Properties of Signature Change Patterns

Sunghun Kim, E. James Whitehead, Jr., Jennifer Bevan
Dept. of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95060 USA
{hunkim, ejw, jbevan}@cs.ucsc.edu

Abstract

Understanding function signature change properties and evolution patterns is important for researchers concerned with alleviating signature change impacts, understanding software evolution, and predicting future evolution patterns. We provide detailed signature change properties by analyzing seven software project histories to reveal multiple properties of signature changes, including their kind, frequency, correlation with other changes, number of parameter changes, and evolution patterns of signature change kinds. We show that signature changes can be used as measurement aid for software evolution analysis.

1. Introduction

In procedural languages like C, the "function" is the primary functional abstraction encapsulating program behavior. In large, evolving software systems, a key question concerns how these functional abstractions fare over time. Designing to accommodate change is a central design principle in software engineering, one that raises the question of how functional abstractions actually do change. Possible changes include function body modification, variable renaming, moving functions from one file to another, and function signature changes [14]. In this paper we perform a detailed examination of the evolutionary behavior of the external interface to functional abstractions: the function signature.

Examining changes to the external interface provides insight into the evolution of the abstraction itself. If we know the most frequent kinds of signature changes, we better understand the common evolutionary stresses affecting the abstraction. Total numbers of signature changes, and correlations between signature changes and function body changes, give insight into the stability of a system's modular decomposition, and hence the stability of the abstraction over time. Furthermore, detailed observation of signature changes might reveal patterns to them, and hence patterns of change to functional abstractions.

Other researchers have observed code changes, though none have examined signature changes in the same level of detail as we do in this paper. Kung et al. identified types of code changes [14] and Counsell et al. discussed the trends of changes in Java code [6]. Both studies identified large granularity change types, such as method body changes, method addition, method deletion and whether the signature changed (but with no differentiation of the different types of signature changes). Their categorization of changes is useful for understanding software changes as an overview, but is insufficiently detailed to develop a strong understanding of functional abstraction evolution.

To provide detailed signature change properties we focus on fine-grained changes in function signatures, categorizing them based on whether they increase, decrease, or do not modify the data flow between caller and callee. We show the properties of function signature change patterns by answering the following research questions: How often do signatures change? What are the common signature change kinds? How often does each kind appear? Does each project have unique signature change patterns? Are there particular sequences of signature change kind?

Understanding function signature change patterns is important to researchers concerned with alleviating signature change impacts, understanding software evolution, and understanding evolution pattern change sequences. We found correlations between signature changes and other types of changes used in such research, such as LOC and function body changes. This result suggests that signature change patterns can enhance other evolution analyses, such as finding hot spots, unstable areas [4], or code decay [8].

Our analysis dataset is seven prominent open source project histories: Apache 1.3 HTTP server, Apache 2.0 HTTP server, Apache Portable Runtime Library, Apache Portable Runtime Library Utility, Subversion, CVS, and GCC (see Table 1). These seven projects are written in the C programming language.

Table 1. Analyzed open source projects. LOC indicates number of lines of .h and .c source files, including comments of the latest transaction. Period shows the project history period for projects for which we directly accessed the SCM repository. # of Txns indicates the number of transactions we extracted.

Project	Software type	LOC	SCM	Period	# of Txns
Apache HTTP 1.3 (A 1.3)	HTTP server	116,398	Subversion	Jan 1996 ~ Mar 2005	7,747
Apache HTTP 2.0 (A 2)	HTTP server modules	104,417	CVS	Jul 1999 ~ Aug 2003	3,877
Apache Portable Runtime (APR)	Portable C library	72,630	Subversion	Jan 1999 ~ Feb 2005	5,990
APR utility (APU)	Portable C library utility	33,660	Subversion	Jul 1999 ~ Feb 2005	1,353
CVS	SCM software	62,415	CVS	Dec 1994 ~ Sep 2003	2,873
GCC	C/C++ compiler	298,236	CVS	Nov 1988 ~ Feb 1993	3,012
Subversion (SVN)	SCM software	185,006	Subversion	Aug 2001 ~ Mar 2005	6,029

For our analysis, we used Kenyon, a data extraction, preprocessing, and storage backend designed to facilitate software evolution research [5]. Using Kenyon, we checked out all transactions of source code from each project, and then extracted facts needed for our analysis, including function signatures, call dependencies, and LOC changes. We grouped signatures by function name and observed patterns of signature changes including signature change frequency distributions (Section 4.1, 4.2, and 4.3), observable signature change patterns (Section 4.4 and 4.5), and correlations with other project aspects (Section 4.6 and 4.7). Based on the observations we introduce nine signature change properties in Section 5.

When function names change, file-based SCM systems like CVS and Subversion do not record the linkage between old and renamed functions. A function name change is only observed as a function deletion and a new function addition. This is a potential source of error in signature change evolution analysis, since it artificially splits the transaction history of a functional abstraction at the point where its name changes. The origin of a renamed function is the same logical function under its old name. To find the origins of functions, Godfrey et al. proposed a human-assisted origin analysis algorithm using dependencies and entity analysis [10, 16]. In a similar vein, we implemented a completely automated origin analysis algorithm to find origins of new functions [11]. We reported preliminary signature change properties in [12], and this paper substantially extends the previous work by applying origin analysis to improve the quality of results, analyzing correlations between signature changes and other changes, providing probability graphs of each change kind, observing the number of parameter changes, and analyzing bug introduction rates of signature changes.

