An Approach for Measuring Software Evolution Using Source Code Features *

Ladan Tahvildari, Richard Gregory, Kostas Kontogiannis
Department of Electrical & Computer Engineering
University of Waterloo
Waterloo, ON, N2L 3G1, Canada
{ltahvild, rwgregor, kostas}@swen.uwaterloo.ca

Abstract

One of the characteristics of large software systems is that they evolve over time. Evolution patterns include modifications related to the implementation, interfaces and the overall system structure. Consequently, system understanding and maintainability tend to degrade over time unless particular attention is paid to measure, assess and evaluate the effects of the evolution activities. Traditionally, the assessment of evolution activities has focused on the architectural level. However, in many cases it is easier to extract low-level program information from the Abstract Syntax Tree rather than to discover the full architecture of a large legacy system. This paper presents techniques for analyzing the evolution of large systems even in cases where no complete architectural views of the system exist, from information obtained solely from the AST. We present experimental results by analyzing the evolution patterns across different versions, of two popular systems, the Apache Web server, and the Bash shell.

1. Introduction

Legacy software systems evolve over time. By having an architectural description for a system, developers are able to share a common mental model and form a high-level understanding of a system. This hides much of the detail that clutters the implementation of a system, allowing software engineers to reason about the entire system in addition to the individual components.

This paper describes a methodology for analyzing the evolution of large systems. In this way, a software engineer can predict the type of evolution to which different software components will be subjected. We present results obtained from experiments which identify low-level source code features that can be used to provide an accurate view of software evolution at the architecture level [10]. We examined two software systems and conducted two independent studies on each. In the first study, we investigated evolution at the architectural level with the focus on three categories: *interface evolution*, *implementation evolution*, and *structural evolution*. In the second study, we analyzed evolution by examining the effect that changes in low-level source code features (control flow in functions, parameter passing, data flow properties, and metrics) had on the overall system architecture. For each system, *evolution distance* measures were defined that determine the degree of change from one version to the next at both the source code and the architectural level. The evolution distances obtained from these two studies for each level were correlated to select the source code features which have the greatest effect on evolution at the architecture level (interface, implementation, structure). Evolution distances were computed based on *insertions*, *deletions*, and *modifications* for source code and architectural entities. These provide a measure of maintainability and of the dissimilarity between system components as a result of change.

This paper is organized as follows: Section 2 presents related work, then Section 3 discusses our techniques for software architecture recovery. In Section 4, we discuss our approach for evolution analysis. Sections 5, 6, and 7 present findings from the evolution analysis of Bash and Apache. Section 8 provides an interpretation of our results and finally Section 9 discusses conclusion, ongoing, and future work.

2. Related Work

One approach to recovering the understanding of a system is to extract its architecture from high-level data flow

*This work was funded by the IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto) and the National Research Council of Canada.

0-7695-0509-0/99 $10.00 © 1999 IEEE

information [2]. A number of architecture design recovery tools have been implemented to help developers understand large software systems. These include PBS [4, 7], GASE [8], and Rigi [13]. These tools provide high-level views that describe the current system design. Moreover, they help developers assess how closely a system's implementation matches its intended structure. Most of these tools aid in reconstructing an architecture based on facts extracted from the source code of a system.

More recently, other models have been proposed as a means for architectural design recovery. Murphy presents in [14] the concept of reflexion models that allow interactions and evolution patterns in large software systems to be distinguished and analyzed. Reflexion models have been successfully used for the re-engineering of large industrial software systems.

Kazman presents an approach for capturing and assessing architectures for evolution and reuse [9]. The *Dali* workbench for architectural extraction supports flexible extraction and fusion of architectural information. This tool enables various types of relevant information to be modeled and provides a means to extract architectural views for re-engineering, analysis, and for comparing architectures.

As part of an ongoing project with researchers from the IBM Center for Advanced Studies, the University of Toronto, and the University of Victoria, Kontogiannis presents in [10] several metrics that describe various software features. These metrics are computed and added as annotations to an AST (Abstract Syntax Tree) extracted with REFINE [1].

