. For example, in the name element set, the order of each character is important in determining if two names are similar. The LCSC Similarity metric checks the order of elements in the sets. Equation 2 defines the LCSC Similarity metrics. LCSC Similarity also ranges from 0 to 1, where 1 indicates two factors are exactly the same.

Equation 2. LCSC Similarity measurement of two factors, A and B

$$LCSC\ Similarity\ \ S_{AB} = \frac{\frac{LCSC_{A-B}}{|A|} + \frac{LCSC_{A-B}}{|B|}}{2}$$

, while $LCSC_{A\text{-}B}$ is the LCSC of A and B, and |A| is the number of elements in the set A.

4.3. Vector Complexity Similarity Metric

We used the five function complexity metrics values listed in Section 3.6 as a similarity factor. To measure the similarity of each function, we used the distance between vectors of the five values, where each vector is defined as ***V***={v1, v2, v3, v4, v5}. If we have two vectors, V and U, we can compute the distance of two vectors. Since other similarities ranges from 0 to 1, we normalize each coordinate value to range from 0 to 1. The similarity of two vectors is defined in Equation 3.

Equation 3. Vector Complexity Similarity of two vectors, V and U

$$Normalize\ function\ \ N(v_n, u_n) = \begin{cases} 1 & if\ |v_n - u_n| > Ave_n \\ (v_n - u_n)/Ave & if\ |v_n - u_n| \le Ave_n \end{cases}$$

$$D = \sqrt{N(v_1,u_1)^2 + N(v_2,u_2)^2 + N(v_3,u_3)^2 + N(v_4,u_4)^2 + N(v_5,u_5)^2}$$

$$Vector\ Similarity\ \ S_{VU} = 1 - D/\sqrt{5}$$

, where Ave_n is average value of the n the elements in all vectors, V_1, V_2, …Vn.

The normalization function normalizes the range of the difference of two coordinates from 1 to 0 using average value of the nth elements in the vectors. We assume the difference of two coordinates is greater than the average value, the two coordinates are far enough to be 1. D indicates the distance of normalized coordinates, and ranges from 0 to $\sqrt{5}$, where 0 indicates two vectors are exactly the same and $\sqrt{5}$ indicates two vectors are different. Finally we compute vector complexity similarity which ranges from 0 to 1, where 1 indicates two vectors are exactly the same.

4.4. Total Similarity Metric

After computing each factor's similarity, the total similarity is computed using a weighted sum of each factor's similarity. We believe the significance of each factor is different, hence each factor is weighted accordingly. Ideally, we would like to select w_i that weigh the fact similarities to produce the best results. However, to judge what is best, we need a set of known good origin relationships to act as a correctness oracle, as described in Section 5. The total similarity equation is defined below.

Equation 4. Total similarity of functions A and B

$$Total\ similarity\ \ ts = \frac{\sum_{i \in F} w_i S_i}{\sum_{i \in F} w_i}$$

F is the set of facts from Table 1, S_i is a similarity measurement (from Equation 1 and Equation 2), and w_i is a weight of each factor's similarity measurements.

4.5. Threshold Value

We use a threshold value to determine an origin relationship. If the total similarity of two functions is greater than the threshold value, we assume there is an origin relationship. We have not yet presented how to select a reasonable threshold value, since it requires judgments against a set of known good origin relationships. In the following two sections we first address issues involved in developing a set of human judged correct origin relationships, and then proceed to use them to develop our weights and threshold values.

5. Experiment

To determine weights for each factor and threshold value we need an origin relationship set to compare. Common SCM systems such as CVS and Subversion do not record renaming of functions. We rely on human judges to determine a known good set of origin relationship pairs. 10 human judges manually examined origin relation candidates and identified origin relationships if present. If more than 70% of the judges agreed on an origin relationship identification, we assume it is a valid origin relationship. Based on the identified origin relationship set from human judges, we decide the weights of similarity factors and the threshold values.

5.1. Making the Oracle Set

In this section we describe the process of making the known good origin relationship set called OracleSet.

First, we chose sample projects and revision periods to perform manual identification. We used two open source projects: the Subversion project from revisions 1–3500, representing about 20% of the total Subversion project history, and the Apache2 project from revision 1-2000, about 20% of the Apache2 project history.