The remaining sections of the paper are as follows. In section 2, we define and organize signature change kinds. We discuss our experiment approach in section 3. Section 4 provides answers to our research questions by performing several detailed signature change analyses. In section 5, we list signature change properties we uncovered during the analysis. We discuss the limitations and applicability of our analysis in section 6 and 7, and conclude in section 8.

2. Signature Change Kinds

Before presenting our results, we describe our fine-grain taxonomy of signature change kinds. Signature change kinds are informally defined in Table 2; more formal definitions of signature changes are in [12].

To define the major categories of our taxonomy, we use a data flow model between a function and a client. A client calls a function by passing arguments (**Arg**) and expecting returns (**R**) as shown in Figure 1. The total data flow across the interface is the union of **Arg** and **R**, defined in Definition 1. We note that data flows can also occur by passing data in global variables.

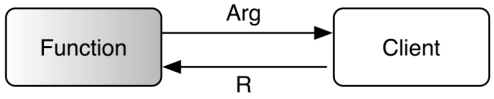

Figure 1. Model of data flow across a function interface.

Definition 1 (Interface Data Flow)

Parameter **Param** = {modifier, type, name, array/pointer, order}
Argument **Arg** = a set of zero or more **Param**
Return parameter **R** = {modifier, type, array/pointer}

DF = **Arg** $\cup$ **R**

Data flow invariant = $|\mathbf{DF}_{old}| = |\mathbf{DF}_{new}|$
Data flow increasing = $|\mathbf{DF}_{old}| < |\mathbf{DF}_{new}|$
Data flow decreasing = $|\mathbf{DF}_{old}| > |\mathbf{DF}_{new}|$

Broadly, when parameters or return values are added, there is an increase in the amount of data flowing across the interface between caller and callee, while parameter deletion or removal of return values results in reduction of data flow. Modifier changes or parameter name changes have no impact on the data flow. The fine-grain taxonomy of signature change kinds based on the data flow model is shown in Table 2 and Table 3.

Table 2. Definitions of signature change kinds.

Change kind	Description
Parameter addition (A)	A parameter is added
Parameter deletion (D)	A parameter is deleted
Parameter ordering change only (OO)	Parameter ordering change without addition or deletion changes
Ordering change by addition (OA)	Parameter ordering change caused by parameter addition
Ordering change by deletion (OD)	Parameter ordering change caused by parameter deletion
Return type change (R)	Return type changes excluding RA and RD
Return type addition (RA)	Return type change from the void data type to other data types
Return type deletion (RD)	Return type change from other data types to the void data type
Primitive type change (T)	A primitive type of a parameter changed
Complex type name change (C)	The name of a complex data type changed
Complex type inner variable addition (CA)	One of members of a complex data type added
Complex type inner variable deletion (CD)	One of members of a complex data type deleted
Function name change (MN)	Function name changed
Parameter name change (N)	Parameter name changed
Parameter modifier change (M)	Modifier of a parameter changed
Concept merge/splitting change (CM/CS)	One or more parameters' concept merged to a parameter or a parameter's concept split to two more parameters
Array/Pointer operation change (P)	Dimensions of pointer or array parameter change

3. Investigation Approach

3.1 Extraction, Refinement, Analysis

Before presenting results from our signature change observations, we briefly describe our analysis process. We used a three phase-process to analyze signature changes using the Kenyon system. The first phase involves checking out all transactions and extracting raw facts. We used Kenyon for this phase. Kenyon checks out all transactions from a SCM repository and invokes a fact extractor that we implemented to extract raw facts such as function signatures, deltas between two transactions, and added/deleted function information. The extracted raw data is stored in a DBMS.

For the projects we analyzed, the transaction history was stored using either the CVS or the Subversion SCM system. An important issue in software evolution research is the extraction of logical transactions from the SCM repository. Since Subversion assigns a transaction number per commit, there is no need to recover transactions for Subversion-managed projects [2]. CVS does not keep the original transaction information, usually requiring a process of transaction recovery [19]. Kenyon provides CVS transaction recovery via a slicing time window algorithm [5].

Table 3. Taxonomy of signature change kinds. A * indicates the item is only manually identifiable and hence is not reported on in this paper.

Data flow invariant	Function name change (MN) Parameter ordering change only (OO) Parameter name change (N) Parameter modifier change (M) *Concept merge/splitting change (CM/CS) Array/Pointer operation change (P) Return type change (R) Primitive type change (T) Complex type name change (C)
Data flow increasing	Parameter addition (A) Ordering change by addition (OA) Return type addition (RA) *Complex type inner variable addition (CA)
Data flow decreasing	Parameter deletion (D) Ordering change by deletion (OD) Return type deletion (RD) *Complex type inner variable deletion (CD)

The second phase refines the extracted raw data in the DBMS. For example, we find origin relationships using raw data such as added/deleted function information and signatures. Section 3.2 below provides additional detail on how we computed origin relationships. The refined data is also stored in the DBMS. The final phase analyzes signature change patterns using the raw data and refined data in the DBMS. The signature data is grouped by function name and applied origin relationships. The grouped signatures are ordered by transaction and stored in a signature change history file.