3. Architectural Design Recovery

Two types of architecture are particularly beneficial to humans trying to understand a software systems: a *conceptual architecture* and a *concrete architecture*. The conceptual architecture shows how developers think about a system; it shows relationships between subsystems that are 'meaningful' to developers. In contrast, the concrete architecture of a system shows the relationships that exist in the implemented system. While the conceptual architecture is easier to understand because it contains only essential relations, the concrete architecture is necessary when making decisions requiring implementation-specific knowledge [1].

An architecture recovery system extracts *facts* from a system implementation, then combines these facts into higher level abstractions. The extracted facts may be in many forms. Researchers have extracted information about function calls, data accesses, and file operations to help reconstruct views of an architecture.

To begin our analysis of a software system, we obtain an abstraction of the source code. Several program representation schemes have been proposed in the relevant literature. We have chosen as a program representation scheme, the program's annotated Abstract Syntax Tree (AST) using the commercial tool REFINE as our primary development environment.

[1] Refine is a trademark of Reasoning Systems Corp.

4. Software Evolution Analysis

4.1. Evolution Categories

We have classified evolution into three main categories: a) *interface evolution* which relates to source code changes that affect the interfaces between functions, modules, or subsystems, b) *implementation evolution* which relates to source code changes that affect the control flow and data flow properties of a given code fragment, and c) *structural evolution* which relates to changes that affect the structure of the system but do not necessarily affect its functionality, control flow, or data flow properties.

4.2. Measuring Interface Evolution

Interface evolution is measured by analyzing the corresponding program features from one version to another. An overall distance between versions is computed. This distance between versions j and k for module M is calculated by

$$E_{jk}^{interface}(M) = \Sigma_i w_i(F_{ij}(M) - F_{ik}(M))$$

where w_i is the weight for feature i and $F_{ij}(M)$ is the value for feature i on version j for module M. For example, if $F_{ij}(M)$ is the number of parameters passed by reference for module M in version j, then the evolution distance is based on the weighted difference between the number of parameters passed by reference for this module and in another version of the system (i.e. version k). The weights were based on previous experience in measuring metric distances [10].

4.3. Measuring Implementation Evolution

Implementation evolution is also measured by computing the weighted sum

$$E_{jk}^{implementation}(M) = \Sigma_i w_i(F_{ij}(M) - F_{ik}(M))$$

where w_i is the weight for feature i and $F_{ij}(M)$ is the value for feature i on module M on version j.

4.4. Measuring Structural Evolution

Structural evolution is measured by computing the weighted sum of the number of additions, deletions, and modifications of functions and libraries between two versions of the software system. In particular,

$$E_{jk}^{structure}(S) = w_{del}D_{jk}(S) + w_{ins}I_{jk}(S) + w_{mod}C_{jk}(S)$$

where $D_{jk}(S)$ is the number of deletions (functions/libraries) between version j and version k for subsystem S and $I_{jk}(S)$ is the number of insertions (functions/libraries) from version j to k, and $C_{jk}(S)$ the number of modified (split or merged) functions and libraries between versions j and k.

4.5. Metrics Analysis

In addition to the the above features for measuring evolution, we also create a vector of orthogonal metrics for each function, module or subsystem in each version [10]. The difference between two versions for a source code entity E is then computed as the Euclidean distance between the metrics vectors for the entity E in versions i and j. We use four metrics that are sensitive to several different control flow and data flow program features. If s is a source code fragment, then the following metric values are computed for each function, module, and subsystem: i) $S\_COMPLEXITY(s) = FAN\_OUT(s)^2$ where $FAN\_OUT(s)$ is the number of individual function calls made within s, ii) $D\_COMPLEXITY(s) = GLOBALS(s)/(FAN\_OUT(s) + 1)$ where $GLOBALS(s)$ is the number of individual declarations of global variables used or updated within s, iii) $MCCABE(s) = e - n + 2$ where e is the number if edges in the control flow graph, and n is the number of nodes in the graph [12], vi) $HALSTEAD$'s software science [6] is based on four metrics which can be computed through an AST (the number of distinct operators; the number of distinct operands; the total number of operator occurrences; and the total number of operand occurrences).