Ten human judges used a graphical user interface that shows origin relationship candidates (two-function pair) with function body and the corresponding revision log. The change log is very useful for human judges. In previous origin analysis related research, change log data is not used for the manual decision process [11, 12, 18, 22]. However, change logs are widely used to map bug numbers to code changes [9, 21]. Change logs sometimes explicitly describe function renaming. For example, the log message for Subversion revision 172 explicitly states

that function '*fulltext_string_key*' is renamed from '*string_key*'.

What kinds of words in the change log indicate origin relationships between functions? We manually observed change logs of 3550 revisions of the Subversion project and 2000 revisions of Apache2 project and found the following origin-relationship-sensitive words. Those words are used to identify origin relationships manually.

Table 2. Origin relationship sensitive words. ++ indicates the words have strong indication of origin relationship. – indicates the words have negative indication of origin relationship.

Indication	Words
++	rename, move, replace, revamp
+	Update, merge, split
-	Remove, delete, add, new

We created guidelines for human judges. The guidelines are based on the understanding of the origin relation definition. Human judges also use change logs. If any of the strong positive (++) origin relationship indication words are used in the change log to describe a function in origin relationship candidates, the functions have an origin relationship. If any of the negative (-) origin relationship indication words are used to describe a function, the function is likely either a new function or a deleted function. If there is no description of a function in the change log, human judges decide the origin relationship by examining the semantic behaviors of the two functions. If the two functions are doing exactly the same thing, we assume the two functions have an origin relationship. These guidelines are summarized in Figure 1.

1. Understand the origin relationship Definition.
2. Examine the change log
   A. A function has an origin relationship if any of the strong positive (++) words or their synonyms are used to describe functions.
   B. A function has no origin relationship if any of negative (-) words or their synonyms are used to describe the change to the function
3. If no decisions in the previous step, examine the old and new function bodies
   A. They have an origin relationship if their semantic behaviors are the same.

Figure 1. Origin relationship manual identification guidelines

The third item in the guideline is subjective, but we trust the experience and capabilities of our judges. Based on the guidelines, 10 judges carefully reviewed the revision logs and function pairs, and determined if each function pair has an origin relationship. The judging process is similar to that used for finding code clone detection by human judges in [23]. All human judges have strong C programming experience, and are graduate students in the computer science department. Two of them have experience of extending the Apache2 server, which is similar to the examined Subversion project.

Even though origin relationships are potentially not one-to-one, we did not consider function merging/splitting cases to simplify the algorithm and validation process. Upon inspection of the Subversion history log (from revision 1 to 3500), we found only one function merge and two function split, which we felt comfortable ignoring. When the judges encountered a merge/split situation, they chose the most similar function among the merged or split functions. After gathering results from the judges we compared each human's judgment, as shown in Table 3. As noted in origin relation Definition, the maximum number of origin relationship for Subversion is 2144 and 1980 for Apache2. For example if there are 2 deleted functions and 15 added functions between revisions, the maximum number of origin relationship is 2 which is the minimum number of between numbers of deleted functions and added function.

On average, judges took 3 hours and 40 minutes to identify 634 origin relationships for Subversion and 1 hour and 15 minutes to identify 99 origin relationships for Apache2. We add an origin relationship to the OracleSet if more than 70% of the human judges agree on origin relationship.

Table 3. Comparing results from 10 human judges

Judge ID #	Apache2		Subversion	
	number of origins	Time	number of origins	Time
1	92	45m	643	3h 30m
2	86	1h	628	4h
3	109	1h 20m	673	4h 30m
4	104	1h	640	2h 45m
5	88	1h	470	4h 20m
6	114	1h 30m	659	5h
7	104	50m	671	3h 20m
8	99	1h	647	3h
9	98	1h	643	3h 30
10	91	1h 20m	662	3h 40m

