SOFTWARE EVOLUTION,

VOLATILITY

AND

LIFECYCLE MAINTENANCE PATTERNS:

A LONGITUDINAL ANALYSIS

Evelyn J. Barry
Carnegie Mellon University
Graduate School of Industrial Administration

A Thesis

April 30, 2001

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

CARNEGIE MELLON UNIVERSITY

GRADUATE SCHOOL OF INDUSTRIAL ADMINISTRATION

DISSERTATION

Submitted in partial fulfillment of the requirements

for the degree of DOCTOR OF PHILOSOPHY

INDUSTRIAL ADMINISTRATION (INFORMATION SYSTEMS)

Title "SOFTWARE EVOLUTION, VOLATILITY AND LIFECYCLE MAINTENANCE PATTERNS: A LONGITUDINAL ANALYSIS"

Presented by EVELYN J. BARRY

Accepted by Sandra A. Slaughter 4/30/2001
Co-Chairs: Professor Sandra Slaughter Date

Professor Chris Kemmerer

Approved by the Dean

3-Feb-01
Douglas Dunn Dean Date

TABLE OF CONTENTS

Table of Contents
List of Figures
List of Tables
Acknowledgements
Abstract
Chapter 1: Introduction 1-1
Chapter 2: Literature Review: Software Evolution 2-1
Chapter 3:
Research Question 1- 3-1
A Multi-Dimensional Measurement of Software Volatility
Chapter 4:
Research Question 2- 4-1
Antecedents of Software Volatility
Chapter 5:
Research Question 3: 5-1
Characteristics of Software Evolution and Lifecycle Maintenance Outcomes
Chapter 6: Conclusion 6-1
Appendices:
Appendix A: A-1
Confirming Evidence for the Laws of Software Evolution

Appendix B: B-1
Data Codification and Sequence Analysis Methodology

Appendix C: C-1
Radial Graph Representations of Gamma Analyses
Using periodicity/deviation classification of software volatility

Appendix D: D-1
Radial Graph Representations of Gamma Analyses
Using periodicity/amplitude/deviation classification of software volatility

LIST OF FIGURES

Table =	*Title*	*page*
	Chapter 1	
1	Software Evolution and Lifecycle Maintenance	1-2
2	Overview of Three Research Questions	1-4
	Chapter 2	
1	Embedded Systems	2-2
	Chapter 3	
1	Evolution of System A	3-9
2	Evolution of System A - Month 1	3-10
3	Evolution of System A - Month 2	3-11
4	Evolution of System A - Month 3	3-12
5	System A Lifecycle Software Volatility	3-13
6	Hypothetical Idealized Stable System	3-32
7a	Lifetime Volatility System 7 - Periodicity	3-33
7b	Lifetime Volatility System 7 - Amplitude	3-33
7c	Lifetime Volatility System 7 - Deviation	3-34
8a	Lifetime Volatility System 23 - Periodicity	3-35
8b	Lifetime Volatility System 23 - Amplitude	3-35
8c	Lifetime Volatility System 23 - Deviation	3-36
	Chapter 4	
1	Antecedent of Software Volatility	4-8
2	Predictive Model for Periodicity	4-19
	Chapter 5	
1	Model of Predictors of Maintenance Outcomes	5-12
	Appendix B	
1	Portion of a Sample Change Log	B-2
2	Coding Flowchart	B-4, B-5
3	Labeling Year-Month by Maintenance Activity	B-8
4	Chronological Vector of Maintenance Activities	B-10
5	Phase Map From Figure 4 Sample Input Vector	B-12
6	Gamma Analysis and Precedence Map	B-13
7	Sample Gamma Analysis and Precedence Ordering	B-17
8	Radial Graph of Volatility Gamma Analysis	B-19

LIST OF TABLES

Table ≠	*Title*	*Page*
	Chapter 2	
1	Review of Software Evolution Research	2-2
2	Descriptive / Analytical Research on the Nature of Software Evolution and Lifecycle Maintenance	2-3
3	Laws of Software Evolution	2-4
4	Empirical Studies on Software Evolution	2-4
	Chapter 3	
1	Mathematical Properties of Proposed Measures	3-9
2	Evaluation of Proposed Measures	3-18
3	Correlations of NAmplitude and % of New Programs	3-25
4	Correlations of NPeriodicity and No. of Modifications	3-29
5	Correlations of NDeviation and the Coefficient of Variation	3-27
6	Correlations Supporting Discriminant Validity	3-28
7	Summary of Linear Regression Estimates for the Software Portfolio	3-30
	Chapter 4	
1	Antecedents of Software Volatility	4-24
2	Descriptive Statistics of Operational Variables	4-25
3	Correlations of Operational Variables	4-26
4	Regression Estimates for Drivers of Periodicity	4-27
5	Hypotheses Test Results	4-28
	Chapter 5	
1	Hypotheses to be Tested	5-13
2	Operational Explanatory Variables	5-17
3	Summary Statistics of Monthly Panel Data Set	5-18
4	Frequency Counts of Binary Variables - Monthly Panel	5-19
5	Summary Statistics of Quarterly Panel Data Set	5-19
6	Frequency Counts of Binary Variables - Quarterly Panel	5-19
7	Correlations for Monthly Panel Data Set	5-20
8	Correlations for Quarterly Panel Data Set	5-21
9	Prediction of Software Processing Error Rate Using Monthly Panel Data	5-22
10	Prediction of Maintenance Costs Using Quarterly Panel Data	5-23
11	Effect of System Complexity on Maintenance Outcomes	5-24
12	Effect of Lifecycle Maintenance Profiles on Maintenance Outcomes	5-25
13	Effect of Software Volatility on Maintenance Outcomes	5-26
14	Summary of Tests of Hypotheses	5-26
	Chapter 6	
1	Summary of Contributions	6-6

LIST OF TABLES (cont'd)

Table =	*Title*	*Page*
	Appendix A	
1	Laws of Software Evolution	A-1
2	Testing the Laws of Software Evolution	A-2
3	Operational Variables	A-3
4	Operational Variables and Expected Relationships	A-4
Law 1	Correlation (change, age)	A-4
	Testing for Normal Distribution	A-4
Law 2	Correlation (complexity, age)	A-5
	Testing for Normal Distribution	A-5
Law 3	Correlation (change size, age)	A-5
	t-tests	A-5
Law 4	Correlation (work rate, age)	A-6
	t-tests	A-6
Law 5	Correlation (change rate/capacity, age)	A-6
Law 6	Testing for Normal Distribution of Change Size	A-7
Law 7	Correlation (faults, age)	A-7
	Testing for Normal Distribution	A-7
	Appendix B	
1	Classification Scheme	B-3
2	High/Low Indicators and Volatility Classifications	B-15
3	Volatility Classification Ordering	B-16

This work is dedicated to my family for their encouragement and support of my efforts and for their patience during completion of my thesis. I want to thank my advisors, Dr. Slaughter and Dr. Kemerer, for their patience and determined efforts to keep me focused on achieving my best. I want to thank the faculty and graduate students who participate in GSIA's IS doctoral seminars for their friendly questions, constructive criticisms and enthusiastic encouragement.

This research was supported in part by the William J. Larimer Fellowship at the Graduate School of Industrial Administration, by faculty development grants from Carnegie Mellon University, by National Science Foundation grants CCR-9988227 and CCR-9988315, by a Research Proposal Award from the Center for Computational Analysis of Social and Organizational Systems, NSF IGERT, and by the University of Pittsburgh Katz Graduate School of Business Institute for Industrial Competitiveness.

This thesis is dedicated to my father, who knew I could do it all along.

SOFTWARE EVOLUTION, VOLATILITY AND LIFECYCLE MAINTENANCE PATTERNS: A LONGITUDINAL ANALYSIS

Thesis Abstract
Evelyn Barry
April 2001

Change is a constant in our world and information systems are no exception. Y2K required massive software modifications and the need for change continues in the application systems for the World Wide Web. Information systems often remain productive for many years, yet change so dramatically that current system characteristics no longer resemble their original implementation.

Software evolution is described as the dynamic behavior, growth and incremental change of information systems throughout their productive lives. Even though software maintenance represents 80% of the lifetime cost of an information system, the IS community has little scientific knowledge explaining how information systems evolve and the consequences of different evolutionary patterns. This work expands our understanding of software evolution by providing quantitative measurement and analysis of software evolution, examining its causes and its consequences.

Software evolution is characterized by *software volatility* and *lifecycle maintenance profiles*. These traits are used to address three research questions: (1) how can software volatility be conceptualized and measured? (2) what are the antecedents of software volatility? and (3) are software volatility and lifecycle maintenance profiles determinants of lifecycle maintenance outcomes?

Formal criteria are applied to rigorously define, evaluate and validate three measures of software volatility: *amplitude*, *periodicity* and *deviation*. Empirical data demonstrate the contingent, discriminant and predictive validity of these measures.

Conceptual models for the second and third questions are developed and empirically tested to analyze the relationships of software evolution to information systems and their lifecycle maintenance processes. Hypotheses are tested by panel regressions based on empirical data from a detailed 20-year maintenance log of software modifications in a portfolio of 23 information systems.

This thesis makes several contributions. A rigorous set of evaluation criteria for software measurement is developed and applied. These analyses describe the relationships connecting amplitude, periodicity and deviation with lifecycle maintenance patterns, and lifecycle maintenance outcomes of processing errors and maintenance costs. These results are strengthened by use of a unique empirical data set ten-times larger than previous longitudinal studies of software evolution. These new insights into software evolutionary processes can be used to advantage by IS researchers and managers.

INTRODUCTION

As we begin the 21st century there is a renewed interest in long-term perspectives for many of the management practices currently in use. We have now had electronic computers for over half a century, and with them, the added responsibility of software management. Many people were dumbfounded by the possibility of a Y2K problem. With the rapidity of changes in information technology, how could we still be dealing with software written in some cases more than 20 years ago?

What IS professionals have long realized, and what many others are now beginning to understand, is that many information systems remain productive for decades. It is estimated that the average enterprise general ledger application system in Fortune 1000 companies is 15 years old (Kalakota and Whinston, 1996, p. 390). The Y2K problem highlighted the continual investment required of organizations to maintain their systems.

With the recognition that information systems are long-lived comes the necessity to understand longitudinal changes occurring in those systems. Information systems must continue to operate efficiently and effectively in dynamic competitive environments. To perform at satisfactory levels, software systems must periodically be adjusted to model changes occurring around them. Software changes may reveal errors of omission or miscommunication, or be the result of requirements for additional functionality. Whether these changes are done to correct flaws in existing code, adapt to the environment, or add functionality, they are generally classified as lifecycle software maintenance.

Software maintenance activities span a system's productive life and can consume as much as 80% of the total effort expended on a system during its lifetime (Bennett, 1996). While researchers recognize the importance of lifecycle maintenance activities and their outcomes, relatively little empirical research has been conducted that examines the type and extent of changes taking place.

The longitudinal perspective required for analysis of lifecycle maintenance leads us to the process of *software evolution.* Belady and Lehman (1976) define software evolution as "... the dynamic behavior of programming systems as they are maintained and enhanced over their life times." Software evolution is of increasing importance as systems in organizations become longer-lived. We refer to those changes as lifecycle maintenance.

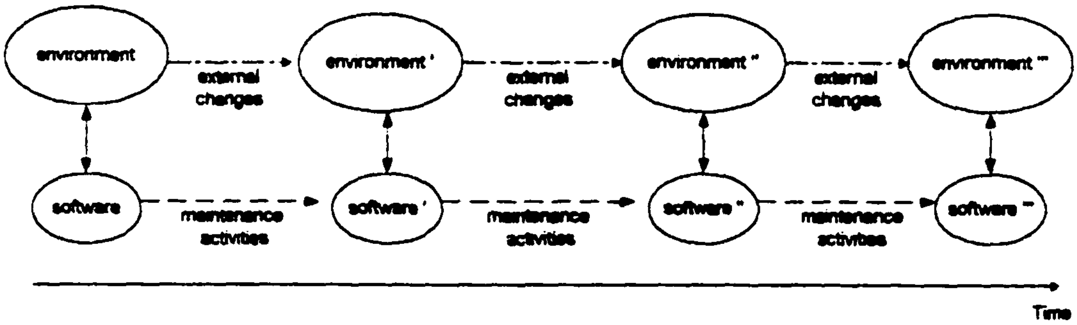


Figure 1: Software Evolution and Lifecycle Maintenance

We observe that lifecycle maintenance activities are the driving force in the longitudinal transformations occurring within an information system. Not every information system displays the same evolutionary changes in behavior. What accounts for these differences? How do these differences affect lifecycle maintenance outcomes such as processing errors and maintenance costs? Do differences in information system behavior affect maintenance outcomes?

The evolutionary process of software change can be described by analyzing software volatility, i.e. the amount and intensity of software change. Some software systems are constantly undergoing major modifications and others remain untouched for months and years at a time. By developing a measure of software volatility and identifying its antecedents, we can expand our understanding of software evolution. We use our measures of software volatility and historical patterns of lifecycle maintenance activities to describe software evolution. We then investigate the relationship between software evolution and software maintenance outcomes. We combine our measures of software volatility with a detailed taxonomy of lifecycle maintenance activities to describe software evolution. We investigate these factors to see what effect they have on outcomes such as processing errors and maintenance costs. This research project addresses three research questions by using a software evolutionary perspective to study longitudinal transformations that information systems undergo during their lifetimes.

1. *How can software volatility be conceptualized and measured?*
2. *What are the antecedents of software volatility?*
3. *Using software volatility and lifecycle maintenance profiles as descriptors of software evolution, are the characteristics of software evolution determinants of lifecycle maintenance outcomes?*

These research questions now become pieces of a puzzle. As shown in Figure 2, each research objective is one step toward painting a complete picture of the roles lifecycle maintenance activities and software volatility play in the evolutionary transformations that occur during the productive lives of information systems.

The next chapter presents a discussion of software evolution with a brief review of relevant literature. The research questions are presented and empirically tested in each of the following three chapters. The results of this research are then summarized in Chapter 6. Additional empirical results and a detailed description of data conversion processes are included in the appendix.

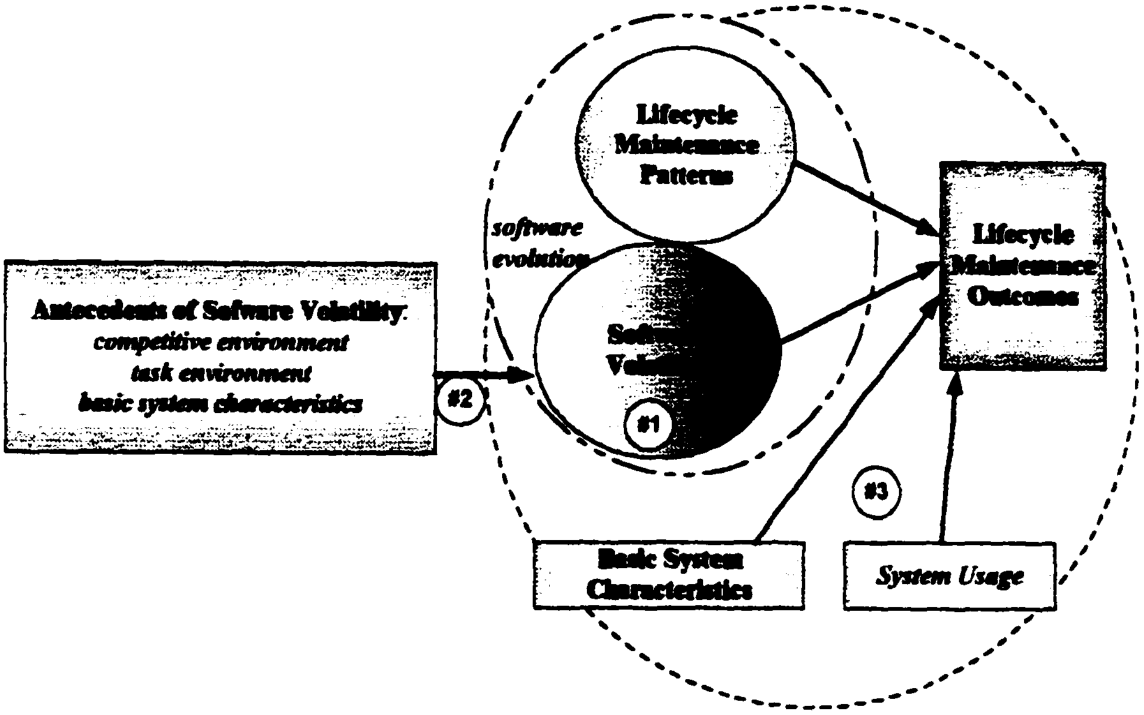

Lifecycle Maintenance Patterns
software evolution
Antecedents of Sofware Volatility:
competitive environment
task environment
basic system characteristics
#2
Soft
Vol
#1
Lifecycle Maintenance Outcomes
#3
Basic System Characteristics
System Usage

Figure 2: Overview of Three Research Questions

REFERENCES

Belady, L.A., and Lehman, M.M., "A Model of Large Program Development", IBM Systems Journal, 1976, No. 3, pp. 225-252.

Bennet, Keith, "Software Evolution: Past, Present and Future", Information and Software Technology, Nov. 1996, Vol. 39, No. 11, pp. 673-380.

Kalakota, R., and Whinston, A.B., Electronic Commerce: A Manager's Guide, Addison-Wesley, Reading, MA, 1996.

CHAPTER 2:

LITERATURE REVIEW - SOFTWARE EVOLUTION

PRIOR RESEARCH ON SOFTWARE EVOLUTION

As with discussions of the evolution of biological entities, analyses of software evolutionary processes must account for both the inherent characteristics of software systems, and the effects of environmental influences on them. Software changes occur for a variety of reasons. Some of these are in response to environmental changes and some are the result of a natural growth in user expectations and functional demands placed on the application.

Prior studies of software evolution assume information systems are open systems, embedded in their respective organizations. Because they are open systems exchanging information with their environment, information systems are open systems, growing and changing during their productie lives in response to their environment. (Scott, 1992; Morgan, 1997). A shown in Figure 1 embedded systems are influenced and changed by their environment, and in turn they influence and change their environment (Lehman and Belady, 1985; Pfleeger, 1998).

Researchers have analyzed software evolution for over three decades. (See Tables 1 and 2) Based on a series of empirical and analytical studies, researchers, Lehman et al., have developed eight laws of software evolution for embedded systems. (Lehman and Belady, 1985, , et al., 1997). (See Table 3) Prior to work by Kemerer and Slaughter (1997, 1999) none of these empirical studies examined data covering more than four years of software evolution. (See Table 4.)

Current research on software evolution is headed in a number of different directions. As reported in a recent Workshop on Empirical Studies of Software Development and Evolution, software evolution is providing a theoretical foundation for analysis of reverse engineering technologies and new perspectives on cost estimation tools. In addition further work is being done on the FEAST/2 (Feedback Evolution and Software Technology) project, further investigating Lehman's eighth law of software evolution, the Law of System Feedback (Harrison, et al., 1999).

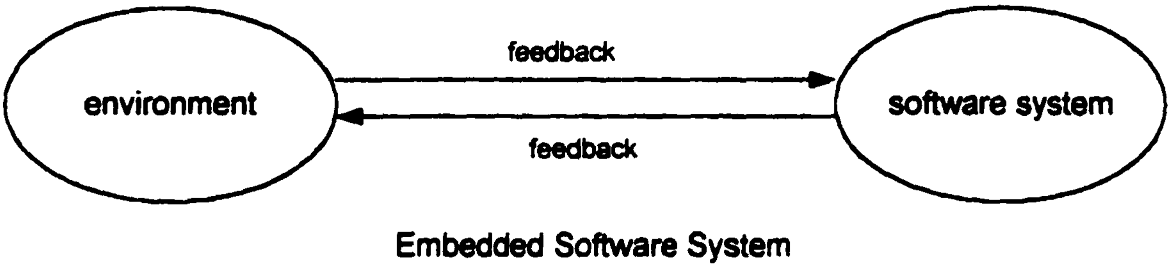


Figure 1: Embedded Systems

Author	year	Title
Bennet	1996	Software Evolution: Past, Present and Future
Schneidewind	1987	The State of Software Evolution
Kemerer	1995	Software Complexity and Software Maintenance: A survey of empirical research
Kemerer & Slaughter	1997	Methodologies for Performing Empirical Studies: Report from the International Workshop on Empirical Studies of Software Maintenance
Cote, Bourque, Oligny & Rivard	1988	Software Metrics: An Overview of Recent Results
Belady	1979	On Software Complexity

Table 1: Reviews of Software Evolution Research

Author	Year	Title	Dependent variable(s)	Independent variable(s)	Conclusion
Lehman	1998	Software's Future: Managing Evolution			Discussion
Lehman	1996	Feedback in the Software Evolution Process	Feedback learning system; success of forward path innovations	system type, software process	benefits from innovative changes to forward path methods limited by feedback occurring in software process
Brooks	1995	<u>Mythical Man Month</u>			general software process management techniques
Perry	1994	Dimensions of Software Evolution			discusses 3 dimensions: domain (real world and its abstractions), experience (from feedback,experimenta-tion), process (methods, technologies, organizations)
Yau, Nicholl, Tsai, & Liu	1988	An Integrated Life-Cycle Model for Software Maintenance			describes software maintenance in 4 phases and concentrates on interphase relationships; describes software in terms of control flow, data flow and data structure; permits analysis of basic properties of software system throughout life-cycle
Lehman	1984	Program Evolution			software process can be studied in its environment - also detailed discussion of SPE classification of systems; discussion of step paradigm for software process
Lehman	1980	On Understanding Laws, Evolution and Conservation in the Large-Program Life Cycle			explain and support 5 laws of software evolution
Woodside	1980	A Mathematical Model for the Evolution of Software	size &complexity of software; efficiency of software process	structural work effort, non-structural work effort, release no., modules produced	develop mathematical models expressing the laws of software evolution; gives laws internal validity
Belady	1978	Staffing Problems in Large Scale Programming			discussion of the type and sequence of work done in software development, and the iterative nature of the process
Lehman	1977	Human Thought and Action as an Ingredient of System Behavior			top-down analysis of the software process; discussion relies on systems science

Table 2: Descriptive / Analytical Research on the Nature of Software Evolution and Lifecycle Maintenance

Laws of Software Evolution	Description
Law of Continuous Change	Systems must continually adapt to the environment to maintain satisfactory performance
Law of Increasing Entropy (later renamed Law of Increasing Complexity)	As systems evolve they become more complex unless work is specifically done to prevent this breakdown in structure
Law of statistically smooth growth (also called the Law of Self Regulation)	The software evolution processes are self-regulating and promote globally smooth growth of an organization's software
Law of invariant work rate (also called Law of Conservation of Organizational Stability)	The organization's average effective global activity rate is invariant throughout system's lifetime
Law of conservation of familiarity	Incremental growth rate of a system is constant to conserve the organization's familiarity with the software.
Law of continuing growth	Functional content of systems must be continually increased to maintain user satisfaction
Law of declining quality	System quality declines unless it is actively maintained and adapted to environmental changes
Law of system feedback	Software evolutionary processes must be recognized as multi-level, multi-loop, multi-agent feedback systems in order to achieve system improvement.

Table 3: Laws of Software Evolution (Lehman, et al., 1997)

Author	Year	Title	Data
Kemerer and Slaughter	1999	An Empirical Approach to Studying Software Evolution	20 years of software modifications for 23 software systems
Kemerer and Slaughter	1997	Determinants of Software Maintenance Profiles: An Empirical Investigation	5488 modifications in 621 software modules in five application systems; approximately 9 years of software changes
Lehman, et al.	1997	Metrics and Laws of Software Evolution: The Nineties View	21 releases of a financial software package
Basili, et al.	1996	Understanding and Predicting the Process of Software Maintenance Releases	25 releases of 10 different software systems
Gefen and Schneberger	1996	The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications	29 months of software problem reports
Cook and Roesch	1994	Real-Time Software Metrics	10 versions of real-time German switching software released over 18 months
Yuen	1987	A Statistical Rationale for Evolution Dynamics Concepts	Modules from OS 360, OMEGA, Executive, BD, B, DOS, CCSS systems
Yuen	1985	An Empirical Approach to the Study of Errors in Large Software Under Maintenance	19 months of data for 5000 'component', 3000 KLOC
Belady and Lehman	1976	A Model of Large Program Development	21 user-oriented releases

Table 4: Empirical Studies of Software Evolution

REFERENCES

Basili, V., Briand, L., Condon, S., Kim, Y.M., Melo, W., and Valett, J., "Understanding and Predicting the Process of Software Maintenance Releases", 18th International Conference on Software Engineering, 1996, Berlin, Germany.

Belady, L.A., "Evolved Software for the 80's", Computer, Vol. 12, No. 2, Feb. 1979, pp. 79-82.

Bennet, K., "Software evolution: past, present and future", Information and Software Technology, Vol. 39, No. 11, Nov. 1996, pp. 673-680

Brooks, F.J., The Mythical Man-Month, Addison-Wesley Publishing Co., 1995.

Cook, C.R. and Roesch, A., "Read-Time Software Metrics", Journal of Systems and Software, Mar. 1994, Vol. 24, No. 3, pp. 223-237.

Cote, V., Bourque, P., Oligny, S., Rivard, N., “Software Metrics: An Overview of Recent Results”, The Journal of Systems and Software, Vol. 8, No. 2, March 1988, pp. 121-131.

Gefen, D., and Schneberger, S.L., "The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications", Proceedings of the IEEE Conference on Software Maintenance, 1996, Monterey, CA.

Harrison, R., Badoo, N. Barry, E., Biffl, S., Parra, A., Winter, B., and Wuest, J., "Workshop and Conference Reports: ESSDE'99 Working Group Report on Directions and Methodologies for Empirical Software Engineering Research", Empirical Software Engineering, December 1999, Vol. 4, No. 4, pp. 405-410.

Kemerer, C.F., "Software Complexity and Software Maintenance: A Survey of Empirical Research", Annals of Software Engineering, Vol. 1, Sept. 1995, pp. 1-22.

Kemerer, C.F. and Slaughter, S.A., "Determinants of Software Maintenance Profiles: An Empirical Investigation", Journal of Software Maintenance, Vol. 9, 1997, pp. 235-251.

Kemerer, C.F. and Slaughter, S.A., 1999, "An Empirical Approach to Studying Software Evolution", IEEE Transactions on Software Engineering, Vol. 25, No. 4, pp. 493-509.

Lehman, M.M., " Human Thought and Action as an Ingredient of System Behavior", Encylcopedia of Ignorance, R. Duncan and M. W. Smith (Eds), Pergamon Press, Oxford, 1977.

Lehman, M.M., "On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle", Journal of Systems and Software, Vol. 1, No. 3, 1980, pp. 213-221.

Lehman, M.M., "Program Evolution", Information Processing and Management, Vol. 20, 1984, pp. 19-36

Lehman, M.M., and Belady, L.A., Program Evolution: Processes of Software Change, Academic Press, London, 1985.

Lehman, M.M., "Feedback in the Software Evolution Process", Information and Software Technology, Vol. 39, No. 11, Nov. 1996, pp. 681-686.

Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E. and Turski, W.M., "Metrics and Laws of Software Evolution - The Nineties View", Metrics '97, the Fourth International Software Metrics Symposium, 1997, Albequerque, NM

Lehman, M.M., "Software's Future: Managing Evolution", IEEE Software, January-February, 1998, pp. 40-44.

Morgan, G., Images of Organization, Sage Publications, Thousand Oaks, CA, 1997.

Perry, D.E., "Dimensions of Software Evolution", IEEE Conference on Software Maintenance, 1994, IEEE.

Pfleeger, S., "The Nature of System Change", IEEE Software, Vol. 15, No. 3, May-June 1998, pp. 87-90.

Schneidewind, N.F., "The State of Software Evolution", IEEE Transactions on Software Engineering, Vol. 13, No. 3, March 1987, pp. 103-110.

Scott, R.W., Organizations: Rational, Natural, and Open Systems 3rd Edition, Prentice Hall, Englewood Cliffs, NJ, 1992.

Woodside, C.M., "A Mathematical Model for the Evolution of software", Journal of Systems and Software, Vol. 1, No. 4, 1980.

Yau, S.S., Nicholl, R.A., Tsai, J., Liu, S., "An Integrated Life-Cycle Model for Software Maintenance", IEEE Transactions on Software Engineering, Vol. 14, No. 8, Aug. 1988, pp. 1128-1144.

Yuen, C.H., "An Empirical Approach to the Study of Errors in Large Software Under Maintenance", 2nd IEEE Conference on Software Maintenance, 1985, Washington, D.C.

Yuen, C.H., "A Statistical Rationale for Evolution Dynamics Concepts", Proceedings of the Conference on Software Maintenance, 1987, Austin, TX.

CHAPTER 3:

RESEARCH QUESTION 1 -

A MULTIDIMENSIONAL MEASUREMENT OF SOFTWARE VOLATILITY

1 INTRODUCTION

Everyone has heard the adage "The only thing constant is change". It is no surprise, therefore, that change is also a constant in software systems. This is probably more true of software systems than other phenomenon due to their often perceived ease of change. In the software engineering community we dealt with change while working on software modifications for Y2K, and we continue to deal with rapid changes while supporting software for the World Wide Web. Not all software systems[1] change in the same way or at the same rate. Some software systems are constantly undergoing major modifications, while others remain untouched for months and years at a time. What accounts for these differences and how can they be analyzed? Identification and understanding of these differences in system evolution can lead to improved abilities to engineer and manage software systems.

Exact definitions and measurement of research variables are essential before more in-depth analysis can be conducted. As defined by Belady and Lehman, software evolution is "the dynamic behavior of programming systems as they are maintained and enhanced over their life times" (Belady and Lehman, 1976). Some researchers have expanded this definition to concentrate on lifecycle maintenance processes, using "evolution" as a synonym for "maintenance" or "modification" (Van Horn, 1980). This view changes the research emphasis to examine processes people use to develop software systems and to follow systems as they progress through iterative releases (Lehman and Ramil, 1999).

In this study, we draw upon Belady and Lehman's original definition of software evolution. In doing so, we re-emphasize the general systems approach to understanding

[1] In this discussion *system* refers to a group of related programs or modules that function together toward a common purpose. We refer to *programs* as elements of a software system. A program is a set of ordered computer commands assembled to accomplish a specific task.

the nature of software systems. As in Kemerer and Slaughter (Kemerer and Slaughter, 1999), this work examines post-implementation system behavior. For over two decades Lehman *et al.* (Belady and Lehman, 1976; Lehman and Belady, 1985) have postulated and tested a series of laws of software evolution describing proposed universal aspects of software system behavior. These laws describe software behavior as dynamic characteristics of change, entropy, growth and quality. *The Law of Continuous Change* was among the first of these laws to be formally stated and is based on experiences of practitioners working with hardware and software (Lehman and Belady, 1985). To understand the dynamic behavior of individual software systems, additional research is needed to build on descriptions of universal behavior and seek explanation and understanding of variations in behavior (Thompson, 1967). While we recognize that all software systems change throughout their productive lives, they do not all change in the same way, or at the same pace. This research identifies *software volatility* as a dynamic characteristic of system behavior. In so doing, this work emphasizes the longitudinal nature of evolutionary processes allowing comparison of the variations in behavior across systems and over time.

Key to understanding the variations in software system is the ability to measure important characteristics of software lifecycle evolution, including software volatility. In applying engineering discipline to the endeavor of software engineering, measurement of work products is considered essential (Tian and Zelkowitz, 1992). This discipline should apply to studies of software *behavior* as well. Hence, the objective of this research is to define a direct measure of software volatility, evaluate the proposed measurement function, and empirically provide evidence that the new metric can be collected. Software volatility measures can then be used to improve our theoretical understanding of software evolution, and to assist practitioners in managing long-lived software systems.

Because software programs generally do not change unless someone directly alters source code, previous measures of software volatility have concentrated on counts of software modifications. For example, versioning is a count of new editions of a software system, e.g. with the releases of the commercial software product *MS Windows NT* each successive version number may represent different amounts of additional or modified source code. With the versions of *Windows NT* implemented between 1993 and 2000, the software system grew from 6.1M to 30M lines of code (LOC) (Hamm and Port, 1999). However, there are typically irregular time intervals and changes in software size between releases. Versioning, therefore, provides only a relatively crude measure of changes taking place. It marks major levels of change, but fails to track the size of change or the periodicity of change.

Using a simple count of software modifications for a measure of volatility fails to consider how often changes occur. If two software systems have both had the same number of modifications, but one is 2 years old and the other is 5 years old, the older system would intuitively be considered the less volatile of the two, *ceteris paribus*. However, a simple rate of software modification over time may still not adequately describe software volatility. Consider two systems with different patterns of lifecycle maintenance such that one system is modified at the end of each month, and the other annually undergoes 12 modifications at one time. The two systems each change at the rate of 12 modifications per year, yet one is in a constant state of flux, and the other remains unchanged for 11 of every 12 months. Hence, more descriptive measures of software volatility must include a measure of the time between source code modifications. System size must also be considered. If two systems report 10 changes of equal size per month, but one system has 500 programs and the other has 10 programs, the former is intuitively less volatile.

We propose a 3-dimensional measure of software system volatility. The first dimension is a measure of software change size or *amplitude*. The second is a measure of how often changes occur, or *periodicity*. By using amplitude and periodicity, we can describe software volatility with a smooth sine curve similar to that used to describe physical systems, e.g. sound waves (Bueche, 1969). However, software system behavior is unlikely to be as consistent as a physical system. A more precise measure of software volatility will include a third dimension, a measure of how closely software volatility follows the implied cyclical pattern. As in the studies of environmental volatility by Wholey and Brittain (1989), we add a third dimension, *deviation*, to indicate how closely system behavior follows the cyclical patterns described by periodicity. Measurements for each of these dimensions can be calculated periodically throughout the productive life of a software system and analyzed to describe changes in system behavior as it evolves.

In section 2 we first briefly review prior research that is relevant to the measurement of software volatility and then formally define a 3-dimensional measure of software system volatility. In sections 3 and 4 we evaluate these dimensional measures and provide empirical support to validate the proposed metrics. This work contributes to our understanding of the lifecycle transformations of software systems by maintaining a system-level perspective while analyzing the full extent of a system's productive life span. Software lifecycle changes have traditionally been tracked at the program level. To more fully understand lifecycle transformations occurring at the level of software systems a more comprehensive approach is required. Software volatility should measure multiple aspects of the changes occurring in a software system throughout its lifecycle. A system-level measure of software volatility can be used for descriptive analysis of system behavior. In addition, this project lays the groundwork for building theories to explain and predict software volatility and analyze its contribution to software product attributes and lifecycle maintenance processes and their outcomes.

2 A MULTI-DIMENSIONAL DEFINITION OF SOFTWARE VOLATILITY

In this section we define the attributes of software volatility and propose measurement functions for each. In section 3 we develop a set of formal evaluation criteria for measurement functions and assess our proposed measures. In section 4 we provide empirical evidence of the validity of these metrics, and in sections 5 and 6 we discuss the implications and application of these measures of software volatility

We start by clearly defining each attribute being measured. The use of a natural language definition in addition to precise mathematical terminology is essential in developing a consensus about what is being measured and how it should be done Finkelstein and Leaning, 1984; Xia, 1999). For wide application and adoption of new measures it is also important that such measures be software programming language and technology independent (Churcher and Shepperd, 1995).

Previous empirical studies of software evolution have measured longitudinal changes in software product attributes and compared those values at different points in time (Banker and Slaughter, 2000). To understand the evolution of software systems and analyze their dynamic behavior, we need to analyze characteristics of software *behavior*.

2.1 Prior Studies of Volatility

While a number of researchers have examined the problem of measurement of software product and software process attributes, there is little empirical research that measures dynamic characteristics of software behavior, particularly, software volatility. Existing studies tend to use basic counts of software modifications as a direct measure (Banker and Slaughter, 2000). In contrast, a predictive model for the logical stability of software is based on other software product attributes (Yau and Collofello, 1980; 1985). In that work the dependent variable is expressed as a rating of the ripple effect, i.e. the effect of changes in other system programs felt by the program being evaluated.

To develop additional perspective on software volatility we turn to other research to see how others have measured change. Some researchers have studied other types of volatility using counts of change incidents Snyder and Glueck, 1982; Dess and Beard, 1984; Stroh, Baumann and Reilly, 1996). Schneidewind's study of process stability examined trends by first calculating a change metric, and then analyzing the trend function to indirectly measure stability (Schneidewind, 1999). Li, Etzkorn and Talburt (2000) examine process instability with empirical measures of object-oriented software evolution during the design phase.

We propose a direct measure of the multi-dimensional aspects of the volatility of software systems. Organizational theorists Wholey and Brittain (1989) describe environmental variation with three dimensions: amplitude, frequency and predictability of variation. A primary premise of our work is that software systems, particularly application systems, model their environments. As the business and technological environment grows and changes, software systems must also change (Lehman and Belady, 1985). This suggests that dimensional characteristics of environmental volatility measures could be adapted to describe software volatility. We define 3 dimensions of software volatility: amplitude, periodicity and deviation. Amplitude describes the magnitude of change and periodicity measures the time interval between software modifications. These two characteristics imply a smooth pattern of software modifications. While many naturally occurring physical phenomena may be described this way, software systems are unlikely to be so well behaved. We need a third dimension, deviation, to describe how closely a system's behavior follows the pattern implied by amplitude and periodicity.

2.2 AMPLITUDE

Amplitude measures the *size* of software modifications. Traditional measures for software size include lines of code (LOC), function point counts, token counts, equivalent size metrics, entity counts, percentages of changed programs and object-oriented methods (Boehm, 1997; Albrecht and Gaffney, 1983; Chidamber and Kemerer, 1994).

Amplitude can be measured as the sum of the size of all modifications made to a software system. Amplitude can be measured for each time period t, as:

$$\text{Amplitude}_t = \sum_{j=1}^{N_t} \text{size}(\text{modification}_j)$$

where N_t is the number of modifications in time period t.

We can use any of the previously validated measures of software size for our measure Amplitude_t. Division of Amplitude_t by the total size of the system creates a bounded scale invariant measure. We refer to this as normalized amplitude, NAmplitude_t.[2]

$$\text{NAmplitude}_t = \text{Amplitude}_t \,/\, (\text{total size of software system at end of time period } t)$$

NAmplitude_t is the normalized measure of amplitude for time period t.

2.3 Periodicity

Periodicity measures time since software modifications (TSM). Manufacturing and production researchers define Mean Time Between Failures (MTBF) as the total unit-hours of operation divided by the total number of failures (Gaither, 1990). MTBF is calculated as a single value for the entire product lifecycle. Studies of software reliability find the Mean Time to Failure (MTTF) as the expected time the next software failure will be observed (Lyu, 1995). By definition, MTBF and MTTF are concerned only with failures or breakdowns.

[2] We use the term *normalize* to refer to a mathematical operation that eliminates units of measurement, i.e. we are creating normalized measures to show relative measures with respect to the maximum possible value. Measures normalized in this fashion will be scale invariant and bounded between 0 and 1.

We are seeking a measure of the time between software modifications, regardless of their purpose, e.g. corrective, adaptive or enhancement. Time since software modification, TSM, is the time (measured as days, weeks, months, etc.) elapsed since the previous modification.

TSM_{jt} = TSM for change event j in time period t,
where t = system age.