A complete list of all metrics used in each of our two studies is shown in Table 1.

4.6. Software Evolution and Maintainability

Another aspect of this study deals with the impact of software evolution on the maintainability of systems which has become one of the most important aspects of software systems. Many models have been defined and implemented to quantify maintainability based on software metrics. Here we present three models that we use to compute maintainability indices.

Metric	Description
Distance	Euclidean distance among elements
Jaccard	Jaccard distance among elements
Fan-In	Number of called functions
Fan-Out	Number of individual function calls
McCabe	McCabe cyclomatic complexity
D-Flow	Number of used and updated global variables
Num-Stm	Total source code statement
Num-Ops	Total operator count
Num-uops	Unique operator count
Num-Opnds	Total operand count
Num-uopnds	Unique operand count
Comments	Number of comments
LOC	Total source input lines
LnCh	Changed lines
Effort	Halstead's program effort

Table 1. List of Metrics Analyzed.

1. A three-metric model based on Fan-Out, data complexity, and McCabe cyclomatic complexity. Further explanation of this index can be found in [15].
   Software Maintainability Index 1 = *125.951 - 3.975 * Fan-Out - 0.999 * D-Com - 1.142 * McCabe*

2. A single metric model based on Halstead's efforts [6]. Further details can be found in [16].
   Software Maintainability Index 2 = *125 - 10 * log(ave-E)*

3. A four-metric model based on Halstead's effort [6], McCabe's $V(G')$, LOC, and CMT. More information can be found in [3].
   Software Maintainability Index 3 = *171 - 3.42 * ln(ave-E) - 0.23 * avg-V(G') - 16.2 * ln(avg-LOC) + 0.99 * avg-CMT*

5. Experimental Studies

In this section, we present the evolution of two public domain legacy systems: the Apache Web server and the Unix shell Bash.

5.1. Bash

Bash is GNU's "Bourne Again SHell" from the Free Software Foundation that is an `sh`-compatible command language interpreter. The reason we choose Bash as a subject of our study was that the source code of four different versions was available and it was sufficiently complex, yet small enough to manage for research.The versions we have chosen to examine include 1.14.7, 2.01, 2.02, and the most recent version 2.03. Table 2 gives some statistics for these four versions.

We found that the conceptual and concrete architectures were consistent between all four versions. Our description of Bash is based largely on the descriptions of the general

Version	LOC	Source Files	Header Files	LOC/file
1.14.7	40,661	40	47	467
2.01	46,179	48	49	476
2.02	47,762	47	51	487
2.03	48,516	47	54	480

Table 2. Basic Statistics of Bash.

functionality of shells, as well as on the structure of the source code. This corresponds most closely to the Pipeline Architecture, as defined by Shaw and Garlan [5] and is illustrated in Figure 1, where the bold arrows represent control and data flow from one stage to the next. The solid arrows show procedure calls among the pipeline stages as well as to the utility and interface modules. As Figure 1 shows, Bash's architecture can be described as follows: i) it reads its input from a file (Shell Scripts), or from a string (Invoking Bash), or from the user's terminal, then breaks the input into words and operators, ii) it parses the tokens into simple and compound commands, iii) it performs the various shell expansions, breaking the expanded tokens into lists of filenames, commands and arguments, and finally iv) it executes the command. All these stages of the pipeline are done through an interface and by accessing various utilities.

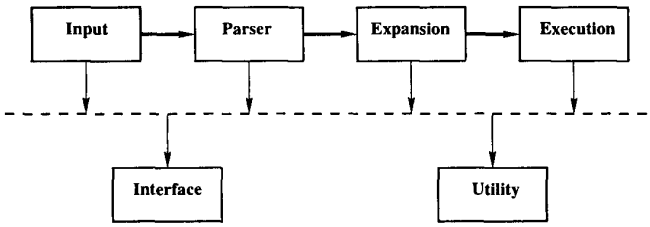

Figure 1. Architecture of Bash.