We manually observed the signature change history file to identify common signature changes. After analyzing the signature change history files from various open source projects, we found the common change kinds shown in Table 2. While most of the change patterns can be automatically identified by a static software analysis, some change kinds, such as concept merging/splitting changes are not automatically identifiable, requiring knowledge of the project and parameter concepts.

Based on these manually identified signature change kinds, we implemented an automatic identifier that reads a signature change history file, and annotates the file based on the identified kinds. After the signature change history file has been annotated, we calculate the frequency of each change kind over a project's history.

We also examine the sequence of signature change kinds of a given function to identify any common patterns in the signature evolution. The results from these analyses are presented in Section 4.

3.2 Origin Analysis

It is natural to use a function name as an identifier to track function signature changes. For example we can observe signature changes of the '*foo*' function over transactions by finding the function name '*foo*' in each transaction's source code. However, function names do change, and existing SCM systems do not track this as a logical change. Ideally, we would like signature changes to be attributed to their logical function, even if the function's name changes over time. In this case, there is an origin relation between the deleted function and the new function [10]. In previous work, we developed an automatic origin analysis algorithm that compares two functions for identity using a total similarity measure [11]. To simplify our signature change analysis, we compute similarities using four facts: function name, parameters, incoming calls, and outgoing calls. Detailed algorithms and similarity metrics are described in [11]. We use the automated origin analysis results for our signature change analysis.

4. Observations of Signature Changes

In this section we discuss the following detailed properties of function signature changes:

Frequency distributions: We characterize the distribution of various signature change types over a project history (Section 4.1), the number of changes per function (Section 4.2), and the number of parameter changes (Section 4.3).

Observable patterns: We try to observe if there are any project-specific patterns (Section 4.4) or noticeable patterns in the sequence of change types (Section 4.5).

Correlations with other project aspects: We observed correlation between signature changes and other changes such as function body changes (Section 4.6) and bug-introducing changes (Section 4.7).

Based on these observations, we introduce nine signature change properties in Section 5.

4.1 Frequency of Change Kinds

After identifying signature change kinds, we computed the frequencies of each kind. Figure 2 shows the signature change kind frequencies of each project as percentages for each change kind. The percentage is calculated by taking the number of observations of a particular change kind, and dividing it by the total number of signature changes observed for that project. For example, in the SVN project, we observed 2347 parameter additions, and 9062 total signature changes, resulting in a frequency percentage of 26%.

Note that a single observed signature change event could match more than one change kind. For example, a single signature change could potentially involve parameter addition, parameter deletion, and ordering changes. Figure 2 shows that the most common change kinds are complex type name changes (average 27.25%), parameter addition (average 24.8%), ordering only changes (average 15.7%), and parameter deletion (average 14%). The array/pointer and primitive type changes are relatively uncommon change kinds. Note that the number of parameter addition changes is almost double the number of parameter deletion changes; note also that if multiple parameters are added or deleted in a single change, only one change count was recorded. Most complex data type name changes are due to refactoring of a complex data type. For example, in APR the complex type name, '*ap_pool_t*' changed to '*apr_pool_t*' (an 'r' was added to 'ap' to make 'apr'). Many other complex data type names for this project had a similar change from 'ap' to 'apr,' indicating a global renaming event.

The number of return type and function name changes of selected projects is shown in Table 5. In this table, the count of total parameter changes is taken from Table 4. The ratios for return type changes and function name changes are similar. For example, return type changes occur at 5~10% of the frequency of total parameter changes.

Table 4. Counts of each change kind for the seven open source projects

Project	Parameter Name change	Only ordering change	Addition	Deletion	Modifier change	Array/ Pointer	Complex type name change	Primitive type change	Total changes
A 1.3	72	67	262	117	207	4	64	25	818
A 2	183	274	572	462	272	33	992	8	2796
APR	263	426	708	261	169	52	2188	113	4180
APU	53	61	99	50	10	1	271	11	556
CVS	37	244	404	299	117	28	10	4	1143
GCC	19	47	114	59	49	1	158	7	454
SVN	354	3510	2347	1137	694	27	973	20	9062