Periodicity is the mean TSM for a system during time period t.

$$Periodicity_t = \frac{1}{N_t}\sum_{j=1}^{N_t} TSM_{jt},$$
where N_t = total number of change events during time period t

To make comparisons of Periodicity_t across systems of different ages, we normalize Periodicity_t by the number of time periods a system has been in existence at the end of time period t. Thus, normalized Periodicity for time period t, NPeriodicity_t, is defined as:

$$\text{NPeriodicity}_t = \text{Periodicity}_t / t.$$

2.4 Deviation

Deviation is the variance of the TSM_j for the change events occurring in time period t. Deviation should express the variation in both amplitude and periodicity. By definition our measure of amplitude, NAmplitude_t will vary over a software system's lifecycle, but not within each time period t. Therefore, the variance in amplitude will not contribute to deviation. However, periodicity, measured by TSM_{jt}, will vary within time period t. We define Deviation as the variance of the TSM_{jt} ($\text{variance}(TSM_j)_t$).. Boundedness can be obtained by calculating the variance of normalized $TSM_{jt,}$ i.e. $NTSM_t = (TSM_{jt})/t$. We refer to the normalized variance as NDeviation_t.

$$\begin{aligned}\text{NDeviation}_t &= \text{variance}(NTSM_t)\\ &= \text{variance}(TSM_{jt}/t)\\ &= (1/t^2)\text{variance}(TSM_{jt})\end{aligned}$$

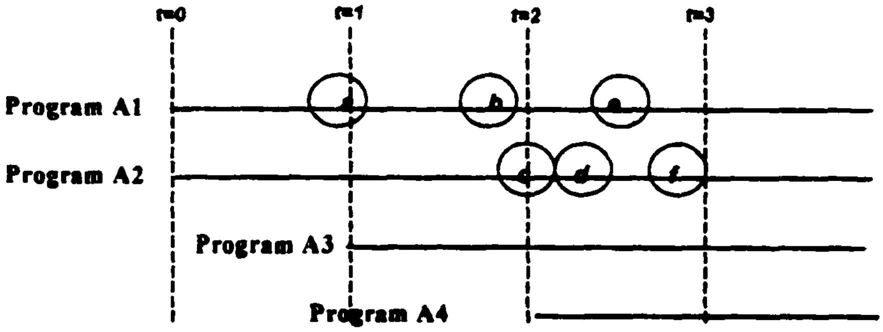
The mathematical properties of the elements of amplitude, periodicity and deviation are shown in Table 1.

Construct element	∈ Real Numbers, for all j, t;	≥ 0, for all j, t	≤ 1, for all j, t
Amplitude	Yes	Yes	No
Namplitude	Yes	Yes	Yes
Periodicity	Yes	Yes	No
NPeriodicity	Yes	Yes	Yes
Deviation	Yes	Yes	No
NDeviation	Yes	Yes	Yes

Table 1: Mathematical Properties of Proposed Measures

2.5 Example Calculation of Software Volatility Measures:

We demonstrate calculation of these measures with the following hypothetical example. Assume system A is implemented with two programs, A1 and A2. The evolution of system A is shown in Figure 1.



***a - f* are change events indicating modifications to programs A1 and A2.**

Figure 1: Evolution of System A

For purposes of this example, assume the programs in system A all use a common programming language, and therefore lines of code (LOC) is an appropriate measure of software size. Assume Program A1 has 1000 LOC and A2 has 1500 LOC. At the end of the first month one software modification (change event *a*) is made to program A1. The modification size is 50 LOC. As shown in Figure 2, we calculate NAmplitude$_1$ = 50/2500 = 0.02, NPeriodicity$_1$ = 1.0/1.0 = 1.00, and NDeviation$_1$ = variance({1.0}) / 1.0 = 0.00.

End of Month 1	Program size (LOC)	Modification size (LOC)	Modification time since modification (in months)
A1	1000	50	1.0
A2	1500		
Total	2500	50	1.0
Amplitude (NAmplitude)	0.02		
Periodicity (NPeriodicity)	1.00		
Deviation (NDeviation)	0.00		

Figure 2: Evolution of System A - Month 1

At the beginning of the second month of operation program A3 is added to the system. Program A3 has 1200 LOC. During the second month of operation (0.8 through the month) a modification (change event *b*) involving 30 LOC is completed on program A1. At the end of the second month program A2 is modified for the first time (change event *c*). The modification involves 500 LOC. As shown in Figure 3, we calculate NAmplitude$_2$ = 1730/3700 = 0.47, NPeriodicity$_2$ = mean({0.8, 2.0, 0.0}) / 2 = 0.47, and NDeviation$_2$ = variance({0.8, 2.0, 0.0}) / 2^2 = 0.25.

End of Month 2	Program size (LOC)	Modification size (LOC)	Modification time since modification (in months)
A1	1000	30	0.8
A2	1500	500	2.0
A3	1200	1200	0.0
Total	3700	1730	2.8
Amplitude (NAmplitude)	0.47		
Periodicity (NPeriodicity)	0.47		
Deviation (NDeviation)	0.25		

Figure 3: Evolution of System A - Month 2

Program A4 is added to the system at the beginning of month 3. Program A4 has 500 LOC. Halfway through the third month program A1 was modified by a software modification of 200 LOC (change event *e*). Two modifications are completed on program A2 with 100 LOC and 50 LOC, respectively. These modifications are completed on day 10 and day 25 of the month, respectively (change events *d* and *f*). As shown in Figure 4, we calculate NAmplitude$_3$ = 850/4200 = 0.20, NPeriodicity$_3$ = mean({0.7,0.3,0.5, 0.0})/3 = 0.125, and NDeviation$_3$ = variance({0.7,0.3,0.5, 0.0})/3^2 = 0.01.

End of Month 3	Program size (LOC)	Modification size (LOC)	Modification time since modification (in months)
A1	1000	200	0.7
A2	1500	100	0.3
		50	0.5
A3	1200		
A4	500	500	0.0
Total	4200	850	1.5
Amplitude (NAmplitude)	0.20		
Periodicity (NPeriodicity)	0.125		
Deviation (NDeviation)	0.01		

Figure 4: Evolution of System A - Month 3

The three dimensional measures of volatility describe changing behavior in the system. In this example we see that over time, amplitude is becoming larger (NAmplitude) and periodicity (NPeriodicity) is becoming shorter. System A starts as a "well-behaved" system with low deviation (NDeviation).

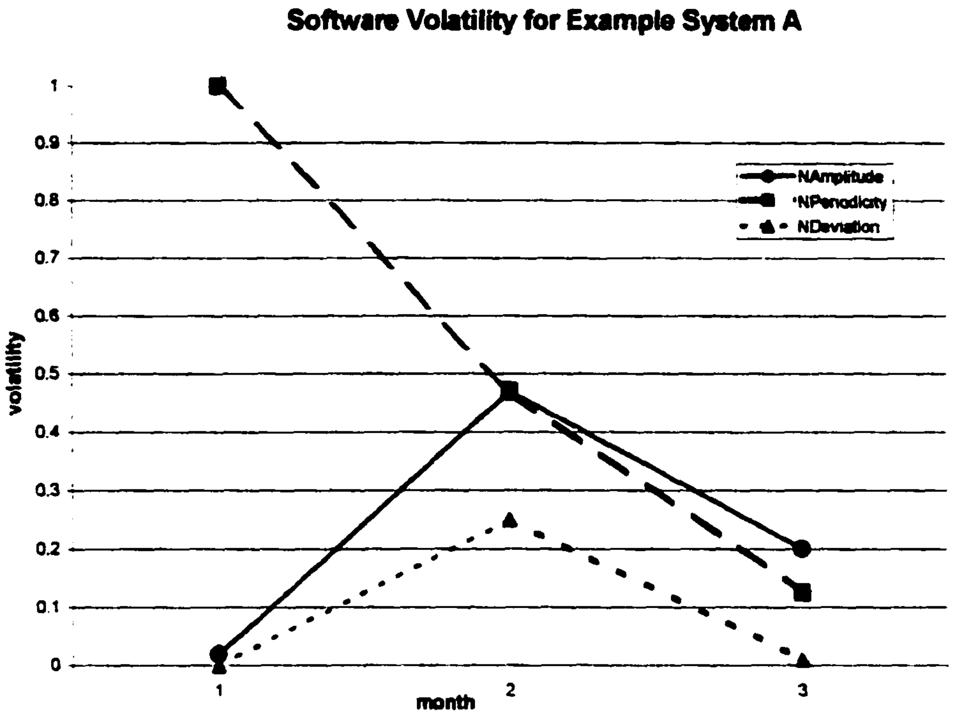


Figure 5: System A Lifecycle Software Volatility

As summarized in Figure 5, software volatility for system A shows an increase in amplitude between the first and second months, and a decrease between the second and third months. Periodicity is the same in months 1 and 2, but sharply decreases in month 3. Deviation increases slightly between the first and second month, and then decreases in the third month. We can infer from this that modifications get larger from month 1 to month 2, then decrease in size between months 2 and 3. Decreasing periodicity between months 2 and 3 indicates that modifications are being made more frequently. An increase in deviation between months 1 and 2 indicates that there is a wider variance in the time intervals between modifications during the second month. Decreases in deviation, as in month 3, indicate a reduction in variance of the length of time intervals, i.e. modifications are being implemented at more regular intervals. Therefore, lower deviation indicates that the intervals between program modifications are nearly equal, and system behavior is becoming more uniform.

In the next section we develop criteria for evaluation of the mathematical properties of software volatility measures. These criteria are then applied to our proposed measures. Subsequently we empirically validate the measures. We then measure software volatility of two commercial systems and interpret their lifecycle software behavior.

3 EVALUATION OF SOFTWARE VOLATILITY MEASURES

The prior sections introduced three software volatility metrics. These measures should be rigorously evaluated and validated to see that they logically behave in a manner consistent with the real world phenomena being studied. We approach this task in two steps. First, we evaluate the measurement functions defined for amplitude, periodicity and deviation for appropriate logical and mathematical properties. Then, we evaluate the convergent, discriminant and predictive validity of these measures with empirical data from a software portfolio of legacy systems. Logical evaluation of the measurement functions, coupled with convergent and discriminant validity, will ensure that our measures of amplitude, periodicity and deviation are valid in a precise sense. Predictive validity is demonstrated empirically by the significance of these measures as explanatory variables in a predictive model. This demonstration expands the validation of these variables to external validity and illustrates the proposed measures' generalizability (Rosenthal and Rosnow, 1991). We proceed with evaluation of the logical and mathematical properties of the measurement functions for the dimensions of software volatility.

We build our criteria from traditional measurement theory and evaluation criteria used for other metrics (Allison, 1978; Weyuker, 1988; Chidamber and Kemerer, 1994). We start by building a set of evaluation criteria to test our proposed measurement functions from criteria used in previous research. Amplitude, periodicity and deviation are characteristics used to describe software volatility. The measurement functions we have defined are direct measures of these attributes. We have defined these measures with natural language to build intuitive understanding of the concepts, and with mathematical precision to reduce confusion and provide repeatable results (Finkelstein and Leaning, 1984; Churcher and Shepperd, 1995; Schnedewind, 1992). Situational context is important in the choice of which metrics and scales to apply Zuse and Bollmann, 1990). As Schneidewind (1992) points out, evaluation criteria should fit the context of intended use of the measure, and set reasonable validation criteria. We have defined measurement functions for three attributes for software volatility. Intuitively, we expect measures of these attributes to be non-negative, and to vary from one system to another and throughout a system's lifecycle.

To determine the criteria we should use for evaluating measures of software volatility, we start by listing attributes we logically expect from such measures. As with the Goal-Question-Metric paradigm (Briand, Morasca and Basili, 1999) the criteria used to evaluate measures of software system behavior must be relevant to our intuitive understanding of amplitude (size) and periodicity (time). We have defined system-level measures of software volatility. The definitions of these measures use aggregate functions to describe lifecycle system behavior. Therefore, we need aggregate measures that reflect combined behaviors. We defined system-level measures to allow the use of these measures in comparing software systems of different sizes, ages and technologies.

Allison (1978) uses several criteria for evaluating the mathematical properties of aggregate measures. Allison's criteria include

(A-1) If all individual elements equal 0, the measure equals 0.

(A-2) If any element > 0, then the measure > 0.

(A-3) The measure is scale invariant.

(A-4) The measure is bounded [3]

Allison's first two criteria are important in our evaluation. Allison developed aggregate measures to describe system-level changes. We are defining aggregate measures to describe changes in a system of programs. Behavior of individual software programs must be reflected by any system-level measure. Thus, we evaluate system-level measures to insure they will (1) be positive if any of the system elements has a positive measure, and (2) will be zero if the measures for all system elements are zero.

The properties of (3) scale invariance and (4) boundedness are essential criteria if software volatility measures are used to analyze lifecycle software behavior and to compare behavior of multiple systems. Scale invariance also makes measurement functions technology independent. This is an important characteristic for the measurement of software behavior as it allows the flexibility of comparing measures of a wider variety of systems and of the same system over time.

Weyucker (1988) identified 9 criteria for evaluating software complexity measures. Although the appropriateness and completeness of these properties have been widely debated, no specific alternative set of evaluation criteria has been proposed (Cherniavhy and Smith, 1991; Churcher and Shepperd, 1995; Roy, 2001). We examine 3 of the 9 properties identified by Weyucker, and also used by Chidamber and Kemerer (1994) to evaluate software complexity metrics. These are (W-1) monotonicity, (W-2) noncoarseness and (W-3) equivalence. The other 6 criteria apply specifically to software

[3] Allison (1978) used a fifth criteria, sensitivity to transfers, to see if a measure is affected by the principle of transfers when income is shifted from one group to another. This criterion applies mainly to measures for economic analyses and is not directly applicable here.

complexity metrics and are not relevant in evaluating system-level measures of software volatility.

The property of (W-1) *monotonicity* implies that the measure of a combined system (P+Q) would be larger than the individual measures of either P or Q alone (Weyuker, 1988). General monotonicity requires only that the measure of (P+Q) be no less than the measures of either P or Q alone (Tian and Zelkowitz, 1992). Logically, monotonicity would apply to absolute measures of volatility. The defined dimensional measures are relative measures normalized against the size or age of the system. Normalization is needed to satisfy the properties of scaled invariance and boundedness. After two subsystems are combined and the combined measures normalized, monotonicity requirements can no longer be applied. As discussed below, measurement qualities of scale invariance and boundedness are important for the analysis of software system behavior over complete life spans and across systems of varying size and age. For the purposes of our work, these qualities are considered more relevant than monotonicity.

We seek measures distinguishing differences in behavior between systems with divergent behavior, i.e. a system that is never modified and one that is modified on a daily basis. In addition, we need measures that will detect changes in lifecycle behavior, i.e. volatility at time t may or may not be equal to volatility measured at a later time, $t-n$. In contrast, if two systems are the same size and age, and both are modified at the same time and with modifications of the same size, we expect software volatility measures for both systems to be equivalent. The definitions for Weyucker's properties (W-2) *noncoarseness* and (W-3) *equivalence* describe these qualities.

Our fifth evaluation property (W-2) *noncoarseness* requires that any proposed metric provide variation in measurement.[4] More precisely, for a given measure, η, there exist two entities, P and Q, for which the measures of those two entities will differ, i.e. given metric η, $\exists$ P, $\exists$ Q, such that $\eta(p) \neq \eta(Q)$.

Similarly, it is also important that software volatility measures exhibit a sixth property, (W-3) *equivalence*. Thus, for a given measure, η, there can exist two entities, P and Q, with the same measure, i.e. given metric η, $\exists$ P $\exists$ Q such that $\eta(P) = \eta(Q)$.

We now evaluate our proposed measures of amplitude, periodicity and deviation against these 6 evaluation criteria. Table 2 summarizes the results of this evaluation.

	Amplitude **NAmplitude**	***Periodicity*** **NPeriodicity**	***Deviation*** **NDeviation**
1. If individual elements all = 0, so does the measure	TRUE	TRUE	TRUE
2. If any element > 0, then the measure > 0	TRUE	TRUE	TRUE
3. Scale invariance and technology indepedence	TRUE	TRUE	TRUE
4. Lower bound **Upper bound**	0 1	0 1	0 1
5. Noncoarseness	TRUE	TRUE	TRUE
6. Equivalence	TRUE	TRUE	TRUE
7. Monotonicity	Not applicable	Not applicable	Not applicable

Table 2: Evaluation of Proposed Measures

3.1 AMPLITUDE OF SOFTWARE VOLATILITY

Evaluating amplitude measurement function NAmplitude according to our criteria we find:

(1) If all individual elements equal 0, the measure equals 0: True - if the size of all modifications = 0, then all $\text{NAmplitude}_t = 0$.

[4] Noncoarseness is similar to the evaluation criteria of discriminative power described by Schneidewind (1992).

(2) If any element > 0, then the measure > 0: True - if at least one modification size > 0, then $\text{NAmplitude}_t > 0$.

(3) Scale invariance and technology independence: True by definition for all normalized measures.

(4) Boundedness: Lower Bound = 0: NAmplitude_t is bounded below by 0; Upper Bound = 1: NAmplitude_t is bounded above by 1 by definition.

(5) Noncoarseness: True - Software systems are of widely varying size, as are their modifications. NAmplitude_t will vary between systems and over time.

(6) Equivalence: True - Two systems, P and Q, of the same size can receive modifications of the same size, making by NAmplitude_t of P equal to NAmplitude_t of Q.

3.2 Periodicity of Software Volatility

Evaluating periodicity measurement function NPeriodicity according to our criteria we find:

(1) If all individual elements equal 0, the measure equals 0. True - if all $\text{TSM}_{jt} = 0$, then all $\text{Periodicity}_t = 0$, and all $\text{NPeriodicity}_t = 0$.

(2) If any element > 0, then the measure > 0.True - if at least one $\text{TSM}_{jt} > 0$, then $\text{Periodicity}_t > 0$, and $\text{NPeriodicity}_t > 0$.

(3) Scale invariance and technology independence - True for all normalized measures, by definition.

(4) Boundedness: *Lower Bound* - Periodicity$_t$ and NPeriodicity$_t$ are bounded below by 0 because all elements are non-negative real numbers: Upper Bound - Periodicity$_t$ is bounded above by *t;* NPeriodicity$_t$ is bounded above by 1; NPeriodicity$_t$ = Periodicity$_t$ /t; Upper bound of NPeriodicity$_t$ = t/t = 1;

(5) Noncoarseness - True. Example: two systems, P and Q, are initially implemented on the same day. Each receives one modification. P is modified on the 15th day of its first month of operation and system Q is modified on the 20th day of its first month of operation. NPeriodicity$_t$ for P is 0.5 and NPeriodicity$_t$ for Q is 0.67. NPeriodicity$_t$ will vary between systems and over time.

(6) Equivalence - True - Two system, P and Q, are initially implemented on the same day. Each receives one modification on the same day. NPeriodicity$_t$ for P will be equivalent to NPeriodicity$_t$ for Q.

3.3 Deviation of Software Volatility

Evaluating deviation measurement function NDeviation$_t$ according to our criteria we find:

(1) If all individual elements equal 0, the measure equals 0. True

(2) If any element > 0, then the measure > 0. True

(3) Scale invariance and technology independence - True for all normalized measures by definition.

(4) Boundedness - *Lower Bound:* by definition, variance and NDeviation$_t$ are bounded by 0. - *Upper Bound:* by definition, NDeviation$_t$ is bounded above by 1.

(5) Noncoarseness: True - Two systems, P and Q, are initially implemented on the same day. Each receives one modification. P is modified on the 15th day of its first month of operation and system Q is twice, once on the 10th day of the month and once on the 15th day of its first month of operation. NDeviation_t for P is 0.0 and NDeviation_t for Q is 0.56. NDeviation_t will vary between systems and across time.

(6) Equivalence - True - Two system, P and Q, are initially implemented on the same day. Each receives one modification on the same day. NDeviation_t for both P and Q is 0.0.

Therefore, our defined measures for amplitude, periodicity and deviation satisfy these 6 evaluation criteria. In section 4 we now proceed to empirically validate these measures. We use empirical data from a longitudinal record of modifications to a software portfolio to test for convergent and discriminant validity. In section 5 we then illustrate predictive validity with a regression of software complexity against our proposed measures.

4 RESULTS FROM EMPIRICAL EVALUATION

4.1 Establishing Validation Criteria

The measurement functions for the amplitude, periodicity and deviation of software volatility should now be evaluated empirically to establish external validity. There is no established set of universally accepted validation criteria for software metrics. As with evaluation criteria for the mathematical properties of software metrics, we now review some of the validation criteria used by other researchers examining software metrics.

Basically, a measure is valid if it accurately characterizes the attribute it claims to measure (Schneiidewind, 1992). Statistical validation of new measures is established by setting validation criteria, identifying appropriate statistical tests with confidence level α, and performing the appropriate tests. It is important to recognize that any particular metric may be valid with respect to certain criteria and invalid with respect to others (Schneidewind, 1992). Validation criteria used for testing new metrics must be relevant to the characteristics being measured.

It is important to consider both the statistical significance of the relationship and the degree of the association between the variables being analyzed (Baroudi and Orlikowski, 1989; Emam and Birk, 2000). We consider a correlation weak if it is statistically insignificant ($p > 0.05$) or has a low correlation ($|\text{correlation}| < 0.40$ and strong correlation in the inverse is true.[5]

There are several types of external validity that may be addressed. This work examines three in particular; convergent validity, discriminant validity and predictive validity. Convergent validity is established by demonstrating a correlation between our new measure and a comparable measure of the same property. Discriminant validity demonstrates independence among the three measures of software volatility. By demonstrating the orthogonality of these measures, discriminant validity shows these measures describe three separate dimensions of software volatility. The criteria for establishing discriminant validity is to show weak correlations of each dimensional measure with each of the other two. Predictive validity can be demonstrated by testing predictive models and obtaining a strong correlation coefficient between independent and dependent variables (Emam and Birk, 2000; Brand, Morasca and Basili, 1999).

[5] The cut-off value for strong correlations (0.40) is somewhat lower than a 0.50 level that might be established for correlation coefficients when testing predictive validity (Donaldson and Weymark, 1980). In this case we seek to establish the existence of meaningful relationships between our measures of amplitude, periodicity and deviation and other comparable measures for those same concepts. Because the comparable measures we have used for convergent validation of periodicity and deviation are not bounded above, the association between our measures and those

4.2 Empirical Validation of Proposed Software Volatility Measures

In this analysis we use empirical data obtained from a large mid-Western retailer with a portfolio of 23 legacy systems, including more than 3,500 programs. Over the course of the portfolio's 20-year history, software maintainers kept a detailed log of every modification to each program by recording implementation date, purpose, type of change and programmer responsible (Kemerer and Slaughter, 1999). The combined maintenance logs for the portfolio provide researchers with the raw data for detailed information of approximately 25,000 individual software modifications.

Convergent validity is supported by substantial correlation with conceptually similar metrics (Rosenthal and Rosnow, 1991). We demonstrate convergent validity of our measures of software volatility by calculating our 3-dimensional measures and any logically comparable measures of amplitude, periodicity and deviation for each system and comparing the correlation between each measure and its counterpart.

The irony of this comparison is that should completely satisfactory alternate measures exist, we would not be defining new ones. In each case, we have identified a comparable measure that should logically behave in a manner consistent with the constructed measurement functions we have introduced *despite* its other potential flaws. A comparable system-level measure for amplitude is the percentage of new programs in the system. Percentage of new programs is a coarser measure than NAmplitude. Percentage of new programs assumes that each program added is the same size as all programs in the system. As we are comparing the proportional change in size of the system with the proportional amount of modified code in the system, these two measures should behave similarly. Periodicity can be measured by the inverse of number of changes per time period. The correlation of NPeriodicity$_t$ and the number of

comparable measures will be weakened. Still we seek the strongest relationship possible, in an effort to define new measurement functions for these phenomena.

modifications per time period can be used to support convergent validity. Number of modifications per time period is a coarser measure of activity in the system. Its reciprocal will give an average of time between modifications for the period.[6] However, number of modifications per time period is not bounded above, as is NPeriodicity$_t$. This difference in the basic properties of NPeriodicity$_t$ and its comparable measure will *weaken* the strength of their association. However, analysis of the correlations between them should still be sufficient to provide evidence of convergent validity of NPeriodicity$_t$. Coefficient of variation measure is logically comparable to our proposed measure of deviation.[7] Coefficient of variation for the TSM of a system during each time period *t* should provide an alternative measure for the degree of variance in the time intervals between modifications. However, coefficient of variation is not bounded above. This difference in the basic properties of our measure of deviation, NDeviation, and its comparable measure will weaken the strength of their association.

These comparable measures are not suitable substitutes for the measures we have defined. In each case these comparable measures fail at least one of the logical and mathematical evaluation criteria. However, they are useful in order to demonstrate convergent validity of our measures of software volatility by calculating our 3-dimensional measures and their logically comparable measures of amplitude, periodicity and deviation for each system and comparing the correlations between each measure and its counterpart.

Table 3 shows the correlation of our measure of NAmplitude and its comparable measure, the percentage of new programs in the system. The portfolio's 23 systems all show these measures to be strongly correlated with statistically significant correlations ($p < 0.05$) of magnitude > 0.40, supporting convergent validity for NAmplitude as a

[6] This is similar to the calculation of MTBF that is based on the number of failures occurring over the full lifespan of the system (Gaither, 1990).

measure of amplitude.

System	Lifespan (months)	Correlation(NAmplitude, % new programs)	p-value
System 1	96	0.8984	0.0000
System 2	201	0.9062	0.0000
System 3	89	0.9662	0.0000
System 4	69	0.9939	0.0000
System 5	235	0.9154	0.0000
System 6	223	0.9219	0.0000
System 7	85	0.9249	0.0000
System 8	234	0.9084	0.0000
System 9	96	0.9918	0.0000
System 10	246	0.9463	0.0000
System 11	62	0.9931	0.0000
System 12	122	0.9755	0.0000
System 13	87	0.9788	0.0000
System 14	189	0.8761	0.0000
System 15	137	0.9481	0.0000
System 16	125	0.9719	0.0000
System 17	73	0.9648	0.0000
System 18	120	0.9805	0.0000
System 19	66	0.9771	0.0000
System 20	195	0.9888	0.0000
System 21	110	0.9925	0.0000
System 22	212	0.9915	0.0000
System 23	129	0.9752	0.0000

Table 3: Correlations NAmplitude and % of New Programs

Table 3 lists the correlations of periodicity measured by NPeriodicity and its comparable measure, the number of modifications each time period. Twenty-one of the 23 systems show statistically significant correlations ($p < 0.05$) with eleven strong correlations of magnitude > 0.40. These empirical results provide support for the convergent validity of NPeriodicity as a measure of periodicity.

[7] Coefficient of variation is a measure of dispersion (Dess and Beard. 1984).

System	Lifespan (months)	Correlation(NPeriodicity, Number of modifications)	p-value
System 1	96	-0.3555	0.0004
System 2	201	-0.6005	0.0000
System 3	89	-0.0587	0.5848
System 4	69	-0.5077	0.0000
System 5	235	-0.3381	0.0000
System 6	223	-0.5545	0.0000
System 7	85	-0.5462	0.0000
System 8	234	-0.5545	0.0000
System 9	96	-0.5685	0.0000
System 10	246	-0.5788	0.0000
System 11	62	-0.3966	0.0014
System 12	122	-0.2753	0.0022
System 13	87	0.1297	0.2312
System 14	189	-0.4780	0.0000
System 15	137	-0.2796	0.0009
System 16	125	0.1876	0.0362
System 17	73	-0.1598	0.1769
System 18	120	-0.3672	0.0000
System 19	66	-0.5201	0.0000
System 20	195	-0.4065	0.0000
System 21	110	-0.8368	0.0000
System 22	212	-0.3576	0.0000
System 23	129	-0.1838	0.0371

Table 4: Correlations of NPeriodicity and Number of Modifications

Table 5 shows that twenty of the portfolio's 23 systems have statistically significant correlations between deviation measured by NDeviation and its comparable measure, coefficient of variation. Twelve of the 23 systems show strong correlations with pair-wise correlations of magnitude greater than or equal to 0.40. These results provide support for convergent validity of NDeviation as a measure of deviation.

System	Lifespan (months)	Correlation(NDeviation, Coefficient of Variation)	p-value
System 1	96	0.3528	0.0004
System 2	201	0.1717	0.0148
System 3	89	0.4264	0.0000
System 4	69	0.7159	0.0000
System 5	235	0.4100	0.0000
System 6	223	0.4391	0.0000
System 7	85	0.1574	0.1502
System 8	234	0.2065	0.0015
System 9	96	0.5521	0.0000
System 10	246	0.1626	0.0106
System 11	62	0.4769	0.0001
System 12	122	0.3545	0.0001
System 13	87	0.3073	0.0038
System 14	189	0.4525	0.0000
System 15	137	0.2281	0.0074
System 16	125	0.1722	0.0549
System 17	73	0.4707	0.0000
System 18	120	0.3851	0.0000
System 19	66	0.4953	0.0000
System 20	195	0.4590	0.0000
System 21	110	0.9559	0.0000
System 22	212	0.5822	0.0000
System 23	129	0.2644	0.0025

Table 5: Correlations of NDeviation and the Coefficient of Variation

Discriminant validity is supported by a lack of correlation between conceptually unrelated measures (Rosenthal and Rosnow, 1991). Discriminant validity among amplitude, periodicity and deviation is demonstrated by weak correlations among NAmplitude$_t$, NPeriodicity$_t$ and NDeviation$_t$.

Correlations among the measures of amplitude, periodicity and deviation are calculated for each month in the lifecycles of the portfolio's 23 systems (Table 6). In nineteen systems there are weak correlations between NAmplitude$_t$ and NPeriodicity$_t$ (10 were statistically insignificant and nine others had $|\text{correlation}| < 0.40$). Twenty-two systems show weak correlations between NPeriodicity_t and NDeviation_t (thirteen were statistically insignificant and nine others had $|\text{correlation}| < 0.40$). All 23 systems have weak correlations between NAmplitude_t and NDeviation_t (22 were statistically insignificant and one other had $|\text{correlation}| < 0.40$). Thus, these data support discriminant validity among these dimensions.

System	Lifespan (months)	Corr(NAmplitude, NPeriodicity) | p-value	CORR(NAMPLITUDE, NDEVIATION) | p-value	Corr(NPeriodicity, NDeviation) | p-value
System 1	96	-0.1677 | (0.1024)	0.0552 | (0.5931)	0.0157 | (0.8794)
System 2	201	-0.3222 | (0.0000)	-0.0079 | (0.9111)	-0.0517 | (0.4662)
System 3	89	-0.1526 | (0.1535)	-0.0269 | (0.8025)	0.1394 | (0.1926)
System 4	69	-0.4731 | (0.0000)	0.0620 | (0.6131)	-0.2750 | (0.0222)
System 5	235	-0.2162 | (0.0008)	-0.0126 | (0.8471)	-0.1533 | (0.0187)
System 6	223	-0.2977 | (0.0000)	0.1540 | (0.0215)	-0.1803 | (0.0069)
System 7	85	-0.6191 | (0.0000)	-0.0097 | (0.9301)	-0.2673 | (0.0134)
System 8	234	-0.2352 | (0.0003)	-0.0234 | (0.7216)	-0.1065 | (0.1042)
System 9	96	-0.4702 | (0.0000)	-0.0106 | (0.9185)	-0.1954 | (0.0564)
System 10	246	-0.2405 | (0.0001)	0.0127 | (0.8423)	-0.0464 | (0.4691)
System 11	62	-0.1459 | (0.2577)	-0.0344 | (0.7909)	-0.0724 | (0.5760)
System 12	122	-0.1486 | (0.1024)	-0.0786 | (0.3897)	-0.0027 | (0.9761)
System 13	87	-0.1831 | (0.0897)	-0.0935 | (0.3891)	0.2052 | (0.0566)
System 14	189	-0.2377 | (0.0010)	-0.0452 | (0.5368)	-0.2813 | (0.0001)
System 15	137	-0.1552 | (0.0702)	-0.1283 | (0.1352)	-0.1282 | (0.1355)
System 16	125	-0.1712 | (0.0563)	0.0093 | (0.9180)	0.7901 | (0.0000)
System 17	73	-0.1563 | (0.1867)	0.0200 | (0.8664)	0.0065 | (0.9563)
System 18	120	-0.1424 | (0.1207)	-0.1358 | (0.1390)	-0.1041 | (0.2578)
System 19	66	-0.3597 | (0.0030)	-0.0310 | (0.8046)	-0.3224 | (0.0083)
System 20	195	-0.1963 | (0.0060)	-0.0436 | (0.5448)	-0.2422 | (0.0006)
System 21	110	-0.5483 | (0.0000)	-0.0123 | (0.8984)	-0.2548 | (0.0072)
System 22	212	-0.2694 | (0.0001)	-0.0073 | (0.9154)	-0.1726 | (0.0119)
System 23	129	-0.1489 | (0.0921)	-0.0349 | (0.6947)	-0.0155 | (0.8617)

Table 6: Correlations Supporting Discriminant Validity

In summary, the new measures have been evaluated for the mathematical properties we desire for aggregate measures of size and time. The empirical data support convergent and discriminant validity. Convergent validity tells us that these measures behave in a manner consistent with other logically comparable measures. Discriminant validity demonstrates that these three measures describe three different attributes of software volatility. We now illustrate the relationship between these measures and a traditional measure of a software characteristic, software complexity. By using a simple predictive model we test a multivariate regression of software complexity against lagged terms for amplitude, periodicity and deviation.

4.3 Predictive Validity

Predictive validity is established by determining the degree to which a trait or characteristic can predict future outcomes. To demonstrate the predictive validity of software volatility, we use a simple model for software complexity. We posit that software volatility in a previous time period will significantly affect software complexity in the current time period. Banker, Davis and Slaughter (1998) propose and support a model demonstrating the link between software maintenance processes and complexity. They show that maintenance activity results in increased levels of software complexity. In the same manner, we posit that increased software volatility from software modifications will result in increased software complexity. Our model uses the lagged software volatility dimensions of amplitude$_{t-1}$, periodicity$_{t-1}$ and deviation$_{t-1}$ as explanatory variables.

$$complexity_t = \beta_0 + \beta_1 amplitude_{t-1} + \beta_2 periodicity_{t-1} + \beta_3 deviation_{t-1} + \varepsilon_t$$

Software system complexity$_t$, normalized by total system size$_t$, is the dependent variable. There are a number of software complexity metrics available (Cook and Roesch, 1994; Harrison, 1990; Pressman, 1992). We ran empirical tests for this model using six different standard complexity metrics.[8] In each case, the total system size measured in lines of code was used to control for system size and allow comparison of results between systems.[9] Coefficients for the explanatory variables were estimated using ordinary least squares estimation procedures. The multivariate regression was estimated for each of the 23 systems, for each of the 6 normalized complexity metrics. In total we estimated 138 equations for our proposed measures of amplitude, periodicity and deviation, and 138 equations for the comparable measures of those same attributes.

The adjusted R^2 for two-thirds of the estimated 138 regressions using our proposed measures was greater than or equal to the adjusted R^2 for the corresponding estimates using the comparable measures. As one would expect, using software volatility to predict software complexity is more significant for more volatile systems. Summarized results of these estimated regressions are summarized in Table 7.

	Predictive Validity		
Software Complexity Metric:	**Average R squared (proposed measures)**		**Average R squared (comparable measures)**
McCabe's	0.2646	>	0.2003
Halstead's n1	0.3530	>	0.2798
Halstead's n2	0.3478	>	0.2668
Halstead's N1	0.2803	>	0.1910
Halstead's N2	0.3160	>	0.2255
Calls	0.3694	>	0.2719

Table 7: Summary of Linear Regression Estimates for the Software Portfolio

Our results show no significant multicollinearity among the measures for amplitude, periodicity and deviation. Low mean Variance Inflation Factors, VIF, indicate a lack of multicollinearity among explanatory variables (Belsley, Kuh and Welsch, 1980). This provides further confirmation of the independence of amplitude, periodicity and deviation as unique dimensions of software volatility.

[8] These measures are McCabe's cyclometric measure, Halstead's primitive measures n1, n2, N1 and N2, and the number of calls (Cook and Roesch, 1994; Harrison, 1990). Each was normalized by the total system LOC at time *t*.

In summary, the three proposed measures (1) improve on standard existing measures in explaining variance of standard software complexity measures and (2) provide support for predictive validity.

5 DISCUSSION

How can managers use these dimensional measures of software volatility to interpret changes in lifecycle system behavior? We start with a graphical representation of amplitude, periodicity and deviation for an idealized completely stable system. Using the measurement function for amplitude, by definition $\text{NAmplitude}_t = 0$ for each time period *t* when no software change events occur. Given the measurement function NPeriodicity_t, as the number of software change events in time period *t* approaches 0, the limit of $\text{NPeriodicity}_t = 1$. Hence, $\text{NPeriodicity}_t = 1$ for any time period *t* in which no software change events occur. Given the measurement function NDeviation_t, as the number of software change events in time period *t* approaches 0, the limit of $\text{NDeviation}_t = 0$, i.e. $\text{NDeviation}_t = 0$ for any time period *t* in which no software change events occur. Hence, $\text{NDeviation}_t = 0$ when no modifications occur in time period *t*. If there are no software modifications in any time period throughout the productive life span of an idealized stable system, $\text{NAmplitude}_t = 0$, $\text{NPeriodicity}_t = 1$ and $\text{NDeviation}_t = 0$, for all *t*. The software volatility for the lifecycle of a hypothetical idealized stable system would be graphed as in Figure 6.

[9] All programs in each system were written in the same language (Jeffrey and Lawrence, 1979).

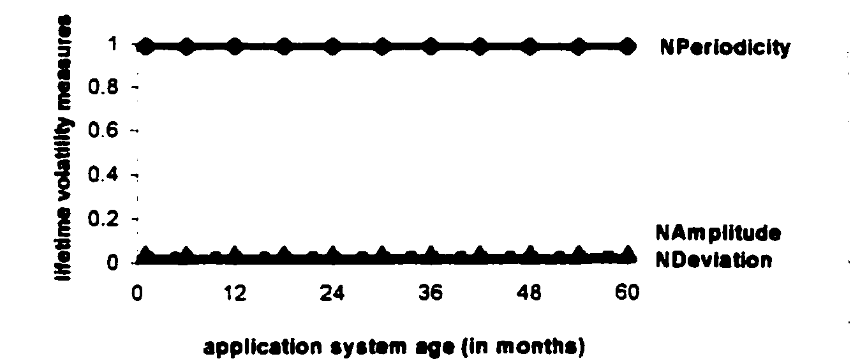


Figure 6: Hypothetical Idealized Stable System

We compare this idealized stable system with two actual systems in our portfolio. System 7 appears to be fairly stable throughout its more than seven year life span. There were two brief periods of volatility. The first one occurred when system 7 was about eighteen months old. Amplitude increased to 0.2 and periodicity became short. The second period of volatility occurred when system 7 was between 65 and 70 months old. Periodicity fell and amplitude increased indicating more frequent and larger modifications. Deviation increased indicating that some programs in the system were changing frequently and others were not. (See Figures 7a, b and c.)