5.2. Apache Web Server

Apache [2] is a free and publicly available Web server and is currently in use on over 50% of all Internet web sites. Apache is developed and maintained by the Apache Group and other volunteers, located in various parts of the world. The system is available for a number of operating systems. There were several reasons why we chose Apache as a subject of our study. We could easily obtain source code for five different versions and we felt that Apache was even more interesting than Bash. In the interest of technical content, we also wanted to use a system that was sufficiently complex, had undergone significant revision, yet was not unmanageably large.

[2] As a side note, since Apache originally contained a series of patches applied to the NCSA code - the result was then referred to as "a patchy server", the source of Apache's present name.

The versions we have chosen to examine include 1.1.1, 1.2.4, 1.2.6, 1.3.4, and the most recent version 1.3.6. Table 3 gives some introductory statistics for these five versions.

Version	LOC	Source Files	Header Files	LOC/file
1.1.1	28,692	48	17	440
1.2.4	31,878	42	19	523
1.2.6	39,966	54	20	619
1.3.4	79,819	102	56	505
1.3.6	75,523	81	40	616

Table 3. Basic Statistics of Apache.

We discovered a consistent conceptual architecture of Apache over all five versions. This is based largely on descriptions of the API, as well as on the structure of the source code. This corresponds most closely to the Development View of Architecture, as defined by Kruchten [11] and is illustrated in Figure 2. In the diagram, the dotted arrows represent uses of modules by the core subsystem. The solid arrows show procedure calls among the core subsystems as well as to the operating system.

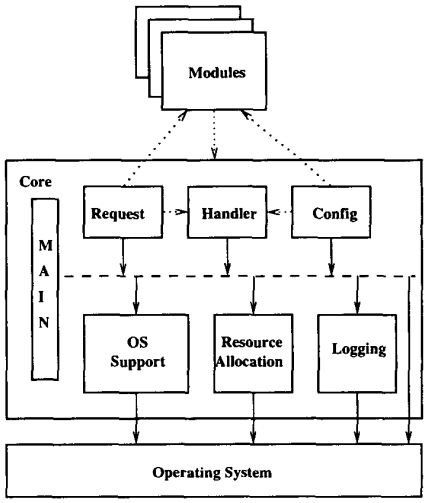

Figure 2. Architecture of Apache.

6. Apache Web Server Architectural Evolution

In this section we present an analysis of the concrete architecture of Apache as it has evolved over three major releases. For our analysis, we used REFINE [10], PBS [7], and Rigi [13]. Our aim was to understand its architectural evolution and then correlate the results with source code features that have the most impact per evolution category (interface, implementation and structure).

In early versions (e.g., 1.1.1 in our study), the Apache Web Server was released as a set of source code files with no directory structure, that is, all files for the entire Apache system were located in one directory. A partial file naming convention allowed for some initial clustering of source code files and libraries. For example module files were prefixed with `mod_` and the system core files were prefixed with `http_`. However, not all functions followed a naming pattern making any analysis based solely on naming conventions inaccurate. The early Apache architecture features two main co-operating subsystems (Apache Modules, Apache Core). This is illustrated in Figure 3. The purpose of the core subsystem is to provide a programming platform that allows an HTTP request to be fulfilled in a flexible way. The purpose of the modules subsystem is to use platform functionality to implement the request handling expected of a modern Web Server.

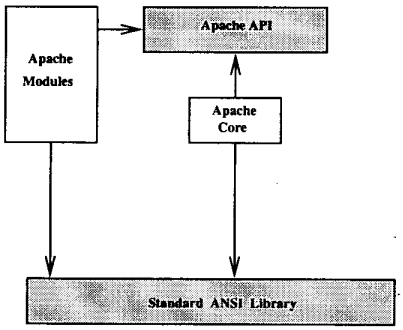

Figure 3. Concrete Architecture of First Generation of Apache.

Our evolution analysis concluded that Apache was augmented in versions 1.2.4 and 1.2.6 with an abstraction of OS functionality. Moreover, a set of routines for customizing standard ANSI C libraries were added. More importantly, in these versions a directory structure was beginning to emerge. A key point is that all modules were placed in their own subdirectory. There were also two other directories: one for regular expression processing functions and another for utility shell scripts. The architecture for versions 1.2.4 to 1.2.6 is illustrated in Figure 4.