Equation 5. Agreement/Accuracy from A and B origin sets from two judges

$$Agreement\,(percentage) = \frac{|A \cap B|}{(|A| + |B|)/2} * 100$$

We define the percentage of agreement/accuracy in Equation 5 to compare accuracy of any two origin relation sets. Using Equation 5, we computed inter-rater agreement to determine the quality of the human judgments, reported in Table 4. For example, in Subversion project judges #1 and #2 agreed on 579 (91%)

origin relationships. Consistent with the observations of Wallenstein et al.[23], we found low agreement between the human judges. The comparatively high agreement of the Subversion project is due to its change log which indicates function remaining. Since the Apache2 project does not provide such change logs, human judges have to identify origin relationships based on the semantics of function behavior. On average there is 84.8% inter-rater agreement for Apache2 and 89.1% for Subversion. To make the OracleSet we assume there is an origin relationship if 70% or more judges agreed on the origin relationship, bringing the total number of origin relationships in OracleSet to 626 for Subversion and 91 for Apache2.

Table 4. Each judge agreement with judge #1 for SVN and Apache2 projects.

	#2	#3	#4	#5	#6	#7	#8	#9	#10
A2	82	86	87	86	84	88	86	84	80
SVN	91	92	93	71	91	92	89	93	90

5.2. Deciding Factor Weights

We compare each similarity factor's output and the OracleSet to decide weights for each factor. Table 5 shows an illustrative sample of six origin candidates, and the respective similarity of selected factors, and the OracleSet (see Table 1 for factor name abbreviations). Our intuition is that if a similarity factor's output is close to the OracleSet, we assume the factor is important. There are several ways to decide if two sets of values are close. We could determine the error between the two sets or compute correlation using Pearson's correlation equation [7].

Table 5. Example similarities of factors and OracleSet.

Candidates	Name	Sig	CcF	Moss	OracleSet
1	0.8	0.7	0.9	0.7	1
2	0.2	0.3	0.0	0.0	0
3	0.3	0.2	0.4	0.3	0
4	0.2	0.5	0.0	0.0	0
5	0.7	0.9	0.8	0.9	1
6	0.4	0.8	1.0	1.0	1

We use accuracy (defined in Equation 5) between a single similarity factor and the OracleSet to decide weights of each factor. To obtain the best similarity of a similarity factor, we vary the threshold value for the factor from 0.1 to 0.9 by steps of 0.01.

For example, to compute the significance of the name factor we vary the threshold value from 0.1 to 0.99. During these processes, suppose the threshold for the name factor is currently 0.5. If the output value of the name factor is greater than 0.5, it predicts it has an origin relationship, so it is set to 1. The prediction set of the name factor for the six candidates in Table 5 is {1, 0, 0, 0, 1, 0}. By comparing the prediction set and the OracleSet, we compute the accuracy of name factor which is 0.8 (2/((2+3)/2)*100).

The accuracy of each similarity factor indicates the significance of each factor. The accuracies of each factor are shown in Table 6 with corresponding threshold values. Note that the threshold values yielding the best accuracy for each factor may vary. The $_{ISC}$ sub text indicates we used ISC similarity metrics (Equation 1) to compute the factor similarity. The default is using LCSC similarity metrics (Equation 2) excluding complexity metrics, MOSS, and CCFinder. The accuracy of each similarity factor indicates the significance. The function body diff, outgoing call set, and function name factors are the most significant factors for deciding origin relationships. In contrast, complexity metrics and CCFinder are less significant factors.

The weights of each similarity factor are decided by the accuracy of the factor. The basic idea is that if a factor leads to increased accuracy, the factor contributes more to the total similarity. Note that all similarity factors using ISC and LCSC similarity equations in Equation 1 and Equation 2 are similar. We excluded all ISC similarity in the rest of experiments.

Table 6. Accuracy and weights of each factor of two projects, Apache2 and SVN. Values in parenthesis indicate the threshold value of each similarity factor.

Similarity Factor	Apache2 Weight	Subversion Weight
Function name	78.1 (0.72)	90.6 (0.68)
Function name$_{ISC}$	77.4 (0.74)	86.0 (0.68)
Incoming call set	71.4 (0.88)	81.9 (0.32)
Incoming call set$_{ISC}$	71.4 (0.88)	81.9 (0.32)
Outgoing call set	86.5 (0.75)	89.5 (0.74)
Outgoing call set$_{ISC}$	85.3 (0.69)	89.5 (0.74)
Signature	76.4 (0.50)	87.4 (0.59)
Signature$_{ISC}$	76.4 (0.50)	87.2 (0.59)
Function body diff	86.2 (0.65)	95.1 (0.60)
Complexity metrics	46.3 (0.84)	85.7 (0.84)
MOSS	79.5 (0.66)	83.3 (0.63)
CCFinder	61.3 (0.24)	75.1 (0.19)

5.3. Deciding Threshold Value

To find the best threshold value, we varied the threshold value from 0.1 to 0.99 (steps of 0.01) and observed agreements between OracleSet and the result using Equation 5 (the resulting set of predicted origin relationship pairs is called PredSet). We found the best threshold value to be 0.6 for Apache2 and 0.52 for Subversion.