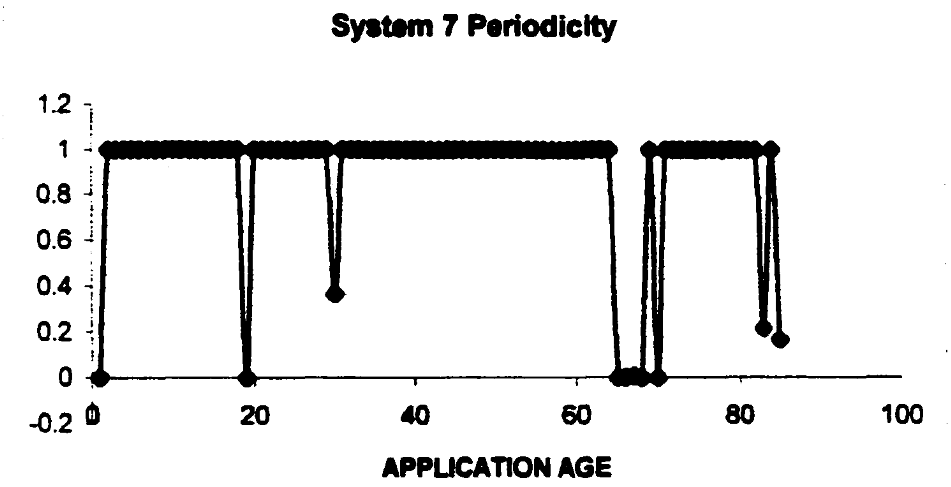


Figure 7a: Lifetime Volatility System 7 - Periodicity

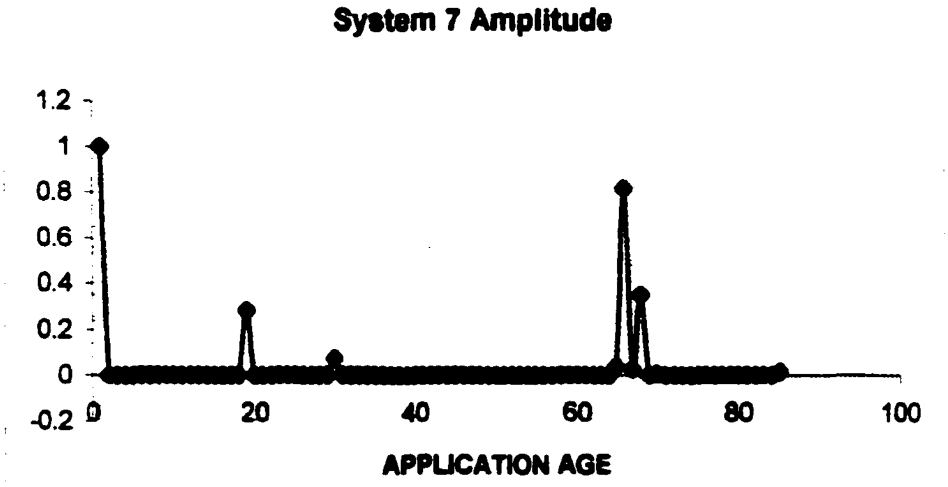


Figure 7b: Lifetime Volatility System 7 - Amplitude

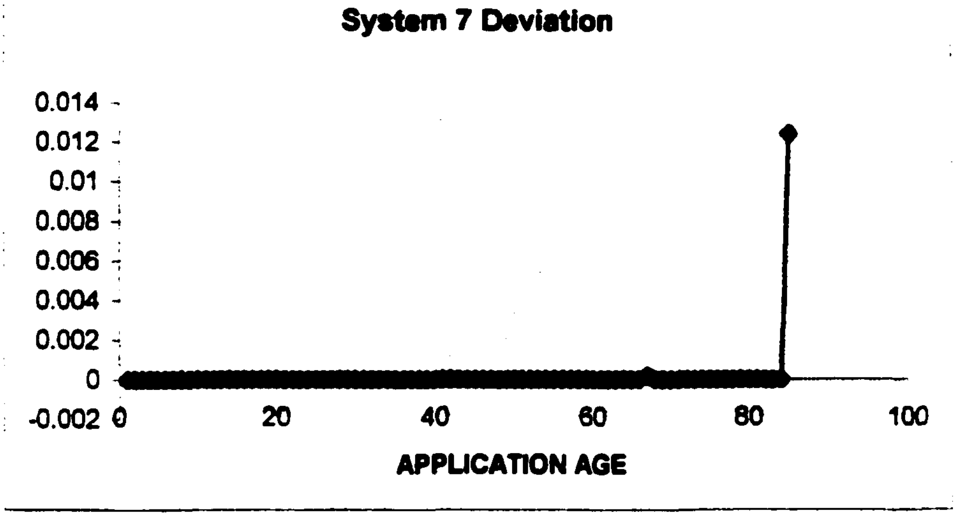


Figure7c: Lifetime Volatility System 7 - Deviation

System 23 appears to be relatively stable for only the first 18 months of its productive life. Starting at approximately 18 months of age the system became volatile with frequent, relatively small software changes for the rest of its more than 10 year life span. Inconsistency of behavior between programs in system 23 is indicated by deviation measured by $\text{NDeviation}_t > 0$ (See Figures 8a b and c.)[10]

[10] When amplitude, periodicity and deviation are plotted on the same graphical scale, changes in deviation are difficult to see. Even though all three are bounded by 0 and 1, the magnitude of deviation as defined tends to be much smaller than the other two.

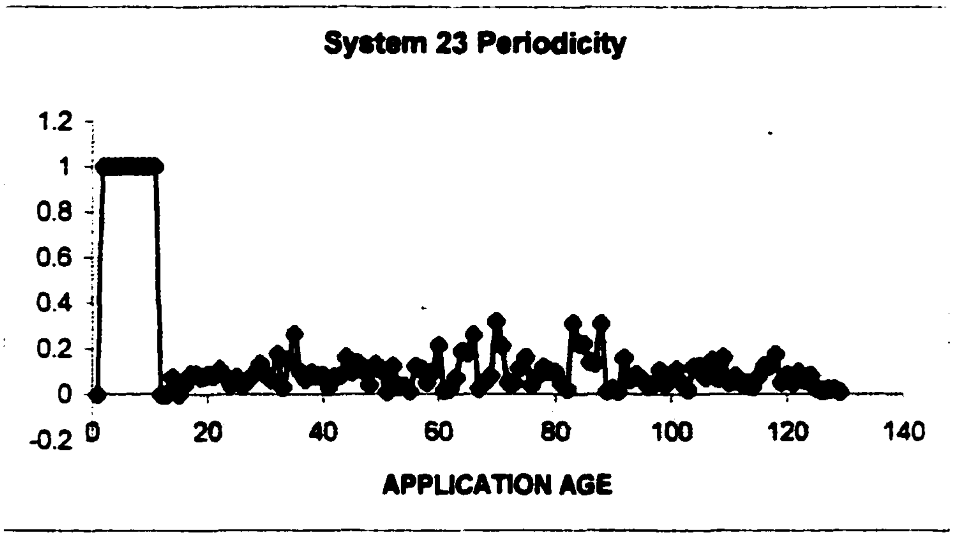


Figure 8a: Lifetime Volatility System 23 - Periodicity

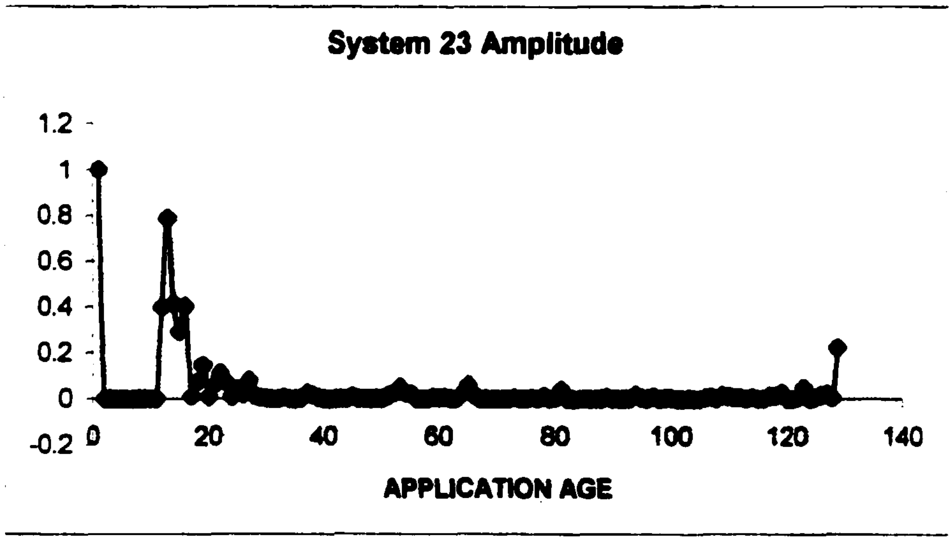


Figure 8b: Lifetime Volatility System 23 - Amplitude

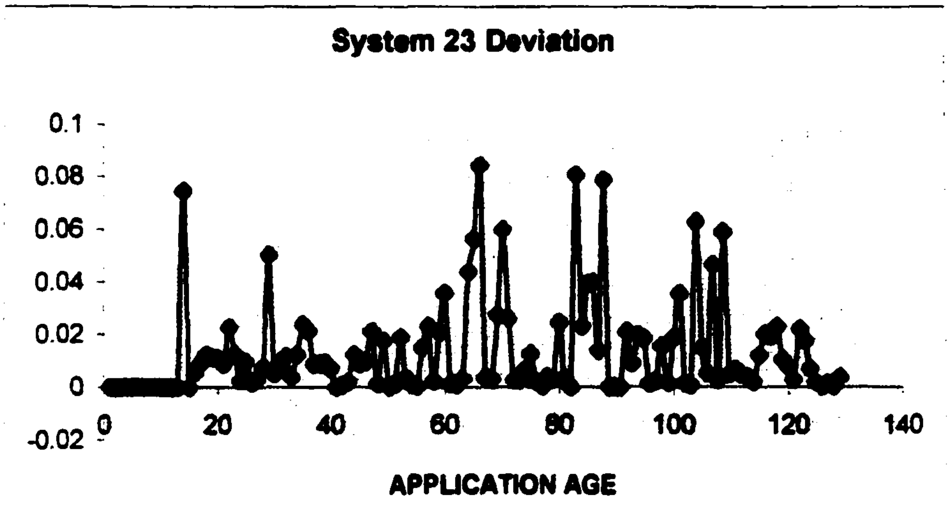


Figure 8c: Lifetime Volatility System 23 - Deviation

We observe that lifecycle maintenance activity started about eighteen months after initial system implementation. Software managers can compare the behavior patterns of system 7 and system 23, and conclude that system 23 will require more constant levels of maintenance support while system 7 requires infrequent support. This information can be useful for resource planning both in the short term, e.g. budgeting system support resources, and in the long term, e.g. as input to the "repair or replace" decision for an application system.

6 SUMMARY

The definition, evaluation and validation of a new system-level measure of software volatility contribute to the collective theory base for software evolution. A system-level multi-dimensional measure of software volatility makes it possible to develop a more complete picture of lifecycle software behavior. By presenting a multi-dimensional measure of software volatility, software system change processes can be analyzed concurrently for the amplitude, periodicity and deviation of software volatility. We defined three measures describing these different attributes of software volatility in order to facilitate its description as a dynamic behavior of software systems. By rigorously evaluating these measures, we establish a set of criteria for evaluation of software volatility measures. Evaluation criteria were developed from measurement literature and applied against our proposed measures. The proposed measures were then validated for convergent and discriminant validity. Their usefulness as predictors was demonstrated with a regression of complexity against lagged values of amplitude, periodicity and deviation. This multi-dimensional system-level software volatility measure provides technology independent measures that allow comparison of system behavioral changes over time and across systems. Interpretation of lifecycle volatility was demonstrated with empirical data for two software systems.

This work can be expanded by analyzing software volatility in a number of ways. The development of these direct, objective measures lays the groundwork for development of theoretical models of software system behavior. Theoretical models of the factors contributing to software volatility can be built and tested with parametric methods for regression analysis. Analyses can be used to build and test models of the drivers of software volatility and examination of the effects of software volatility on lifecycle software maintenance outcomes such as costs and errors.

REFERENCES

Albrecht, A.J., and Gaffney, J.E., Jr., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", IEEE Transactions on Software Engineering, vol. 9, no. 6, pp. 639-648, Nov. 1983.

Allison, P.D., "Measures of Inequality", American Sociological Review, vol. 43, pp. 865-880, Dec. 1978.

Banker, R.D. and Slaughter, S.A., "The Moderating Effects of Structure on Volatility and Complexity in Software Enhancement", Information Systems Research, vol. 11, no. 3, pp. 219-240, Sept. 2000.

Baroudi, J.J. and Orlikowski, W.J., "The Problem of Statistical Power in MIS Research", MIS Quarterly, pp. 87-105, Mar. 1989.

Blady, L.A. and Lehman, M.M., "A Model of Large Program Development", IBM Systems Journal, no. 3, pp. 225-252, 1976.

Belsley, D.A., Kuh, E., and Welsch, R.E., Regression Diagnostics: Identifying Influential Data and Sources of Collinearity, John Wiley and Sons, 1980.

Boehm, B.W., "Software Engineering Economics", Software Project Management: Readings and Cases, C.F. Kemerer (Ed.), R. D. Irwin/McGraw-Hill, 1997.

Briand, L., Morasca, S., and Basili, V.R., "Defining and Validating Measures for Object-Based High-Level Design", IEEE Transactions on Software Engineering, vol. 25, no. 5, pp. 722-743, Sept./Oct. 1999.

Buecbe, F., Introduction to Physics for Scientists and Engineers, McGraw-Hill, 1969.

Cherniavsky, J.C. and Smith, C.H., "On Weyucker's Axioms for Software Complexity Measures", IEEE Transactions on Software Engineering, vol. 16, no. 6, pp. 636-638, June 1991.

Chidamber S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, June 1994.

Churcher, N.I., and Shepperd, M.J., "Comments on 'A Metrics Suite for Object-Oriented Design' ", IEEE Transactions on Software Engineering, vol. 21, no. 3, pp. 263-265, March 1995.

Cook, C.R. and Roesch, A., ":Real-time Software Metrics", Journal of Systems and Software, vol. 24, no. 3, pp. 223-237, 1994.

Dess, G.G. and Beard, D.W., "Dimensions of Organizational Environments", Administrative Science Quarterly, vol. 29, pp. 52-73, 1984.

Donaldson, D., and Weymark, J.A., "A Single-Parameter Generalization of the Gini Indices of Inequality", Journal of Economic Theory, vol. 22, pp. 67-86, 1980.

Emam, K. El, and Birk, A., "Validating the ISO/IEC 15504 Measure of Software Requirements Analysis Process Capability", IEEE Transactions on Software Engineering, vol. 26, no. 6, pp. 541-566, June 2000.

Finkelstein, L., and Leaning, M.S., "A Review of the Fundamental Concepts of Measurement", Measurement, vol. 2, no. 1, pp. 25-34, Jan.-Mar. 1984.

Gaither, N., Production And Operations Management: A Problem-Solving and Decision-Making Approach, 4th Edition, The Dryden Press, 1990.

Hamm, S., and Port, O., "The Mother of All Software Projects", Business Week, February 22, 1999, pp. 69-76.

Harrison, W., "Using Metrics to Allocate Testing Resources in a Resource Constrained Environment", Portland State University Department of Computer Science, 1990.

Jeffrey, D.R., and Lawrence, M.J., "An Inter-Organizational Comparison of Programming Productivity", Proceedings of the 4th International Conference on Software Engineering, 1979.

Kemerer, C.F. and Slaughter, S.A., "An Empirical Approach to Studying Software Evolution", IEEE Transactions on Software Engineering, vol. 25, no. 4, pp.1-17, 1999.

Lehman, M.M., and Belady, L.A., Program Evolution: Processes of Software Change, Academic Press, 1985.

Lehman, M.M., and Ramil, J.F., "The Impact of Feedback in the Global Software Process", The Journal of Systems and Software, vol. 46, no. 2-3, pp. 123-134, April 15, 1999.

Li, W., Etzkorn, L., Davis, C. and Talburt, J., "An Empirical Study of Object-Oriented System Evolution", Information and Software Technology, vol. 42, no. 6, pp. 373-381, April 15, 2000.

Lyu, M.R., Handbook of Software Reliability Engineering, IEEE Computer Society Press, 1995.

Pressman, R.S., Software Engineering: A Practitioner's Approach, 3rd Edition, McGraw Hill, 1992.

Rosenthal, R. and Rosnow, R.L., Essentials of Behavioral Research: Methods and Data Analysis, 2nd Edition, McGraw-Hill, Inc., 1991.

Roy, Gursaran and Gurdev, On the Applicability of Weyuker Property 9 to Objet-Oriented Structural Inheritance Complexity Metrics, IEEE Transactions on Software Engineering, vol. 27, no. 4, pp. 381-384, April 2001.

Schnedewind, N.F., "Methodology for Validating Software Metrics", IEEE Transactions on Software Engineering, vol. 17, pp. 253-266, 1992.

Schnedewind, N.F., "Measuring and Evaluating Process Using Reliability, Risk, and Test Metrics", IEEE Transactions on Software Engineering, vol. 25, no. 4, pp. 769-781, Nov./Dec. 1999.

Snyder, N.H., and Glueck, W.F., "Can Environmental Volatility be Measured Objectively?", Academy of Management Journal, vol. 25, pp. 185-192, 1982.

Stroh, L., Baumann, B.J., and Reilly, A., "Agency Theory and Variable Pay Compensation Strategies", Academy of Management Journal, vol. 39, no. 2, pp. 751-767, 1996.

Thompson, J.D., Organizations in Action, McGraw Hill Book Company, 1967.

Tian, J., and Zelkowitz, M.V., "A Formal Program Complexity Model and Its Application", Journal of Systems and Software, vol. 17, pp. 253-266, 1992.

Weyuker, E., "Evaluating Software Complexity Measures", IEEE Transactions on Software Engineering, vol. 14, pp. 1357-1365, 1988.

Wholey, D.R. and Brittain, J., "Characterizing Environmental Variation", Academy of Management Journal, vol. 32, no. 4, pp. 867-882, 1989.

Van Horn, E.C., "Software Must Evolve", Software Engineering, vol. 1, H. Freeman and P.M. Lewis, (Eds.), Academic Press, 1980.

Xia, F. Xia, "Look Before You Leap: On Some Fundamental Issues in Software Engineering Research", Information and Software Technology, vol. 41, no. 10, pp. 661-672, 1999.

Yau, S.S. and Collofello, J., "Some Stability Measures for Software Maintenance", IEEE Transactions on Software Engineering, vol. 6, no. 11, pp. 545+, Nov. 1980.

Yau, S.S. and Collofello, J., "Design Stability Measures for Software Maintenance", IEEE Transactions on Software Engineering, vol. 11, no. 9, pp. 849-856, Sept. 1985.

Zuse, H., and Bollmann, M.P., "Software Metrics: Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Measures", SIGPLAN Notices, vol. 24, no. 8, pp. 23-33, 1990.

CHATER 4:

RESEARCH QUESTION 2 -

ANTECEDENTS OF SOFTWARE VOLATILITY

INTRODUCTION

"Why do we have to keep *fixing* this software? Why can't you write a *good* system so we won't need these *constant changes*?" Most software managers have heard these comments from their users. The truth is that change is a constant in our world and information systems are no exception. Because they are embedded in their respective organizations, information systems affect, and are affected by, the organizations they serve (Lehman and Belady 1985; Pfleeger, 1998). Organizations must make constant adjustments to survive in a habitually changing competitive environment (Porter, 1980; Davis and Olson, 1985; Morgan, 1997). Information systems must also evolve to provide the information their organizations need to remain competitive. Because information systems must provide required information in a timely and accurate manner to the people and organizations that need it, the systems must constantly be maintained and enhanced to satisfy the information requirements of a perpetually changing organization.

Even facing these constant changes, many systems operate productively for decades. It is estimated that the average age for enterprise general ledger application systems in Fortune 1000 companies is 15 years old (Kalakota and Whinston, 1996).

Some information systems change a great deal during their productive lifespans, and others remain unchanged for months and years at a time. Does this automatically mean systems that change are *bad*, and those that never change are *good*? Is change *always* something to be avoided? If an information system remains stable and fails to change with its environment, the system may cause a drag on the organization and hinder organizational success (Truex, Baskerville and Klein, 1998). Therefore, it is important to understand the nature of change and the associated change processes for information systems.

Growth and change of information systems is accomplished through lifecycle software maintenance. To understand the nature of information system change we must understand the "dynamic behavior of programming systems as they are maintained and enhanced over their life times", i.e. *software evolution* (Belady and Lehman, 1976). Software change is a characteristic of the behavior of information systems as they evolve throughout their productive lifecycles. *Software volatility* describes software changes occurring as a result of lifecycle maintenance. By envisioning a longitudinal model of an information system changing to keep pace with changes in its environment, we see a system evolving along with its environment. Analysis of software volatility throughout a system's lifecycle, and across different systems, can improve our understanding of software change and system behavior. With this increased insight into software evolution researchers and managers can enhance their understanding of software evolution and improve management of lifecycle maintenance processes.

Some environments are more volatile than others. Some organizations change more than others. Some tasks are more variable than others. These differences result in differences in the volatility of information systems. At each level through this progression, entities cope with changes in surrounding environments through the dynamics of interfaces to each of those environments. The objective of this research is to identify those dynamic environmental attributes that drive software volatility in information systems. We start by examining factors in the competitive environment and work progressively inward toward more localized factors at the task environment and basic system levels, i.e. the system's inner environment. All lifecycle maintenance activities, including corrective and adaptive modifications, enhancements and new program creations, serve as mechanism for software change (Pfleeger, 1998). Thus, we recognize that these lifecycle maintenance activities are catalysts for software volatility. Therefore, we build our conceptual model from two sets of prior research: research on drivers of software volatility, and research on drivers of lifecycle maintenance and software change.

In the remainder of this paper we build on this discussion by defining dependent and explanatory variables to develop a conceptual model of the antecedents of software volatility. After examining prior research and grounding theory for the influence of each of these concepts, we establish operational variables for these concepts. Seven hypotheses describe the relationship of each of these variables to our measure of software volatility.

Empirical data obtained from the 20-year maintenance logs of a large company are used to test the model through regression analysis. Regression estimates for the full lifecycle maintenance records of 23 information systems are analyzed. Results indicate that attributes of environmental interfaces at all levels drive software volatility.

By maintaining a system-level perspective we have built a predictive model for software volatility. Using this model, researchers can broaden and deepen their understanding of the transforming processes and dynamic behavior observed during software evolution. Managers can improve their ability to anticipate change and design adaptable systems while maintaining a lifecycle perspective for system support resource requirements. We begin by reviewing the relevant literature on software change and change processes in software evolution.

SOFTWARE EVOLUTION

Software change and change processes have been studied in a number of contexts. One perspective for analyzing and understanding lifecycle software change is the study of software evolution. By its very nature software evolution occurs incrementally over long periods of time. Based in general systems theory, studies of software evolution emphasize longitudinal descriptions of system characteristics and the change processes affecting them. Using analytical methodologies a number of researchers apply a top-down systems approach describing the processes that affect information systems and the transformational forces that influence them (Lehman, 1977; Woodside, 1980; Lehman, 1980, 1981, 1984, Yau, Nicholl, Tsai and Liu, 1988; Perry, 1994; Lehman, 1998).

Based on a series of empirical and analytical studies, Lehman et al. have developed eight laws of software evolution for embedded systems (Lehman and Belady, 1985; Lehman, et al., 1997). Much of the research on software evolution has sought to support these laws using relatively short data collection periods for operating systems software. (Lehman and Belady, 1985; Lehman, et al., 1997). Four of the eight laws on software evolution describe changes in system characteristics, while the other four deal with the interfaces and exchange of information between organizations and their embedded information systems (Lehman, et aL, 1997).

Current research on software evolution is headed in a number of different directions. Software evolution is providing a theoretical foundation for analysis of reverse engineering technologies and new perspectives on cost estimation tools. In addition, further work is being done on the FEAST/2 (Feedback Evolution and Software Technology) project, further investigating Lehman's eighth law, the Law of System Feedback (Lehman, et al., 1997).

SOFTWARE VOLATILITY

Another research perspective has concentrated on change processes and the definition and measurement of software volatility. Prior research on volatility relating to information systems has described software volatility as change in software product, or as change in software process. Practitioners routinely track software product change with version numbers. Traditional system-level versioning fails to track the size or frequency of software changes. Researchers often rely on token counts of modifications to measure software product change (Butcher, 1997; Banker and Slaughter, 2000). Yau and Collofello (1980; 1985) developed a measure of system instability by calculating logical ripple effect based on counts of cyclomatic complexity in software modules.

Software process volatility is measured by counting changes in data models or objects during software design and development (Marche, 1993; Li, et al., 2000). Heales (2000) develops a software volatility index to measure effort spent on deep structural changes during software change processes. However, all of these measures fail to answer the question about *how often* information systems are changed.

Existing measures of token counts of modifications over time are usually maintained at the program or module level. We define an aggregate measure of software volatility at the system-wide level that can be calculated at different times throughout productive system lifecycles. By recognizing the connection between an information system and its environment, we build a model describing antecedents of software by examining dynamic factors in that environment. Our measure of software volatility is used to empirically test this model.

Software volatility is a characteristic of the dynamic behavior observed in software evolutionary processes. This dynamic behavior implies system change. Some systems change frequently, and some seldom change. Each information system evolves and changes at its own pace. We identify information system change as software volatility so we can identify and analyze the differences in the timing of these changes. To concentrate on the time dimension of this software volatility we measure the intervals between software modifications. Increases in software volatility will manifest themselves as changes that will occur at shorter, more frequent, intervals. Decreases in software volatility will be seen as less frequent, longer, intervals between changes. This could be measured as Mean Time Between Failures (MTBF) as has been done in some software research studies (Lyu, 1996; Gaither, 1990). However, the measures used in software reliability engineering are only modeled for corrective software modifications. For studies of software evolution we need to consider all software modifications, regardless of the motivation for change, e.g. corrective, adaptive, enhancement and new program creation. To facilitate an analysis of changes in lifecycle system behavior, the time dimension measure of software volatility needs to be calculable at specified time periods throughout a system's productive life span. MTBF is generally calculated once, or only a few times, during the useful life of an artifact (Gaither, 1990).

We use *periodicity* as a measure of the time dimension of software volatility to describe mean time intervals between software modifications (Barry and Slaughter, 2000). Periodicity can be calculated as an aggregate measure at the system-wide level at any time interval needed for our analysis. This provides a descriptive measure to identify frequency of change. Periodicity will tell researchers and managers how often lifecycle software maintenance activities occur in each productive system. As we analyze software volatility we note that increased software modifications occur at more frequent intervals with decreasing periodicity. Alternatively, increasing periodicity indicates decreased levels of software volatility.

Software volatility can also be described by amplitude and deviation (Barry and Slaughter, 2000). Amplitude is defined as the size of software change, and deviation indicates the variation in behavior among systems. This research uses one dimension of software volatility, periodicity, as our dependent variable for two reasons.

First, the body of literature providing a foundation to identify antecedents of software volatility is most closely linked with token counts of modifications. Modification counts are logically comparable to the reciprocal of periodicity (Barry and Slaughter, 2000). Therefore, we can construct our model using hypotheses with rationale from this body of research. Because our previous work demonstrated discriminant validity among periodicity, amplitude and deviation, the dimensions of software volatility (Barry and Slaughter, 2000). The independence of periodicity, amplitude and deviation requires separate models for each of the dimensions of software volatility. To maintain a strong focus for the current investigation we pursue a single line of investigation.

The objective of this research is to identify the factors that drive software volatility as measured by periodicity, as in Figure 1. In the next section we review prior work on drivers of software change to develop a conceptual model predicting software volatility.

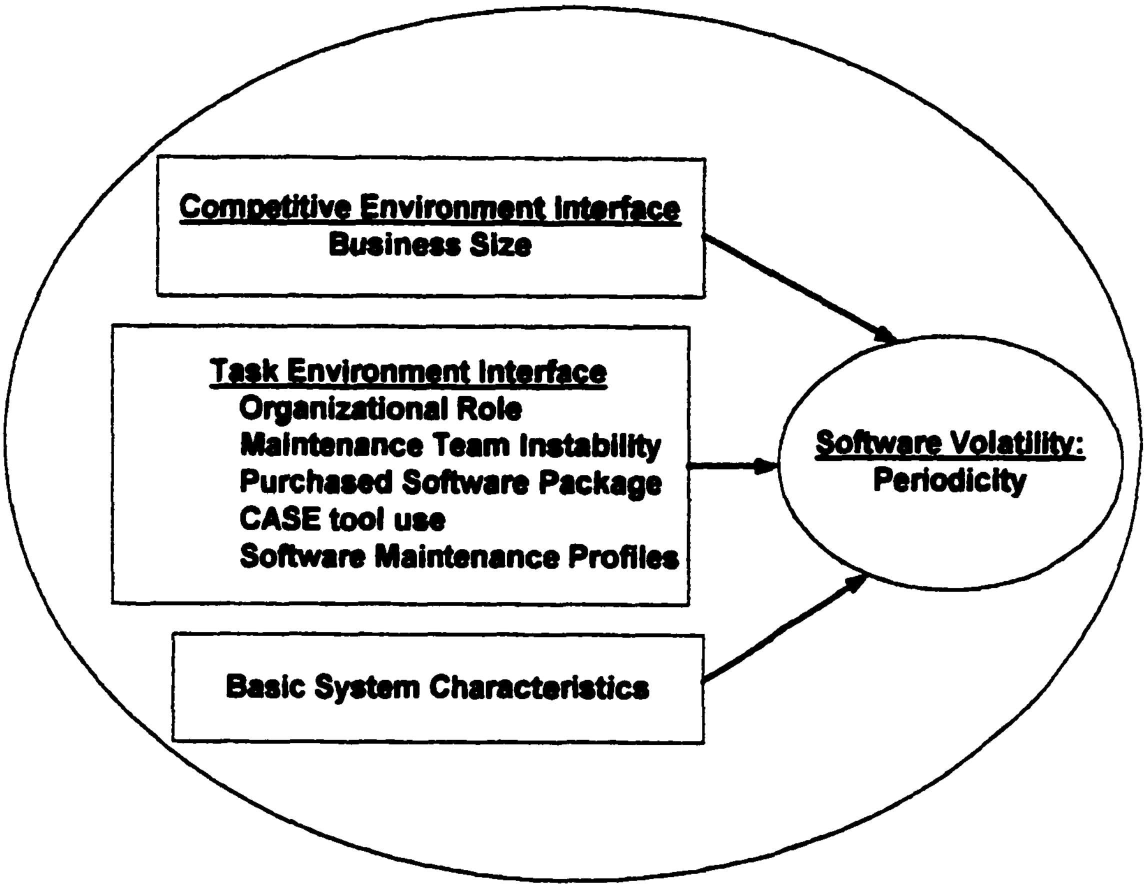

Competitive Environment Interface
Business Size
Task Environment Interface
Organizational Role
Maintenance Team Instability
Purchased Software Package
CASE tool use
Software Maintenance Profiles
Basic System Characteristics
Software Volatility:
Periodicity

Figure 1: Antecedents of Software Volatility

ANTECEDENTS OF SOFTWARE VOLATILITY

Each of the existing studies of software volatility uses a different definition and measure of software volatility and, therefore, as a consequence, predictive models for software volatility identify a wide variety of explanatory factors. Models predicting volatility of software products concentrate on those characteristics driving software change (Butcher, 1997; Banker and Slaughter, 2000; Yau and Collofello, 1980, 1985). Those models predicting volatility of software process focus analysis on attributes of both the software process and software product (Marche, 1993; Li, et al., 2000; Heales, 2000).

We draw from two bodies of literature to develop our model. First, we recognize that information systems are embedded in the overall general environment. As environments change, these information systems must change and evolve to remain productive. The mechanisms for this change is lifecycle software maintenance activities. Hence, we also examine existing models identifying drivers of software maintenance or software modifications. We start with the interface to the general competitive environment and progress to the localized task environment interface with the information system. We then examine the essential characteristics of information systems to assess their influence on levels of lifecycle software volatility.

Competitive Environment Interface

All organizations exchange resources with their environments. Organizations viewed as open systems participate in this exchange on a great many levels. More closed organizations may only exchange output with the outside world. However, most organizations operate as open systems and are viewed that way. As open systems, organizations rely on surrounding environments for resources, including information resources, needed to succeed (Scott, 1992). To maintain their productivity, organizations must change to keep pace with the dynamic nature of their competitive environment (Porter, 1980; Morgan, 1997; Highsmith, 2000). Organizations faced with dynamic environments have an increased need for informational resources to meet competitive challenges. These increased requirements can be met by increasing the internal resources available. Resources required for survival are the most immediate and relevant focus for organizations interfacing with their competitive environment (Dess and Beard, 1984).

An organization's ability to compete in the business environment and obtain needed resources can be crucial to organizational success. Size can be used as an indicator of the demand for an organization's products and services. Size is a dimension of organizational structure. Size indicates an organization's ability to compete against its cohorts for outside resources. Size is a variable on the interface between an organization and its environment. Size measures how much work an organization does (Scott, 1992).

Companies grow incrementally by building on their own success. Some large changes in company size occur through merger and acquisition, or divestiture of smaller companies or large company subdivisions. Any of these changes may affect information requirements and necessitate information system modification. If a business is involved in mergers and acquisitions, its information systems may need to be enhanced to provide services for new functional areas and increased services for a larger and more diverse constituency. These changes result in modification of information systems and increases in software volatility. Thus, we state the following hypothesis:

> *H1: Increasing business size will increase software volatility, i.e. decrease periodicity.*

Task Environment Interface

Each organization faces a number of varied tasks for its survival and success. Task environments are created to denote the parts of the organization relevant to, or potentially relevant to, accomplishment of these tasks (Thompson, 1967). Information systems are the tools organizations use to solve problems and accomplish necessary tasks. Thus, changes in the task environment directly affect software volatility. Primary characteristics of any task environment define the task and identify its domain. They include problem complexity and the number and variety of its constituency. These characteristics are frequently reflected in the information systems created to support these tasks, and in the processes used to develop and maintain those systems.

We capture differences in task domain by identifying the functional domain of each information system. The relationship between an information system and its organization is the primary task environment interface. This relationship is a system's organizational role. This role can be described several ways. A system can be identified by the functional business area it supports, e.g. human resources, operations, etc. The timing and quantity of changes in different business areas will be reflected in the volatility of systems supporting those functions. There are a number of ways to describe different functional roles performed by information systems. A system may serve a technical core function or a boundary-spanning function (Thompson, 1967; Scott, 1992), and there are different demands for the content and timeliness of information provided by boundary-spanners and non-boundary-spanners (Aldrich and Herker, 1977). Boundary-spanning information systems create and distribute information for users inside and outside an organization. Boundary-spanning information systems need to respond more often and more quickly to changes in an organization's external environment and produce new and different types of information as it become available. These systems may create annual financial reports for stockholders, monthly statements and special sales flyers for customers and online Just-In-Time delivery and inventory information for suppliers. Non-boundary-spanning information systems create information to be used within the organization, e.g. an organization's payroll system.

Information systems can also be used to provide buffers between external competitive forces and internal resources by influencing demand, leveling supply and demand of resources and product, forecasting and adjusting activities, and aiding with the organization's technical core (Scott, 1992).

Whether we examine the specific business area a system supports, the extent to which a system serves in a boundary-spanning capacity, or the strategic goal it facilitates, we are assessing the role the information system plays in its organization. Information systems spanning the boundary between one group and another, either within an organization or between an organization and the outside world, must be flexible and changeable to accommodate all stakeholders. Thus, boundary-spanning functional roles promote more volatility in the information systems supporting them. We formally state the following hypothesis:

> *H2: Information systems with boundary-spanning roles will have higher software volatility relative to those with non-boundary spanning roles, i.e. decreased periodicity relative to those systems with non-boundary-spanning roles.*

A number of studies have analyzed task team and task processes as dimensions of task environment volatility because they serve as dynamic elements of the task environment interface (Dess and Beard, 1984). Both the maintenance team and the processes used to develop and maintain information systems serve as mechanisms for changing software.

Prior research on software maintenance and software evolution has shown that team factors significantly influence software maintenance processes (Perry, 1994; Slaughter, 1995; Dekleva, 1992; Kemerer, 1995). We theorize that team factors also affect software volatility. No one is as knowledgeable about source code as is the source code author (Sacks, 1994). When other programmers try to modify source code, they often have difficulty because they are less familiar with code written by someone else.

Thus, changes in programmer-program assignments can result in increased errors and unnecessary modifications. Programmers unfamiliar with source code are likely to change the code more often, change more code than necessary, and change more programs than needed. If maintenance team members are uninformed of concurrent maintenance activities, software modifications may need to be rewritten and re-tested, forcing increases in software volatility. This *maintenance team instability*, i.e. changes in assignments and membership of the maintenance team supporting a system, can increase software volatility. The following hypothesis results:

> *H3: Increased maintenance team instability will increase software volatility, i.e. decrease periodicity.*

Task processes and procedures represent the standard operating procedures and processes used by an organization to accomplish specific tasks. We concentrate on those processes in the task environment relating to development and maintenance of information systems.

Software development practices have been shown to affect the levels of software volatility and lifecycle software maintenance (Lientz and Swanson, 1980; Banker and Slaughter, 2000; Heales, 2000). To begin examining differences in development processes we ask who developed the system. Software is often purchased from outsourcers because managers believe the organization lacks necessary in-house resources to create a reliable and efficient system (Lacity and Hirschheim, 1993; Kirkpatrick and van Scoy, 1993). Purchased packages are often assumed to require less lifecycle maintenance and expected to have reduced levels of software volatility. Under contractual agreement, the vendor often restricts maintenance of purchased software packages. Source code may be available for modification only to vendor personnel. In addition, many purchased information systems have lifecycle maintenance scheduled and controlled by the vendors. Outsourcers control lifecycle maintenance activities for purchased packages by scheduling modifications less frequently for large portions of the system, and install new versions of many system programs at the same time resulting in decreased software volatility, i.e. increased periodicity. This leads us to the following hypothesis:

> *H4: Purchased packages have decreased software volatility, i.e. increased periodicity.*

Structured development practices encourage the design of structured systems. Computer-Assisted System Engineering (CASE) tools reinforce the use of structured system design and controlled development methodologies (Low and Leenanuraksa, 1999). CASE tool proponents emphasize the time and effort saved by software developers and maintainers in dealing with source code (Martin, 1989). These tools make it possible to reduce maintenance effort even while increasing the changes occurring in the source code. CASE tools encourage re-engineering and replacing source code rather than maintenance of existing code (Martin, 1989). CASE tools are used to help implement a single design philosophy in an organization throughout its many projects and information systems. CASE tools improve system documentation by facilitating the creation and revision of complete current system documentation (Hoffer, George and Valacich, 1996). Thus, CASE tools facilitate software maintenance processes. Availability of CASE tools promotes change because the tools make it relatively easy to change the software. This will tend to make software modifications more frequent and decrease periodicity. Therefore, we assert that use of CASE tools will increase software modifications and software volatility, as measured by decreased periodicity.

H5: Increased use of CASE tools increases software volatility, i.e. decreases periodicity.

Lifecycle maintenance represents the largest force driving software modification. As much as 80% of the effort spent on information systems is expended during post-implementation lifecycle maintenance. The historical pattern of these incremental software changes can be used to describe a software change process composed of a variety of maintenance activities. Differences in patterns of these activities distinguish lifecycle maintenance processes used from one system to another. We refer to these historical patterns as lifecycle maintenance profiles. Prior research has shown that lifecycle maintenance profiles may vary widely from one information system to another, even among systems within the same organization. Research on software process volatility has shown the significance of prior modification profiles (Heales, 2000). For some systems empirical tests have demonstrated the significant contribution these profiles make in predicting software processing errors (Barry, Kemerer and Slaughter, 1999). Software changes accomplished through addition, change or deletion of source code will all result in some level of software faults (Malaiya and Denton, 1999). These software faults will require correction, precipitating software modification and increasing software volatility. Hence, we assert that lifecycle maintenance profiles are significant drivers in a generalized predictive model of software volatility. Thus, we state the following hypothesis:

H6: Increases in lifecycle maintenance profiles will increase software volatility, i.e. decrease periodicity.

Basic System Characteristics

Basic system characteristics are those attributes used to describe the essence of an information system. If we look at the interface of an information system and its environment, we ultimately view the inner core, i.e. the inner environment, as the substance and organization of the system itself (Simon, 1994).

Information systems are among the most complex and abstract of any artifacts humans have created (Simon, 1994; Brooks, 1995). Basic system characteristics are inextricably linked to the characteristics of the tasks they address. An information system is an abstract construct of interlocking concepts representing data sets and relationships. The inherent properties of information systems are often reduced to measures of their complexity, size and age (Brooks, 1995).

Previous research on software volatility has identified some software characteristics relating to the volatility of software products, including structure and complexity (Banker and Slaughter, 2000). Software complexity is a basic software product characteristic. Software complexity has been linked to software product volatility (Yau and Collofello, 1980; Banker and Slaughter, 2000). Increases in software complexity are also associated with increased levels of software maintenance (Banker, et al., 1997). We recognize the complexity of an information system is a mixture of task complexity and the complexity of its implemented solution. Total complexity is a basic description of the system we analyze (Wood, 1986; Banker, Davis and Slaughter, 1998). Thus, increases in software complexity will increase necessary software maintenance and, in turn, increase software volatility, i.e. decrease periodicity. We formally state the following hypothesis:

H7: Increased software complexity will increase software volatility, i.e. decrease periodicity.