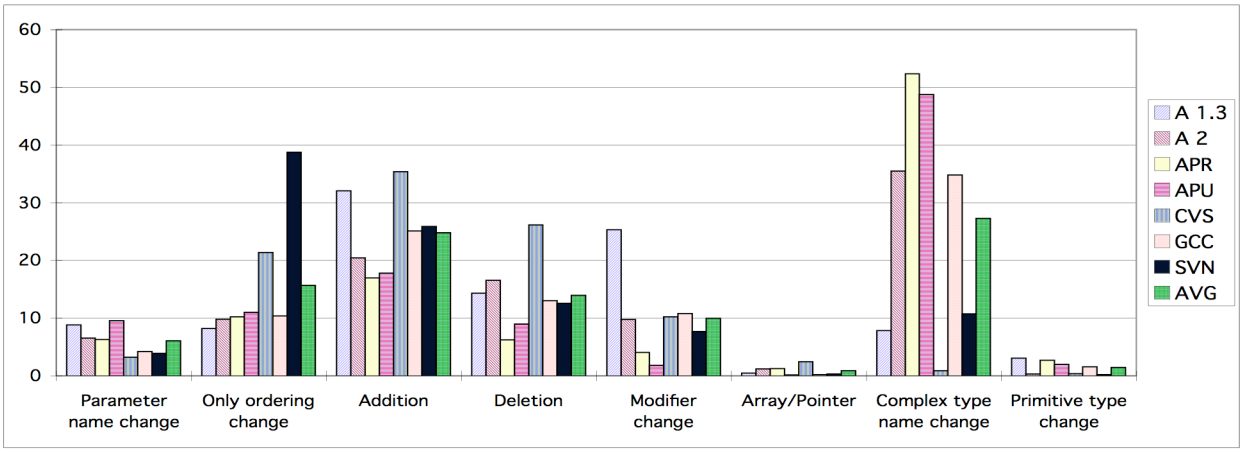

Figure 2. Per-project and average frequency for each signature change kind for seven open source projects.

Table 5. Numbers of parameter, return type, and function name changes of selected projects

	A1.3	A2	APR	APU
Total parameter changes	818	2796	4180	556
Return type changes	49	190	571	63
Function name changes	413	201	885	184

4.2 Distribution of Signature Changes

To show the distribution of signature changes across functions, we counted the number of functions in SVN having *n* signature changes, with *n* varying from 0 to 19 changes. SVN has 5925 unique functions after applying the origin analysis result. Figure 3 shows that 3434 functions (58%) never changed their signature and 94% of the functions had fewer than three signature changes.

Table 6 shows the percentage of functions whose signature never changed, and the percentage of functions whose signature changed less than three times. For example, in A2 57% of function signatures never changed, and 90% of function signatures changed less than three times. Note that GCC has a small total number of changes, 454, and most signatures changed only once or twice, resulting in higher figures than other projects.

Table 6. Percentages of signatures that never change, and change less than three times. For example, 56% of function signatures in A 1.3 never changed, and 96% of them changed less than three times.

A 1.3	A2	APR	APU	CVS	GCC	SVN
56/96	57/90	49/86	67/96	61/97	92/99	58/94

Another interesting ratio of signature changes can be obtained by comparing the number of signature changes and number of function body changes. Table 7 shows the body change to signature change ratios for each project. For example, in the Apache 1.3 project, on average, a function's signature changes after its body has changed 7.6 times. The correlation between function signature and function body is discussed further in Section 4.6. The ratios for CVS and GCC are relatively high. These two projects have a relatively small number of signature changes (CVS: 1143, GCC: 454) in spite of their relatively high number of transactions (CVS: 2873 transactions, GCC: 3012 transactions).

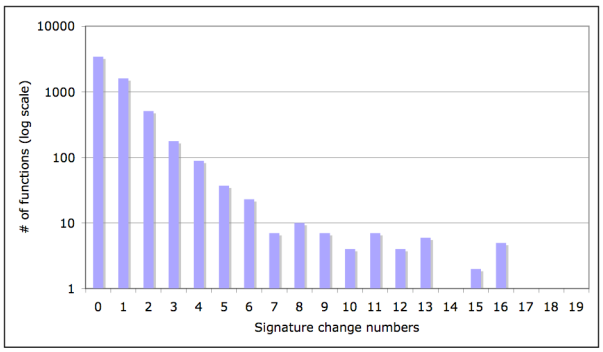

Figure 3. Numbers of signature changes of functions in the Subversion project. The x-axis is number of signature changes, and the y-axis is the number of functions (log scale).

Table 7. Function body : signature change ratios

	A 1.3	A2	APR	APU	CVS	GCC	SVN
Ratio	7.6:1	5.95:1	3.5:1	6.3:1	14.9:1	13.2:1	6.2:1

4.3 Number of Parameters

In Figure 2, we observed that the frequency of parameter addition is almost twice that of parameter deletion. Based on this, we hypothesized that the average number of arguments would grow over time. If true, we felt this would be a strong indicator of code decay, since "more parameters" is a rough measure of inter-function coupling (more data being passed) and

function complexity (more data means more processing of that data). However, it was not entirely clear that the number of parameters would grow, since a parameter deletion event can delete more than one parameter, and hence a small number of deletions could remove more parameters than a large number of additions. Figure 4 shows the average number of arguments over the transaction history of four projects.

There was no strong trend towards increasing numbers of parameters. Apache 1.3 and Apache 2 show a slow increase in parameter count, while APR rose, dipped, then rose again. APU was similarly jagged, with a sharp increase, then a slow taper.