6. Validation

In this section we validate our algorithm using the OracleSet described in Section 5. First we compute the

weights and threshold values for two projects, Subversion and Apache2. We then compute our algorithm's accuracy. Second, we divide two project revisions into three periods: train, test A, and test B periods. We train weights and threshold value in the train period and apply trained weights and threshold to the test A and test B periods. Finally, we combine all possible similarity factors and compute accuracy of combined factors see if adding more factors leads to improved results.

6.1. Algorithm Accuracy

We showed the process of deciding the weights of each factors and threshold values using OracleSet. We computed the total similarity of origin candidates using weights in Table 6. Using the threshold value from Section 5.3, we determined if the candidates have an origin relationship; all candidates with origin relationships are placed in a set called PredSet. We compared the PredSet and OracleSet and computed the accuracy of the origin relationships of the two projects, as shown in Table 7.

Table 7. PredSet accuracy of the two projects.

Projects	Apache2	SVN
Accuracy	87.8	91.1

Note that the accuracy of PredSet is better than the average accuracy among human judges shown in Table 4 (84.8% for Apache2 and 89.1% for Subversion)

6.2. Applying Trained Weights and Threshold

We wondered if the trained weights and threshold value from one project and period are reusable for the other project periods. To determine this we first divided two project periods into three periods, as shown in Figure 2.

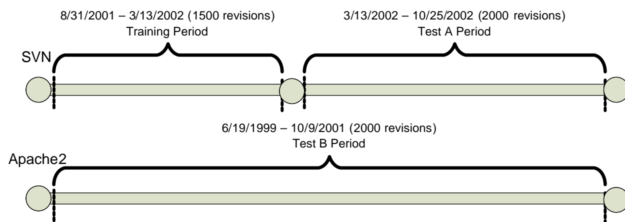

Figure 2. Training, test A, and test B periods

Then we computed weights and threshold using the data in the training period following the process described in Section 5, with results in Table 8.

The best threshold value for the trained period is 0.47. Using the weights and threshold value, we applied the algorithm to test A and test B periods. The accuracy of each period is shown in Table 9. The accuracy of the training period is lower than that of the test A. One of the reasons is that the first couple of revisions of Subversion project have renaming changes with lots of function inside changing.

Table 8. Trained weights using data in the training period

Factor	Accuracy/Weight from the train period
Function name	86.7
Incoming call set	78.3
Outgoing call set	82.3
Signature	80.5
Function body diff	90.2
Complexity metrics	82.1
MOSS	73.0
CCFinder	67.4

Table 9. Accuracy of each period using the trained weights and threshold value from the training period.

Period	Training	Test A	Test B
Accuracy	83.7	96.8	86.5

The accuracies using trained weights and threshold values in the train period is similar to the accuracies using their own trained weights and threshold value (See Table 7). The results strongly suggest that trained values from one period are reusable for other periods and projects. While the weights hold well for these two projects, we ideally would like to test against other projects to see if these high accuracies are consistent.

6.3. Using More/Less Factors

In Section 4 we identified many possible factors to decide origin relationships. We wanted to determine whether or not adding more factors would lead to improved accuracy.

Since it is not feasible to get models of each factor, we tested the combinations of all possible factors using the brute force method. For example we combine all possible combinations of factors and computed the accuracy of each combination using different threshold values (from 0.1 to 0.99). We applied the method for the Apache2 project. The top 10 accuracy combinations and total similarity are shown in Table 10. Note that using all factors defined in Equation 4 does not have the best accuracy (ranked 50).