Software size is also significant in predicting occurrence of software faults and software modifications (Kemerer, 1995; Banker, Datar, Kemerer, and Zweig, 1993). Larger programs and systems contain larger numbers of faults and require more modifications to correct those faults. System age is another basic system characteristic. Lehman et al. (1997) state three laws of software evolution describing system changes related to software aging.[1] Work on software process volatility has also identified the significance of software size and age (Heales, 2000). Analyses of software evolutionary processes imply that software volatility increases with age. As information systems age we expect an increasing divergence between them and their environments. Resolution of these discrepancies requires software modification resulting in increased software volatility. System size and system age are exogenous variables included as control variables in our predictive models of software volatility.

Our seven hypotheses are summarized in Table 1. Directional relationships specified in hypotheses H1 through H7 are diagrammed in Figure 2.

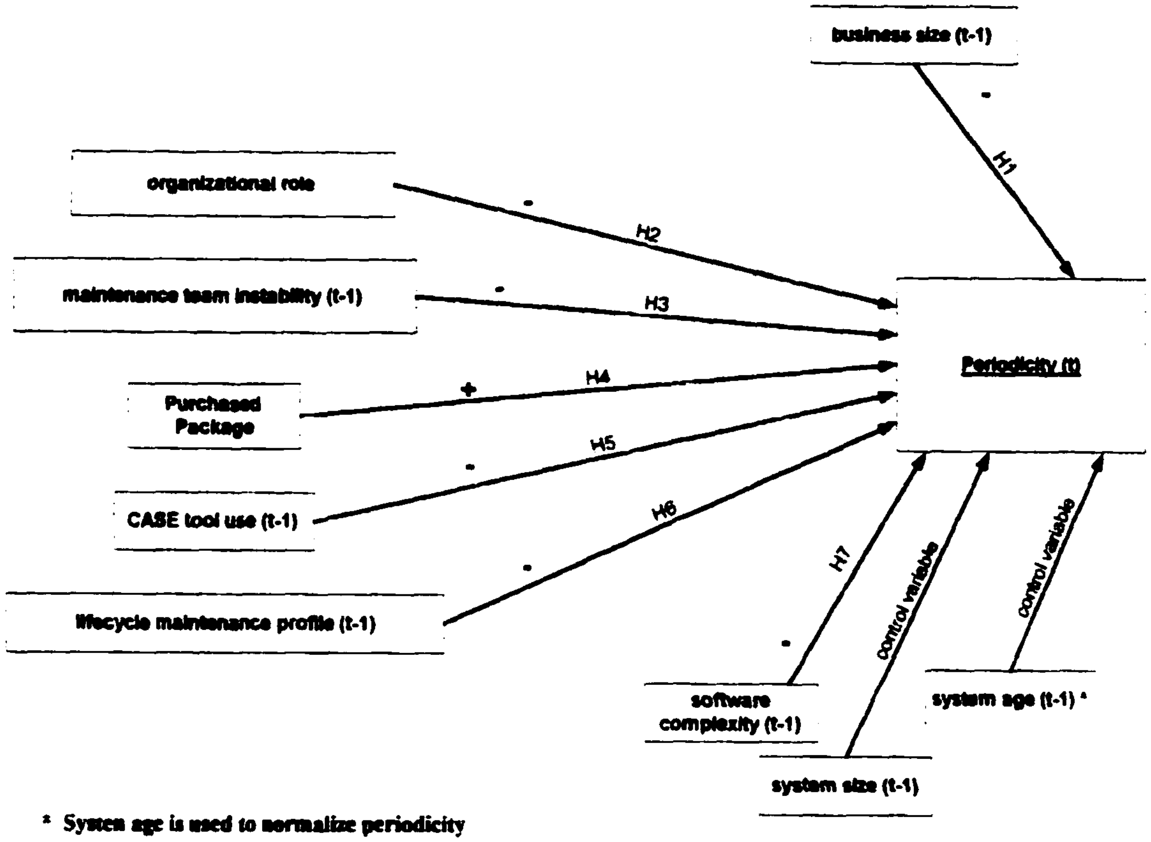
[1] These are the Law of Continuous Change (the 1st law), the Law of Increasing Entropy (the 2nd law), and the Law of Continuing Growth (the 6th law) (Belady and Lehman 1985).

The model for software volatility, *V*, we will be empirically testing is:

$$V = \beta_0 + \sum_{i=1}^{7} (\beta_i X_i) + \varepsilon$$

Where

β_0 = constant term
$\beta_1 X_1$ = coefficient and explanatory variable for business size (as in H1)
$\beta_2 X_2$ = coefficient and explanatory variable for role (as in H2)
$\beta_3 X_3$ = coefficient and explanatory variable for maintenance team instability (as in H3)
$\beta_4 X_4$ = coefficient and explanatory variable for purchased packages (as in H4)
$\beta_5 X_5$ = coefficient and explanatory variable for CASE tool use (as in H5)
$\beta_6 X_6$ = coefficient and explanatory variable for maintenance profiles (as in H6)
$\beta_7 X_7$ = coefficient and explanatory variable for complexity (as in H7)
ε = error term.



* Systen age is used to normalize periodicity

Figure 2: Predictive Model for Periodicity

METHODOLOGY

Research Site

The research site is a publicly owned mid-Western retailer with a portfolio of 23 information systems, including 3500+ software programs. This portfolio supports work for human resources, fiscal, operations and merchandising business functions. The company supports this large and varied software portfolio with its centralized Information Systems (IS) department. The IS department has separate development and maintenance units.

Data

During the software portfolio's 20-year history, the IS department maintained a detailed log of every modification made to each program, providing researchers with detailed information about 25000+ individual change events, i.e. any software modification for correction, adaptation, enhancement or creation of new programs (Kemerer and Slaughter, 1999).

Other available system characteristics include counts of programs, paragraphs, lines of code and each of Halstead's primitive measures (Conte, Dunsmore and Shen, 1986). Each program is flagged to indicate that CASE tools were used during its development or maintenance support. A binary variable indicates systems purchased as software packages. The indicator was set by detecting a vendor's name as the source code author.

Measures

Operational definitions for the model's dependent and explanatory variables are listed in Table 2. Each variable is measured for each month of the productive life span of each information system.

The dependent variable periodicity is measured as the system-wide average time interval between software modifications, relative to system age. Periodicity is calculated monthly to allow analysis of variation in software volatility throughout the productive lifecycle of an information system. We measure periodicity relative to system age to allow analysis across systems and throughout a system's lifecycle.

Explanatory variables in our predictive model of software volatility are grouped in three categories: (1) attributes of the competitive environment interface, (2) attributes of the task environment interface, and (3) basic system characteristics.

(1) Attributes of the Competitive Environment Interface: A company can adjust to changes in the competitive environment through growth, mergers and acquisitions, and divestiture of subdivisions. Changes in *Business Size* can be used to indicate these changes. We use annual revenue as a measure of business size[2]. Annual revenue is adjusted by the consumer price index (CPI) to correct for general economic conditions over the span of this longitudinal study.[3]

(2) Attributes of the Task Environment Interface: A number of attributes can be used to describe the task environment interface. The *role* an information system plays in an organization can be associated with the functional domain of the information system. Aldrich and Herker (1977) discuss the tension when acting as liason between groups from inside and outside an organization. They show that people and systems functioning in boundary-spanning roles face increased volatility in information requirements. This led us to hypothesize that information systems fulfilling boundary-spanning roles will have increased levels of software volatility. We operationalize these roles with dummy variables to indicate information system ownership by different functional areas of the company: human resources, operations, merchandising and fiscal. These functions each respond to the information needs of a different constituency. The human resources function serves internal stakeholders and would need information processing for such

[2] Annual revenue was obtained from each year of this publicly-owned company's annual report to stockholders.

things as payroll and benefits. We would expect information systems supporting the human resources function to facilitate internal information requirements and be a non-boundary-spanning function. In contrast, information requirements for the fiscal area would be set by a large and widely diverse group of stakeholders. Fiscal systems are required to produce specialized accounts payable systems and reports for the annual report of stockholders. We expect these information systems to support a boundary-spanning function. We use a fixed-effects model in our parameter estimates to distinguish the functional domain for information systems in the portfolio.

The composition of an organization's software maintenance team can be used to describe *maintenance team instability*. The detailed information in the maintenance logs for the organization's portfolio allows us to count the number of times lifecycle maintenance activities are completed by a programmer different from the programmer previously assigned to support that program. A count of these programmer swaps is used as the operational variable to describing maintenance team instability. This variable is aggregated at the system level by summation of the programmer swaps for each program in the system, for each month in the system life span.

A simple binary variable is used to identify which systems were *purchased software packages*. This will indicate those information systems where the development process was outsourced.

Development and maintenance processes are also described by the *use of CASE tools* in an information system. Each program in our portfolio was marked as using, or

[3] All financial data adjusted by CPI reported by U.S. government and reported in the 1999 World Almanac and Book of Facts, p. 111.

not using, a commercial CASE tool during development and maintenance. System CASE-tool-use is aggregated as the monthly count of programs in a system developed or maintained by CASE tools divided by the number of programs in the system, i.e. the portion of each system's programs using CASE tools. This calculation allows CASE-tool-use to vary from system to system and throughout the productive life span of individual systems.

Maintenance processes are described by *software maintenance profiles*. Historical patterns of lifecycle software maintenance activities are classified by motivation for the modifications: corrective, adaptive, enhancement and new program creation. These variables were operationalized for our empirical tests by calculating the proportionate mix of corrective, adaptive, enhancement and new program creations for each month in each system's life span.

(3) Basic System Characteristics: Three essential characteristics of information systems are complexity, size and age. *Software complexity* can be broken down as component, coordinative and dynamic complexity (Wood, 1986; Banker, Davis and Slaughter, 1998). We use the following software product metrics for these complexities and normalize them by system size, i.e. total lines of code (LOC). Component complexity is operationalized as normalized system total unique operands, i.e. Halstead's n2/(LOC). Coordinative complexity is operationalized using normalized system total McCabe's cyclomatics, i.e. total cyclomatics/(LOC). Dynamic complexity is operationalized as normalized program calls, i.e. total calls/(LOC).

The operational measure of periodicity is normalized relative to system age. Therefore, we do not include system age as a separate explanatory variable in our predictive model. The control variable for system size is operationalized as current total LOC.

	Construct	Operational Variable	Description	Unit Of Analysis Varies...
H1	Business size	Annual revenue	Company's total annual revenue adjusted by CPI	By year
H2	Organizational role	4-way fixed effects for business area supported	dummy variables to indicate one of 4 business areas: human resources, fiscal, operations, merchandising	By system
H3	Maintenance team instability	Programmer swap count	Count of each program modification completed by someone other than the previous programmer to maintain that program	By system by month
H4	Purchased software package	Package	Binary variable 1 = purchased package 0 = not purchased	By system
H5	CASE tool development	CASE tool use	System-wide average of CASE-tool-use indicators	By system by month
H6	Maintenance profiles	Corrective mix	portion of maintenance activities classified as corrective	By system by month
		Adaptive mix	portion of maintenance activities classified as adaptive	
		Enhancement mix	portion of maintenance activities classified as enhancement	
		New program mix	Portion of activities classified as new program creation	
H7	complexity	Normalized Component complexity	System-wide count of Halstead's n2 (unique operands) normalized by system size	By system by month
		Normalized coordinative complexity	System-wide count of McCabe's cyclomatics normalized by system size	
		Normalized dynamic complexity	System-wide count of program calls normalized by system size	
	System size	Total LOC	Control variable - current total lines of code in system	By system by month

Table 1: Antecedents of Software Volatility

Our data set was built using the variables in Table 1 for the full productive lifecycle of the 23 information systems in the company's portfolio. Data collection started for each information system on the date of its initial implementation, and continued until the end of the data collection period or until the system was no longer in use, whichever came first. The result is an unbalanced panel data set containing 3201 observations. We include lagged explanatory variables, i.e. for time period *t-1*, to accommodate our model predicting software volatility, i.e. periodicity, in time period *t*. This slightly reduces our panel to 3178 observations.

RESULTS

Descriptive Statistics

Table 2 reports descriptive statistics for each of the operational variables used in our predictive model. Table 3 reports the inter-correlations. In the software portfolio there are four information systems for the human resources area, seven for operations, two for merchandising and ten for the fiscal area. Four of the 23 systems in the software portfolio are purchased packages.

Variable -all systems 3178 observations in 23 systems	Mean	Std. Dev.	Min.	Max.
Software volatility - periodicity	0.4826652	0.459903	0	1
Annual revenue	9487.367	3232.399	3443.309	14715.38
Programmer swap count	3.323474	6.278237	0	68
CASE-tool-use	0.1633094	0.2640052	0	1
Corrective mix	0.0674294	0.1577636	0	1
Adaptive mix	0.0407775	0.226535	0	1
Enhancement mix	0.3510842	0.392479	0	1
New program creation mix	0.1073055	0.258088	0	1
Component complexity: n2 / lines of code	2188174	0608182	1191962	3940193
Coordinative complexity: McCabe's cyclomatics / lines of code	58445	25487	337224	1938202
Dynamic complexity: Calls / lines of code	0079633	0054584	0	0304348
Total LOC	188705.2	262514.7	187	1279163
System age (in months)	84.06671	59.31983	2	246

Table 2: Descriptive Statistics of Operational Variables

By definition and the construction of operational variables, there are upper and lower bounds on each of the relative measures, i.e. periodicity and each of the mix proportions. We note a relatively high correlation between CASE tool use and both coordinative and component complexity, and among the three measures of compkexity. We will expand our analysis of this after reviewing parameter estimate from regression results. We will clarify this further and eplain the implications later in the discussion section.

	β1	β3	β5	β6(1)	β6(2)	β6(3)	β6(4)	β7(1)	β7(2)	β7(3)
annual revenue β1	1									
programmer swap count β3	0.2444	1								
CASE-tool-use β5	0.3387	0.3286	1							
corrective mix β6(1)	0.2184	0.1017	0.2754	1						
adaptive mix β6(2)	0.1419	0.3384	0.2584	0.0431	1					
enhancement mix β6(3)	0.4719	0.3707	0.4039	0.1079	0.1221	1				
new program creation mix β6(4)	-0.0444	0.1033	0.0183	-0.0796	-0.0525	-0.1447	1			
component complexity β7(1)	-0.3209	-0.2390	-0.6943	-0.2363	-0.1474	-0.3342	-0.0521	1		
coordinative complexity β7(2)	-0.3272	-0.2354	-0.4401	-0.1815	-0.1434	-0.3318	-0.0390	0.5380	1	
dynamic complexity β7(3)	0.1405	0.1865	0.5752	0.1573	0.1078	0.2193	0.0128	-0.2752	-0.1867	1
Total LOC	0.4647	0.3738	0.7817	0.2698	0.2553	0.3816	0.0043	-0.5927	-0.3643	0.4190

Table 3: Correlations of Operational Variables

Parameter Estimates

A predictive model for periodicity was estimated using Generalized Least Squares methods. As is often the case with panel data, i.e. pooled time series data, we found evidence of serial correlation. A panel-specific correction for AR1 level serial correlation was employed after the Breusch-Godfrey test confirmed autocorrelation (Johnston, 1984). Separate regressions were run for each system's time series data. These regressions reported a wide variation in Durbin-Watson statistics, indicating some systems had strong serial correlation, and some were hardly affected. This indicated that a panel specific correction would be more appropriate than using the same AR1 correction for the entire panel. This was confirmed by comparison of Wald statistics from estimates using AR1 corrections against the Wald statistics from estimates using panel specific AR1 corrections.

Estimated parameters for our predictive model of periodicity are reported in Table 4 and results of hypothesis tests are reported in Table 5. Empirical tests support all but one of our hypotheses. Residuals were examined for outliers by identifying observations resulting in residuals more than three standard deviations from the mean residual (Neter, Wasserman and Kutner, 1990; Belsley, Kuh and Welsch, 1980; Baroudi and Orlikowski, 1987). After removing 16 outliers, parameter estimates remained consistent with our original results.

As hypothesized, we find that increases in business size, team instability and lifecycle maintenance are associated with increased software volatility, i.e. decreased periodicity.

The estimated coefficient for component complexity, i.e. [N2/system-total LOC] has a sign opposite from that hypothesized (H7). We elaborate on this unexpected result and provide a possible explanation in our discussion section.

Log likelihood = -667.5318 N = 3178	Wald = 4866.67	*** ≈ p ≤ 0..001 ** ≈ p ≤ 0.05	
Operational variable	Estimated β	p-value	
Constant	0.9524293	0.000	***
Annual revenue (t-1)	0.0000229	0.000	***
Business area ~ human resources	-0.0456381	0.137	
Business area ~ fiscal	-0.0771880	0.006	***
Business area ~ operations	-0.0523706	0.031	**
Programmer swap count (t-1)	-0.0070583	0.000	***
Purchased package	0.1764679	0.000	***
CASE-use (t-1)	-0.0954437	0.094	
Corrective mix (t-1)	-0.4380241	0.000	***
Adaptive mix (t-1)	-0.4148038	0.000	***
Enhancement mix (t-1)	-0.4895813	0.000	***
New program creation mix (t-1)	-0.4275023	0.000	***
Component complexity: n2 / lines of code (t-1)	0.4609087	0.001	***
Coordinative complexity: McCabe's cyclomatics / lines of code (t-1)	0.1815908	0.638	
Dynamic complexity: Calls / lines of code (t-1)	-1.5444250	0.305	
Total LOC (t)	-0.0000001	0.002	***

Table 4: Regression Estimate for Drivers of Periodicity

	Supported? *** ≈ p ≤ 0..001 ** ≈ p ≤ 0.05		Predicted sign	Hypotheses concerning Competitive Environment Interface:
H1	Yes	***	-	Increasing business size will increase software volatility.
H2	Yes	**	-	Information systems with boundary-spanning roles have increased volatility, i.e. decreased periodicity, relative to those with non-boundary-spanning roles.
H3	Yes	***	-	Increased maintenance team instability will increase software volatility.
H4	Yes	***	+	Purchased packages have decreased software volatility.
H5			-	Increased use of CASE tools increases lifecycle software volatility.
H6	Yes	***	-	Increases in software maintenance profiles will decrease periodicity.
H7	No		-	Increased software complexity will decrease periodicity.

Table 5: Hypotheses test results

DISCUSSION

We built our conceptual models for drivers of software volatility based upon the literature in software evolution and lifecycle software maintenance. We used a measure of periodicity of lifecycle software maintenance activities as a measure of software volatility. By emphasizing the close connection between information systems and their environments, we built a predictive model for the antecedents of software volatility, i.e. periodicity. Focusing on environmental influences, these antecedents are identified from the dynamic attributes of interfaces between information systems and the competitive and task environments. We also included attributes of the basic information system to represent the core inner environment of all systems. Parameter estimates for our predictive model of periodicity lend strong support to this approach. In the following paragraphs we discuss the results for characteristics of each environmental level.

Competitive Environment Interface

Business size operationalized as annual revenue is negatively related to periodicity of software volatility. Thus, growing businesses will face increased software volatility and more frequent software modifications to their information systems portfolio. This provides support for hypothesis 1. While perhaps intuitive, this result (1) alerts managers to build this effect into their cost planning, and (2) allows us to interpret the other effects in the model with greater confidence.

Task Environment Interface

We hypothesized that the organizational role influences the volatility of an information system supporting that role (H2). The predictive model demonstrates the significance of organizational role, i.e. functional domain, supported by each information system. Information requirements for each information system vary according to the tasks assigned. As a result, we expected information systems supporting boundary-spanning activities to have higher levels of software volatility as compared to those supporting non-boundary-spanning activities. Boundary-spanning activities share information between organizations.

The organizational role of the information system is a significant driver in our empirical model as indicated by the significance of the group of fixed-effects variables designating business area (F = 24.398, p-value = 0.00) (Greene, 1997). The functional domains supported by information systems in four business areas have distinctly different levels of software volatility (F = 21.49173, p-value 0.00) (Greene, 1997). Using the merchandising group as a reference group, we observe that the fiscal systems have the lowest periodicity and are, thus, the most volatile. This is as we expected for boundary-spanning systems like the information systems supporting the fiscal domain.

As hypothesized, increases in maintenance team instability will increase software volatility and decrease the time intervals between software modifications, i.e. decrease periodicity (H3). Managers should be mindful of the effect of changing the programmer-program assignments for lifecycle maintenance. Increased swapping of programmer assignments will cause more software modifications than if the program's sequential modifications are handled by the same programmer. The implicit knowledge programmers collect as they familiarize themselves with a program and modify it is not likely to be easily or completely transferred. If the program is assigned to a different programmer each time modifications are required, each person must build their knowledge of the program for each change. Unnecessary modifications may result as newly assigned programmers familiarize themselves with the source code.

As expected, purchased software packages are modified less frequently and have lower periodicity, than information systems developed in-house (H4). Even though these results seem counter-intuitive, they are consistent with results in Banker and Slaughter (2000), showing that the use of CASE tools promotes increased levels of software volatility (H5). Dekleva (1992) describes use of CASE tools as one way to judge the amount of structure in system design, associating strong structuring techniques with use of CASE tools. Banker & Slaughter (2000) show that more highly structured systems have higher levels of volatility, i.e. more modifications. The results we obtain demonstrate a negative relationship between CASE tools and periodicity. our parameter estimate, though marginally significant, indicates that CASE tool use results in more frequent system modification, providing further support for the Banker and Slaughter result. Thus, CASE tools increase software volatility. This seems counter-intuitive until we remember that CASE tools were never designed to stabilize software, only to make it easier and less time-consuming to change.

Increases in lifecycle maintenance profiles increase software volatility, i.e. decrease periodicity (H6). We operationalized historical patterns of lifecycle maintenance profiles with four variables representing the proportion of lifecycle maintenance activities devoted to each motivating maintenance category: corrective, adaptive, enhancement and new program creation, respectively. Their combined values indicate the presence, or absence, of lifecycle maintenance activity in time $t-1$. We used historical patterns of lifecycle maintenance activities, i.e. software maintenance profiles, for time $t-1$ to predict software volatility. Our results indicate that lifecycle maintenance activity in time period $t-1$ will increase software volatility during time period t.

Basic System Characteristics

Prior research shows that increased software complexity will increase software maintenance effort (Banker, Datar, Kemerer and Zweig, 1997). We hypothesized that increased complexity will increase software changes, thus increasing software volatility and decreasing periodicity (H7). Software complexity was operationalized three ways; as component complexity, as coordinative complexity, and as dynamic complexity. This reflects the types of cognitive complexity programmers face in creating task solutions with their source code. Our parameter estimates yield a positive coefficient for component complexity and insignificant coefficients for coordinative and dynamic complexities.

Component complexity is measured as Halstead's n2 normalized by system total LOC. Because Halstead's n2 counts unique data elements, high levels of component complexity indicate information systems with relatively high levels of data-intensity. There is research to indicate data intensive systems are more stable than those based on process-driven models (Martin, 1989; Hoffer, George and Valacich, 1996). Our model estimates for component complexity appear to support those findings.

Coordinative complexity and dynamic complexity measure the complexity of decision branching in each program, and the call structure between programs within the system, respectively. To make the inherent complexity of any task easier to deal with, problem solvers often reduce complexity by breaking the task into smaller chunks (Simon, 1994). By encouraging the creation of smaller, reusable programs, CASE tools promote this same approach in re-engineering and maintaining software (Martin, 1989; Low and Leenanuraksa, 1999). This change in design will also affect measures of component and dynamic complexity.

CASE tools generate source code by using heuristics designed to create systems with a large number of short reusable programs. These programs are accessed by program calls. Thus, systems relying on CASE tools for source code generation will have higher levels of dynamic complexity. CASE tools are used to generate programs with simpler logic flow. These programs will have reduced coordinative complexity. Once again, we note the relatively high correlation between the use of CASE tools and measures of system complexity (corr(CASE-tool-use, component complexity) = -0.6943; corr(CASE-tool-use, coordinative complexity) = -0.4401; corr(CASE-tool-use, dynamic complexity) = 0.5752). Pair-wise correlations also indicate that systems with more programs generated by CASE tools are larger and have, on average, larger programs (corr(CASE-tool-use, system size) = 0.7786; corr(CASE-tool-use, average program size) = 0.8394; corr(CASE-tool-use, programs in system) = 0.4115). These correlations lead us to conclude that, in general, CASE tools generate systems with more programs of smaller size. These programs have lower values of component and coordinative complexity. They have higher complexity, i.e. they have increased levels of program calls per LOC. By encouraging reuse of code, CASE tools generate programs to perform generalized functions and use program calls to access those programs from other programs in the system.

To check for the effects of multicollinearity among the measures of complexity and CASE-tool-use, we re-estimated the model by omitting each of the four variables individually. The results were consistent with those we obtained with the full model in Table 4.

CONCLUDING REMARKS

This research contributes to the breadth and depth of our understanding of the antecedents of software volatility. Our analysis indicates that dynamic attributes of the environmental interfaces can be used to predict periodicity in software volatility. We view an information system's environment as organized in successive layers, i.e. the competitive environment, the task environment and the inner environment of the system itself. We find that dynamic attributes describing an information system's interfaces to each of these environments are significant drivers of software volatility.

Increasing business size can increase volatility by shortening the interval between software modifications, i.e. decreasing periodicity. Increases in business size can indicate the acquisition or creation of new business units. These additions will have new or changed information requirements. In addition, constituencies for existing systems may increase. The need to satisfy information requirements for additional users will result in an increased need for software modifications and increases in software volatility.

When organizations are viewed as open systems, the boundaries between them become difficult to identify. Depending on organizational structure, there are boundaries within organizations, between sections, departments and divisions. Information is often shared between subdivisions or departments, or between a company and its strategic partners. In the aftermath of mergers and acquisitions, information systems that had previously been viewed as non-boundary spanning may become boundary- spanning. Recognition of the tie between software volatility and the functional domain it supports can include the need for flexibility in supporting boundary-spanning activities at any level of an organizational hierarchy. Use of fixed-effects variables for business area classifications capture a number of differences in the volatility in task environments and their association with information system behavior. A more detailed classification of the task performed and its associated functional domain would provide researchers with a greater understanding of this source of volatility.

Purchased software will be modified less frequently, i.e. software packages have increasing periodicity. The decision to buy a software package is often based on an expectation of improved software quality and less need for software maintenance. Our findings appear to confirm this. Many software vendors control lifecycle maintenance by grouping modifications and releasing sets of changes, or new system versions at one time. This would result in lengthened intervals between modifications and increased periodicity.

We speculate that CASE tool use may be interfering with the effect complexity has on software volatility. The objective of CASE tools is to reduce the effort needed for software development and modification. However, CASE tools generate larger systems with more short programs. The individual programs are less complex, but have increase levels of dynamic complexity, i.e. more program calls per LOC. CASE tools also encourage re-engineering and regeneration of source code. Because CASE tools allow code generation with relatively little programmer effort. Thus, CASE tools break the connection between size and complexity of software and effort required to create and to maintain that software. For the same reason, we believe the same interference is affecting the results we obtain in examining the relationships between sofware volatility and CASE tools.

We can use historical patterns of software maintenance activities, i.e. software maintenance profiles, to predict software volatility. Our results indicate that lifecycle maintenance activity in the previous time period will increase subsequent levels of software volatility.

Increased component complexity, indicating data-intensive systems, will decrease software volatility and increase periodicity. Systems with relatively high numbers of data elements are likely to be relatively stable.

Implications for Future Research

This research expands our understanding of software evolutionary changes by searching for drivers of software volatility, i.e. software change. By relating the time dimension of software volatility to the dynamic attributes of environments surrounding an information system we have identified significant factors affecting software volatility, i.e. periodicity. We learned that factors from each level of the general environment contribute to software lifecycle changes.

In our discussions about software volatility we find it very easy to make implicit assumptions about software volatility. Software volatility is often viewed as *bad*, or *something to be avoided*. We don't really know this to be true. Future work should examine the effect of software volatility on software maintenance outcomes. Researchers should also examine the moderating effect of software volatility on the influence of other factors in predicting lifecycle maintenance costs or processing errors.

Implications for Practice

We return to our original questions: "Why do we have to keep *fixing* this software? Why can't you write a *good* system, so we don't need these *constant changes*?" We often assume software volatility, i.e. software change, is bad, and should be avoided. By recognizing the connection between information systems and their surrounding environments, we see that change is often unavoidable. With the continued presence of long-lived systems we understand that unchanging information systems can have negative consequences. Consequently, practitioners should view software evolution and software volatility as inevitable. Our identification of the drivers of software volatility can help software managers by focusing attention on those drivers within a manager's control while anticipating resources needed for software lifecycle maintenance task.

This work has demonstrated the effects that managerial decisions concerning software sourcing, CASE tool use and staffing assignments can have on software volatility and system behavior. It is too easy to make the assumptions that all software volatility is bad and immediately leads to increased maintenance costs. These valid questions are beyond the scope of this work. However, the ability to anticipate levels of software volatility will help managers become more proactive in dealing with lifecycle software maintenance.

By measuring software volatility and identifying the factors driving volatility, researchers and practitioners can all improve their understanding of the transformations occurring during software evolution. Knowing which factors influence software volatility, researchers and managers can focus on controllable factors to improve management of software evolutionary processes.

REFERENCES

Aldrich, H. and D. Herker, "Boundary Spanning Roles and Organizational Structure", Academy of Management Review, Vol. 3, 1977.

Banker, R.D., S.M. Datar, C.F. Kemerer, and D. Zweig, November 1993, "Software Complexity and Maintenance Costs", Communications of the ACM, Vol. 36, No. 1,November, 1993, pp. 81-93.

Banker, R., G.B. Davis, and S.A. Slaughter, "Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study", Management Science, Vol. 44, No. 4, Apr. 1998.

Banker, R.D., and S.A. Slaughter, "The Moderating Effects of Structure on Volatility an Complexity in Software Enhancement", Information Systems Research, Vol. 11, No. 3, Sept. 2000, pp. 219-240.

Baroudi, J.J., and W.J. Orlikowski, " The Problem of Statistical Power in MIS Research", MIS Quarterly, March, 1989, pp.87-105.

Barry, Evelyn J., Kemerer, Chris F., and Slaughter, Sandra A., "An Empirical Analysis of Software Evolution Profiles and Outcomes", Proceedings of the International Conference on Information Systems, Charlotte, NC, December 1999.

Barry, Evelyn J., and Slaughter, Sandra A., "Measuring Software Volatility: A Multi-Dimensional Approach" (extended abstract), Proceedings of the International Conference on Information Systems, Brisbane, Australia, December 2000.

Belady, L.A. and M.M. Lehman, "A Model of Large Program Development", IBM Systems Journal, Vol. 3, 1976, pp. 225-252.

Belsley, D.A., E. Kuh, and R.E. Welsch, Regression Diagnostics: Identifying Influential Data and Sources of Collieanity, John Wiley and Sons, New York, 1980.

Brooks, F.J., The Mythical Man-Month, Addison-Wesley Publishing Co. 1995.

Butcher, G., 1997, Addressing Software Volatility in the System Life Cycle, Ph.D. Dissertation, Colorado Technical University, UMI#9815557.

Conte, S., H. Dunsmore, V. Shen, Software Engineering Metrics and Models, Benjamin/Cummings, Menlo Park, CA, 1986.

Dekelva, S.M., "The Influence of the Information Systems Development Approach on Maintenance", MIS Quarterly, 1992, pp. 355-372.

Davis, G.B., M.H. Olson, Management Information Systems: Conceptual Foundations, Structure, and Development, 2nd Edition, McGraw-Hill Book Company, 1985.

Dess, G.G. and D.W. Beard, "Dimensions of Organizational Environments", Administrative Science Quarterly, Vol. 29, 1984, pp. 52-73.

Gaither, N., Production And Operations Management: A Problem-Solving and Decision-Making Approach, 4th Edition, The Dryden Press, 1990.

Greene, William H., Econometric Analysis, Third Edition, Prentice Hall, Upper Saddle River, NJ, 1997.

Heales, J., "Factors Affecting Information Systems Volatility", *Proceedings of the International Conference on Information Systems 2000*, Brisbane, Australia, 2000,.

Highsmith, J.A., III, Adaptive Software Development: A Collaborative Approach to Managing Complex Systems, Dorset House Publishing, NY, 2000.

Hoffer, J.A., J.F. George, and J.S.Valacich, Modern Systems Analysis and Design, The Benjamin/Cummings Publishing Company, Inc., Reading, MA, 1996.

Johnston, J., Econometric Methods, Third Edition, McGraw-Hill, Inc., New York, 1984.

Kalakota, R., and A.B. Whinston, Electronic Commerce: A Manager's Guide, Addison-Wesley, Reading, MA, 1996.

Kemerer, C.F., "Software complexity and software maintenance: A survey of empirical research", Annals of Software Engineering, Vol. 1, Sept. 1995, pp. 1-22..

Kemerer, Chris F. and Slaughter, Sandra A., 1999, "An Empirical Approach to Studying Software Evolution", IEEE Transactions on Software Engineering, Vol. 25, No. 4, 1999, pp. 493-509.

Kirkpatrick, R.J., and R. Van Scoy, "Potential Risks to Software Development Projects from the Use of COTS Components", Proceedings of 5th Annual Software Technology Conference, Salt Lake City, Utah, 1993.

Lacity, M.C. And R. Hirschheim, Information Systems Outsourcing: Myths, Metaphors and Realities, John Wiley and Sons, New York, 1993.

Lehman, M.M., "Human Thought and Action as an Ingredient of System Behavior", Encyclopedia of Ignorance, R. Duncan and M. W. Smith (Eds), Pergamon Press, Oxford, 1977.

Lehman, M.M., "Programs, Life Cycles and Laws of Software Evolution", Proceeding of IEEE Special Issue on Software Engineering, Vol. 68, No. 7, 1980, pp.1160-1176.

Lehman, M.M., "Programming Productivity - A Lifecycle Concept", Proceeding CompCon '81, 1981, pp. 232-241.

Lehman, M.M., "Program Evolution", Information Processing and Management, Vol. 20, 1984, pp.19-36.

Lehman, M.M., "Software's Future: Managing Evolution", IEEE Software, 1998, pp. 40-44.

Lehman, M.M., and L.A. Belady, Program Evolution: Processes of Software Change, Academic Press, London, 1985.

Lehman, M.M., J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski, "Metrics and Laws of Software Evolution - The Nineties View", Metrics '97, the Fourth International Software Metrics Symposium, Albequerque, NM, 1997.

Li, W., L. Etzkorn, D. Davis, and J. Talburt, "An Empirical Study of Object-Oriented System Evolution", Information and Software Technology, Vol. 42, No. 6, 2000, pp. 373-381.

Lientz, B.P., and E.B. Swanson, Software Maintenance Management, Addison-Wesley, Reading, MA, 1980.

Low, G., and V. Leenanuraksa, "Software Quality and CASE Tools", Proceedings of Software Technology and Engineering Practice, STEP '99, Pittsburgh, PA, 1999, pp. 142-150.

Lyu, M.R., Handbook of Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1996.

Malaiya, Y.K., and J. Denton, "Requirements Volatility and Defect Density", Proceedings 10th International Symposium on Software Reliability Engineering, 1999, pp. 285-94.

Marche, S., "Measuring the Stability of Data Models", European Journal of Information Systems, Vol. 2, No. 1, 1993.

Martin, J., Information Engineering: Book I Introduction, Prentice Hall, Englewood Cliffs, NJ, 1989.

Morgan, G., Images of Organization, Sage Publications, Thousand Oaks, CA, 1997.

Neter, J., W. Wasserman, and M.H. Kutner, Applied Liner Statistical Models Regression, Analysis of Variance, and Experimental Design, 3rd Edition, Richard D. Irwin, Inc., Burr Ridge, IL, 1990.

Perry, D.E., "Dimensions of Software Evolution", IEEE Conference on Software Maintenance, IEEE, 1994.

Pfleeger, S., "The Nature of System Change", IEEE Software, Vol 15, No. 3,1998, pp. 87-90.

Porter, M.E., Competitive Strategy: Techniques for Analyzing Industries and Competitors, The Free Press, New York, 1980.

Sacks, M., On the Job Learning in the Software Industry: Corporate Culture and the Acquisition of Knowledge, Quorum Books, Westport, CT, 1994.

Scott, R.W., Organizations: Rational, Natural, and Open Systems 3rd Edition, Prentice Hall, Englewood Cliffs, NJ, 1992.

Simon, H.A., The Sciences of the Artificial, The MIT Press, Cambridge, MA, 1994.

Slaughter. S.A.., Software Development Practices and Software Maintenance Performance: A Field Study, Ph.D. Dissertation, University of Minnesota, 1995.

Thompson, J.D., Organizations in Action, New York, McGraw Hill Book Co., 1967.

Truex, D.P., R. Baskerville, and H. Klein, "Growing Systems in Emergent Organizations", Communications of the ACM, Vol. 42, No. 8, 1998, pp. 117-123.

Wood, R.E., "Task Complexity: Definition of a Construct", Organizational Behavior and Human Decision Processes, Vol. 37, 1986, pp. 60-82.

Woodside, C.M., "A Mathematical Model For the Evolution of Software", Journal of Systems and Software, Vol. 1, No. 4, 1980.

World Almanac and Book of Facts 1999, World Almanac Books, 1999, Mohawk, NJ, 111.

Yau, S.S., and J. Collofello, "Some Stability Measures for Software Maintenance", IEEE Transactions on Software Engineering, Vol. 6, No 11,1980, pp. 545+.

Yau, S.S., and J. Collofello, "Design Stability Measures for Software Maintenance", IEEE Transactions on Software Engineering, Vol.11, No. 9 ,1985, pp. 849-856.

Yau, S.S., R.A. Nicholl, J.J. Tsai, and S.S. Liu, "An Integrated Life-Cycle Model for Software Maintenance", IEEE Transactions on Software Engineering, Vol.14, No. 8, 1988, pp. 1128-1144.

CHAPTER 5:

RESEARCH QUESTION 3 -

CHARACTERISTICS OF SOFTWARE EVOLUTION AND LIFECYCLE MAINTENANCE OUTCOMES

INTRODUCTION

Information systems (IS) managers universally cope with the task of lifecycle software maintenance. Despite the importance of software maintenance, IS managers deal with management of software maintenance in a predominantly reactive, rather than proactive, manner. This is partially due to the difficulty in forecasting lifecycle maintenance outcomes and predicting lifecycle maintenance resource requirements.

Many information systems serve their organizations for upwards of fifteen years (Kalakota and Whinston, 1996) and outlive the tenure of the programmers and IS managers that develop them (Swanson and Dans, 2000). To forecast lifecycle maintenance outcomes, IS managers need to deal with currently implemented systems. The system characteristics and management decisions from system development may no longer be available. Current legacy systems may vary from those originally implemented. In fact, an information system may change so much that original characteristics may no longer resemble the current system. Task and organizational environments can change dramatically during the years an information system is in productive use. Lifecycle maintenance processes are used to enhance information systems allowing them to evolve in parallel to their surrounding environment (Pfleeger, 1998). We recognize these life-long transformation processes as *software evolution*.

As Swanson and Dans recently observed, lifecycle maintenance activities are forward-focused procedures striving to lengthen the productive life of an information system (Swanson and Dans, 2000). Likewise, in this research we focus on prediction of lifecycle maintenance outcomes to improve lifecycle maintenance management by enhancing predictive models of maintenance outcomes, i.e. processing errors and maintenance costs.

By sustaining a forward perspective we use information about a current information system and its recent changes. The objective of our research is to use basic characteristics of the current system and characteristics of its recent software evolution to predict lifecycle maintenance outcomes, i.e. software processing errors and lifecycle maintenance costs. What effect does software evolution have on future maintenance costs? If two information systems are described with identical size and complexity, should we expect them to have the same maintenance costs and error rates? Do their different lifecycle maintenance histories, i.e. their different patterns of software evolution, affect subsequent error rates and maintenance costs?

In this study we examine the relationship between software evolution as described by and lifecycle maintenance outcomes. Software evolution is formally defined as the "dynamic behavior of programming systems as they are maintained and enhanced over their life times" (Belady and Lehman, 1976). We describe software evolution with two main characteristics, i.e. lifecycle maintenance profiles and software volatility.