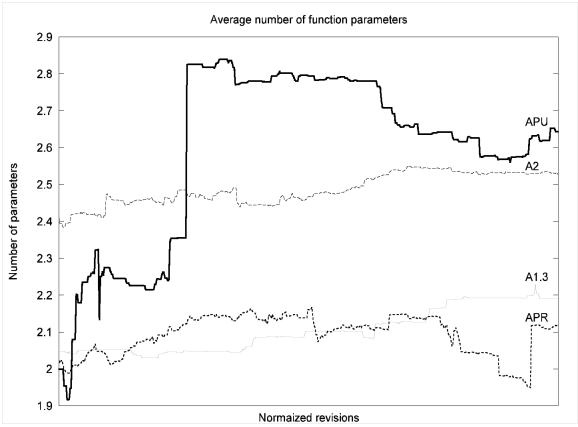

Figure 4. Average number of parameters vs. transaction number for four projects

We wondered if this phenomenon could be explained by the existence of a "per-project threshold" for the number of function parameters. The idea here is that developers have an upper bound for the length of parameter lists they will tolerate. If the number of parameters gets close to the threshold value, developers will split or refactor the function. To determine if this threshold effect was at work, we took the Apache 2 project (which shows slow growth) and observed the maximum, minimum, and current count of parameters for each function. Figure 5 shows rough trends in the growth and shrinkage of function parameter lists for Apache 2; functions that have a current value (parameter number in the latest revision) greater than their minimum have an upward arrow (the parameter list grew), the others have a downward arrow (the parameter list shrank). Alas, there is scant evidence for this threshold hypothesis, with more functions growing than shrinking, especially at high parameter counts.

What Figure 5 does indicate is that frequency distributions may be a better characterization of parameter list length than an average number of parameters, since the data is inherently quantized. Figure 6 shows the frequency distribution of parameter list length for 4 projects. It shows that the most common parameter list lengths are 1, 2 and 3, with a long tail of longer list lengths.

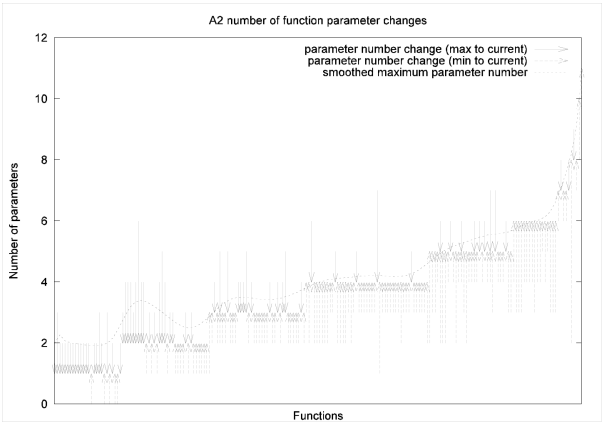

Figure 5. Each position in the x-axis represents a function. The y-axis represents the count of parameters. An arrow shows the parameter number change, max/min to current parameter count, per a function. The smoothed line shows the interpolated maximum number of parameters.

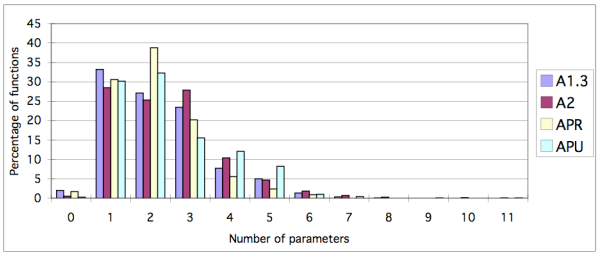

Figure 6. Parameter count frequency showing number of parameters vs. percentage of functions.

4.4 Unique Signature Change Patterns

We have shown that different projects have different signature change patterns. But does each project have its own unique signature change pattern? If that is the case, we can analyze part of a project's history to reveal the patterns for the entire project. Based on previous change patterns, we could roughly predict future change patterns, since they would be similar. We explored this hypothesis by taking two projects, Subversion and Apache 1.3, and observed signature change kind frequencies by analyzing the first 100, 200, 300, 500, 1000, 1500, 2000, 3000, 5000 and all transactions (6029 for SVN and 7747 Apache 1.3) to see if there is a unique signature change pattern for each project.

Figure 7 shows the percentage of major signature change kinds by analyzing various transaction numbers. Clearly SVN and Apache 1.3 have different signature change patterns. For example, SVN has almost 50% ordering-only changes, as compared to fewer than 10% for Apache 1.3.