The top accuracy is much higher than total similarity. Note that the top accuracy combination includes the '*com factor*', one of the insignificance factors. We need further research to find general factor combination for the best accuracy, but this result strongly recommends using carefully selected factor combinations leads to improved accuracy.

Table 10. Top accuracy combinations of the Apache2 project.

rank	Factor combination	Accuracy
1	body, name, sig, com	91.0
2	body, ccf, name, sig, com	90.5
3	body, name, out, sig	90.4
4	body, ccf, in, name, sig, com	90.3
5	body, name, out	90.2
6	body, name	90.2
7	body, name, out, sig, com	90.1
8	name, out, sig, com	90.0
9	body, ccf, in, name, out, sig, com	90.0
10	body, ccf, in, name	89.9
50	Total similarity	87.8

7. Applications of Origin Relationship Detection

Any software evolution analysis at the function level can apply our algorithm to provide more accurate analysis results. We highlight some applications including: instability analysis, signature change analysis, and code clone detection analysis.

In prior research, we performed evolution analysis of function signature by observing signature changes of each function across a project history [16]. To follow the same function throughout the revision history we used function names as identifiers. But when a function name changed, we lost the previous signature change histories of the function. Origin relationship detection helps to link the same function even after function names change.

Instability analysis [5] is a technique to identify dependence-related maintenance intensive code regions by observing project history. Various metrics can be used to indicate such regions including number of changes and number of authors of an entity. If we were to perform instability analysis at the function level without origin relationship detection, when unstable functions changed their name, we would lose the ability to track the instability across the name change.

For code clone evolution research such as [15], identifying origin relationships is also important. To see the evolution of code clone areas, it is necessary map entities over revisions. The physical location of code clones is likely to change over revisions. Using function level code clone areas with origin relationship detection can be used to observe code clone evolution.

8. Threats to Validity

The results presented in this paper are based on selected revision periods of two open source projects. Other open source or commercial software projects may not have the same properties we presented here. We only analyzed projects written in the C programming language; software written in other programming languages may have different origin relationship patterns. Some open source projects have revisions that cannot be compiled or contain syntactically invalid source code. In these cases, we had to guess the function boundary or skip the invalid parts of the code. We ignored '#ifdef' statements because we cannot determine the real definition value. Ignoring '#ifdef' caused us to add some extra signatures, which will not be compiled in the real program. Our validations are based on an OracleSet derived from the opinions of 10 human judges. The decisions of human judges are subjective and our threshold (70%) for inter-rater agreement many not be causative enough.

9. Conclusions and Future Work

We proposed an automatic algorithm to find origin relationships between revisions using various similarity factors including function name, incoming call set, outgoing call set, signature, body, MOSS, and CCFinder. 10 human judges manually identified origin relationships. If more than 70% of the judges agree on an origin relationship, we assume it is a valid origin relationship. Based on human judgments, we made an OracleSet to validate our algorithm. We performed significance analysis of each factor and used the significance indication as weights for each factor. Based on the significance of factors we identified dominant similarity factors such as text diff, outgoing call set, and function name. In contrast, the complexity metrics and CCFinder similarity factors are insignificant.

We validated the accuracy of the algorithm using two projects: Subversion and Apache2. We found our algorithm has 87.8% and 91.1% accuracy for Apache2 and Subversion respectively. Compared to the average accuracy among human judges (89.1% for Subversion and 84.8% for Apache2), our algorithm accuracy is better than the average accuracy among human judges. Second we validate how the trained weights of similarity factors and the threshold value can be reused for other periods and projects. We conclude that the trained weights and threshold value from one period are reusable in others. Finally, we validate whether adding more factors leads to improved accuracy using brute force combinations of similarity factors. We found that using more factors does not improve the accuracy. Careful selection of factor combinations can improve accuracy. It is also possible to reduce the computational cost of determining origin pairs without appreciable loss of accuracy. Finding general factor combinations remains as future work.

For future work we need to investigate some machine learning techniques such as regression training to find models and weights of factors.

We applied our algorithm to projects written in C. Some programming languages such as Java, and C++ support function name overriding. One revision of source

code can have the same function (method) names. If some of the same function names evolve by changing names or signatures, it is challenging to identify the same entities between revisions. Similarity factors we listed here may not work very well in programming languages which support function name overriding. Applying our algorithm and finding suitable factors for such languages remains future work.