At the third generation of Apache, such as in releases 1.3.4 and 1.3.6, a set of macros were added to provide "cluster" functionality offered by the API and specific subsystems such as the core. In these versions a robust directory structure was now in place. The architecture for versions 1.3.4 to 1.3.6 features four cooperating subsystems and is illustrated in Figure 5. The *Platform* subsystem provides a base set of helper functionality and extends the standard library by providing an interface for the core subsystem and an interface for the API. The *OS Abstraction* subsystem hides the underlying differences between operating systems and implements functionality that is absent on certain operating systems. The other two subsystems have similar functionality to that of previous versions.

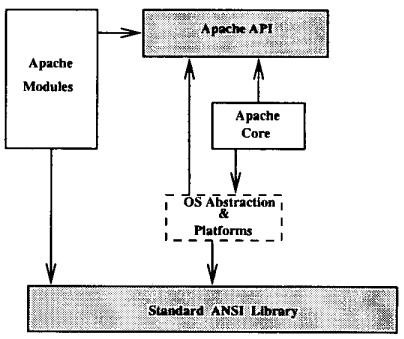

Figure 4. Concrete Architecture of the Second Generation of Apache.

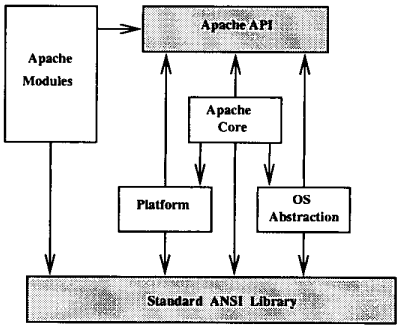

Figure 5. Concrete Architecture of the Third Generation of Apache.

Once we had an analysis of the concrete architecture over five major releases (1.1.1 - 1.3.6) of the system we could examine how the architectural evolution patterns relate to changes at the source code level.

7. Apache Web Server Source Code Evolution

From our analysis of the source code as described in Sections 4 and 5, we discovered a set of program features where the distances measured across pairs of versions show a high correlation with distances obtained using architec-

Version	AST Size	Code Size
1.1.1	8.5 MB	28.6 KLOC
1.2.4	14.4 MB	31.8 KLOC
1.2.6	14.6 MB	39.9 KLOC
1.3.4	21.2 MB	79.8 KLOC
1.3.6	20.1 MB	75.5 KLOC

Table 4. Size of AST for Five Versions of the Apache Web Server.

tural features (interface changes, implementation changes, structural changes). These features include Fan-in, Fan-out, Number of Statements, Data usage, and McCabe cyclomatic complexity. Our analysis was on the evolution of subsystems concerned with the operating system abstraction and different platform utilities. Over the course of evolution, the structure of the modules remained unchanged, although the number of modules increased. Although the conceptual architecture remained consistent throughout each version, an inspection of the source code and the use of reverse engineering techniques indicated that the concrete architecture underwent a gradual change. As we described earlier, the program's AST has been chosen as an abstract representation scheme for this study. Table 4 gives the size of AST for five different versions of Apache. This is the first aspect of evolution that we encountered after beginning our analysis.

After building the AST for each version, evolution, and the effects that source code changes had on system characteristics, was measured based on three primary categories as described previously. Table 5 illustrates the major evolution patterns for Apache.