Maintenance profiles describe what type of lifecycle maintenance activities have occurred, and software volatility describes when changes occur, how large they are, and how consistently changes permeate the software system. Maintenance profiles are historical patterns of software maintenance activities. The type of change motivation, i.e. corrective, adaptive, enhancement and new program creation, categorizes these activities.

We describe software volatility as a multi-dimensional phenomenon with attributes of periodicity, amplitude and deviation. Periodicity tells us how often software changes. Amplitude tells use how much software changes. Deviation indicates the relative variance in length of change intervals for programs in the system.

Using the results of this study, researchers can observe the effect of software evolution on lifecycle maintenance outcomes. IS Managers can apply these results to improve their budgets for software maintenance resources. With an improved ability to predict software processing errors and maintenance costs, IS managers can include anticipated system error rates when looking at decisions concerning software system repair or replacement.

In the next section we combine the relevant literature on software maintenance outcomes and software evolution to present eight hypotheses as a basis for our proposed model predicting lifecycle maintenance outcomes. Dependent and explanatory variables are operationalized, and the model empirically tested using panel regressions. Separate panel regressions estimate model parameters for prediction of maintenance costs and software processing errors. Our results indicate that IS managers can use traditional software product attributes and descriptors of lifecycle maintenance profiles and software volatility to predict software maintenance outcomes.

SOFTWARE MAINTENANCE OUTCOMES

Post-implementation lifecycle maintenance of information systems accounts for as much as 80% of the lifetime costs of an information system (Bennet, 1996). For many organizations, lifecycle maintenance activities consume more IS resources than new development (Swanson and Dans, 2000). The resources expended in lifecycle maintenance can strain budgets and prevent organizations from having the time and money needed for new software development. When processing errors occur, managers frequently chase obscure, yet pressing problems with few diagnostics describing the cause of those problems (Swanson and Beath, 1990). If software managers could predict the frequency of production problems they will face, they could become more proactive, and their ability to plan and manage their work would be greatly enhanced.

We examine existing models of software maintenance outcomes. These outcomes include software processing errors and lifecycle maintenance costs. Each time maintenance activities modify software there is a change in system processing that can lead to errors. We also include costs, in dollars and in hours of effort. After examining current models of software maintenance outcomes, we build our model based on basic characteristics of currently implemented systems and elements of recent software evolutionary processes.

DRIVERS OF MAINTENANCE OUTCOMES

At any point in time an information system is the cumulation of an implemented information system and post-implementation software evolution. We start by examining factors that determine software product characteristics.

Basic System Characteristics

Brooks (1995) describes the basic characteristics of an information system as its complexity, size and age. Complexity has been shown to be a significant factor contributing to software maintenance outcomes. Increased complexity has been associated with increased software errors, increased software faults and increased effort for lifecycle software maintenance (Shen, et al., 1985; Banker, et al., 1991; Takahashi, 1997, Banker Davis and Slaughter, 1998; Banker and Slaughter, 2000; Banker, et al., 2000; Graves, et al., 2000). Increased software complexity is associated with programs that are more difficult to maintain and enhance. Extra effort is required to understand what the program source code is intended to accomplish and why it needs to be changed (Heales, 2000). Increased complexity makes it more difficult for programmers to change existing code or add functionality without disturbing the logical flow of processing in the original design. Thus, increased complexity will subsequently lead to an increase in processing errors and increase maintenance costs because the result will be even more complex programs that are difficult to maintain without causing errors. This leads to the following hypothesis:

> *H1: Increased system complexity will increase software processing errors and lifecycle maintenance costs.*

System size and age are inherent attributes of an information system. These are both system characteristics shown to relate to software processing errors and maintenance costs (Lehman, et al., 1997; Davis and Olson, 1985; Graves, et al., 2000; Banker, et al., 2000; Heales, 2000; Eick, et al., 2001). We will use measures of software size and average program age as control variables in this study.

Software Evolution

Software evolution is defined as the "dynamic behavior of programming systems as they are maintained and enhanced over their life times" (Belady and Lehman, 1976). Software evolution can be described by the accumulative effect of lifecycle maintenance activities on information systems after their implementation. Lifecycle modifications are small incremental changes that gradually transform a system. Rather than having the revolutionary impact of new system implementation, these changes are evolutionary, gradually transforming information systems to stay productive for the organization.

Software evolution has been studied from the general systems theory approach for several decades, e.g. see Lehman and Belady, 1985, etc. Empirical research has led to a series of laws describing behavior of information systems. Now we use characteristics of software evolution to help with the IS management problem of predicting lifecycle maintenance outcomes. We describe software evolution with two main characteristics, i.e. lifecycle maintenance profiles and software volatility. Lifecycle maintenance profiles describe what types of changes are made to the system and drive evolutionary processes. Software volatility measures the extent, timing and predictability of those changes.

Basic system characteristics and software evolution are used as explanatory variables in our conceptual model for software maintenance outcomes. Every information system is subjected to software evolution to one degree or another. We use software volatility to describe dynamic system behavior, and lifecycle maintenance profiles to describe maintenance processes driving software evolution.

Lifecycle Maintenance Profiles

Past research has used information system histories and prior maintenance activities to predict future levels of software maintenance costs and software faults or errors (Biyani and Santhanam, 1998; Gefen and Schneberger, 1996; Lientz and Swanson, 1980; Banker, et al., 2000; Banker and Slaughter, 2000). Similarly we seek a quantifiable descriptor of the type of lifecycle maintenance work previously done to help anticipate future outcomes.

We use software maintenance profiles as attributes describing the processes driving the transformations occurring as part of software evolution. These activities can be classified according to their motivation: corrective, adaptive and enhancements (Lientz and Swanson, 1980). We can further classify maintenance activities by functional subcategories, i.e. data handling, logic, computation, initialization, user interface and module interface (Barry, Kemerer and Slaughter, 1999).

A number of researchers have presented taxonomies of lifecycle maintenance activities to describe major types of maintenance work (Swanson and Beath, 1990; Lientz and Swanson, 1980; Pressman, 1992). IEEE standards have listed these as corrective, adaptive and enhancement. Some authors include categories for perfective and preventive maintenance (Pressman, 1992). The empirical work that has been done tends to report most of the maintenance effort as perfective (Lientz and Swanson, 1980, p. 68; Barry, Kemerer and Slaughter, 1999).

To develop a taxonomy of lifecycle maintenance activities we examine the primary motivations for software maintenance. Thus, we establish main activity categories corresponding to the original classifications of *corrective*, *adaptive* and *enhancement*. We add a fourth main category for *new program creation.*

Historical patterns of lifecycle maintenance activities are referred to as software maintenance profiles. Each system has its own history and unique software maintenance profile. We theorize that different types of maintenance activities will have different effects on future software processing errors and maintenance costs. Based on software reliability models, increased corrective activities should lead to reduced future levels of software processing errors and thus, lower maintenance costs (Lyu, 1996). We state the following hypothesis:

> *H2: Increased corrective maintenance profiles will decrease software processing errors and lifecycle maintenance costs.*

Adaptive maintenance activities change software programs to conform to changes in their surrounding environment. As opposed to enhancements, adaptive modifications add no new functionality to an information system. These software modifications are only intended to preserve the *status quo*. Continuing to operate a system after it no longer conforms to new technological circumstances can cost an organization (Truex, Baskerville and Klein, 1998). Even though an information system may continue to operate without processing errors, additional expense may be encountered as additional software maintenance or manual processing to compensate for a system that is not up-to-date. Thus, adaptive activities would eliminate the need for this additional software maintenance or manual processing to cover the gap between an old information system and the current business environment. Thus, we state the following hypothesis:

H3: Increased adaptive maintenance profiles will decrease software processing errors and lifecycle maintenance costs.

Enhancement activities change software programs by adding new functionality to an information system. New program creations expand the functionality of a system by adding new programs to a system. Changing and adding new source code to a system from enhancements or new programs is likely to introduce software faults and result in increased processing errors for later periods (Malaiya and Denton, 1999). Increases in processing errors will necessitate software modiications to correct those newly introduced software faults. We propose the following hypotheses:

H4: Increased enhancement profiles will increase software processing errors and lifecycle maintenance costs.

H5: Increased new program creation maintenance profiles will increase software processing errors and lifecycle maintenance costs.

Software Volatility

Software volatility describes dynamic behaviors by measuring software change. Software volatility is a characteristic of the dynamic behavior of the programming systems as they evolve, i.e. as they are maintained and enhanced throughout their productive life spans (Belady and Lehman, 1976). This lifecycle software change is in many respects inevitable. It is necessary for systems to keep pace with the changing environments surrounding them (Lehman and Belady, 1985; Pfleeger, 1998). These evolutionary processes are important. Information systems that fail these transformations can cause a drag on their organizations, resulting in a cost of stability (Truex, Baskerville and Klein, 1998). We include software volatility in our model to determine what its effect on maintenance outcomes.

Prior research has shown that software volatility affects software maintenance costs and errors (Butcher, 1997; Banker and Slaughter, 2000; Yau and Collofello, 1980, 1985; Banker, et al., 2000). Heales (2000) develops a software volatility index to measure effort spent on deep structural changes during software change processes. Malaiya and Denton (1999) use analytical methods to show that software change, i.e. volatility, results in increased levels of software errors. Other researchers have used empirical studies to show that prior software changes result in increased amounts of maintenance effort and software errors (Biyani and Santhanam, 1998; Lientz and Swanson, 1980; Eick, et al., 2001; Banker, et al., 2000).

Software change or volatility is a multi-dimensional phenomena and it should be described by a multidimensional measure to show how often software changes, how much it changes and how predictably changes occur. Due to the close ties between information systems and their environment we use multi-dimensional measures of software volatility based on a multi-dimensional measure of environmental volatility developed by Wholey and Brittain (1989). We describe software volatility with a multi-dimensional measure containing periodicity, amplitude and deviation.

Periodicity describes how often information systems change. Amplitude describes how much information systems change. Deviation describes how predictably the systems behave (Barry, Kemerer and Slaughter, 2001). We define measures that can be calculated at each time interval, e.g. week, month or quarter, during a system's post-implementation productive life span.

Thus, periodicity, amplitude and deviation can capture variations in lifecycle behavior of information systems.

Periodicity describes time intervals between software modifications. This is the system-wide mean of time intervals between software modifications for the time period being studied, e.g. week or month. We measure periodicity relative to system age to allow analysis across systems and throughout a system's lifecycle. Increased levels of software modification are believed to lead to an increase in processing errors (Graves, et al., 2000). Increases in processing errors will by necessity lead to increased maintenance costs in order to correct the cause of those errors. As we analyze software volatility we note that decreasing periodicity indicates that software modifications occur at more frequent intervals. We use periodicity at time *t-1* to predict processing errors and maintenance costs at time *t*. Remembering that shortened intervals between modifications indicates increased software volatility, we are led to the following hypothesis:

H6: Decreased periodicity will increase software processing errors and lifecycle maintenance costs.

Amplitude describes the magnitude of change by measuring the total size of system modifications each time period. We establish a relative system-level measure of amplitude as the change in application system size and normalizing by total system size. Several size metrics are available, including token counts of executable lines of code (LOC), function points, model objects and entities (Boehm, 1984; Albrecht and Gaffney, 1983; Grady, 1987; Symons, 1988). We use amplitude as a relative measure of how much software has changed. Increased amplitude in time period *t-1* is used to predict counts of processing errors and lifecycle maintenance costs for time period *t*. More modifications, i.e. greater amplitude, in time period *t*-

l leads to a greater likelihood of introduction of software errors, resulting in an increase in software processing errors and subsequent maintenance costs for time period *t*. This leads to the following hypothesis:

> *H7: Increased amplitude will increase software processing errors and lifecycle maintenance costs.*

Deviation describes the variance of the time intervals between software modifications. This measure indicates how consistent the change interval is for programs in an information system. A high deviation indicates that the system has a few programs modified at short intervals and some with very long intervals. A high deviation indicates that intervals between software modifications vary widely across programs in the system. The behavior of the system is harder to predict. Expertise needed for software changes will vary as well. This could lead to an increase in mistakes while source code is changed, and unnecessary effort expended when programmers try to support a larger subset of programs in the system. This could lead to increased software processing errors and increased maintenance costs. Therefore, we pose the following hypothesis:

> *H8: Increased deviation will increase software processing errors and lifecycle maintenance costs.*

Control Variables

Prior research indicates that as application software usage increases, so does the detection of processing errors (Biyani and Santhanam, 1998; Yuen 1985; Dekleva, 1992). System usage influences the number of software processing errors uncovered (Banker, et al., 2000). Software that is not executed will not have any errors detected, nor will it require software maintenance. Thus, we include application usage as a control variable.

Our eight hypotheses are summarized in Table 1. Directional relationships are diagrammed in Figure 1. Our predictive model for software maintenance outcomes will be empirically tested and results summarized in the next two sections.

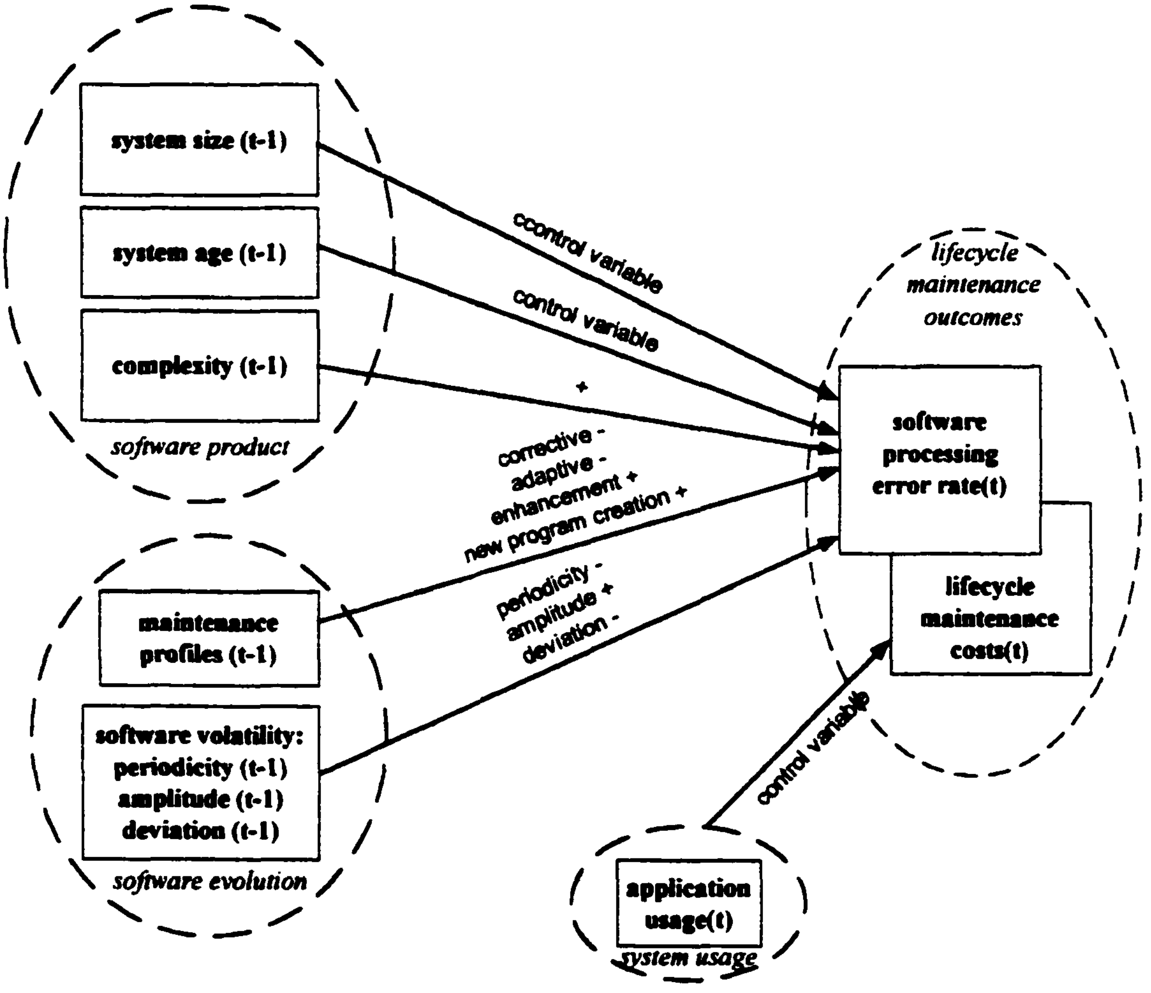


Figure 1: Model of Predictors of Maintenance Outcomes

METHODOLOGY

To test the hypotheses in Table 1 we estimate the following models:

$$\textit{Error-rate}_t = \beta_0 + \beta_1\,\textit{complexity}_{t-1} + \beta_2\,\textit{corrective profile}_{t-1} + \beta_3\,\textit{adaptive profile}_{t-1} + \beta_4\,\textit{enhancement profile}_{t-1} + \beta_5\,\textit{new program creation profile}_{t-1} + \beta_6\,\textit{periodicity}_{t-1} + \beta_7\,\textit{amplitude}_{t-1} + \beta_8\,\textit{deviation}_{t-1} + \beta_9\,\textit{system size} + \varepsilon$$

$$\textit{Costs}_t = \beta_0 + \beta_1\,\textit{complexity}_{t-1} + \beta_2\,\textit{corrective profile}_{t-1} + \beta_3\,\textit{adaptive profile}_{t-1} + \beta_4\,\textit{enhancement profile}_{t-1} + \beta_5\,\textit{new program creation profile}_{t-1} + \beta_6\,\textit{periodicity}_{t-1} + \beta_7\,\textit{amplitude}_{t-1} + \beta_8\,\textit{deviation}_{t-1} + \beta_9\,\textit{system size} + \beta_{10}\,\textit{application usage} + \varepsilon$$

	Hypothesis	**Test**
H1	Increased system complexity will increase software processing errors and lifecycle maintenance team costs.	H1: $\beta_1 > 0$
H2	Increased corrective maintenance profiles decrease software processing errors and lifecycle maintenance costs.	H2: $\beta_2 < 0$
H3	Increased adaptive maintenance profiles decrease software processing errors and lifecycle maintenance costs.	H3: $\beta_3 < 0$
H4	Increased enhancement profiles will increase software processing errors and lifecycle maintenance costs.	H4: $\beta_4 > 0$
H5	Increased new program creation profiles will increase software processing errors and lifecycle maintenance costs.	H5: $\beta_5 > 0$
H6	Decreased periodicity will increase software processing errors and lifecycle maintenance costs.	H6: $\beta_6 < 0$
H7	Increased amplitude will increase software processing errors and lifecycle maintenance costs.	H7: $\beta_7 > 0$
H8	Increased deviation will increase software processing errors and lifecycle maintenance costs.	H8: $\beta_8 > 0$

Table 1: Hypotheses to be tested

Research site:

The research site is a large national retailer with a software portfolio of 23 legacy systems of 3500+ programs. The retailer has a large, centralized Information Systems (IS) department that handles information processing for all of its various department stores. The Retailer's IS department has separate development and maintenance units. Software maintainers keep a detailed log of every modification made to each module by recording implementation date, purpose, type of change and programmer responsible.

Other available characteristics of each program include measures of system size, age and complexity. Application usage statistics include number and types of transactions processed (online vs. batch). These factors are combined with outcome measures for processing errors and maintenance costs.

Measures

Operational definitions for each of the model's dependent and independent variables are listed in Table 2. Our predictive model uses explanatory variables for time period *t-1* to predict outcomes in time period *t*.

Software Maintenance Outcomes

Maintenance outcomes are measured as software processing error rates and maintenance costs. Counts of software processing errors are token counts of abnormal terminations during transaction processing, i.e. abends. We measure software processing error rate as the number of abends per transaction processed (# abends / # transactions). Lifecycle maintenance costs are measured as hours of effort expended for all lifecycle maintenance activities each quarter for each system of the software portfolio.

Basic System Characteristics

We include software complexity as a key descriptor of a system. Because an information system is conceived and created as a tool for problem solution, the complexity of a system reflects the complexity of the task it addresses. Thus, descriptors of cognitive complexity describe the complexity of a problem, as well as the complexity of its solution.

Cognitive complexity breaks system complexity into three types: coordinative, component and dynamic complexity (Wood, 1986; Banker, Davis and Slaughter, 1998). Coordinative complexity examines the logic flow within each program of the system. Component complexity examines the data intensity of a system. Dynamic complexity corresponds to the overall complexity of the entire system by measuring the linkages between programs or elements in the system.

Coordinative complexity is operationalized as the system-wide total of cyclomatic measures of logic flow complexity (Gill and Kemerer, 1991). We operationalize component complexity as the system-wide number of unique operands in the system, i.e. Halstead's n2 summed over all programs in a system. Dynamic complexity is operationalized as the system-wide total number of program calls. Each of the complexity measures is normalized with respect to system size.

System size and age are control variables in our model. In addition, system size is used to normalize complexity measures.

Lifecycle Maintenance Profiles

Lifecycle maintenance profiles summarize historical patterns of the types of lifecycle maintenance activities that have occurred. We use four main activity categories relating to the motivation for each system modification: corrective, adaptive, enhancement and new program creation. Our empirical data provides a detailed log allowing classification of system modifications. The centralized systems development and maintenance staff maintained in-house standards requiring a record of who made the who made each software modification, when it was implemented, what was modified and why. These maintenance logs were maintained in a special section of the source code in each program throughout the software portfolio. System counts for each category and subcategory are aggregated by system.

We operationalize lifecycle maintenance profiles to indicate the main type of activity occurring in each system each time period. Once again, we use profiles from time period *t-1* to predict outcomes in time period *t*. To do this, we count activities for each main activity category, and calculate the proportion of lifecycle maintenance activities for each category that time period. We established four binary variables to indicate which main category of lifecycle maintenance activity was predominant for each system in that period.[1] For example, suppose system XXX had ten system modifications this month. Five (50%) were enhancements, three (30%) were new program creations, one (10%) was corrective and one (10%) was adaptive. We set our four profile variables as follows:

Profile - corrective	0
Profile - adaptive	0
Profile - enhancement	1
Profile - new program creation	0

As with the other explanatory variables in our model, we use *t-1* profile values to predict maintenance outcomes in time period *t*.

Software Volatility

Software evolution is described by two attributes: software volatility and lifecycle maintenance profiles. We describe software volatility with normalized measures of periodicity, amplitude and deviation. Periodicity is operationalized as the mean time interval between system modifications. Amplitude is the total change in system size normalized with respect to total system size. Deviation is the variance in the lengths of time intervals between system modifications. Periodicity and deviation are normalized with respect to system age.

Periodicity, amplitude and deviation are aggregate measures calculated for each time period in our empirical data. Empirical data to predict maintenance costs are aggregated quarterly. Tests of models predicting software processing errors are aggregated monthly. We use values of periodicity, amplitude and deviation for time period *t-1* to predict maintenance costs in time period *t*.

[1] Ties are handled by assigning the value to the first non-zero proportion.

	Construct	Operational variable	Description	Unit of analysis varies ....
H1	System complexity	Normalized Component complexity	System-wide count of Halstead's n2 (unique operands) normalized by system size	By system by quarter
		Normalized coordinative complexity	System-wide count of McCabe's cyclomatics normalized by system size	
		Normalized dynamic complexity	System-wide count of program calls normalized by system size	
H2 H3 H4 H5	Software maintenance profiles	Profile of main activity for software lifecycle maintenance	Binary variable to indicate this is most prevalent type of lifecycle maintenance activity this quarter - there are four variables for each of four main categories of maintenance activities (corrective, adaptive, enhancement, new program creation)	By system by quarter
H6	Software volatility	Periodicity	Mean time interval between software modifications normalized with respect to system age	By system by quarter
H7	Software volatility	Amplitude	Change in system size normalized by system size	By system by quarter
H8	Software volatility	Deviation	Variance in length of time intervals between software modifications normalized with respect to system age	By system by quarter

Table 2: Operational Explanatory Variables

Data

The retailer's software portfolio includes 23 applications, 21 with batch processing and 18 with online processing. A detailed log recorded all lifecycle maintenance activities in each of the 3500+ programs. Data include what modifications were made, who made them and when each software modification was implemented. Quarterly data is available for maintenance costs, maintenance effort, vendor costs, transactions processed online and batch, and processing errors online and batch. Using these data a panel data set was built for the 23 applications covering 10 quarters. Two systems were not in production for the full 10 quarters. Missing values caused some observations to have irreconcilable values. These records were dropped prior to regression estimates. This leaves an unbalanced panel with 192 observations.

Total monthly processing errors are available for each system for 31 months. Processing errors, i.e. abends, are categorized separately for online and batch processes. Likewise, we tested our model with separate regression estimates of online and batch abends using an unbalanced panel data set with 688 observations.

RESULTS

Descriptive Statistics

Table 3 lists summary statistics for the monthly panel data set. Frequency counts for the binary profile variables are listed in Table 4. Summary statistics for the relevant variables in the quarterly panel data set are listed in Table 5. Table 6 lists frequency counts for the binary profile variables in the quarterly panel data set. Correlations are listed in Tables 7 and 8.

Variable	*N*	*Mean*	*Std. Dev.*	*Minimum*	*Maximum*
# online transactions (t)	665	603136.8	972288.4	0	9167362
# batch transactions (t)	665	764.7235	1073.836	0	7184.2
Total function points (t-1)	665	2364.647	1603.668	273	5482
Average program age (in months)	665	71.61239	48.63827	18.55263	230.5
Total Cyclomatics / total LOC (t-1)	665	0.0516164	0.0110005	0.0353499	0.0810833
Total n2 / total LOC (t-1)	665	0.2013791	0.553793	0.1239131	0.3558856
Total calls / total LOC (t-1)	665	0.0082747	0.0048862	0.0008251	0.0156295
Software volatility - periodicity	665	0.265109	0.3805316	0	1
Software volatility - amplitude	665	0.0080517	0.042706	0	0.8145953
Software volatility - deviation	665	0.0076945	0.0194198	0	0.1994759

Table 3: Summary statistics of monthly panel data set

Profile (t-1)	N	Freq = 0	Freq = 1	% = 0	% = 1
Profile - corrective	665	639	26	96.09	3.91
Profile - adaptive	665	655	10	98.50	1.50
Profile - enhancement	665	231	434	34.74	65.26
Profile - new programs	665	632	33	95.04	4.96

Profile enhancement subcategory (t-1)	N	Freq = 0	Freq = 1	% = 0	% = 1
Data handling	665	519	146	78.05	21.95
Logic	665	415	250	62.41	37.59
Computation	665	662	3	99.55	0.45
Initialization	665	662	3	99.55	0.45
User interface	665	643	22	96.69	3.31
Module interface	665	665	0	100.00	0.00

Table 4: Frequency counts of Binary variables - monthly panel

Variable	*N*	*Mean*	*Std. Dev.*	*Minimum*	*Maximum*
Ln(lifecycle maintenance hours) (t)	180	4.756355	4.83937	-18.42068	7.907651
Ln(# batch transactions) (t)	194	4.528112	7.755914	-18.42068	9.807131
Ln(# online transactions) (t)	194	7.865331	12.35231	-18.42068	16.88541
Total function points (t-1)	194	2385.247	1657.912	273	5482
Average program age (in quarters)	199	32.95181	18.29137	7.3333	72.5
Total Cyclomatics / total LOC (t-1)	194	0.0529539	0.010193	0.0386465	0.0810837
Total n2 / total LOC (t-1)	194	0.2042366	0.0542459	0.1388885	0.3558856
Total calls / total LOC (t-1)	194	0.0076707	0.0049431	0.0008251	0.0154845
Software volatility - periodicity	194	0.2048901	0.3086835	0.007	1
Software volatility - amplitude	194	0.0311604	0.0973654	0	0.8268304
Software volatility - deviation	194	0.0172412	0.0360195	0	0.3100103

Table 5: Summary statistics of quarterly panel data set

Profile (t)	N	Freq = 0	Freq = 1	% = 0	% = 1
Profile - corrective	194	187	7	96.39	3.61
Profile - adaptive	194	192	2	98.97	1.03
Profile - enhancement	194	36	158	18.56	81.44
Profile - new programs	194	187	7	96.39	3.61

Profile enhancement subcategory (t-1)	N	Freq = 0	Freq = 1	% = 0	% = 1
Data handling	194	147	47	75.77	24.35
Logic	194	124	70	63.92	36.08
Computation	194	187	7	96.39	3.61
Initialization	194	190	4	97.94	2.06
User interface	194	188	6	96.91	3.09
Module interface	194	194	0	100.00	0.00

Table 6: Frequency counts of Binary Variables - quarterly panel

Monthly panel data set	total function points (t-1)	Average program age (t-1)	Total cyclomatics / total LOC (t-1)	Total n2 / total LOC (t-1)	Total calls / total LOC (t-1)	Profile - corrective (t-1)	Profile - adaptive (t-1)	Profile - enhancement (t-1)	Profile - new programs (t-1)	Software volatility - periodicity (t-1)	Software volatility - amplitude (t-1)
Average program age (t-1)	-0.2104	1									
Total cyclomatics / total LOC (t-1)	-0.1161	0.1917	1								
Total n2 / total LOC (t-1)	-0.2555	0.0311	0.6081	1							
Total calls / total LOC (t-1)	0.1110	-0.2397	0.5571	-0.7124	1						
Profile - corrective (t-1)	-0.0324	-0.0049	0.0152	-0.0324	0.0330	1					
Profile - adaptive (t-1)	0.0243	-0.0566	0.0467	0.0847	-0.0840	-0.0257	1				
Profile - enhancement (t-1)	0.0743	0.1654	-0.1454	-0.2843	-0.3519	-0.3519	-0.2398	1			
Profile - new programs (t-1)	0.1441	-0.1081	0.0573	-0.0632	-0.0427	-0.0427	-0.0291	-0.3975	1		
Software volatility - periodicity (t-1)	-0.2521	-0.1728	0.1207	0.4348	-0.0354	-0.0354	0.0071	-0.5351	-0.1078	1	
Software volatility - amplitude (t-1)	0.1128	-0.0682	-0.0406	-0.0679	-0.0600	-0.0600	-0.0170	-0.1194	0.2579	-0.1259	1
Software volatility - deviation (t-1)	0.0810	-0.1314	-0.0073	-0.1492	-0.0368	-0.0368	0.1154	0.0823	-0.0128	0.0363	0.0264

Table 7: Correlations for monthly panel data

Quarterly panel data set	# batch transactions (t)	# online transactions (t)	total function points (t-1)	Average program age (t-1)	Total cyclomatics / total LOC (t-1)	Total n2 / total LOC (t-1)	Total calls / total LOC (t-1)	Profile - corrective (t-1)	Profile - adaptive (t-1)	Profile - enhancement (t-1)	Profile - new programs (t-1)	Software volatility - periodicity (t-1)	Software volatility - amplitude (t-1)
# online transac-tions (t)	0.1770	1											
Total function points (t-1)	0.2752	0.1508	1										
Average program age (t-1)	0.5579	0.0702	0.2169	1									
Total cyclomatics / total LOC (t-1)	-0.0759	0.0228	-0.2961	-0.0868	1								
Total n2 / total LOC (t-1)	0.0044	0.1218	-0.2306	0.0745	0.5468	1							
Total calls / total LOC (t-1)	0.0717	0.0557	0.2662	0.2382	-0.5803	-0.6168	1						
Profile - corrective (t-1)	-0.1194	-0.1248	-0.0343	-0.1766	0.0622	0.0702	-0.1728	1					
Profile - adaptive (t-1)	-0.0291	-0.0399	-0.0814	-0.0811	-0.0655	-0.0746	0.0585	-0.0153	1				
Profile - enhancement (t-1)	0.1689	0.1888	0.0839	0.2448	0.2773	-0.0975	0.1706	-0.5685	-0.2112	1			
Profile - new programs (t-1)	-0.0820	-0.0319	0.1001	-0.0691	0.1018	-0.0771	0.0521	-0.0309	-0.0115	-0.4260	1		
Software volatility - periodicity (t-1)	-0.2127	-0.1910	-0.2171	-0.2992	0.3553	0.1806	-0.2665	-0.3945	0.0585	-0.6633	0.1020	1	
Software volatility - amplitude (t-1)	-0.0637	-0.0465	0.0077	-0.0841	0.0299	-0.0349	-0.0795	0.2936	-0.0229	-0.2133	0.1597	0.2465	1
Software volatility - deviation (t-1)	-0.2164	-0.0798	-0.0694	-0.3055	0.1507	-0.0151	-0.0489	-0.0237	0.0840	0.0766	-0.0621	0.1065	0.0148

Table 8: Correlations for quarterly panel data

Parameter Estimates

Model parameters were estimated with generalized least square regressions for panel data sets. Two separate regressions were estimated one using monthly data for predicting software processing error rates, and one using quarterly data for predicting lifecycle maintenance costs. As is common with pooled time-series data sets, autocorrelation was indicated by diagnostic tests, i.e. the Breusch-Godfrey test for serial correlation (Johnston, 1984). To correct for this we used panel-specific AR1 methods for the correction of serial correlation. This provides separate AR1 correction for each group in the panel, i.e. each system in the portfolio. The same correction for serial correlation was used in both regression estimates.

Table 9 reports parameter estimates for software processing errors. There is no commonly established functional form to describe the relationship between maintenance outcomes and characteristics of software evolution and basic characteristics. A linear transformation produced a better fit than either a semi-log or log-linear transformation. We elaborate on our results in the discussion section.

Predict: processing error rate
N = 522
Log likelihood = 2196.769
Wald = 113.13
	Estimated coefficient	p-value	*** ≈ $p \leq 0.001$
			** ≈ $p \leq 0.05$
			* ≈ $p \leq 0.10$
				Hypothesis tested / supported?

Constant	0.0126385	0.000	***	
Total function points (t-1)	-0.0000005	0.000	***	
Average program age (t-1)	-0.0000161	0.002	***	
Total cyclomatics (t-1) normalized by LOC	-0.0761168	0.080	*	H1 no
Total n2 (t-1) normalized by LOC	0.0042215	0.543		H1
Total calls (t-1) normalized by LOC	-0.3143885	0.000	***	H1 no
Profile - corrective (t-1)	-0.0030525	0.003	***	H2 yes
Profile - adaptive (t-1)	-0.0036036	0.005	***	H3 yes
Profile - enhancement (t-1)	-0.0029690	0.000	***	H4 no
Profile - new program creation (t-1)	-0.0045835	0.000	***	H5 no
Software volatility - periodicity (t-1)	-0.0047772	0.000	***	H6 yes
Software volatility - amplitude (t-1)	0.0452957	0.000	***	H7 yes
Software volatility - deviation (t-1)	0.0030350	0.705		H8

Table 9: Prediction of software processing error rate using monthly panel data

Predict: ln(lifecycle maintenance hours)
N = 199
Log likelihood = -573.9624
Wald = 85.91
	Estimated coefficient	p-value	*** ≈ $p \leq 0.001$
			** ≈ $p \leq 0.05$
			* ≈ $p \leq 0.10$
				Hypothesis tested / supported?

Constant	25.8279400	0.000	***	
Usage - transactions, batch (t)	0.0006823	0.153		
Usage - transactions, online (t)	-0.0000005	0.190		
Total function points (t-1)	0.0000143	0.968		
Average program age (t-1)	0.0533362	0.421		
Total cyclomatics (t-1) normalized by LOC	170.6134000	0.027	**	H1 yes
Total n2 (t-1) normalized by LOC	-134.3394000	0.000	***	H1 no
Total calls (t-1) normalized by LOC	-1016.9020000	0.000	***	H1 no
Profile - corrective (t-1)	2.1484380	0.281		H2
Profile - adaptive (t-1)	-0.5163811	0.883		H3
Profile - enhancement (t-1)	-1.1262410	0.496		H4
Profile - new program creation (t-1)	0.8682899	0.622		H5
Software volatility - periodicity (t-1)	-5.0545150	0.014	**	H6 yes
Software volatility - amplitude (t-1)	6.5823710	0.048	**	H7 yes
Software volatility - deviation (t-1)	0.9321104	0.921		H8

Table 10: Prediction of maintenance costs using quarterly panel data

The results of the regression estimates for maintenance costs are listed in Table 10. A test for type of distribution function for maintenance costs showed they were not normally distributed.[2] A semi-log transformation was used for the cost panel regressions after confirmation with the Box-Cox test for data transformation (Greene, 1997; Neter, Waserman and Kutner, 1990). Our results support hypotheses H6 and H7, with mixed results for H1. Inferences drawn from these results will be expanded in the discussion section.

DISCUSSION

The objective of this research is to determine if IS managers can use attributes of an information system and its evolutionary record to forecast software maintenance outcomes, i.e. maintenance costs and processing errors. By recognizing the progressive function of lifecycle maintenance we focus our research on the attributes of an information system to see what we can learn from aspects of the system's recent software evolution. We combine our understanding of quantifiable characteristics of software evolution with basic system characteristics to build a predictive model for lifecycle maintenance outcomes. This work examines the effect of these explanatory variables with separate model regression estimates for software processing errors and maintenance costs.

Explanatory variables chosen for our investigation represent information available to IS managers charged with responsibility for system lifecycle support. An improved ability to predict processing errors and maintenance costs can assist managers with resource planning, staff assignments and cost controls. Enhanced predictions of costs and errors can increase an IS manager's ability to make repair / replace decisions for information systems.

First, we examine the effect basic system characteristics have on software maintenance outcomes. Tests of hypothesis H1 indicate we can use some types of system complexity to

[2] Maintenance costs are measured as hours of effort spent on maintenance.

predict maintenance outcomes. We examine the effects of coordinative, component and dynamic complexity as explanatory variables in our predictive model (Table 11).

	Explanatory variable	Errors	Costs
↑	Coordinative complexity = normalized cyclomatics	↓	↑
↑	Component complexity = normalized n2	Ns	↓
↑	Dynamic complexity = normalized calls	↓	↓

Table 11: Effect of System Complexity on maintenance outcomes[3]

Component and dynamic complexities decrease costs and processing errors, while coordinative complexity increases costs and processing errors. Systems with higher levels of component complexity, i.e. normalized n2 = total n2/total LOC, are data intensive and, thus, more stable than systems with lower levels of data intensity (Martin, 1989; Hoffer, George and Valacich, 1996). These systems would have lower errors rates, require fewer modifications and fixes, and be less expensive to maintain. Increased levels of dynamic complexity, i.e. normalized calls = total calls/total LOC, indicate more structured system design with fewer software faults and easier maintenance. Our results indicate that processing errors will decrease and maintenance costs increase when there is an increase in coordinative complexity, i.e. when normalized cyclomatics increases where normalized cyclomatics = total cyclomatics/total LOC. Coordinative complexity places a higher burden on maintenance programmers. Modifications take longer to implement and, therefore, result in higher costs. At the same time, it may be that the increased time and care devoted to those modifications are completed with fewer errors. Thus, we have mixed support for hypothesis H1.

Next we examine the effect software evolution can have on software maintenance outcomes. Our model uses two main characteristics of software evolution: lifecycle maintenance profiles and software volatility. We use lifecycle maintenance profiles to describe the type of activities occurring in software evolutionary processes. We analyze our results for hypotheses H2 though H5 in Table 12.

	Explanatory variable	Errors	Costs
↑	Profile - corrective (t-1)	↓	Ns
↑	Profile - adaptive (t-1)	↓	Ns
↑	Profile - enhancement (t-1)	↓	Ns
↑	Profile - new program creation (t-1)	↓	Ns

Table 12: Effect of lifecycle maintenance profiles on maintenance outcomes

As expected, increases in corrective and adaptive maintenance activities will decrease future processing errors. Hypothesis H2 and H3 for software processing errors are supported. Hypothesis H4 predicts that increasing enhancements will increase both processing errors and maintenance costs. Our results indicate that an increase in enhancements at time *t-1* will result in a reduction of processing errors at time *t*. Hypothesis H5 predicts that an increase in new program creations will increase both processing errors and maintenance costs. Our results indicate that an increase in new program creations at time *t-1* will decrease processing errors at time *t*. Our parameter estimates support hypotheses H2 and H3, and contradict hypotheses H4 and H5.

Software volatility is described with three dimensions, periodicity, amplitude and deviation. Tests of hypotheses H6 through H8 will indicate whether we can use dimensions of system volatility to predict lifecycle maintenance outcomes. As shown in Table 13, our results provide support for hypotheses H6 and H7..