10. Acknowledgements

Thanks go to our ten human judges. We thank Mark Slater and the anonymous reviewers for their valuable feedback on this paper. We especially thank the Storage Systems Research Center at UCSC for allowing the use of their cluster for our research. Work on this project is supported by NSF Grant CCR-01234603, and a Cooperative Agreement with NASA Ames Research Center.

11. References

[1] A. Aiken, "A System for Detecting Software Plagiarism," 2005 http://www.cs.berkeley.edu/~aiken/moss.html

[2] Apache Foundation, "The Apache HTTPD Server Project," 2003 http://httpd.apache.org/

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. the International Conference on Software Maintenance*, Bethesda, Maryland, 1998, pp. 368.

[4] B. Behlendorf, C. M. Pilato, G. Stein, K. Fogel, K. Hancock, and B. Collins-Sussman, "Subversion Project Homepage," 2005 http://subversion.tigris.org/

[5] J. Bevan and J. E. James Whitehead, "Identification of Software Instabilities," *Proc. 2003 Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, Canada., Nov. 13-16, 2003, pp. 134-145.

[6] E. Burd and J. Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance," *Proc. the Second IEEE Intl. Workshop on Source Code Analysis and Manipulation (SCAM'02)*, 2002, pp. 36-43.

[7] R. E. Courtney and D. A. Gustafson, "Shotgun Correlations in Software Measures," *Software Engineering Journal*, vol. 8, no. 1, pp. 5-13, 1992.

[8] S. h. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *Proc. the IEEE International Conference on Software Maintenance*, 1999, pp. 109.

[9] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," *Proc. 2003 Int'l Conference on Software Maintenance (ICSM'03)*, September, 2003, pp. 23-32.

[10] GNU, "Diffutils Project Home Page," 2003 http://www.gnu.org/software/diffutils/diffutils.html

[11] M. Godfrey and Q. Tu, "Tracking Structural Evolution using Origin Analysis," *Proc. the International Workshop on Principles of Software Evolution*, Orlando, Florida, 2002, pp. 117 - 119.

[12] M. W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Trans. on Software Engineering*, vol. 31, no. 2, pp. 166- 181, 2005.

[13] D. S. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *Journal of the ACM (JACM)*, vol. 24, no. 4, pp. 664 - 675, 1977.

[14] 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*, vol. 28, no. 7, pp. 654 - 670, 2002.

[15] M. Kim and D. Notkin, "Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones," *Proc. Int'l Workshop on Mining Software Repositories (MSR 2005)*, Saint Louis, Missouri, USA, May 17, 2005, pp. 17-21.

[16] S. Kim, E. J. Whitehead, Jr., and J. Bevan, "Analysis of Signature Change Patterns," *Proc. Int'l Workshop on Mining Software Repositories (MSR 2005)*, Saint Louis, Missouri, USA, May 17, 2005, pp. 64-68.

[17] K. Kontogiannis, "Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics," *Proc. the Fourth Working Conference on Reverse Engineering (WCRE '97)*, 1997, pp. 44.

[18] G. Malpohl, J. J. Hunt, and W. F. Tichy, "Renaming Detection," *Proc. The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, Grenoble, France, September 11 - 15, 2000, pp. 73.

[19] S. Schleimer, D. Wilderson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," *Proc. the ACM SIGMOD International Conference on Management of Data*, San Diego, California, 2003, pp. 76 - 85.

[20] Scientific Toolworks, "Maintenance, Understanding, Metrics and Documentation Tools for Ada, C, C++, Java, and FORTRAN," 2005 http://www.scitools.com/

[21] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *Proc. Int'l Workshop on Mining Software Repositories (MSR 2005)*, Saint Louis, Missouri, USA, May 17, 2005, pp. 24-28.

[22] Q. Tu and M. W. Godfrey, "An Integrated Approach for Studying Architectural Evolution," *Proc. Intl. Workshop on Program Comprehension (IWPC 2002)*, Paris, June, 2002, pp. 127.

[23] A. Wallenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems Creating Task-relevant Clone Detection Reference Data," *Proc. 2003 Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, Canada., Nov. 13-16, 2003, pp. 285.