File Name	Subsys Name	Versions	Types of Evolution
alloc.c	Core	1.2.4 & 1.2.6 1.2.6 & 1.3.4	Implntn, Interface Interface, Structural
buff.c	Core	1.1.1 & 1.2.4 1.2.4 & 1.2.6 1.2.6 & 1.3.4	Interface Implementation Interface, Structural
http_config.c	Core	1.1.1 & 1.2.4 1.2.4 & 1.2.6 1.2.6 & 1.3.4	Structural Interface Implntn, Interface, Str
http_core.c	Core	1.1.1 & 1.2.4 1.2.4 & 1.2.6 1.2.6 & 1.3.4	Structural, Interface Implementation Interface, Structural
http_main.c	Core	1.1.1 & 1.2.4 1.2.4 & 1.2.6 1.2.6 & 1.3.4	Structural, Implntn Structural Interface, Structural
http_protocol.c	Core	1.1.1 & 1.2.4 1.2.4 & 1.2.6 1.2.6 & 1.3.4	Structural, Implntn Structural, Implntn Structural, Implntn
mod_access.c	Modules	1.2.6 & 1.3.4	Implementation
mod_alias.c	Modules	1.2.6 & 1.3.4	Interface
mod_cern_meta.c	Modules	1.2.6 & 1.3.4	Interface
mod_cgi.c	Modules	1.1.1 & 1.2.4	Interface
mod_dir.c	Modules	1.2.6 & 1.3.4	Interface
mod_include.c	Modules	1.1.1 & 1.2.4 1.2.4 & 1.2.6	Implntn, Structural Structural
mod_info.c	Modules	1.2.6 & 1.3.4	Interface, Structural
mod_negotiation.c	Modules	1.1.1 & 1.2.4 1.2.6 & 1.3.4	Implntn, Structural Structural
mod_rewrite	Modules	1.2.4 & 1.2.6 1.2.6 & 1.3.4	Structural Interface
mod_status.c	Modules	1.1.1 & 1.2.4 1.2.6 & 1.3.4	Interface Implntn, Structural
util.c	Core	1.1.1 & 1.2.4 1.2.6 & 1.3.4	Implntn, Structural Structural
util_md5.c	Core	1.2.6 & 1.3.4	Interface
util_script.c	Core	1.3.4 & 1.3.6	Interface

Table 5. Major Apache Web Server Evolution Patterns.

8. Interpretation of Results

The objective of this study was to identify those source code features that highly correlate with evolution patterns as manifested per evolution category (interface, implementation, structure) at the architecture level. We aimed for the identification of low-level primitives within a particular group that were highly correlated with features that described evolution at the architecture level.

Our analysis showed a number of low-level features that provide an accurate indication of the impact that changes have at the architectural level. Our empirical studies showed that the selected features exhibit low correlation with each other, thus each metric adds useful information. The source code features identified include: global variables used or defined (interface evolution), function calls (interface and implementation evolution), defined/used parameters passed by reference and by value (interface evolution), control flow graph (implementation evolution), local data usage (implementation and structure evolution).

Table 8 provides a detailed list of evolution distances computed. A comparison between the entries in Table 5 and Table 8 shows that Fan-Out and Fan-In indicate Interface changes in evolution. McCabe cyclomatic complexity and Halstead's metrics (we included only Halstead's program effort here) indicate Implementation changes in evolution. Finally, we observed Structural changes based on the Distance and Jaccard values. On the other hand, the numbers confirm what we saw on visual inspection. Using the following three selected examples, we are able to show our findings. A comparison between Table 8 and Table 5 for *http_config.c* between versions 1.1.1 and 1.2.4 shows a high value for the Distance metric that confirms the Structural changes in evolution. We also found a high value for McCabe complexity between versions 1.2.6 and 1.3.4 in *mod_access.c* which indicates the implementation changes in evolution. High values for Fan-In and Fan-Out

Feature Altered	Correlation using M1	Correlation using M2	Correlation using M3
Overall Distance	−0.12	−0.04	−0.07
Fan-In	0.11	0.05	0.08
Fan-Out	X	0.06	−0.03
McCabe	X	0.04	X
D-Flow	X	0.05	0.04
# of Stm.	0.13	X	0.07
LOC	0.47	X	X
# of Comment.	0.31	0.29	X

Table 6. Selected Source Code Features Correlated With Maintainability.

in *mod_info.c* between versions 1.2.6 and 1.3.4 are a good indication of interface changes in architecture. A summary of this discussion can be found in Table 7 for Apache. We obtained similar results for the Bash case study as well.

Evolution Category	Related Source Code Feature
Interface	Fan-In Fan-Out
Implementation	McCabe Halstead Comments
Structural	Distance Jaccard

Table 7. Evolution Category vs. Related Source Code Feature.