	Explanatory variable	Errors	Costs
↑	Periodicity	↓	↓
↑	Amplitude	↑	↑
↑	Deviation	ns	Ns

Table 13: Effect of software volatility on maintenance outcomes

[3] Empty cells in tables 11-13 indicate estimated parameters were insignificant, i.e. p-value > 10%.

As expected, decreased periodicity will increase error rates and maintenance costs. Decreasing periodicity indicates the software is more volatile, i.e. more software modifications are occurring. More software modifications result in a higher likelihood of errors. Increases in errors will require software maintenance for program fixes with an increase in maintenance costs.

Hypothesis H7 is supported by empirical results for both processing errors and maintenance costs. Increasing amplitude, i.e. relative size of software change, will increase the rate of occurrence of errors.

Four of eight hypotheses were supported, one had mixed support and two hypotheses were contradicted. Table 14 summarizes test results for all eight hypotheses.

Hypothesis	Support?	
H1	Mixed	Increased system complexity will increase software processing errors and lifecycle maintenance costs.
H2	Yes	Increased corrective maintenance profiles decrease software processing errors and lifecycle maintenance costs.
H3	Yes	Increased adaptive maintenance profiles decrease software processing errors and lifecycle maintenance costs.
H4	No	Increased enhancement profiles will increase software processing errors and lifecycle maintenance costs.
H5	No	Increased new program creation profiles will increase software processing errors and lifecycle maintenance costs.
H6	Yes	Decreased periodicity will increase software processing errors and lifecycle maintenance costs.
H7	Yes	Increased amplitude will increase software processing errors and lifecycle maintenance costs.
H8		Decreased deviation will increase software processing errors and lifecycle maintenance costs.

Table 14: Summary of tests of hypotheses

CONCLUDING REMARKS

Change is inevitable. We recognize that to keep pace with changing requirements information systems must also change and evolve. Slow incremental software transformations can be described by software evolution. Some authors equate software evolution to software lifecycle maintenance.

We describe software evolution with software volatility and lifecycle maintenance profiles. Lifecycle maintenance profiles describe what type of work is being done. Software volatility measures how often modifications are made (periodicity), how much is modified (amplitude), and how consistently programs are modified in a system (deviation). We use these characteristics with the basic system characteristics of an information system to predict lifecycle maintenance outcomes, i.e. maintenance costs and software processing errors.

Implications for research

Post-implementation software maintenance activities account for most of the total lifetime costs of software systems as they continue to evolve. It is important to understand the drivers of these maintenance outcomes and improve a manager's ability to control these costs. If software evolution, i.e. software volatility and lifecycle maintenance profiles, affect maintenance outcomes, IS managers need to know.

The objective of this research is to determine if software maintenance outcomes, i.e. errors and costs, are driven by information system characteristics and descriptors of software evolution. We developed a conceptual model based on eight hypotheses describing the relationships among software maintenance outcomes and information system characteristics, lifecycle maintenance profiles and software volatility.

Our analysis demonstrates that we can use our knowledge of system characteristics and recent software evolution, to predict maintenance costs and processing errors

Implications for practice

In discussions of software volatility we often assume that software volatility, i.e. software change, is bad, should be prevented and that it leads to increased costs. We must be careful not to jump to conclusions. Leaving information systems unchanged can actually cause problems if they no longer satisfy the information requirements of their organizations. Another challenge for IS managers is that information systems generally have a longer tenure than the programming team assigned to maintain them. Traditional models explaining software maintenance costs and forecasting processing errors use explanatory variables determined during system development and implementation. IS managers must deal with the current information system. Choices about buying off-the-shelf software, CASE tools and support team staffing may not even be available.

This work provides an IS manager with information about the relationships among the current information system, its recent change history and future lifecycle maintenance outcomes, i.e. maintenance costs and software processing errors. An understanding of the overall evolutionary processes and their relationship with future maintenance costs and processing errors can assist managers in forecasting software maintenance budgets and workload. Improvements in the prediction of these outcomes will improve managers' abilities to make the repair / replace decision when considering replacement of aging information systems with newer technologies.

Contributions

This work extends our knowledge of the relationship between prior lifecycle maintenance activities and future maintenance outcomes. Researchers can use this as a motivation for further work in describing and analyzing the type, sequence, quantity and timing of maintenance activities.

REFERENCES

Albrecht, A.J. and Gaffney, J.E. Jr., “Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation”, IEEE Transactions on Software Engineering, Vol. 9, No. 6, Nov. 1983, pp. 639-648.

Banker, R.D., Datar, S.M., Kemerer, C.F., and Zweig, D., "Software Complexity and Maintenance Costs", Software Project Management: Readings and Cases, C.F. Kemerer, (Ed.) Irwin Book Team, Chicago IL, 1997, pp. 521-538. .(reprinted from 1991)

Banker, R.D., Datar, S.M., Kemerer, C.F., and Zweig, D., "Software Errors and Software Maintenance Management", Information Technology and Management, forthcoming, 2000.

Banker, R., Davis, G.B., and Slaughter, S.A., "Software Development Practices, Software Complexity, and Software Maintenance Performance: A field study", Management Science, Vol. 44, No. 4, Apr. 1998.

Banker, R.D., and Slaughter, S.A., "The Moderating Effects of Structure on Volatility an Complexity in Software Enhancement", Information Systems Research, September 2000, Vol. 11, No. 3, pp. 219-240.

Barry, E.J., Kemerer, C.F., and Slaughter, S.A., "An Empirical Analysis of Software Evolution Profiles and Outcomes", Proceedings of the International Conference on Information Systems, Charlotte, NC, December 1999.

Barry, E.J., Kemerer, C.F., and Slaughter, S.A., "A Multidmensional Measurement of Software Volatility", CMU - GSIA, working paper, 2001.

Belady, L.A., and Lehman, M.M., A Model of Large Program Development", IBM Systems Journal, No, 3, 1976, pp. 225-252.

Bennet, K., "Software evolution: past, present and future", Information and Software Technology, Vol. 39, No. 11, Nov. 1996, pp. 673-680.

Biyani, S., and Santhanam, P., Exploring defect data from develpment and customer usage on software modules over multiple releases, Yorktown Heights, NY, IBM T. J. Watson Research Center, 1998.

Boehm, B.W., "Software Engineering Economics", Software Project Management: Readings and Cases, C.F. Kemerer, (Ed.) Irwin Book Team, Chicago IL, 1997, pp. 55-85

Brooks, F.J., The Mythical Man-Month, Addison-Wesley Publishing Co., 1995.

Butcher, G., Addressing Software Volatility in the System Life Cycle, PhD Dissertation, Colorado Technical University, 1997, UMI#9815557.

Davis, G.B, and Olson, M.H., Management Information Systems: Conceptual Foundations, Structure, and Development, 2^{nd} Edition, McGraw-Hill Book Company, 1985.

Dekelva, S.M. "The Influence of the Information Systems Development Approach on Maintenance", MIS Quarterly, September 1992, pp. 355-372.

Eick, S.G., Graves, T.L., Karr, A.K., Marron, J.S., and Mockus, A., "Does Code Decay? Assessing the Evidence from Change Management Data ", IEEE Transactions on Software Engineering, January 2001, Vol. 27, No. 1, pp. 1-12.

Grady, R.B., "Measuring and Management Software Maintenance", IEEE Software, Vol. 4, Sept. 1987, pp. 35-45.

Gefen, D., and Schneberger, S.L., "The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications", Proceedings of the IEEE Conference on Software Maintenance, 1996, Monterey, CA.

Gill, G.K., and Kemerer, C.F., "Cyclomatic Complexity Density and Software Maintenance Productivity", IEEE Transactions on Software Engineering, Vol. 17, No. 12, December, 1991, pp.1284-1288.

Graves, T.L., Karr, A.K., Marron, J.S., and Siy, H., "Predicting Fault Incidence Using Software Change History", IEEE Transactions on Software Engineering, July 2000, Vol. 26, No. 7, pp. 653-661.

Greene, Wm H., Econometric Analysis, third edition, Prentice Hall, Upper Saddle River, NJ, 1997.

Heales, J., "Factors Affecting Information Systems Volatility", ICIS 2000 Proceedings, Brisbane , Australia, December 10-13, 2000,*forthcoming*.

Hoffer, J.A., J.F. George, and J.S.Valacich, 1996, Modern Systems Analysis and Design, The Benjamin/Cummings Publishing Company, Inc., Reading, MA.

Johnston, J., Econometric Methods, Third Edition, McGraw-Hill, Inc., New York, 1984.

Kalakota, R., and Whinston A.B., Electronic Commerce: A Manager's Guide, Addison-Wesley, Reading, MA, 1996.

Kemerer, C.F. and Slaughter, S.A., "Methodologies for Performing Empirical Studies: Report from the International Workshop on Empirical Studies of Software Maintenance", Empirical Software Engineering, Vol. 2, No. 2, 1997(1), pp. 109-118.

Kemerer, C.F. and Slaughter, S.A., "Determinants of Software Maintenance Profiles: An Empirical Investigation", Journal of Software Maintenance, Vol. 9, 1997(2), pp. 235-251.

Kemerer, C.F. and Slaughter, S.A., "A Longitudinal Analysis of Software Maintenance Patterns", ICIS 1997 Proceedings.

Lehman, M.M., and Belady, L.A., Program Evolution: Processes of Software Change, Academic Press, London, 1985.

Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E. and Turski, W.M., "Metrics and Laws of Software Evolution - The Nineties View", Metrics '97, the Fourth International Software Metrics Symposium, 1997, Albequerque, NM.

Lientz, B.P., and Swanson, E.B., Software Maintenance Management, Addison-Wesley, Reading, MA, 1980.

Lyu, M.R., Handbook of Software Reliability Engineering, IEEE computer Society Press, Los Alamitos, CA, 1996.

Malaiya, Y.K., and Denton, J., "Requirements Volatility and Defect Density", Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No. PR0044), p. xii+304, 285-94. ISBN 0760594434, IEEE Computer Society, Los Alamitos, CA

Martin, J., Information Engineering: Book I Introduction, Prentice Hall, Englewood Cliffs, NJ, 1989.

Neter, J., Wasserman, Wm, and Kutner, M.H., Applied Liner Statistical Models Regression, Analysis of Variance, and Experimental Design, 3rd Edition, Richard D. Irwin, Inc., Burr Ridge, IL, 1990.

Pfleeger, S., "The Nature of System Change", IEEE Software, Vol. 15, No. 3, May-June 1998, pp. 87-90.

Pressman, R.S., Software Engineering: A Practioner's Approach, 3rd Edition, McGraw-Hill,New York, NY, 1992.

Shen, V.Y., Yu, T.J., Thebaut, S., and Paulsen, L.R., "Identifying Error-Prone Software - An Empirical Study", IEEE Transactions on Software Engineering, Vol. SE-11, No. 4, April, 1985, pp. 317-323.

Swanson, E.B. and Beath, C.M., "Departmentalization in Software Development and Maintenance", Software Project Management: Readings and Cases, C.F. Kemerer, (Ed.) Irwin Book Team, Chicago IL, 1997, pp. 539-553. (Reprinted from 1990)

Swanson, E. B., and Dans, E., "System Life Expectancy and the Maintenance Effort: Exploring their Equilibrium", MIS Quarterly, June, 2000, Vol. 24, No. 2, pp. 277-297.

Symons, C.R., "Function Point Analysis: Difficulties and Improvements", IEEE Transactions on Software Engineering, Vol. 14, No. 1, Jan. 1988, pp. 2-11.

Takahashi, R., "Software Quality Classification Model Based on McCabe's Complexity Measure", The Journal of Systems and Software, Vol. 38, No. 1, July 1997, pp. 61-69.

Truex, D.P., Baskerville, R. and Klein, H., "Growing Systems in Emergent Organizations", Communications of the ACM, August 1998, Vol. 42, No. 8, pp. 117-123.

Wholey, D.R., and Brittain, J., "Characterizing Environmental Variation", Academy of Management Journal, Vol. 32, No. 4, 1989, pp. 867-882.

Wood, R.E., Task Complexity: Definition of the Construct", Organizational Behavior and Human Decision Processes, 1986, Vol. 37, pp. 60-82.

Yau, S.S., and Collofello, J., "Some Stability Measures for Software Maintenance", IEEE Transactions on Software Engineering, Vol. 6, No. 11, Nov. 1980, pp. 545+.

Yau, S.S., and Collofello, J., "Design Stability Measures for Software Maintenance", IEEE Transactions on Software Engineering, Vol. 11, No. 9, Sep. 1985, pp. 849-856.

Yuen, C.H., "An Empirical Approach to the Study of Errors in Large Software Under Maintenance", 2nd IEEE Conference on Software Maintenance, 1985, Washington, D.C.

CHAPTER 6: CONCLUSION

CONTRIBUTIONS, IMPLICATIONS AND FUTURE WORK

Contributions and Implications

This thesis makes several contributions to our common understanding of software evolution. This is the first study to measure and analyze differences in evolutionary transformations of systems. Empirical studies of software evolution face particular challenges due to the longitudinal nature of the evolution phenomenon. The results for this study of software evolution are strengthened by use of a unique empirical data set ten-times larger than previous longitudinal studies in this area. Prior research on software evolution has concentrated on development and confirmation of laws of software evolution. This research goes beyond the laws describing universal behavior of software systems to build models for analyzing differences in system behavior, i.e. software evolution.

This work provides a fresh approach for studying the evolutionary process of software change. By defining and validating a multi-dimensional measure of software volatility we can expand available methodologies. Studies of volatility from other disciplines are compared and contrasted with software change processes. We define software volatility as a multi-dimensional concept. Software volatility is described by periodicity (change interval length), amplitude (change size), and deviation (change interval predictability). Evaluation criteria are developed and rigorously applied to the newly defined measures of periodicity, amplitude and deviation. Validity of these new measures is tested empirically using a 20-year history of software modifications for lifecycle maintenance in 23 information systems. The measures are found to have both convergent and discriminant validity. Predictive validity is demonstrated with a model for software complexity. We are able to empirically show that periodicity, amplitude and deviation are predictors of software complexity. Our empirical tests show that these dimensions of software volatility are better predictors than traditional software product metrics. Multi-dimensional measures of software volatility, i.e. amplitude, periodicity and deviation, are relatively easy to calculate and can be aggregated to vary by system and time period. These measures lend themselves to longitudinal studies necessary for understanding the evolutionary processes taking place in software lifecycle maintenance.

The second research question addressed in this project is to identify antecedents of software volatility. Recognizing the close tie between information systems and their working environments, we examine attributes of the competitive environment, the task environment, and the basic information system's inner environment to build a conceptual model of antecedents for software volatility as measured by periodicity. We build a conceptual model based on seven hypotheses. These hypotheses are empirically tested using longitudinal data from a 20-year log of lifecycle maintenance activities for 23 information systems. We find that elements from each facet of an information system's environment contribute significantly in determining levels of software volatility.

The third objective addressed in this research is to examine the relationship between characteristics of software evolution and lifecycle maintenance outcomes as measured by software processing error rates and lifecycle maintenance costs. Lifecycle maintenance is motivated by a desire to extend the life of an existing information system. To maintain this same progressive focus, we acknowledge that all prior development and lifecycle maintenance work has created the currently implemented system. We ask if attributes of an information system and its recent software evolution are determinants of lifecycle maintenance outcomes. We build a conceptual model based on eight hypotheses describing the relationships between maintenance outcomes, i.e. processing errors and maintenance costs, and descriptors of recent software evolution and basic information system characteristics. This theory is supported with empirical data for a two-and-one-half-year history of lifecycle maintenance outcomes and lifecycle maintenance activities for a portfolio of 23 information systems. By using three dimensions of software volatility, and historical patterns of the types of lifecycle maintenance activities recently executed, IS managers can gain significant insight into future levels of processing errors and maintenance costs.

This research also makes practical contributions to the work faced by practitioners. Combining the results obtained from each of the three research questions, we provide IS managers with software volatility measures that can be calculated with a basic spreadsheet application. By tracking the motivation of lifecycle maintenance activities and the timing and size of software modifications implemented, IS managers can gain insight into system behavior. These insights will help in anticipating resource requirements for lifecycle maintenance support. In addition, the use of current system characteristics and recent software evolutionary processes to predict lifecycle maintenance outcomes, can expand the tools available for assisting with managerial decisions to repair or replace an information system. Contributions of this research project are summarized in Table 1.

This research presented a number of challenges that allow it to make substantial contributions to the understanding of software volatility and its relationship to the evolutionary process of continuous change. First, we defined and measured software volatility. Most prior discussions of software volatility have used counts of modifications to describe the volatility of programs and systems. We define, evaluate and validate a three-dimensional volatility measure to track software evolution during the full lifecycle of an information system.

A study of software volatility is also particularly challenging because little theory has been developed to guide our investigation. This work is a unique opportunity to use a particularly large data set to gain insight into incremental changes occurring in information systems as they age. This work began with an inductive approach. We adapted measures and concepts from research in other areas. Research from the fields of economics, manufacturing processes and software reliability was useful. Once software volatility was defined, this new quantitative measure was used to identify factors contributing to software volatility. Software volatility as a characteristic of recent software evolution was used to model determinants of software maintenance outcomes.

Future Work

The study of software evolution and the management of evolutionary processes fall in the intersection of software engineering and project management. Analyses in these fields must recognize information systems as the economic output of software producing organizations. The unique characteristics of software as a product, and the unique resources required for its creation, present researchers and practitioners with a number of interesting problems. Our recognition of the longevity of information systems and their constant modification dictates that research in these areas maintain a longitudinal perspective. The study of software evolution is the study of change. With expanded understanding of change processes, we can deal with questions of how and when to prepare for change, what types of systems change the most, and how these changes can be dealt with in the most effective and efficient manner. Now that we have a more fully descriptive measurement of software volatility, and methods for analyzing lifecycle maintenance patterns, we can build our understanding of software evolution and its effect on information systems. Both IS researchers and IS practitioners must recognize that software changes and evolves as it ages. Software evolutionary processes are neither good nor bad, but they are inevitable. With the ability to measure this volatility, we can understand what causes change, and anticipate the consequences.

	Research Question 1: **Defining a Multi-Dimensional Measure of Software Volatility**	**Research Question 2:** **Antecedents of Software Volatility**	**Research Question 3:** **Characteristics of Software Evolution and Lifecycle Maintenance Outcomes**
Topic	Multi-dimensional system-level measure of software volatility • Periodicity (how often?) • Amplitude (how big?) • Deviation (how well behaved?)	Predictive model: antecedents of software volatility • Software characteristics • Maintenance profiles (history) • Environmental factors	Predictive model: determinants of lifecycle maintenance outcomes • Basic System Characteristics • Software Evolution • Maintenance profiles • Software volatility
Contribution	• Full picture of lifecycle volatility - system-level measure, multi-dimensional, direct, objective, measures • Measurement provides basis for theory and testing • Gain perspective on life cycle behavior of software systems	• understanding evolutionary behavior • identify and understand driving forces of software volatility • help software engineers and managers design for change	• Link between software volatility and maintenance errors and costs • identify behavioral patterns in software lifecycles • Improves manager's ability to predict errors and costs
Importance to researchers	Direct objective measure, provide foundation for new theory	Start to explain differences in lifecycle system behavior	• Identify patterns in system evolution and link to lifecycle maintenance outcomes • Predictive validation of software volatility measures
Importance to MBA students	Lifecycle perspective on system management	Emphasizes need to design for change	Shows link between lifecycle maintenance outcomes and software volatility
Importance to undergraduates	System-level perspective on software behavior	Explain differences in software behavior	Demonstrate link between lifecycle maintenance costs and errors and system factors
Importance to industry	Can only manage what can be measured	Lifecycle perspective for system support resource requirements	Improve forecasting of lifecycle maintenance outcomes

Table 1: Summary of Contributions

APPENDIX A:

CONFIRMING EVIDENCE FOR LAWS OF SOFTWARE EVOLUTION

CONFIRMING EVIDENCE FOR LAWS OF SOFTWARE EVOLUTION

Lehman et al. (1997) developed a set of laws describing the evolution of software systems. Development of these laws occurred over 25 years, and was based on a series of empirical studies. Most of these studies used data from relatively short data collection periods (less than 3 years) and concentrated on the behavior of software for operating systems. In some cases the laws were confirmed by analyzing the same data used to formulate the laws (Yuen, 1987). The availability of a longitudinal set of data covering 23 distinct application systems affords us a unique opportunity to independently confirm those laws. Empirical data from the research site (Kemerer and Slaughter, 1997; 1999) offer an opportunity to investigate the first seven laws of software evolution.[1]

Laws of Software Evolution	Description
Law of Continuous Change	Systems must continually adapt to the environment to maintain satisfactory performance
Law of Increasing Entropy (later renamed Law of Increasing Complexity)	As systems evolve they become more complex unless work is specifically done to prevent this breakdown in structure
Law of statistically smooth growth (also called the Law of Self Regulation)	The software evolution processes are self-regulating and promote globally smooth growth of an organization's software
Law of invariant work rate (also called Law of Conservation of Organizational Stability)	The organization's average effective global activity rate is invariant throughout system's lifetime
Law of conservation of familiarity	Incremental growth rate of a system is constant to conserve the organization's familiarity with the software.
Law of continuing growth	Functional content of systems must be continually increased to maintain user satisfaction
Law of declining quality	System quality declines unless it is actively maintained and adapted to environmental changes
Law of system feedback	Software evolutionary processes must be recognized as multi-level, multi-loop, multi-agent feedback systems in order to achieve system improvement

Table 1: Laws of Software Evolution (Lehman et al., 1997)

[1] Law 8: the Law of System Feedback, cannot be tested with this data. Tests for system feedback require pre- and post-test data collection, similar to that planned in the FEAST research projects (Lehman and Ramil, 1999).

Each system can be analyzed individually using the full lifecycle history. Portfolio level analyses can be run by using the unbalanced panel data set used to investigate research question 2.

Laws of Software Evolution	Test Variable	Prediction[2]
Law of Continuous Change	Change events/month	Positive
	MTSM	Positive
Law of Increasing Entropy / Law of Increasing Complexity	Complexity	Increasing over time
Law of Statistically Smooth Growth	Change events / programmer-month	Constant over time
Law of Invariant Work Rate	Change in size / month	Constant over time
		Constant over time
	Change events / programmer-month	
Law of Conservation of Familiarity	Change in portfolio size / month	Constant over time
Law of Continuing Growth	Size	Increasing over time
Law of Declining Quality	Errors	Increasing over time
		Increasing over time
	Change events/month	

Table 2: Testing the Laws of Software Evolution

[2] Predictions will be checked with pair-wise correlations.

Variables needed to verify Laws of Software Evolution

Concept	Operational variable	Definition or name in *.dta file
age	portfolio age	Months since earliest system implementation
age	Avg. system age	Mean system age in portfolio
age	system age	Applage
age	Avg. program age	Mean program age in a system
age	Avg. LOC age	Mean LOC age in a system
Change	Periodicity	NMTSM
Change	Amplitude	NORMLOC
Change	# change events	Allchg
Change size	LOC added	LOC
Change size	New programs added	Creations
Change size	Cyclomatics added	Cyclom
Change size	Operands added	n2
Change size	Calls added	calls
Complexity	System total cyclomatic	Totcyclom
Complexity	System total n2	Totn2
Complexity	System total calls	Totcalls
Complexity	Normalized cyclomatic	Nrmcyclom
Complexity	Normalized n2	Nrmn2
Complexity	Normalized calls	Nrmcalls
System size	Total LOC	Totloc
System size	No. of programs	Module_count
Capacity unit	month	1 month
Capacity unit	Programmer-month	Programmer-count * 1 month
Work rate	Changes per month	Allchg / programmer count = wr1
Work rate	Change size per programmer-month	LOC / programmer count = wr2
Work rate	Change size per programmer-month	Programs / programmer count = wr3
Work rate	Change size per programmer-month	Cyclom / programmer count = wr4
Work rate	Change size per programmer-month	N2 / programmer count = wr5
Work rate	Change size per programmer-month	Calls / programmer count = wr6
Change rate	Change size	Programs created
Software faults	No. Of corrections	Sumcorr
Software faults	Total changes	Allchg

Table 3: Operational Variables

Law	Concept	Correlation expected	Regression expected	Plot needed
1	Change	Corr (change,age) > 0	Change = f(age) $\beta_{age} <> 0$	Change vs. age
2	Complexity	Corr (complexity. Age) > 0	Complexity = f(age) $\beta_{age} > 0$	Complexity vs. age
2	Deviation	Corr (deviation, age) > 0		Deviation vs. age
3	Change size	Corr (change size, age) > 0 T-test (change size = $\mu_{change\ size}$)		Change Size vs. age
4	Work rate = work / capacity unit	Corr (work rate, age) > 0 T-test (workrate = $\mu_{workrate}$)		Work rate vs. age
5	Change rate = Change in Portfolio size / capacity unit	Corr (change rate, portfolio age) T-test (change rate = $\mu_{change\ rate}$)		Change rate vs. portfolio age
6	System size	Corr (size, age) > 0	Size = f(age) $\beta_{age} > 0$	Size vs. age
7	Software faults	Corr (faults, age) > 0	Faults = f(age) $\beta_{faults} > 0$	Faults vs. age

Table 4: Operational Variables and Expected Relationships

Confirming Evidence

Law 1: Law of continuous change

	System age	Average program age	Average LOC age
Periodicity	-0.3239	-0.1272	0.1576
Amplitude	-0.1530	-0.1609	-0.2275
# change events	0.1495	0.0275	-0.1376

Correlation (change, age)

	N	Mean	Std dev.	k-s z	2-tailed p
Periodicity	3201	0.4802	0.4601	16.9010	0.0000
Amplitude	3201	0.0282	0.1203	23.0357	0.0000
# change events	3201	8.9510	10.2678	17.6562	0.0000

Testing for Normal Distribution

Law 2: Law of Increasing Entropy

	System age	Average program age	Average LOC age
System total cyclomatic	0.2932	0.0990	0.1194
System total n2	0.3526	0.1263	0.1151
System total calls	0.1554	-0.0412	0.0592
Normalized cyclomatic	-0.0680	0.0348	0.1333
Normalized n2	-0.0916	-0.0128	0.0802
Normalized calls	-0.0738	-0.1902	-0.0807

Correlation (complexity, age)

	N	Mean	Std dev	k-s z	2-tailed p
System total cyclomatic	3201	9138	11236	11.783	0.0000
System total n2	3201	32132	37361	11.1436	0.0000
System total calls	3201	2116	3628	15.832	0.0000
Normalized cyclomatic	3201	0.0566	0.0229	7.6711	0.0000
Normalized n2	3201	0.2187	0.0608	4.5047	0.0000
Normalized calls	3201	0.0080	0.0055	4.8847	0.0000

Testing for Normal Distribution

Law 3: Law of Statistically Smooth Growth

	System age	Average program age	Average LOC age
LOC	0.0052	-0.0650	-0.0903
Creations	0.0678	-0.0254	-0.1372
Cyclom	0.0002	-0.0646	-0.0903
N2	0.0166	-0.0631	-0.1075
calls	-0.0037	-0.0755	-0.0774

Correlation (change size, age)

t-tests

H_0: LOC = mean_{loc}	hypothesis cannot be rejected
H_0: creations = $\text{mean}_{\text{creations}}$	hypothesis cannot be rejected
H_0: cyclomatics = $\text{mean}_{\text{cyclomatics}}$	hypothesis cannot be rejected
H_0: n2 = mean_{n2}	hypothesis cannot be rejected
H_0: calls = $\text{mean}_{\text{calls}}$	hypothesis cannot be rejected

Law 4: Law of Invariant Work Rate

	System age	Average program age	Average LOC age
# changes / programmer months	-0.1312	-0.0586	-0.1126
LOC added/ programmer months	-0.1251	-0.1158	-0.1428
Programs changes / programmer months	-0.0839	-0.0215	-0.0566
Cyclomatics added / programmer months	-0.1458	-0.1267	-0.1649
N2 added / programmer months	-0.1418	-0.1302	-0.1783
Calls added / programmer months	-0.1007	-0.1076	-0.0965

Correlations (work rate, age)

t-tests

H_0: # changes/programmer-months = mean$_{\text{\# changes/programmer-months}}$
hypothesis cannot be rejected
H_0: LOC/programmer-months = mean$_{\text{loc/programmer-months}}$
hypothesis cannot be rejected
H_0: programs changed/ programmer-months = mean$_{\text{programs changed/ programmer-months}}$
hypothesis cannot be rejected
H_0: cyclomatics/ programmer-months = mean$_{\text{cyclomatics/ programmer-months}}$
hypothesis cannot be rejected
H_0: n2/ programmer-months = mean$_{\text{n2/ programmer-months}}$
hypothesis cannot be rejected
H_0: calls/ programmer-months = mean$_{\text{calls/ programmer-months}}$
hypothesis cannot be rejected

Law 5: Law of Conservation of Familiarity

	System age	Average program age	Average LOC age
# programs created / programmer months	-0.1553	-0.1220	-0.2025
LOC added/ programmer months	-0.1251	-0.1158	-0.1428

Correlations (change rate / capacity, age)

Law 6: Law of Continuing Growth

Correlations (size, age) [same correlation test as Law 3]

	N	Mean	Std dev	k-s z	2-tailed p
LOC	3201	2956	13969	23.547	0.0000
Creations	3201	1.0997	3.3372	20.935	0.0000
Cyclom	3201	139.834	674.903	23.6454	0.0000
N2	3201	492	2028	22.8564	0.0000
calls	3201	32.748	182.107	24.2515	0.0000

Testing for normal distribution

Law 7: Law of Declining Quality

	System age	Average program age	Average LOC age
# corrections	0.0750	-0.0089	-0.1337
# modifications	0.1495	0.0275	-0.1376

Correlation (faults, age)

	N	Mean	Std dev	k-s z	2-tailed p
# corrections	3201	8.951	18.2678	17.6562	0.0000
# modifications	3201	0.9528	2.3004	20.9571	0.0000

Testing for normal distribution

REFRENCES

Kemerer, C.F. and Slaughter, S.A., "Methodologies for Performing Empirical Studies: Report from the International Workshop on Empirical Studies of Software Maintenance", Empirical Software Engineering, Vol. 2, No. 2, 1997, pp. 109-118.

Kemerer, C.F. and Slaughter, S.A., 1999, "An Empirical Approach to Studying Software Evolution", IEEE Transactions on Software Engineering, Vol. 25, No. 4, pp. 493-509.

Lehman, M.M. and Ramil, J.F., "The Impact of Feedback in the Global Software Process", The Journal of Systems and Software, April 15, 1999, Vol. 46, No. 2,3, pp. 123-134.

Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E. and Turski, W.M., "Metrics and Laws of Software Evolution - The Nineties View", Metrics '97, the Fourth International Software Metrics Symposium, 1997, Albuquerque, NM.

Yuen, C.H., "A Statistical Rationale for Evolution Dynamics Concepts", Proceedings of the Conference on Software Maintenance, 1987, Austin, TX.

APPENDIX B:

DATA CODIFICATION AND SEQUENCE ANALYSIS METHODOLOGY

APPENDIX:
MAINTENANCE ACTIVITY CLASSIFICATION AND SEQUENCE ANALYSIS

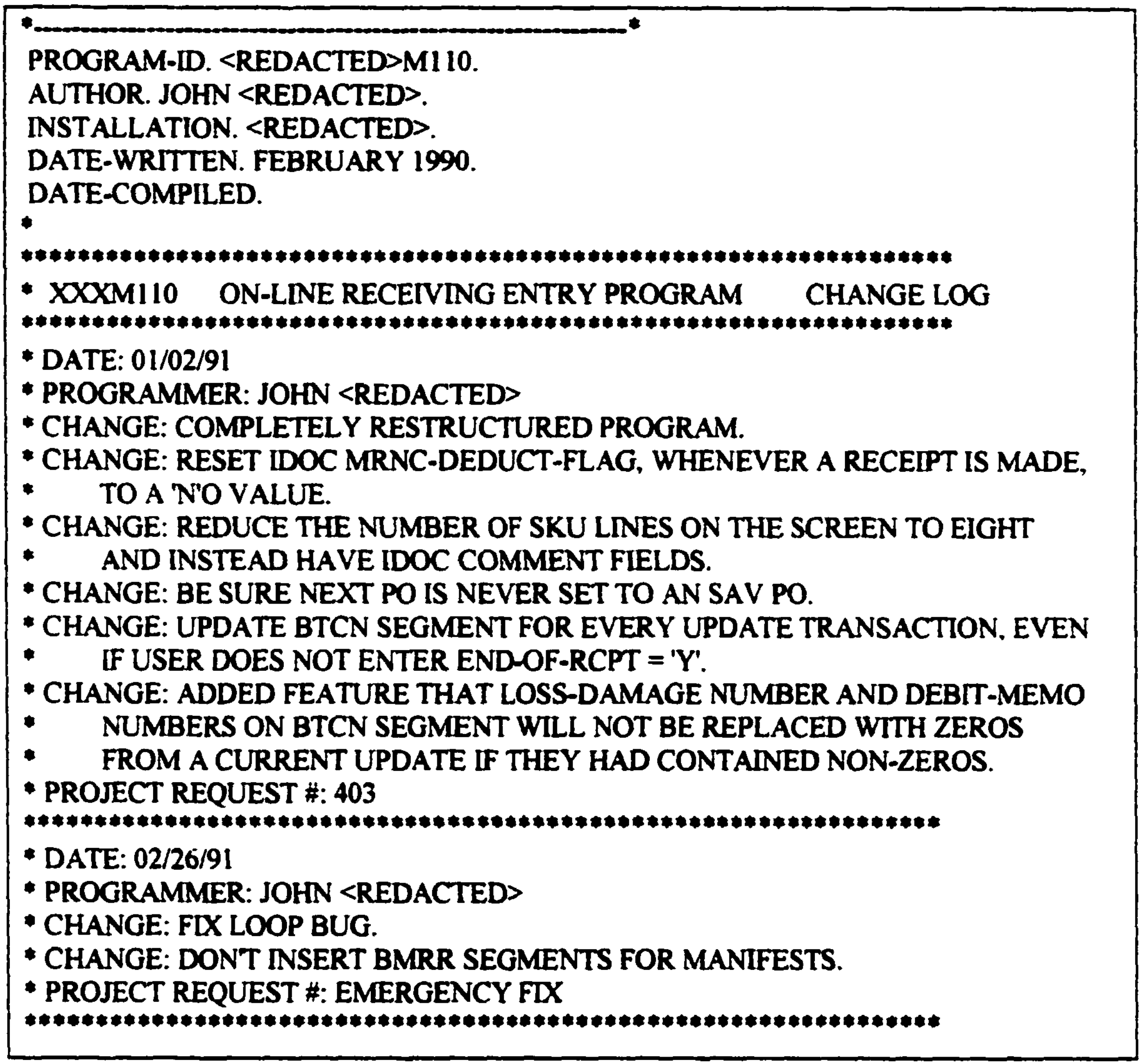
This is a step-by-step description of the data collection and codification processes used in this research. The following discussion concentrates on the collection and codification of maintenance activities and sequence analysis of those activities and associated levels of software volatility.

Maintenance Activity Classification

Step1: convert source code modification log to change history records

Software change events were extracted from maintenance logs written by system support programmers each time they updated a system in the portfolio. Logs were kept for more than 25,000 changes to 3,800 programs in 23 different information systems from the beginning of the early 1970's, when many of the systems were originally written, until the end of the data collection period, June 1993. The information systems represent more than two-thirds of the functionality accomplished by the full complement of the Retailer's systems. The data available in the change logs includes the original program creation date and author, program function, the maintenance project description, the programmer making a change, the date of the change, and the description of the change. In addition, the change logs reference the user project request. In many cases, the changes made to the systems are in response to emergency situations rather than user requests.

The data codification process captures the source and type of change made. Change event history records are dated by modification implementation date. For an example of a change log, see Figure 1. Such documentation allows the unit of analysis for this research to be the individual *change event*,[1] of which there were approximately 25,000 during this period.



```
*-----------------------------------------------------*
 PROGRAM-ID. <REDACTED>M110.
 AUTHOR. JOHN <REDACTED>.
 INSTALLATION. <REDACTED>.
 DATE-WRITTEN. FEBRUARY 1990.
 DATE-COMPILED.
*
*******************************************************************
* XXXM110    ON-LINE RECEIVING ENTRY PROGRAM      CHANGE LOG
*******************************************************************
* DATE: 01/02/91
* PROGRAMMER: JOHN <REDACTED>
* CHANGE: COMPLETELY RESTRUCTURED PROGRAM.
* CHANGE: RESET IDOC MRNC-DEDUCT-FLAG, WHENEVER A RECEIPT IS MADE,
*       TO A 'N'O VALUE.
* CHANGE: REDUCE THE NUMBER OF SKU LINES ON THE SCREEN TO EIGHT
*       AND INSTEAD HAVE IDOC COMMENT FIELDS.
* CHANGE: BE SURE NEXT PO IS NEVER SET TO AN SAV PO.
* CHANGE: UPDATE BTCN SEGMENT FOR EVERY UPDATE TRANSACTION, EVEN
*       IF USER DOES NOT ENTER END-OF-RCPT = 'Y'.
* CHANGE: ADDED FEATURE THAT LOSS-DAMAGE NUMBER AND DEBIT-MEMO
*       NUMBERS ON BTCN SEGMENT WILL NOT BE REPLACED WITH ZEROS
*       FROM A CURRENT UPDATE IF THEY HAD CONTAINED NON-ZEROS.
* PROJECT REQUEST #: 403
*******************************************************************
* DATE: 02/26/91
* PROGRAMMER: JOHN <REDACTED>
* CHANGE: FIX LOOP BUG.
* CHANGE: DONT INSERT BMRR SEGMENTS FOR MANIFESTS.
* PROJECT REQUEST #: EMERGENCY FIX
*******************************************************************
```

Figure 1: Portion of a Sample Change Log

Change logs were used to codify change events for each program or subprogram in the portfolio based on the classification scheme for identifying software maintenance activities. A complete breakdown of maintenance activity categories is shown in Table 1. This is the lifecycle maintenance activity taxonomy with additional categories delineating

[1] Change events include any add, change or delete of program source code, and implementation of any new

add, change or deletion of source code for each of the sub-types of activities. Capturing data at this detailed level allows for more flexibility in subsequent analyses.