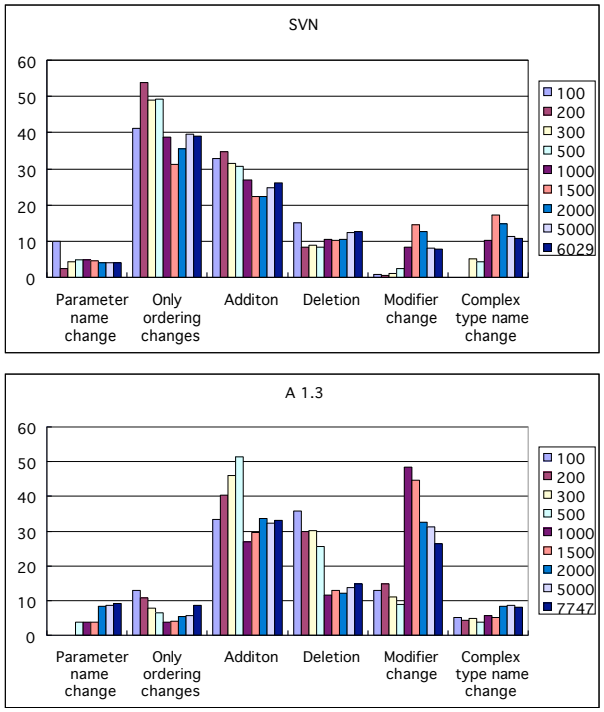

Figure 7. Major change kind frequency with various transaction ranges

However, within the same project, the frequency results from analyzing various transaction ranges such as 1-100, 1-200, and 1-300 generally converge. When the first 1000~1500 transactions are used as training data, the percentage of each change kind is almost the same as the result for the whole history. We need to observe other projects, but the results strongly suggest that each project has its own unique signature change pattern and analyzing the first 1000 ~ 1500 transaction (about one year in calendar time for these two projects) will reveal this pattern.

4.5 Signature Change Sequences

We wondered whether a signature change kind depends on the previous changes. For example, consider a hypothetical project where a return type change (R) is 80% likely if the previous three changes are, in order, a parameter addition change (A), parameter deletion change (D), and an ordering change (O). For this project, when this known signature change evolution sequence occurred (A, D, O), we could make predictions about the likely next signature change (an R).

To determine whether the probability of a change depends on previous changes, we computed all possible conditional probabilities of signature changes, and show the probabilities using graphs similar to Bayesian Networks [1]. For example, when the signature of a function changes in this order: A, D, O, R, and A (See Table 3 for the change pattern abbreviations), we generate a change sequence, '*ADORA*'. We examined all signature change sequences whose length is larger than two. We assumed that change sequences with fewer than three changes are rarely associated with common evolution paths. After having an array of the sequences, we computed probabilities for each change such as A, O, C, and D. We also computed probabilities of each change if previous changes are certain kinds, such as A, O, C and D. The probability graphs of APR and Apache2 projects are shown in Figure 8. The graphs show only high probability nodes to reduce clutter. The total number of analyzed change patterns of APR is 1494, and Apache2 is 1082.

The results show that the probability of specific signature change kinds occurring does vary based on previous changes. For example, in APR the probability of change kind C is 0.94 if the two previous changes were O, and then C. In this case, if we observe O and then C changes, the next change will very likely be C. The two graphs (a) and (b) have different probabilities, but we believe the probabilities are consistent within a single project as we discussed in Section 4.4.

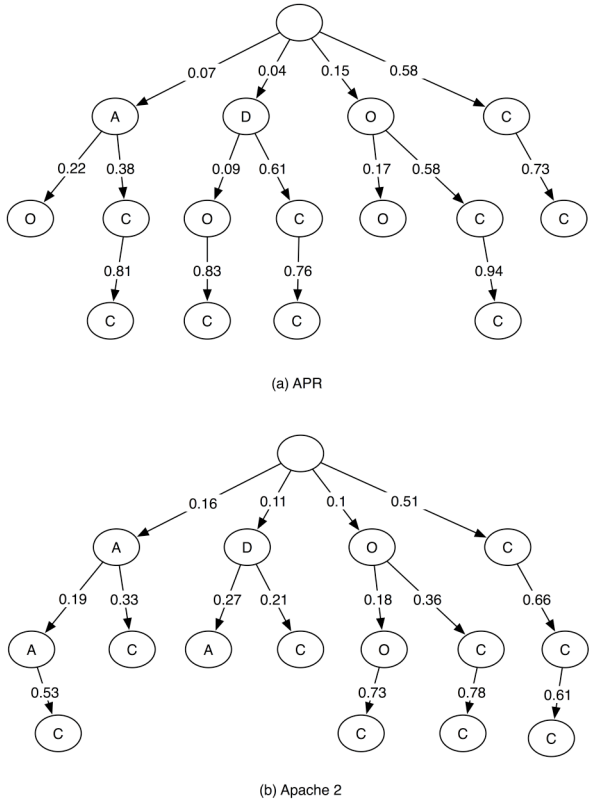

Figure 8. Probability graphs of APR and Apache2. Each node indicates change kind and Edge shows evolution path. A number in the arrow indicate the probability to get to the node.

4.6 Correlations

If there are correlations between signature changes and other changes, such as changes in LOC or changes in the number of function body modifications, we could use signature change patterns for other software analysis such as identifying unstable areas [4] of code or code decay areas [8]. For example, if a certain function's signature changes many times over a project's history, it may indicate the function is not stable or the function is decayed.

To measure correlations between signature changes and other changes, we used Pearson's correlation measure [7]. First, we observed the number of function changes and signature changes to see if there are correlations. We also observed correlations between LOC changes and signature changes. Table 8 shows the Pearson's correlation value for the correlation between signature changes and LOC changes, and signature changes and function body changes. All projects have similar correlations between signature changes and LOC changes, except APR and APU. Since APR and APU are libraries, their maintainers work to avoid changing APIs, changing signatures only when it is absolutely necessary. When they decide to change signatures, they change many signatures at the same time. Note that APR and APU have correlations between signature changes and body changes.