Finally, we correlated the distances obtained using the all identified source code features for both case studies (Bash & Apache) and the distances obtained using architectural level features indicate a high correlation (0.65 for Bash across four versions and 0.837 for Apache across five versions). This result indicates that the source code features we have considered provide a measure of change at the architecture level. The conclusion that can be drawn from the other tables is that the evolution categories can be interpreted based on the values of metrics computed from the source code.

As a final experiment, we measured how maintainability was affected by evolution. Maintainability was measured using the three models discussed in the sections above. Our results indicate a low correlation between the selected source code feature changes and maintainability. This result is illustrated in Table 6. The "X" in each entry of the Table 6 indicates that the correlation using a maintainability index (column) for a feature altered (base metric located in the same row) is not considered. As we described in Section 4.6, the three models depend explicitly upon some of the metrics which we listed. So we did not include the correlations for those metrics because the corresponding maintainability index is computed using the feature altered. Since this has biased the correlation results, we omit these entries.

The conclusion that can be drawn from these low correlations is that maintainability is a complex notion that can not be quantified accurately by just looking at how individual source code features have been altered from one version to another.

9. Conclusion

In this paper we discussed a technique for measuring software evolution. The primary focus of this work was on the identification of a set of source code features that can be used for the understanding of evolution patterns in a large software system. We have classified evolution patterns in three main categories; interface evolution, implementation evolution, and structure evolution. Our experimental results stem from the analysis of several versions of two large systems, namely the Apache Web server and the Unix shell Bash. Our findings indicated that interface evolution is best characterized both at the source code and architectural level by the Fan-In and Fan-Out per function or module. The implementation evolution is best characterized by the McCabe cyclomatic complexity measure, the Halstead effort metric, and the number of comments metric. Finally, structural evolution is best characterized by the overall distance metric, calculated as the weighted sum of the number of deletions, insertions and modifications of functions/libraries per module (file) between two different versions. Finally we assessed the hypothesis that maintainability can be predicted by examining how individual source code features are altered from one version to another. Our experiments indicated that maintainability due to evolution is a complex quantity that can not be predicted by individual source code features. Currently, we are working on the assessment of our methodology and the proposed evolution patterns using other legacy systems in co-operation of the IBM Toronto Laboratory, Center for Advanced Studies.

10. Acknowledgments

We would like to thank Prashant Patil of the University of Waterloo for his help in installing and using REFINE and Rigi. We would also like to thank Professor Richard Holt, also of the University of Waterloo for his feedback on a preliminary version of this paper.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