Corrective	Enhancement/Perfective
Data Handling *(CorrectDat)*	Data Handling: Add, Change, Delete
Logic/Structure *(CorrectLog)*	*(EnhDatAdd, EnhDatChg, EnhDatDel)*
Computation *(CorrectCom)*	Logic/Structure: Add, Change, Delete
Initialization *(CorrectInit)*	*(EnhLogAdd, EnhLogChg, EnhLogDel)*
User Interface *(CorrectUserI)*	Computation: Add, Change, Delete
Module Interface *(CorrectModI)*	*(EnhComAdd, EnhComChg, EnhComDel)*
Adaptive	Initialization: Add, Change, Delete
Data Handling *(AdaptData)*	*(EnhIniAdd, EnhIniChg, EnhIniDel)*
Logic/Structure *(AdaptLogic)*	User Interface: Add, Change, Delete
Computation *(AdaptComp)*	*(EnhUsrIAdd, EnhUsrIChg, EnhUsrIDel)*
Initialization *(AdaptInit)*	Module Interface: Add, Change, Delete
User Interface *(AdaptUserI)*	*(EnhModIAdd, EnhModIChg, EnhModIDel)*
Module Interface *(AdaptModI)*	New Program *(NewProgram)*

Table 1: Classification Scheme *(Codes in parentheses)*[2]

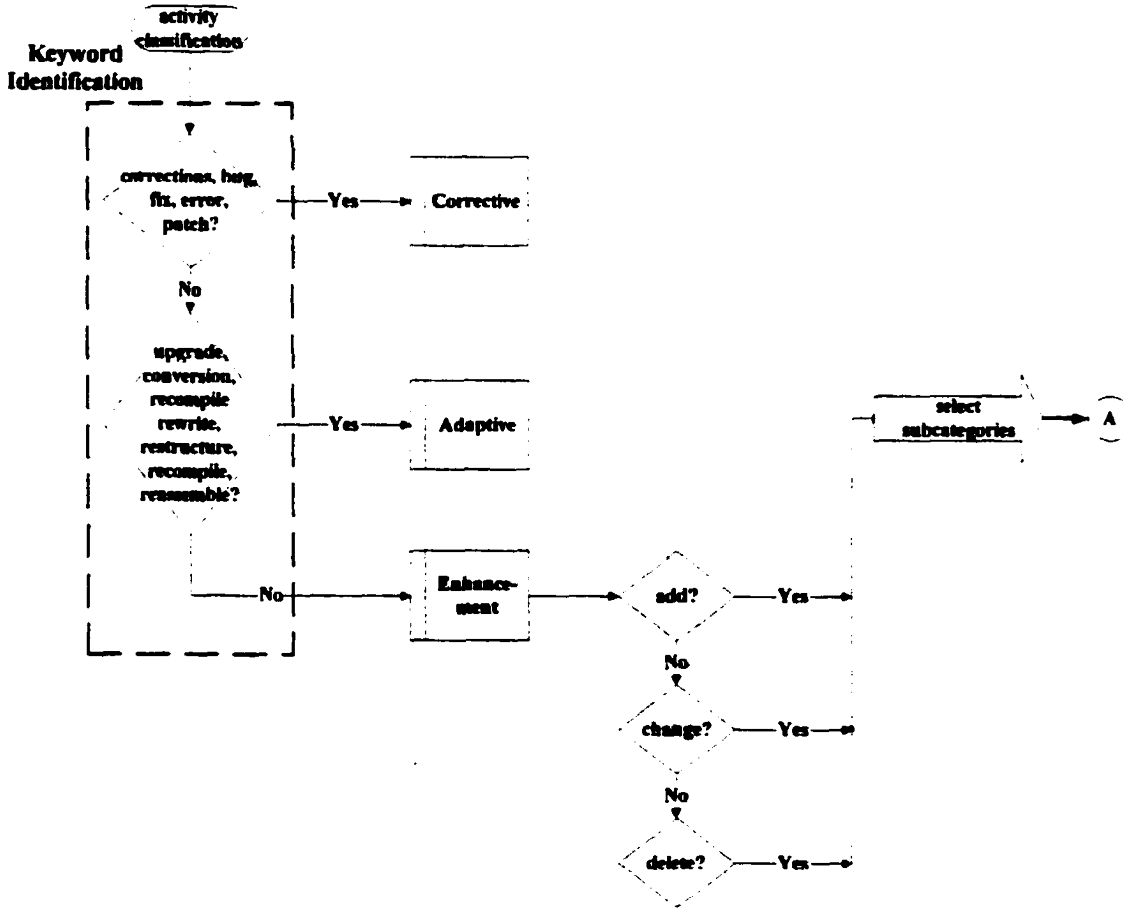
To classify each event in the change logs, a content analytic approach was adopted using a combination of latent and manifest coding techniques. Manifest coding involves looking through the text of the change log for visual occurrences of certain key words. Latent coding identifies the underlying meaning in text of the change log when key words are not sufficient to categorize events. Both approaches to coding will be necessary to account for possible inconsistencies in how the maintenance programmers logged their maintenance activities.

Multiple data coders were employed to content analyze the change logs. The coders were selected based upon their in-depth knowledge of the information systems field so they could identify terms and acronyms, and categorize events accurately. Because of the sensitivity of data-dependent research to error, it is important that measures be as reliable and valid as possible. Therefore, the data capturing procedures

programs in the system.

[2] Adapted from Rombach, Ulery and Vallett (1986)

employed a number of methodologies designed to maximize interrater reliability and to assess and improve coding validity. A coding flowchart was employed to provide a consistent way to classify change events. Each coder was instructed in a standard set of coding procedures. For consistency, coders referred any change event that could not be classified using the flowchart to the principal investigators for resolution. As these cases arose, adjustments were made to the coding flowchart. See Figure 2 for the final version of the coding flowchart.



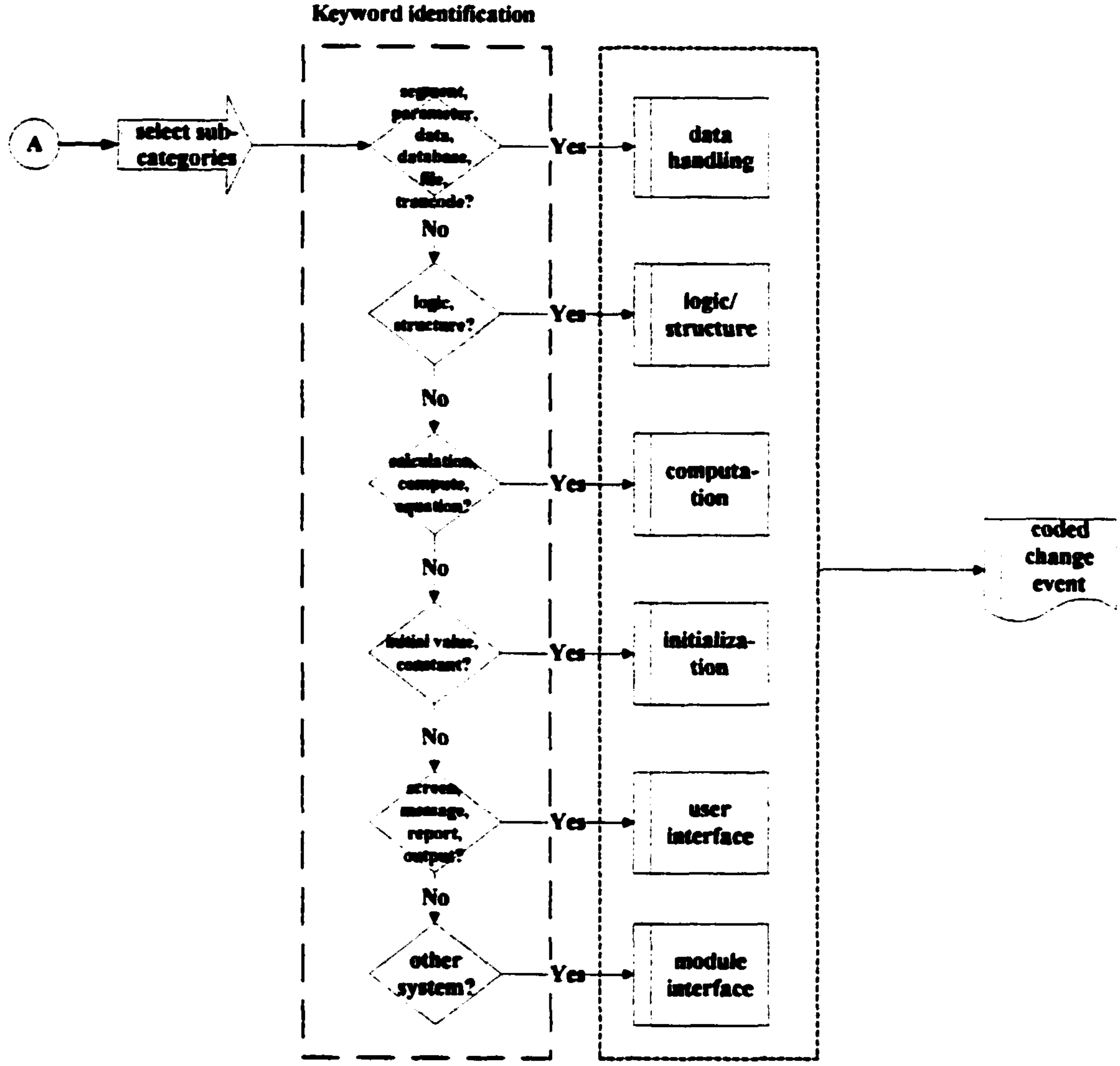


Figure 2: Coding flowchart

To increase intercoder consistency, several trial data coding processes were performed. In these trials, the primary investigators randomly selected a set of maintenance events. After each coder independently coded those selected maintenance events, the Cohen coefficient of agreement, Cohen's K, for nominal scales was computed to assess the relative pair-wise agreement between coders (Emam, 1999). Systematic differences in coding after each trial were discussed and resolved. The coders independently classified another set of maintenance events. When sufficient interrater reliability has been achieved, the maintenance events for the different information systems were divided among the coders. "Coder drift" or exhaustion was investigated by analyzing a sample of each coder's work near the end of the coding process and checking it against the principal investigator's coding of the same events. Tests for coder drift were repeated throughout the data coding process at periodic intervals. As another validity check, the principal investigators randomly inspected coded events to see if there were any degradations in accuracy. Finally, change logs were compared where possible with related data from the Retailer's maintenance activity tracking system, to ensure that the coded change logs are capturing the maintenance activity. All of these measures helped to insure the reliability and validity of the change history records.

The change event history records for each system were recorded in a spreadsheet. The researcher then had a series of 23 spreadsheets, containing one record for each date that each program was modified. If an individual program showed multiple change events on the same day, the maintenance activity entries represented the count of activities occurring in that program on that day. These spreadsheets will be referred to as the system change history file.

Step2 - start with change history records and create sum by date

The system change history file was sorted by year-month of change implementation, and counts in each category of maintenance activity aggregated by month. The resulting spreadsheet was referred to as the sum-by-date file. The range of dates covered in the sum-by-date file for each system varied according to the earliest program creation date for each system.

The sum-by-date file for each system was inspected for missing months. Any year-months missing were inserted into the sum-by-date file for each system. Counts for each maintenance activity category were set to zero on all inserted records.

Step3 - heuristic to identify maintenance activity for the month

Starting with the sum-by-date file for each system, each month was classified according to the major type of lifecycle maintenance performed that month. The classifying heuristic identified which maintenance activity type had the maximum frequency (count) in that month and labeled the year-month record accordingly. The label categories were *new, stable, corrective, adaptive* and *enhancement.* If two categories had equal frequencies, the phase name was set according to this same ordered list. For each year-month classified *enhancement*, a subcategory was assigned by a similar heuristic for six subcategories: *logic, data, computation, initialization, user interface* or *module interface* (Barry, Kemerer, Slaughter, 1999). Again, if there were equal frequency counts, the subcategory was labeled in this same order. See Figure 3 for an example using this heuristic for part of one system's activity identification process.

Change-YYMM	phase	genphase	new	stable	sum-correct	sum-adapt	sum-enhance	sum-enh-add	sum-enh-chg	sum-enh-delete	sum-enh-data	sum-enh-logic	sum-enh-compute	sum-enh-intfc	sum-enh-user	sum-enh-module
9206	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9207	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9208	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9209	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9210	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9211	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9212	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9301	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9302	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9303	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9304	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9305	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9306	logic	enhance	0	0	0	0	1	1	0	0	0	1	0	0	0	0
9307	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9308	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9309	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9310	new	new	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9311	logic	enhance	0	0	0	0	1	1	0	0	0	1	0	0	0	0
9312	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9401	new	new	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9402	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9403	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9504	new	new	1	0	0	0	2	2	0	0	2	0	0	0	0	0
9505	data	enhance	0	0	0	0	1	1	0	0	1	0	0	0	0	0
9506	logic	enhance	0	0	0	0	1	0	0	1	0	1	0	0	0	0
9507	correct	correct	0	0	2	0	1	0	1	0	0	1	0	0	0	0
9508	data	enhance	0	0	0	0	1	1	0	0	1	0	0	0	0	0
9509	logic	enhance	0	0	0	0	2	0	1	1	0	2	0	0	0	0
9510	data	enhance	0	0	0	0	4	4	0	0	2	0	0	0	2	0
9511	data	enhance	0	0	0	0	2	1	1	0	1	0	0	0	1	0
9512	logic	enhance	0	0	0	0	1	0	1	0	0	1	0	0	0	0
9601	new	new	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9602	logic	enhance	0	0	0	0	2	1	1	0	0	2	0	0	0	0
9603	logic	enhance	0	0	0	0	5	3	2	0	1	3	1	0	0	0
9604	logic	enhance	0	0	0	0	5	1	2	2	1	4	0	0	0	0
9605	new	new	1	0	0	0	5	3	2	0	1	4	0	0	0	0
9606	logic	enhance	0	0	1	0	2	2	0	0	1	1	0	0	0	0
9607	new	new	1	0	0	0	1	0	1	0	1	0	0	0	0	0
9608	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9609	new	new	1	0	0	1	0	0	0	0	0	0	0	0	0	0
9610	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9611	new	new	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9612	data	enhance	0	0	0	0	1	0	1	0	1	0	0	0	0	0
9701	data	enhance	0	0	3	3	4	2	2	0	3	1	0	0	0	0
9702	logic	enhance	0	0	0	0	5	2	1	2	2	3	0	0	0	0
9703	new	new	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9704	new	new	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9705	data	enhance	0	0	0	0	2	0	1	1	2	0	0	0	0	0
9706	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9707	new	new	1	0	1	0	0	0	0	0	0	0	0	0	0	0
9708	logic	enhance	0	0	0	0	4	0	4	0	2	2	0	0	0	0
9709	stable	stable	1	1	0	0	0	0	0	0	0	0	0	0	0	0
9710	logic	enhance	0	0	0	0	1	0	1	0	0	1	0	0	0	0
9711	logic	enhance	0	0	0	0	9	1	8	0	3	6	0	0	0	0
9712	new	new	1	0	0	0	2	1	1	0	0	1	0	0	0	1
9801	new	new	1	0	1	0	4	1	3	0	0	2	1	0	1	0
9802	new	new	1	0	0	2	3	0	3	0	1	1	0	0	1	0
9803	new	new	1	0	0	0	6	2	3	1	3	1	1	0	1	0
9804	data	enhance	0	0	2	0	5	4	1	0	3	2	0	0	0	0
9805	logic	enhance	0	0	4	0	42	12	28	2	8	26	2	0	4	2
9806	logic	enhance	0	0	3	0	20	7	13	0	5	10	2	0	3	0
9807	logic	enhance	0	0	0	0	9	2	6	1	3	3	0	0	2	1
9808	logic	enhance	0	0	0	0	8	0	4	4	2	5	0	1	0	0
9809	logic	enhance	0	0	2	0	8	1	7	0	2	3	2	0	1	0
9810	logic	enhance	0	0	0	0	8	0	5	3	0	5	2	0	1	0
9811	logic	enhance	0	0	0	0	13	8	3	2	4	7	1	0	1	0
9812	logic	enhance	0	0	0	0	20	2	17	1	3	17	0	0	0	0
9901	logic	enhance	0	0	2	0	12	5	6	1	3	7	0	0	2	0
9902	logic	enhance	0	0	4	0	13	2	9	2	6	6	0	0	1	0
9903	new	new	1	0	1	1	2	0	2	0	1	1	0	0	0	0
9904	new	new	1	0	0	0	6	3	2	1	3	3	0	0	0	0
9905	data	enhance	0	0	0	1	8	6	2	0	4	2	0	0	2	0
9906	data	enhance	0	0	2	1	11	7	4	0	4	2	1	0	4	0
9907	logic	enhance	0	0	1	0	6	2	4	0	1	2	1	0	2	0
9908	data	enhance	0	0	4	4	39	15	21	3	19	8	6	0	6	0
9909	data	enhance	0	0	3	3	23	7	11	5	13	8	0	0	2	0
9910	logic	enhance	0	0	0	3	10	3	5	2	2	7	0	0	0	1
9911	correct	correct	0	0	7	0	5	2	3	0	1	3	0	0	1	0
9912	logic	enhance	0	0	0	5	9	1	8	0	2	3	1	0	3	0
0001	data	enhance	0	0	2	0	14	8	6	0	7	6	1	0	0	0
0002	logic	enhance	0	0	1	0	19	10	9	0	9	10	0	0	0	0
0003	logic	enhance	0	0	0	0	4	2	0	2	1	2	0	0	1	0
0004	new	new	1	0	0	0	13	5	5	3	4	7	1	0	0	1
0005	logic	enhance	0	0	1	1	11	2	8	1	3	7	0	0	0	1
0006	data	enhance	0	0	0	0	17	9	6	2	6	5	1	0	2	3
0007	new	new	1	0	0	0	4	2	2	0	2	2	0	0	0	0
0008	logic	enhance	0	0	0	0	5	1	3	1	2	2	0	0	0	1
0009	logic	enhance	0	0	3	0	7	1	6	0	3	3	0	0	1	0
0010	logic	enhance	0	0	6	0	22	11	7	4	8	9	3	0	2	0
0011	data	enhance	0	0	2	0	12	4	8	0	5	4	2	0	1	0
0012	data	enhance	0	0	0	0	7	1	5	1	3	2	0	0	2	0
0101	data	enhance	0	0	0	0	14	7	7	0	10	1	1	0	2	0
0102	logic	enhance	0	0	0	0	10	2	5	3	3	5	0	0	0	2
0103	user	enhance	0	0	1	0	2	0	2	0	0	0	0	0	2	0
0104	logic	enhance	0	0	0	0	2	1	1	0	0	1	1	0	0	0
0105	logic	enhance	0	0	0	0	7	2	2	3	3	3	0	0	1	0
0106	data	enhance	0	0	0	0	11	2	8	1	6	4	0	0	1	0
0107	logic	enhance	0	0	1	0	7	2	3	2	2	4	0	0	1	0
0108	logic	enhance	0	0	0	0	9	0	8	1	3	4	0	0	1	1

Figure 3: labeling year-month by maintenance activity

Step-4 - chronological vector of maintenance profile

After each record in the system sum-by-date file had been identified by maintenance activity type, a chronological vector of maintenance activities was extracted. Each element in the vector represented a month's dominant activity. Separate chronological vectors of activities were created for each system. The vector corresponding to the example shown in Figure 3 is displayed in Figure 4.

```
phase
stable
stable
stable
stable
stable
stable
stable
stable
stable
stable
stable
stable
logic
stable
stable
stable
new
logic
stable
new
stable
stable
new
data
logic
correct
data
logic
data
data
logic
new
logic
logic
logic
new
logic
new
stable
new
stable
new
data
data
logic
new
new
data
stable
new
logic
stable
logic
logic
new
new
new
new
data
logic
logic
logic
logic
logic
logic
logic
logic
logic
logic
new
new
data
data
logic
data
data
logic
correct
logic
data
logic
logic
new
logic
data
new
logic
logic
logic
data
data
data
logic
user
logic
logic
data
logic
logic
```

Figure 4: chronological vector of maintenance activities

Step5 - phase map for maintenance activities

For each system, the chronological activity vector was input into *Winphaser* software to begin a longitudinal sequence analysis of information system behavior. The resulting categories of dominant monthly activities were used to create a chronological vector of phases for each system. These vectors were then individually analyzed by *Winphaser*[3] software to produce both phase and gamma maps (Pelz,1985; Kemerer and Slaughter, 1997; 1999). Phase mapping techniques analyze nominal data to identify sequences of similar categories. These sequences identify similarities in sequences of nominal data to show patterns of behavior. *Winphaser* maps the input vector of nominal data elements to a phase map to help analyze and identify patterns in the nominal input vector. *Winphaser* allows the user to vary phase length from one sequence analysis to another. By changing the phase length, researchers can simplify resulting sequential patterns and improve the confidence level of resulting phase maps. *Winphaser* identifies sequences of like activities with phases of specified phase length. If activities are so varied that none of the types are predominant, *Winphaser* creates a *Pending* phase. Smaller phase lengths create fewer pending phases in the phase mapping.

The input vector shown in Figure 4 was sequentially analyzed by *Winphaser* to produce the phase map shown in Figure 5.

[3] *Winphaser* software is used for sequence analysis of nominal data. *Winphaser* was written by Michael Holmes at University of Utah, as adapted from Holmes and Poole, 1991.

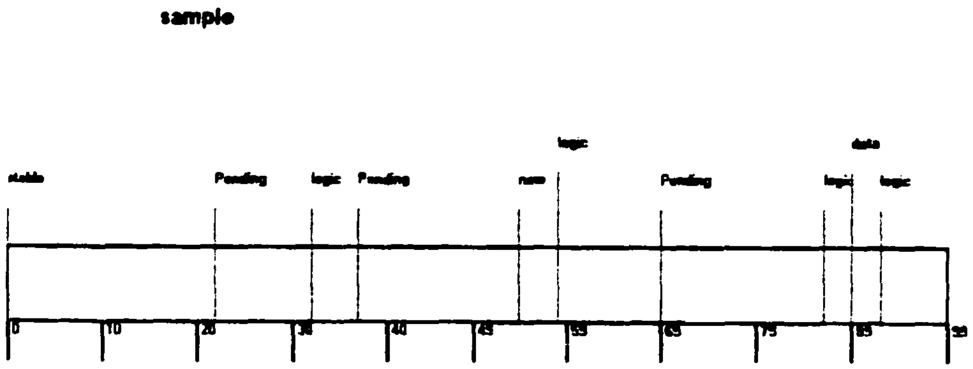


Figure 5: phase map from Figure 4 sample input vector (optional phase length = 3)

Step6 - gamma map for precedence ordering of maintenance profiles

Winphaser also provided a gamma analysis and precedence mapping to identify the predominant phase order. Phase length was set to insure an average confidence level of 50% for the gamma analysis for the input vector of maintenance activities in each system's history. The same phase length was used for both the activity phase map and gamma analysis. The heuristic used for setting phase length was to find the smallest phase length that allowed at least a 50% average confidence in the precedence ordering, thus creating the fewest pending phases with the required level of confidence in the identified phase ordering. Phase lengths varied from system to system. Figure 6 shows the gamma analysis and precedence map established by the input vector in Figure 4.

```
WinPhaser Gamma Analysis

SAMPLE~1  09-24-1999 14:14:50  L3  N100  D1-8  99  (Full Length)

Phase Frequency

data           3
logic         26
new            4
Pending       44
stable        22

Precedence Counts

               data     logic       new  Pending    stable

data              0        57        12      132        66
logic            21         0        84      787       572
new               0        20         0      108        88
Pending           0       357        68        0       968
stable            0         0         0        0         0

Pairwise Gamma Scores

               data     logic       new  Pending    stable

data           .000      .462     1.000    1.000     1.000
logic         -.462      .000      .615     .376     1.000
new          -1.000     -.615      .000     .227     1.000
Pending      -1.000     -.376     -.227     .000     1.000
stable       -1.000    -1.000    -1.000   -1.000      .000

Separation Scores

               data     logic       new  Pending    stable

              0.865     0.613     0.711    0.651     1.000

Precedence Scores

               data     logic       new  Pending    stable

             -0.865    -0.382     0.097    0.151     1.000

Phase Diagram

        stable***   Pending**       new**      logic**      data***

 *   .25 < separation < .50
 **  .50 < separation < .75
 *** .75 < separation
```

Figure 6: gamma analysis and precedence map

Software Volatility Analysis:

Step1 - sort change history records

The change history file for each information system was sorted: primary sort system name, secondary sort change date.

Step2 - calculate time since previous software modification

The sorted change history records were expanded to include a calculated field for time since previous software modification. The elapsed time since the prior change record for that program is calculated.

Step3 - re-sort change history records

The resulting expanded change history file was sorted by year-month.

Step4 -calculate MTSM and change dispersion

Change history records were aggregated by the time unit of analysis.[4] In most cases the unit of analysis is monthly and the records were aggregated by year-month of change implementation. The Mean of Time since Software Modification, MTSM, was calculated for each year-month, and the variance for each MTSM was recorded as the change dispersion for the same year-month.

Step5 - set periodicity, amplitude and deviation high/low indicators

For each information system, the lifetime mean periodicity, amplitude and deviation were calculated. High/low indicators[5] for each dimension are set for each month.

[4] This discussion relies on measures aggregated by month. Software volatility measures to investigate research question 3 on maintenance costs calculate software volatility on a quarterly basis.

[5] Periodicity indicators are set for long/short to indicate time intervals longer or shorter than average.

Step6 - identify volatility classification for each month as follows:

Two coding schemes for nominal classification of software volatility are used. The first scheme classifies volatility by examining characteristics of the periodicity (timing) and deviation (predictability) of software modifications. The second scheme uses all three dimensions of software volatility. Scheme 2 classified systems behavior by amplitude/periodicity/deviation.

Classification	Periodicity level	Deviation level
A	Short	Low
B	Short	High
C	Long	Low
D	Long	High

Volatility Classification Scheme 1: Periodicity/Deviation

Classifica-tion	Amplitude	Periodicity	Deviation	Description
I	Low	Long	Low	Least volatile: occasional small modifications occurring in a well-behaved pattern
II	Low	Long	High	Occasional small modifications with wide variance of behavior among system programs
III	Low	Short	Low	Constant small modifications occurring in a well-behaved pattern
IV	Low	Short	High	Constant small modifications with wide variance of behavior among system programs
V	High	Long	Low	Occasional large modifications occurring in a well-behaved pattern
VI	High	Long	High	Occasional large modifications with wide variance of behavior among system programs
VII	High	Short	Low	Constant large modifications occurring in a well-behaved pattern
VIII	High	Short	High	Most volatile: constant large modifications with wide variance of behavior among system programs

Volatility Classification Scheme 2: amplitude/.periodicity/deviation

Table 2: High/low indicators and volatility classifications

Step7 : chronological vector of volatility classifications

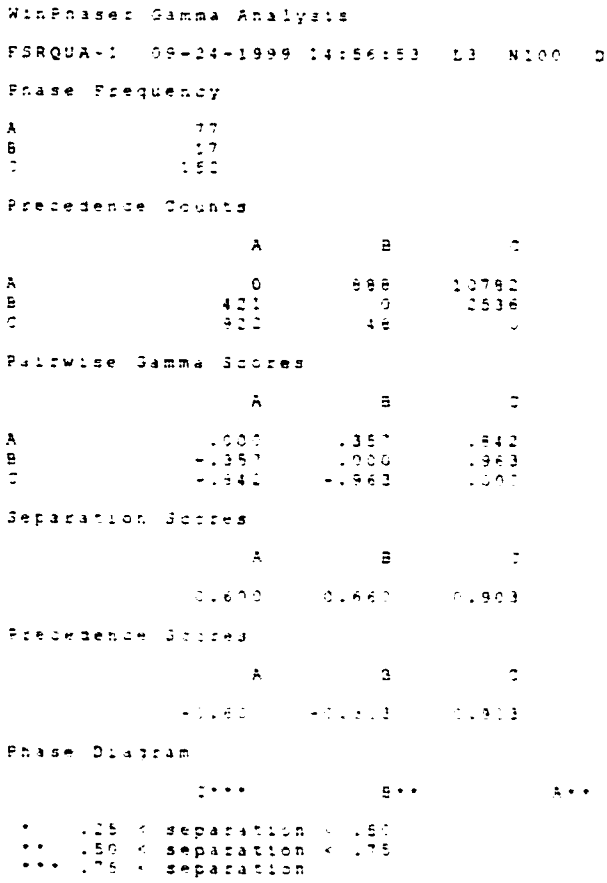
As with the chronological vector of activity phases, a similar vector of volatility classifications is created for each information system. After the appropriate classification scheme is selected, a chronological vector is created for the life span of the information system. This vector is used as an input file for *Winphaser* mapping and gamma analysis.

Step8 : phase map of volatility classifications

Winphaser sequence analysis was run for the chronological vector of volatility classifications associated with each system.

Step9 : gamma map for precedence ordering of volatility classifications

This same sample data was used as input for gamma analysis and precedence ordering produced by *Winphaser*. The 4-level classification of volatility was used as a basis for the sample gamma analysis in Figure 7.



```
WinPhaser Gamma Analysis

FSRQUA-1  09-24-1999 14:56:53  L3  N100  D1-8  246  (Full Length)

Phase Frequency

A                 77
B                 17
C                152

Precedence Counts

                   A          B          C

A                  0        888      10782
B                421          0       2536
C                922         46          0

Pairwise Gamma Scores

                   A          B          C

A               .000       .357       .842
B              -.357       .000       .963
C              -.842      -.963       .000

Separation Scores

                   A          B          C

               0.600      0.660      0.903

Precedence Scores

                   A          B          C

              -0.600     -0.303      0.903

Phase Diagram

                C***         B**          A**

  *    .25 < separation < .50
  **   .50 < separation < .75
  ***  .75 < separation
```

Figure 7: Sample Gamma analysis and precedence ordering

The example gamma analysis in Figure 7 identified 77 software volatility A phases, 17 software volatility B phases and 152 software volatility C phases. The precedence ordering reports that most of the time the system's volatility travels from type C to type A to type B.

__Step10 - radial graphs of stability quadrants__

To show ordering and relative magnitude of volatility classifications the precedence ordering and volatility phase frequency reported from the gamma analysis were normalized and displayed in the form of a radial graph. First, the classifications were ordered by severity and degree of volatility from a software manager's planning perspective. The ordering listed in Table 3 was based on Classification scheme 1 using periodicity/deviation.

Volatility classification	Order
C	5
D	4
Pending	3
A	2
B	1

Table 3: Volatility Classification Ordering

The length of each phase was normalized as the proportion of that type relative to all types identified by the gamma analysis. For the sample in Figure 7, the normalization is as follows:

$$A: 77/(77+17+152) = 0.30$$

$$B: 17/(77+17+152) = 0.07$$

$$C: 152/(77+17+152) = 0.63$$

The resulting radial graph is displayed in Figure 8.

Radial graphs were created for each information system to allow visual comparison of the changes in volatility over system life spans. A set of graphs was created for each classification scheme. They are displayed in the next two appendices.

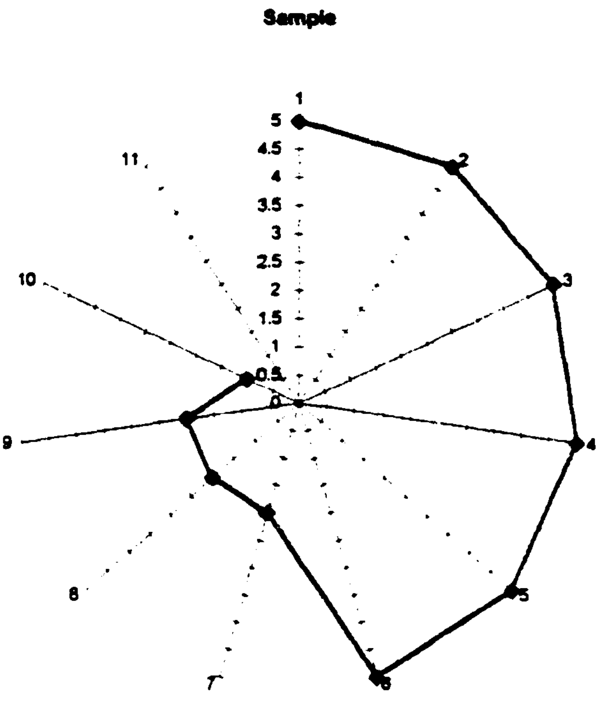


Figure 8: Radial graph of volatility gamma analysis

REFERENCES

Barry, E.J., Kemerer, C.F., and Slaughter, S.A., "An Empirical Analysis of Software Evolution Profiles and Outcomes", Proceedings of the International Conference on Information Systems, Charlotte, NC, December 1999.

Emam, Khaled El, "Benchmarking Kappa: Interrater Agreement in Software Process Assessments", Empirical Software Engineering, 1999, No. 4, pp. 113-133.

Holmes, M.E., and Poole, M.S., "Longitudinal analysis", in S. Duck and B. Montgomery (Eds.), Studying interpersonal interaction, New York, Guilford, 1991, pp.286-302.

Kemerer, C.F., and Slaughter, S.A., "Determinants of Software Maintenance Profiles: An Empirical Investigation", Journal of Software Maintenance, 1997, Vol. 9, pp. 235-251.

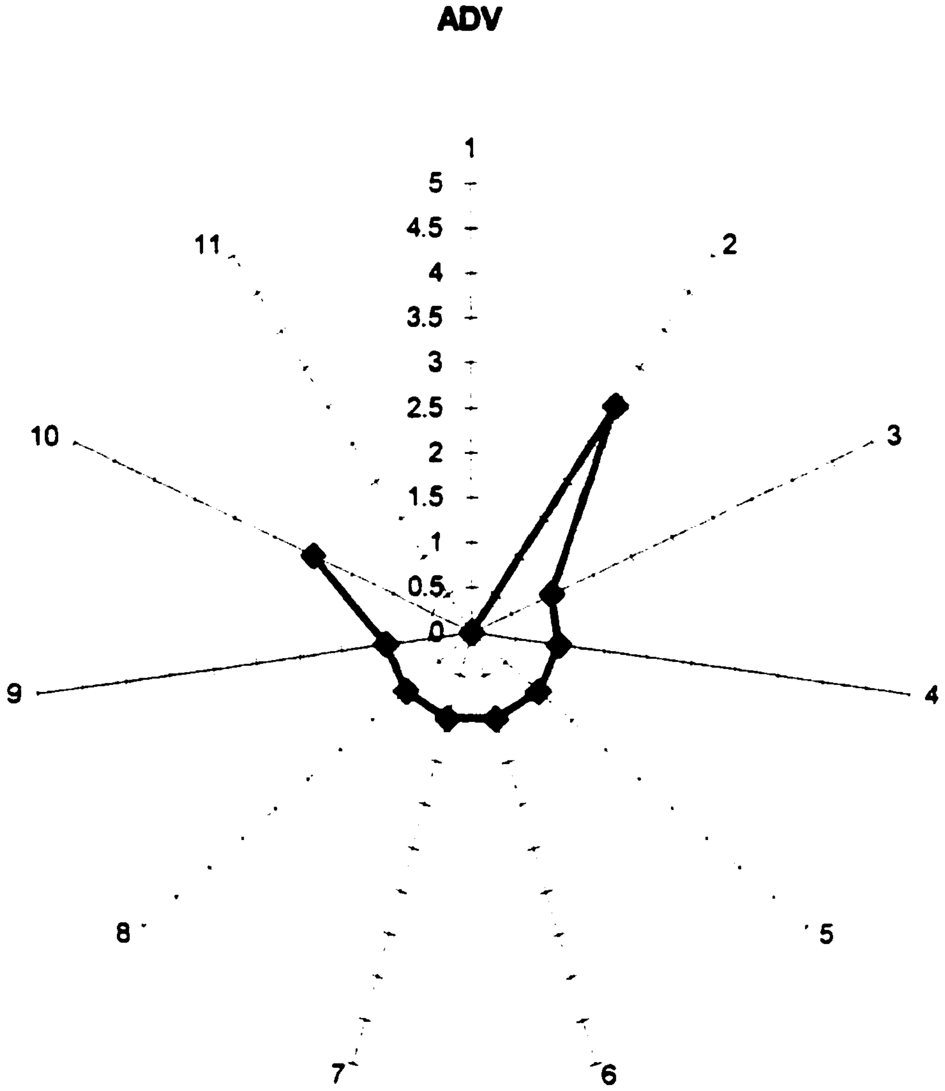
Kemerer, C.F., and Slaughter, S.A., "An Empirical Approach to Studying Software Evolution", IEEE Transactions on Software Engineering, July-Aug. 1999, Vol. 25, No. 4, pp.493-509.

Pelz, Donald C., "Innovation Complexity and the Sequence of Innovating Stages", Knowledge, Creation, Diffusion, Utilization, March 1985, Vol. 6, No. 3, pp. 261-291.