Table 8. Correlations of signature changes and changed LOC and function body changes

	A 1.3	A2	APR	APU	CVS	GCC	SVN
LOC	0.38	0.73	0.08	0.17	0.51	0.57	0.44
Body	0.44	0.47	0.62	0.51	0.45	0.19	0.65

Overall there are weak correlations between signature changes and other changes. One of the reasons that they have weak correlation is that only 57% of function signatures (on average) ever change, as shown in Section 4.1. Since there are correlations between signature changes and other kinds of changes, we can use signature change patterns for software analysis such as identifying unstable areas or detecting code decay, instead of LOC or other change patterns. Determining other ways in which signature change patterns can improve software maintenance analysis remains future work.

4.7 Signature Change and Bug Introduction

We analyzed two of our projects, Apache 1.3 and Subversion, to determine if there is a correlation between code modifications where a signature is changed, and increased frequency of bugs. We identify bug introducing changes using the fix-inducing identification algorithm described in [13, 15]. We group changes in two folds: changes with signature change and changes without signature change. Then we count the number of buggy changes in two folds.

Consider the change history for the function '*foo()*' as shown in Figure 9. The change log at transaction n (Rev n) states "Fixed issue #355", which indicates that it is a fix change. It means the function at transaction n-1 has one or more problematic lines, which are fixed in transaction n by changing them. When were the problematic lines added in the first place? SCM systems such as CVS [3] and Subversion [2] provide an annotation feature that shows information about when each line of a file was modified, and by whom. Using the SCM annotation information, we can determine when the problematic lines were initially added. Suppose the problematic lines were added in transaction 3. This means the file at transaction 2 does not have the problematic lines, so they were added in the change between transaction 2 and 3. This change introduced a bug into the software, and hence we call it a bug-introducing change. All other changes, such as the change between transactions 1 and 2 in Figure 9, are clean changes. Details on the algorithm used for identifying bug-introducing changes are given in [15].

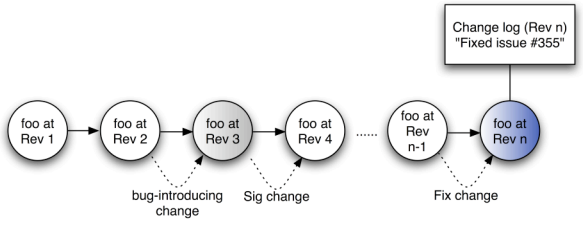

Figure 9. The foo function change history

To determine whether signature-modifying changes introduce more (or fewer) bugs, we separate changes into two groups: signature-modifying changes and non-signature-modifying changes. Then we identify and count the numbers of bug introducing changes in each group. Table 9 shows bug-introducing change counts and rates of the two groups (signature and non-signature changes) of selected projects. For example, Apache 1.3 has 6,373 non-signature changes. Among them, 1,084 (17%) are buggy changes. Apache 1.3 has 526 signature changes, and among them 124 (23.6%) of them are buggy changes. For both projects, the signature change group has a higher percentage of changes that are buggy.

Table 9. Bug-introducing change counts and percentages of signature and non-signature changes of the selected projects and transactions. We analyzed changes in Apache 1.3 (A1.3) and Subversion (SVN) during the first 5000 transactions.

Project	Non-signature changes			Signature changes		
	# of changes	# of buggy changes	% of buggy changes	# of changes	# of buggy changes	% of buggy changes
A1.3	6,373	1,084	17	526	124	23.6
SVN	20,193	763	3.78	4,903	256	5.22

The bug introduction percentages of signature changes are higher than those of non-signature changes. In the case of Apache 1.3, the difference is high enough (6%) to be significant. For Subversion, the difference is 1.4%, which may not be significant. These results are indicative that changes that modify signatures carry a greater risk of introducing bugs than those that modify only function bodies. Repeating this analysis on a larger dataset is necessary to fully establish this correlation.

5. Properties of Signature Changes

Based on the results presented in Section 4, we can now define some general properties of software signature changes. The properties we define here are based on results of signature analysis of relatively small number of projects, but can serve as a basis for comparison between our signature change properties and those found by others.

Property 1. The most common signature change kinds are, in order, complex data type, parameter addition, parameter ordering, and parameter deletion.

Property 2. Over half of all function signatures never change. 90% of functions change signatures less than three times.

Property 3. For every 5- to 15 function body changes, there is a signature change.

Property 4. The average number of parameters remains relatively constant over time.

Property 5. Functions typically have parameter lists with 1, 2, or 3 parameters.

Property 6. There are weak correlations between signature changes and other changes, such as LOC and function body changes.

Property 7. Each project has its own signature change patterns, and these patterns can be discovered after analyzing the first 1000 to 1500 transactions.

Property 8. Probability of a change kind depends on previous changes.

Property 9. Signature changes tend to introduce more bugs than non-signature (only function body) changes.

6. Threats to Validity

The results presented in this paper are based on a selected set of seven open source projects. This set includes major open source projects, but other open source or commercial software projects may not have the same properties we presented here. We analyzed only projects written in the C programming language; software written in other procedural programming languages may have different signature change patterns. Projects written in Object Oriented Programming (OOP) languages such as Java, C++, or C# will certainly have different signature change properties, because the language abstractions go beyond simple functional boundaries.

Some open source projects have transactions that cannot be compiled and contain syntactically invalid source code. In these cases, we had to guess at the signatures 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.