File Name	Ver. No.	Distance	Jaccard	Fan-In	Fan-Out	McCabe	Comments	Effort
alloc.c	1.2.4 & 1.2.6	27.84	0.05	−51	9	−0.04	2	3771292
	1.2.6 & 1.3.4	172.66	0.3	−23.41	149.16	0.12	95	1839352
buff.c	1.1.1 & 1.2.4	15.81	0.15	−25	15	−0.42	7	263245
	1.2.4 & 1.2.6	45.64	0.21	21	21	−0.3	24	1108760
	1.2.6 & 1.3.4	115.66	0.45	147	101	−1.1	45	−26107
http_config.c	1.1.1 & 1.2.4	431.76	0.2	−196	5	0.41	17	1334032
	1.2.4 & 1.2.6	20.21	0.04	2	5	−0.06	3	136799
	1.2.6 & 1.3.4	234.1	0.33	81	70	−0.05	34	673318
http_core.c	1.1.1 & 1.2.4	105.41	0.2	−65	31	0.13	4	805059
	1.2.4 & 1.2.6	31.52	0.02	−102	−10	0.03	3	4023333
	1.2.6 & 1.3.4	322.22	0.38	−191	130	0.51	47	3201147
http_main.c	1.1.1 & 1.2.4	192.63	0.21	−78	11	0.17	36	1019391
	1.2.4 & 1.2.6	112.31	0.13	−144	−19	0.1	63	774695
	1.2.6 & 1.3.4	339.77	0.52	19.26	121.4	0.24	253	1454764
http_protocol.c	1.1.1 & 1.2.4	512.33	0.33	−314.7	−0.94	0.66	55	2412540
	1.2.4 & 1.2.6	213.53	0.14	−289	−14	−0.07	30	1537560
	1.2.6 & 1.3.4	324.96	0.25	−161	25	0.24	21	881969
mod_access.c	1.2.6 & 1.3.4	22.67	0.13	−87	−7	1.63	9	292238
mod_alias.c	1.2.6 & 1.3.4	33.95	0.29	4	24.66	−0.02	3	83449
mod_cern_meta.c	1.2.6 & 1.3.4	24.06	0.5	11	18	−0.34	4	63237
mod_cgi.c	1.1.1 & 1.2.4	99.96	0.7	25	28	−2.55	5	388213
mod_dir.c	1.2.6 & 1.3.4	83.85	0.88	134	98	−0.859	−14	−1677215
mod_include.c	1.1.1 & 1.2.4	116.43	0.32	−27.51	33.15	3.02	18	4447567
	1.2.4 & 1.2.6	878.06	0.06	−258	−21	0.62	24	1631413
mod_info.c	1.2.6 & 1.3.4	191.32	0.5	11	11	−3.31	6	109057
mod_negotiation.c	1.1.1 & 1.2.4	115.12	0.39	−28	45.64	0.78	87	1602544
	1.2.6 & 1.3.4	116.14	0.14	−0.51	7	0.26	63	1479001
mod_rewrite.c	1.2.4 & 1.2.6	2024.36	0.07	−538	−129	0.31	75	4323709
	1.2.6 & 1.3.4	565.49	0.3	132.74	86.60	−0.25	44	1552086
	1.3.4 & 1.3.6	51.41	0.04	−65	10	−0.06	2	562233
mod_status.c	1.1.1 & 1.2.4	9.57	0.25	5	7	−2.36	5	23244
	1.2.6 & 1.3.4	1973.17	0.33	−301	−61	3.77	5	738303
util.c	1.1.1 & 1.2.4	58.68	0.21	−10.65	54	7.46	19	497504
	1.2.4 & 1.2.6	15.02	0.04	−10	7	−0.157	5	222769
	1.2.6 & 1.3.4	121.85	0.31	62	98	0.13	42	1128237
util_md5.c	1.2.6 & 1.3.4	13.43	0.39	21	14	0.04	0	3500
util_script.c	1.2.6 & 1.3.4	60.95	0.33	43.53	34.53	−0.96	23	185551

Table 8. Notable Changes in Features Across Versions.

[2] I. Bowman, R. Holt, and N. Brewster. *Linux as a Case Study: Its Extracted Software Architecture.* ICSE 99, May 1999.

[3] D. Coleman, B. Lowther, and P. Oman. The application of software maintainability models in industrial software systems. *The Journal of Systems and Software*, 29:3–16, 1995.

[4] P. Finnigan et al. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[5] D. Garlan and M. Shaw. *An Introduction to Software Architecture.* World Scientific Publishing Co., 1993.

[6] M. Halstead. *Elements of Software Science.* Elsevier North-Holland Inc., 1977.

[7] R. Holt. *Software Bookshelf: Overview and Construction.* Available at http://www-turing.cs.toronto.edu/pbs/papers/bsbuild.html, 1997.

[8] R. Holt and J. Pak. *GASE: Visualizing Software Evolution-in-the-Large.* WCRE 96: Working Conference on Reverse Engineering, November 1996.

[9] R. Kazman and J. Carrière. *View Extraction and View Fusion in Architectural Understanding.* 5th International Conference on Software Reuse, June 2-5, 1998.

[10] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. *IEEE Proceedings of WCRE97*, pages 44–54, 1997.

[11] P. Kruchten. The 4+1 views model of architecture. *IEEE Software*, pages 42–50, November 1995.

[12] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[13] H. Muller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem identification. *Software Maintenance and Practice*, 5:181–204, 1993.

[14] G. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *Computer*, 30(8):29–36, 1997.

[15] S. Muthanna. *Assessing Maintainability of Industrial Software Systems Using Design Level Metrics.* Master's Thesis, Department of Systems Design Engineering, University of Waterloo, 1997.

[16] P. Oman and J. Hagemeister. Constructing and testing of polynomials predicting software maintainability. *The Journal of Systems and Software*, 24:251–266, 1994.