7. Applicability

In general, function signature changes occur for many reasons, from behavior-neutral data restructuring to modifications of the abstraction that the function is meant to represent. The ability to classify the kinds and patterns of signature changes allows software evolution researchers to enhance their own results. Instability analysis [4] could use signature change analysis and the inferred data flow changes to segregate instabilities based on collapsing data isolation from those based on an evolving concept of program functionality. Systems that use logical coupling or association rules to produce a history-based change-impact set, such as Gall et al. [9], Zimmerman et al. [18], or Ying et al. [17] could use the project-specific change patterns to provide more context for their suggestions. For example, instead of simply "users who changed X also changed Y", an addendum of "but the only change in Y was because X.foo() changed its signature" would certainly be insightful, and indeed could affect the overall categorization of returned suggestions. The benefit of signature change analysis is not limited to software evolution analysis but is also applicable to software maintenance. Based on these results, a more complete ontological framework that includes a conceptual meaning for each parameter with its data type certainly seems possible. With such a framework, it may be possible to accommodate ordering changes and parameter deletion changes by automatically generating glue code that resolves the signature mismatch problem.

8. Future Work and Conclusions

Among change kinds, the most common change kinds are complex type changes, parameter addition, ordering-only changes, and parameter deletion. In future work we hope to use this result to alleviate the impact of signature changes, by automatically generating "glue code" to temporarily bridge mismatched signatures. Observing signature change properties of projects in OOP languages is also future work.

We have provided a detailed taxonomy and categorization of signature change kinds found in C functions. We also presented the nine key properties we discovered during this analysis. Among these properties, we found that there exist correlations between signature and function body changes (Property 6). Property 7 reveals that each project has a unique signature change pattern, and we only need approximately the first 1500 transactions to find it. The patterns found from these training transactions can be used to predict future changes patterns. We also found that changing signatures introduces more bugs than changing only function bodies (Property 9). While several of our results will be strengthened with the addition of more analyzed datasets, our work so far indicates that signature change analysis could eventually be used to enhance the results of other types of software evolution and maintenance research.

9. Acknowledgements

We thank Kai Pan and Mark Slater for their valuable feedback on this paper. We especially thank Kevin Greenan, and 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.

10. References

[1] E. Alpaydin, *Introduction to Machine Learning*: The MIT Press, 2004.

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

[3] B. Berliner, "CVS II: Parallelizing Software Development," Proc. of Winter 1990 USENIX Conference, Washington, DC, pp. 341-351, 1990.

[4] J. Bevan and E. J. Whitehead, Jr., "Identification of Software Instabilities," Proc. of 2003 Working Conference on Reverse Engineering (WCRE 2003), Victoria, Canada, pp. 134-145, 2003.

[5] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005.

[6] S. Counsell, Y. Hassoun, R. Johnson, K. Mannock, and E. Mendes, "Trends in Java Code Changes: the key to Identification of Refactorings?," Proc. of 2nd Int'l Conference on Principles and Practice of Programming in Java, Kilkenny City, Ireland, pp. 45-48, 2003.

[7] R. E. Courtney and D. A. Gustafson, "Shotgun Correlations in Software Measures," *Software Engineering Journal*, vol. 8, pp. 5-13, 1992.

[8] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions on Software Engineering*, vol. 27, pp. 1-13, 2001.

[9] H. Gall, K. Hajek, and M. Jazayeri, "Detection of Logical Coupling Based on Product Release History," Proc. of International Conference on Software Maintenance (ICSM '98), pp. 190-198, 1998.

[10] M. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Transactions on Software Engineering*, vol. 31, pp. 166-181, 2005.

[11] S. Kim, K. Pan, and E. J. Whitehead, Jr., "When Functions Change Their Names: Automatic Detection of Origin Relationships " Proc. of 12th Working Conference on Reverse Engineering (WCRE 2005), Pennsylvania, USA, 2005.

[12] S. Kim, E. J. Whitehead, Jr., and J. Bevan, "Analysis of Signature Change Patterns," Proc. of Int'l Workshop on Mining Software Repositories, St. Louis, Missouri USA, 2005.

[13] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, Jr., "Automatic Identification of Bug Introducing Changes," Proc. of 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), Tokyo, Japan, 2006.

[14] D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change Impact Identification in Object Oriented Software Maintenance," Proc. of Int'l Conference on Software Maintenance, Victoria, BC Canada, pp. 202-211, 1994.

[15] J. Sliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, USA, pp. 24-28, 2005.

[16] Q. Tu and M. W. Godfrey, "An Integrated Approach for Studying Architectural Evolution," Proc. of Int'l Workshop on Program Comprehension (IWPC 2002), Paris, France, 2002.

[17] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Transactions of Software Engineering*, vol. 30, pp. 574-586, 2004.

[18] T. Zimmerman, P. Weissgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," Proc. of Int'l Conference on Software Engineering (ICSE '04), Edinburgh, Scotland, UK, pp. 563-572, 2004.

[19] T. Zimmermann and P. Weißgerber, "Preprocessing CVS Data for Fine-Grained Analysis," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2004), Edinburgh, Scotland, 2004.