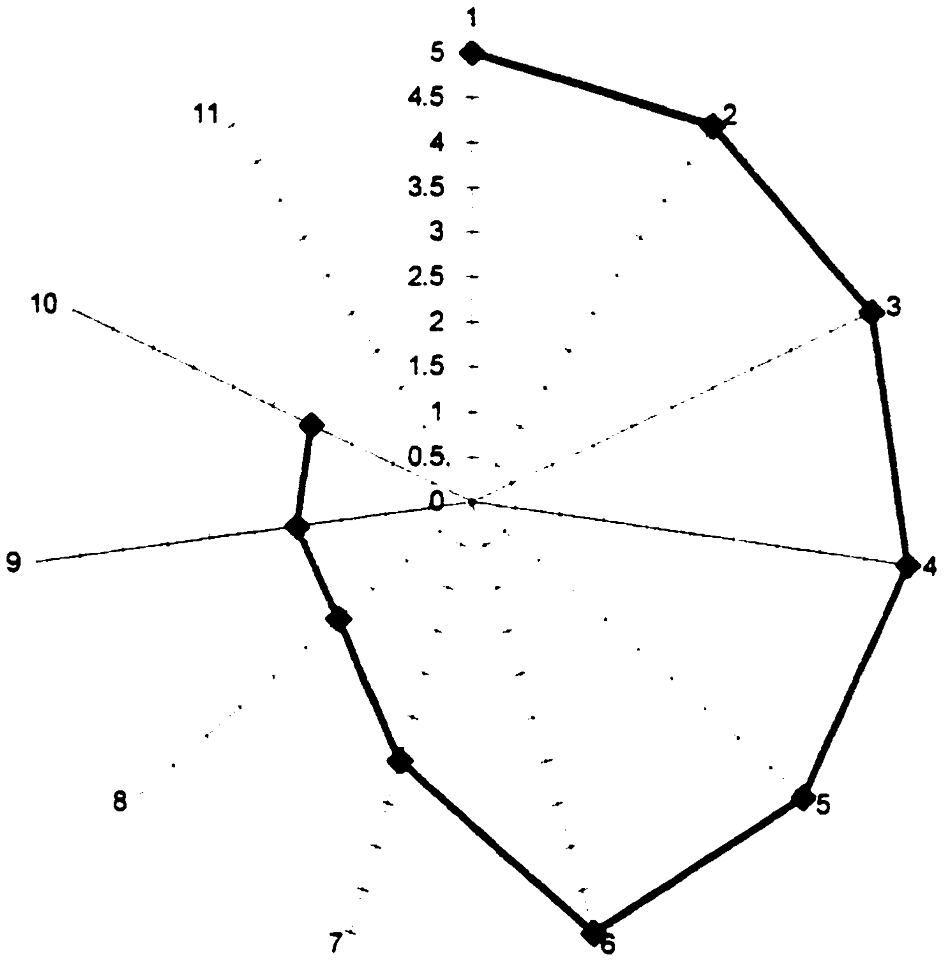
APPENDIX C:

RADIAL GRAPH REPRESENTATIONS OF GAMMA ANALYSES

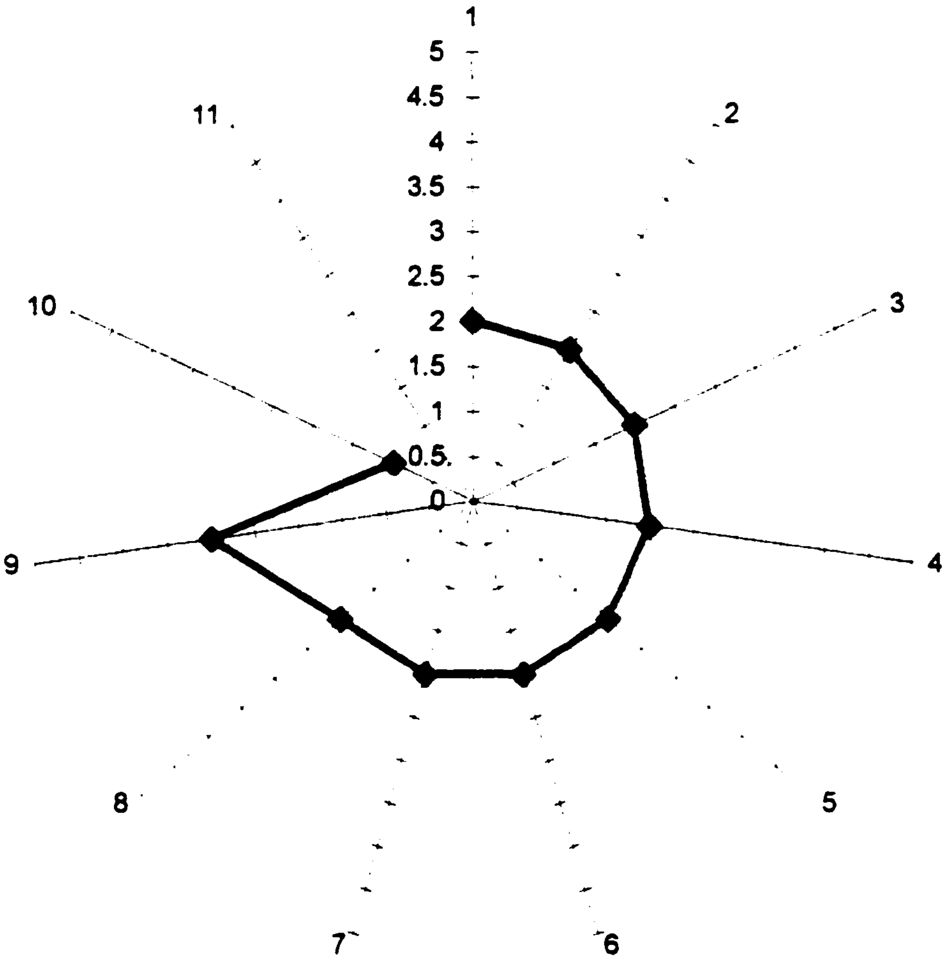
Using Periodicity Deviation Classification of Software Volatility


ADV
1
2
3
4
5
6
7
8
9
10
11
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0

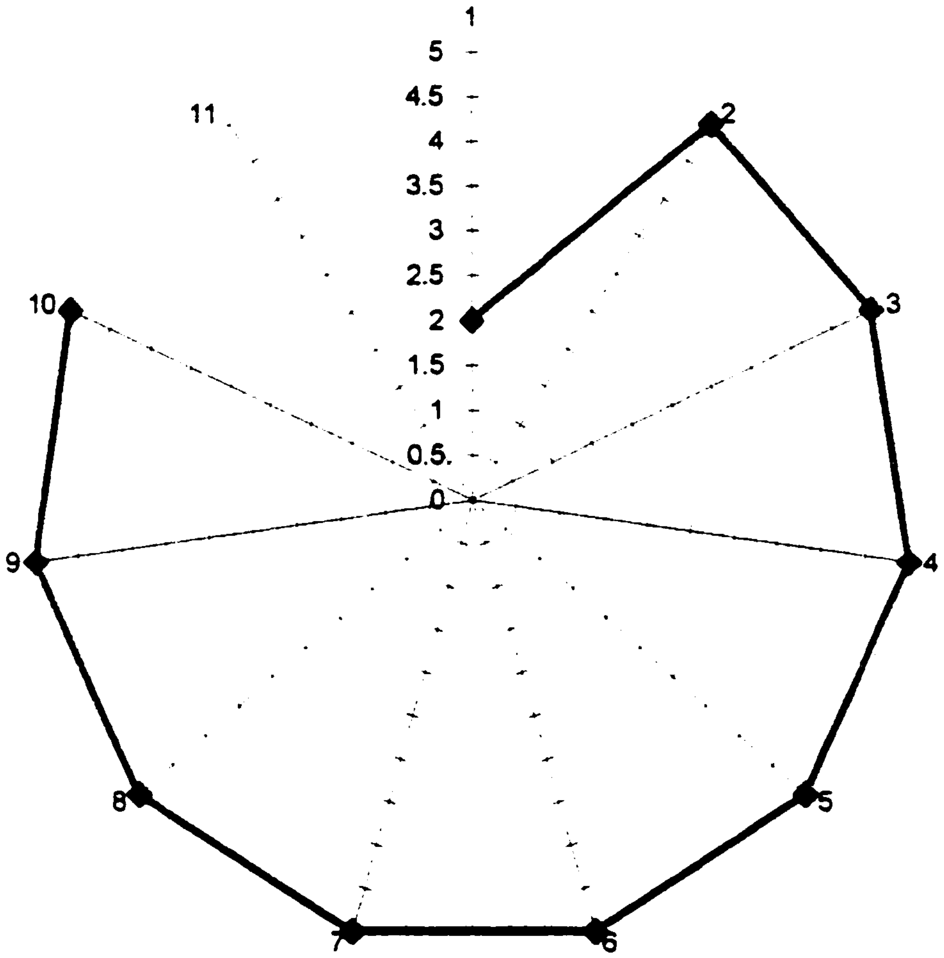
APX



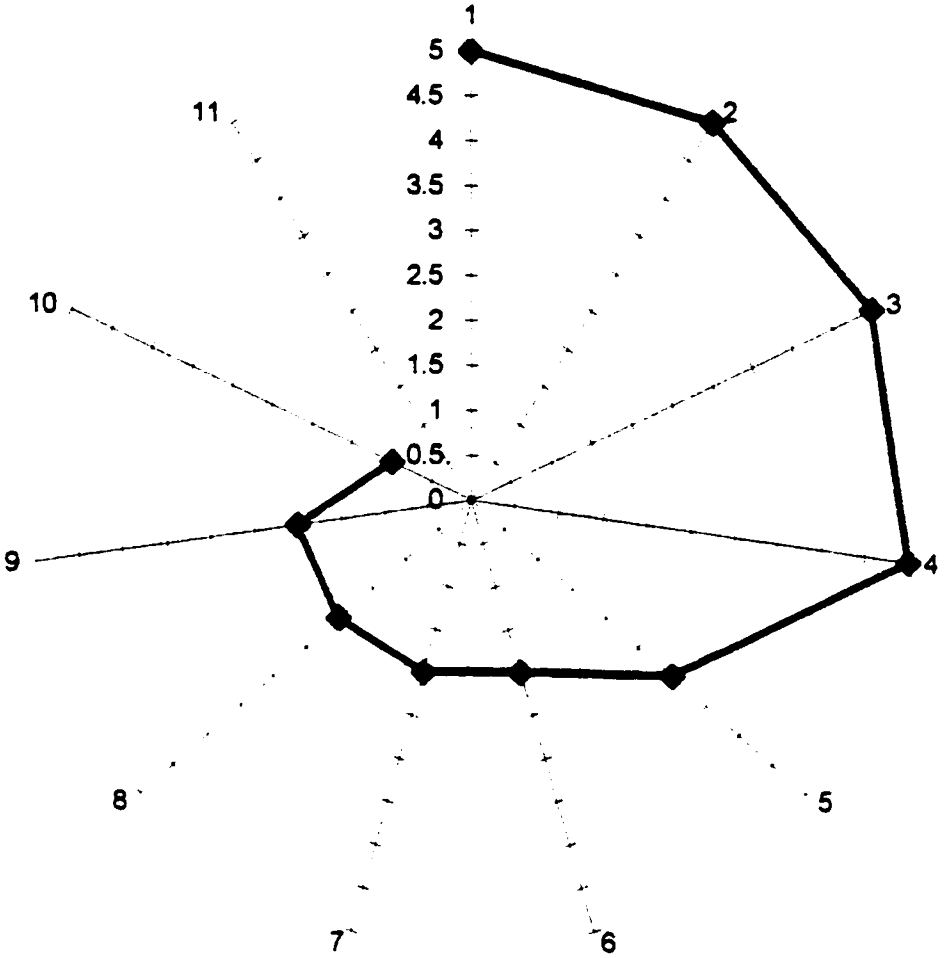
ARA



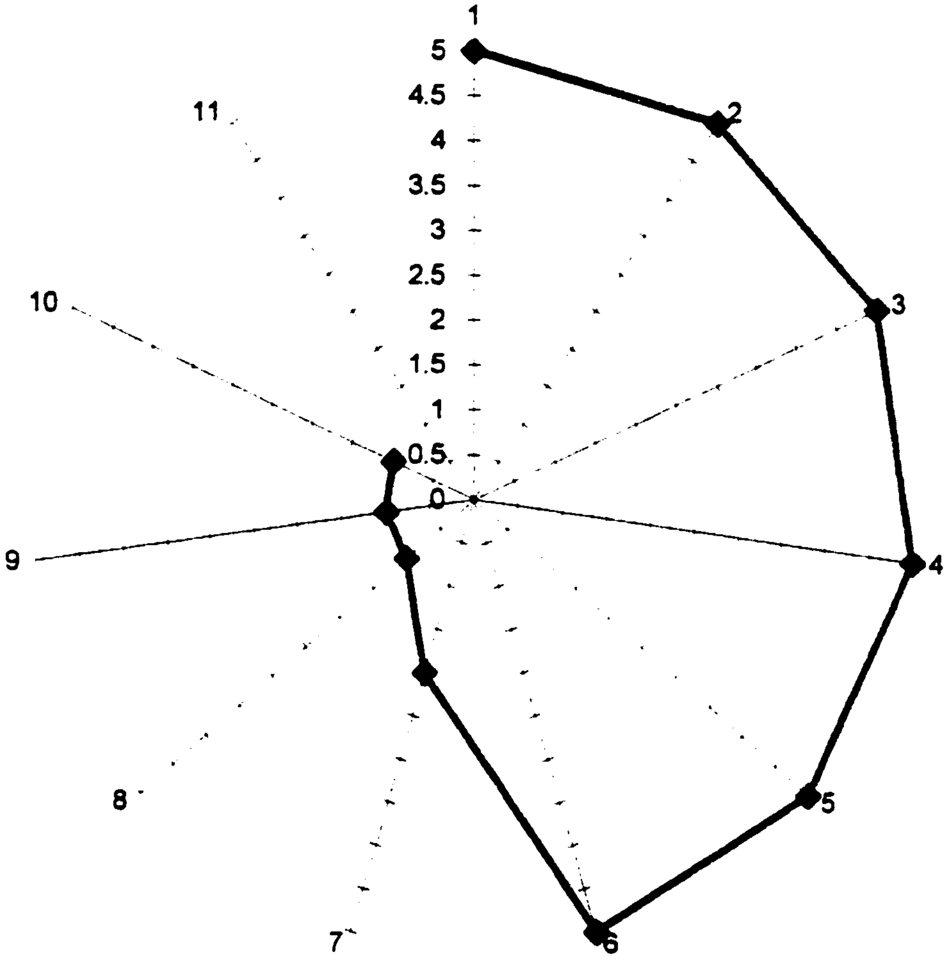
ARI



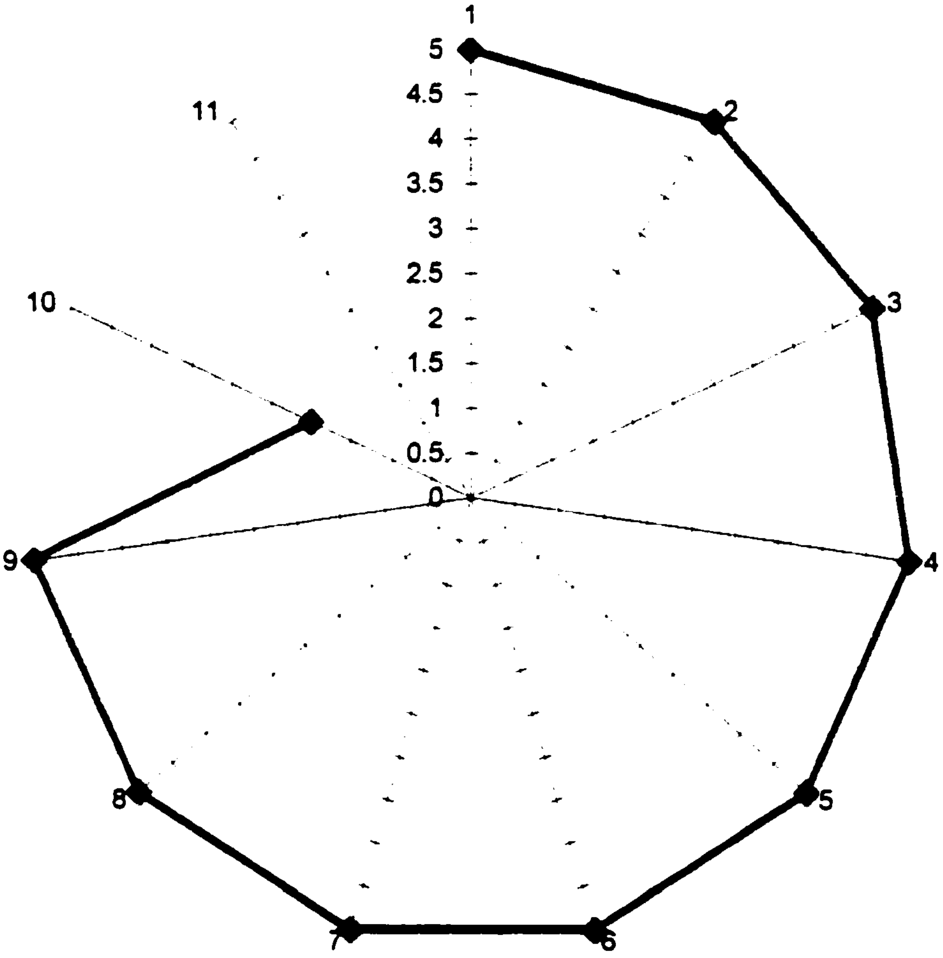
ARS

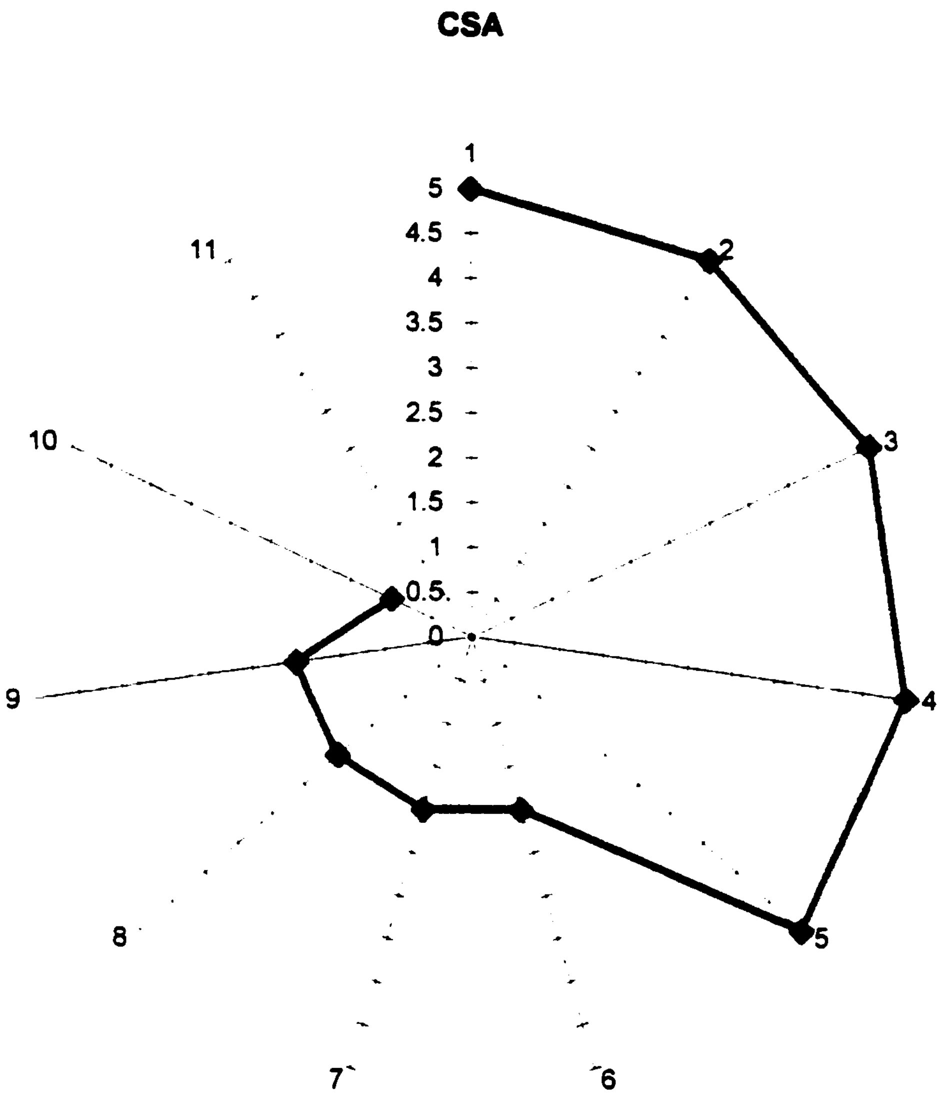


BTK

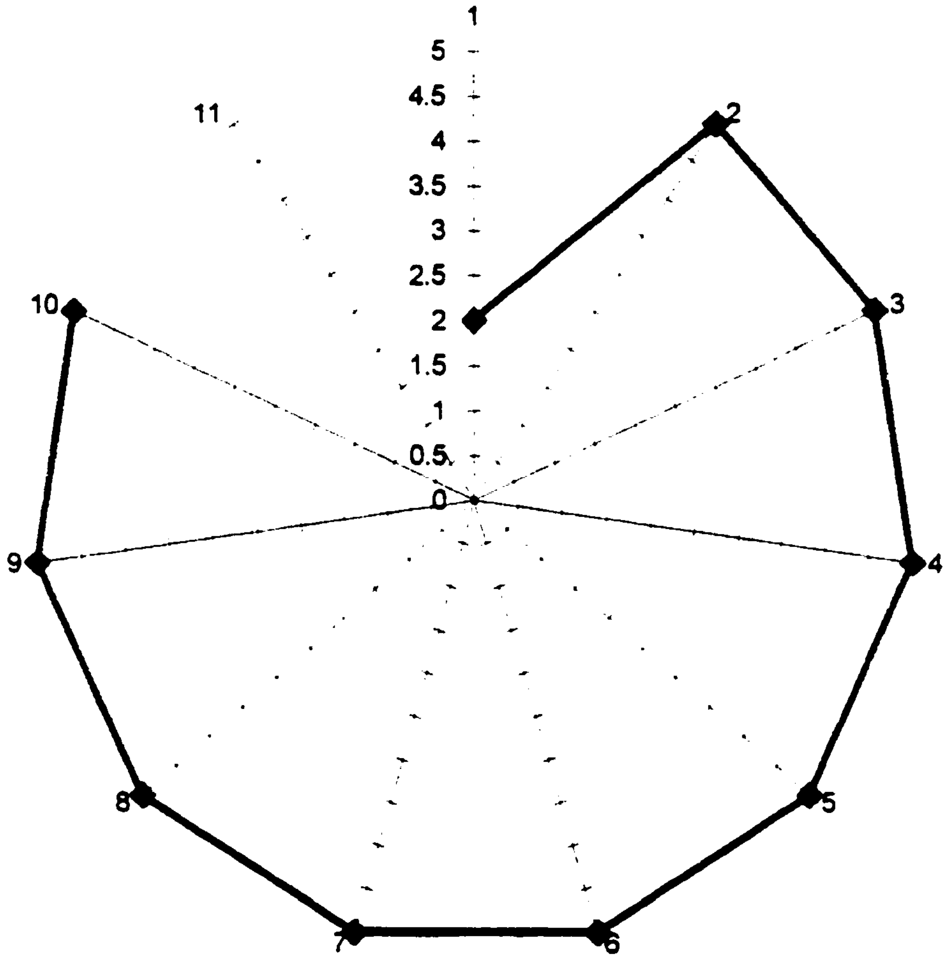


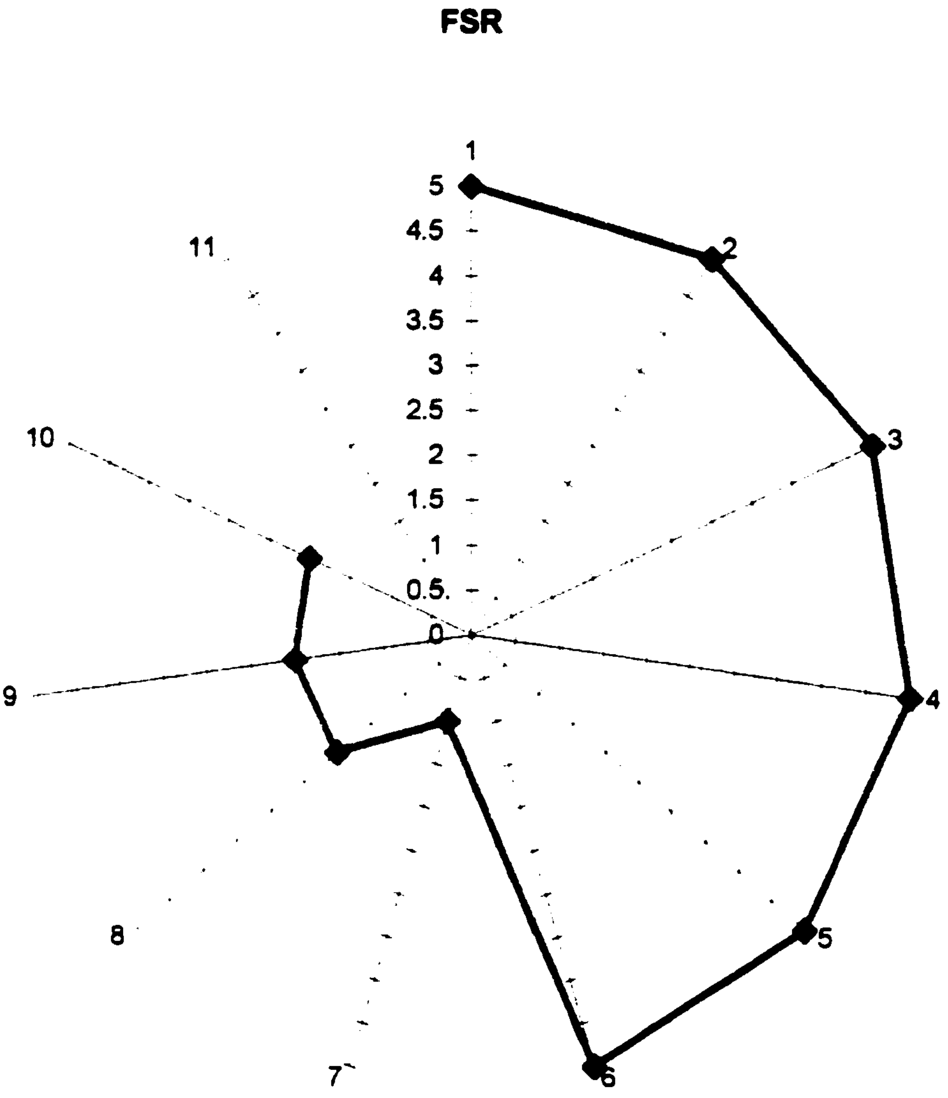
CPM



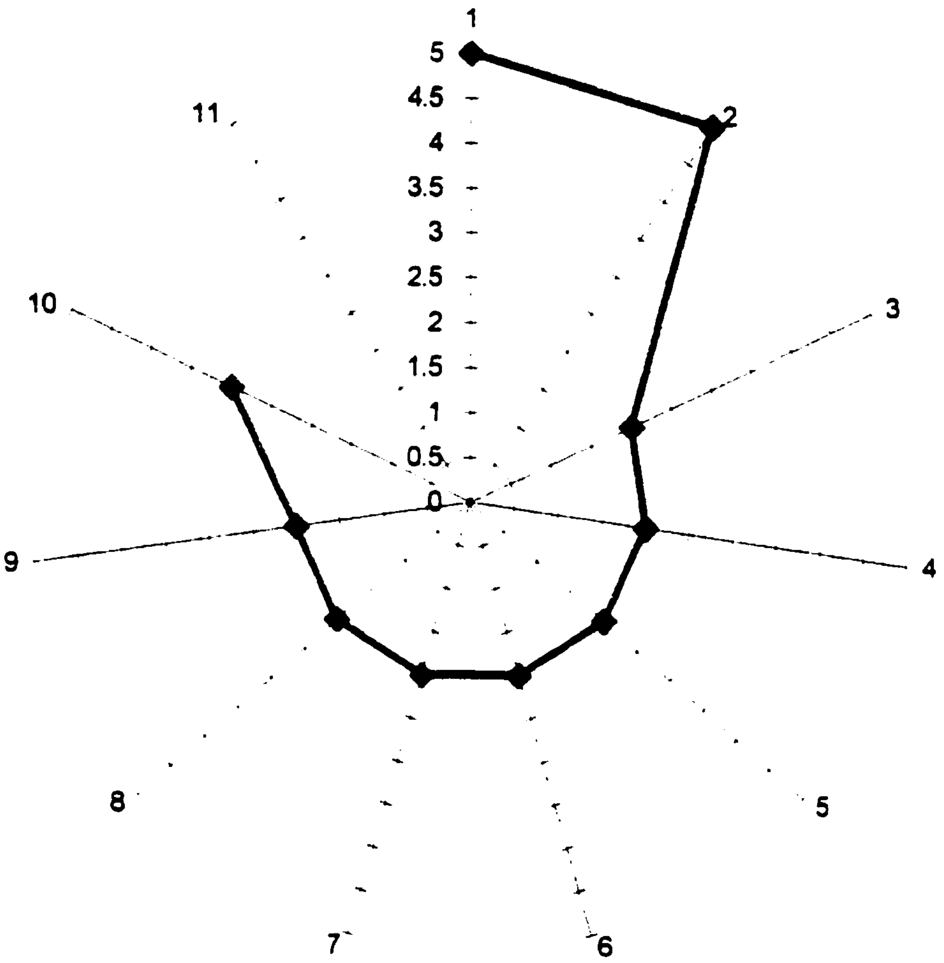

CSA
1
2
3
4
5
6
7
8
9
10
11
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0

FAM

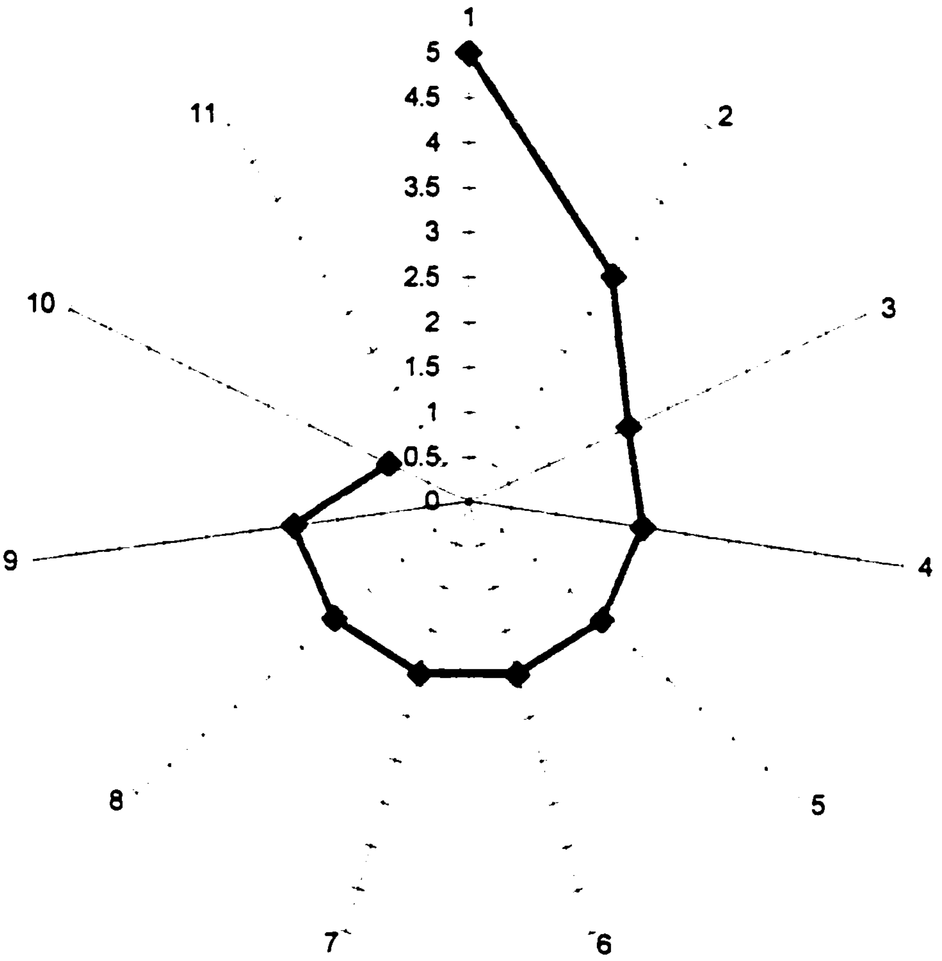


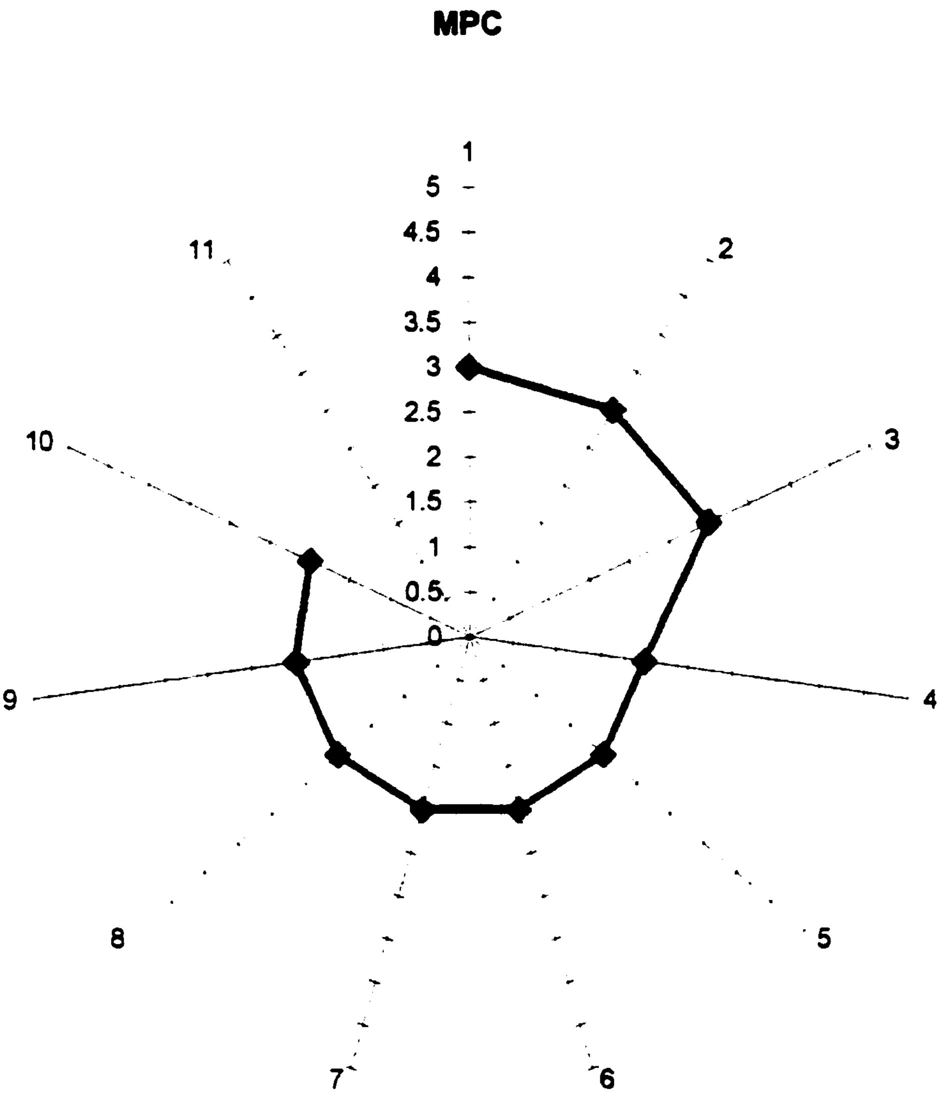

FSR
1
2
3
4
5
6
7
8
9
10
11
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0

GLM

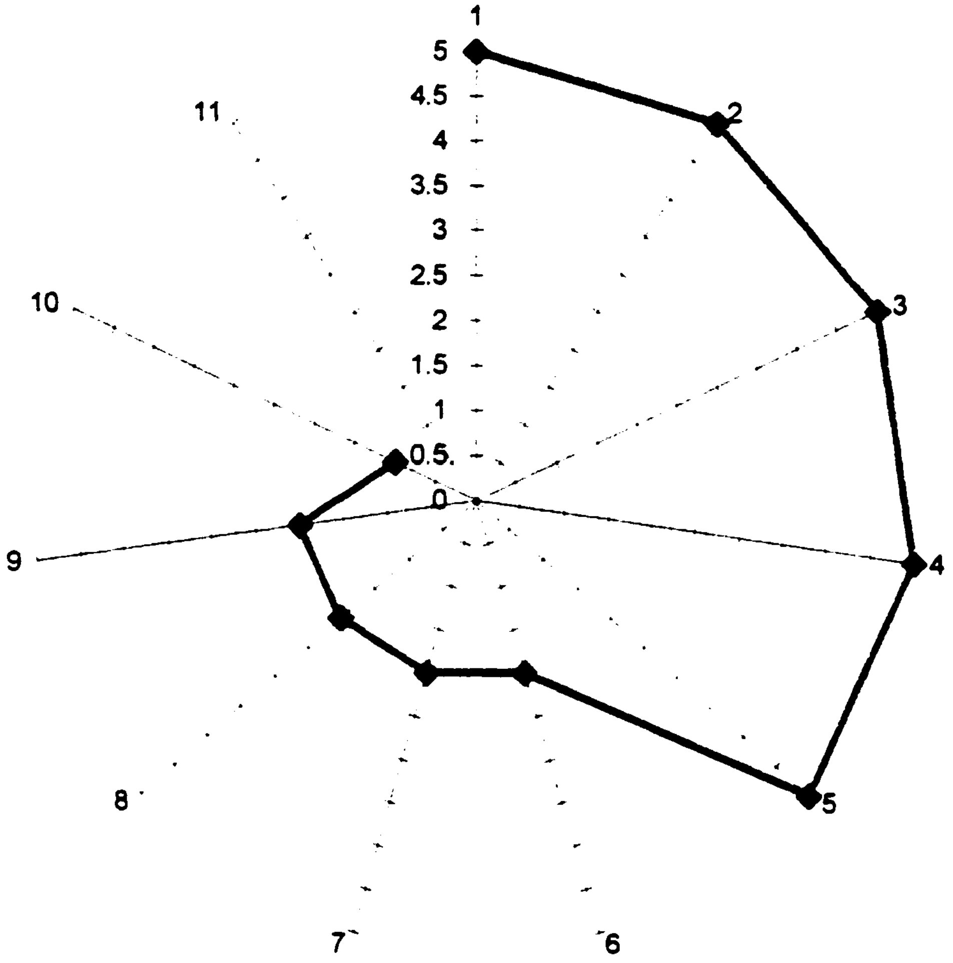


MAN

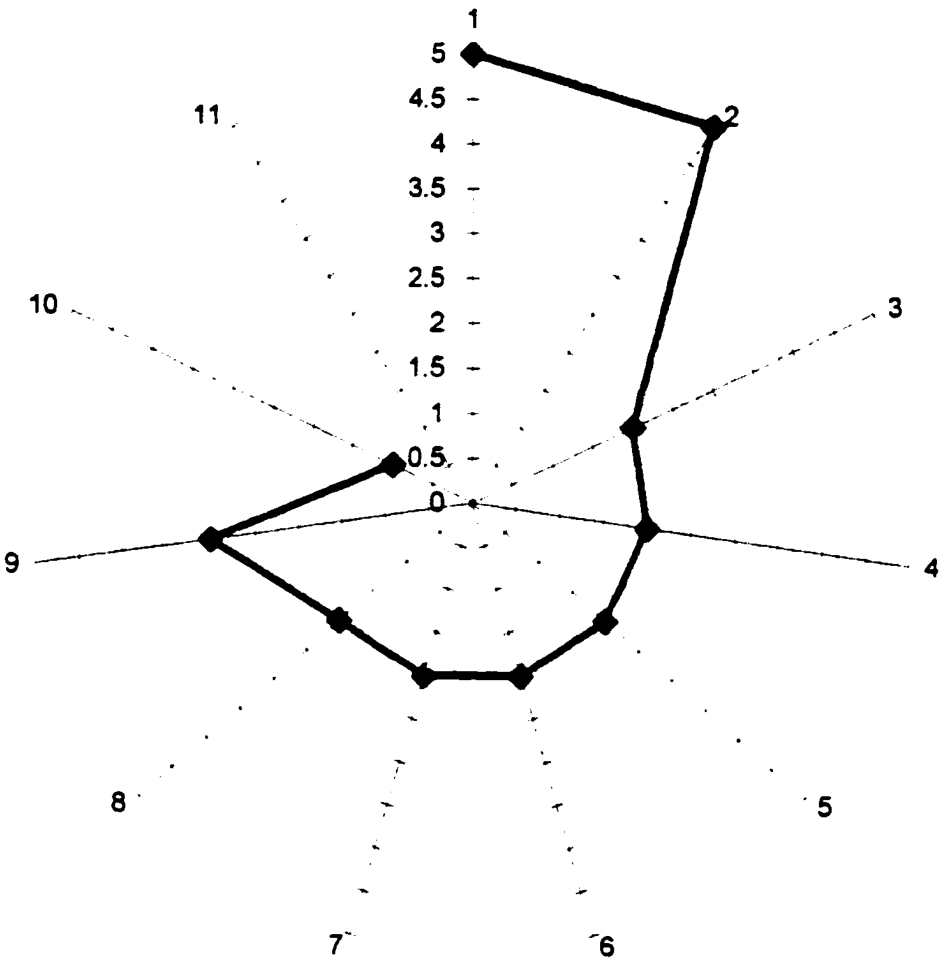



MPC
1
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
2
3
4
5
6
7
8
9
10
11

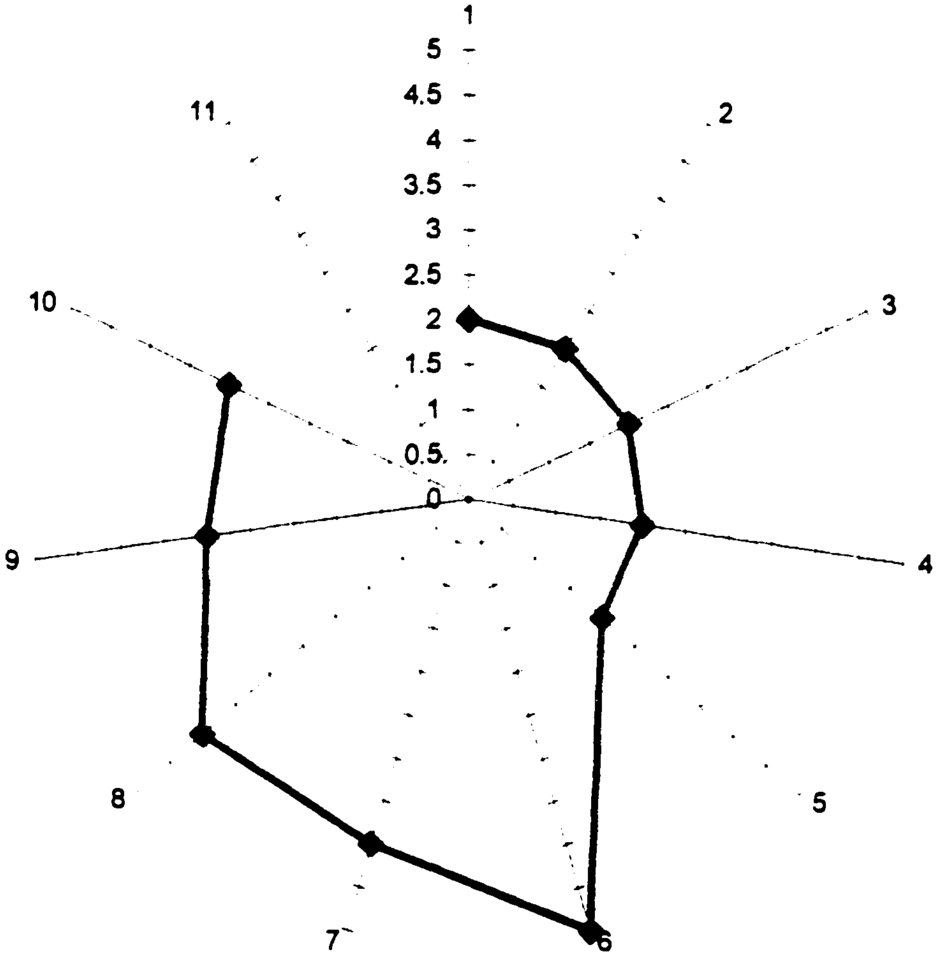
MPF



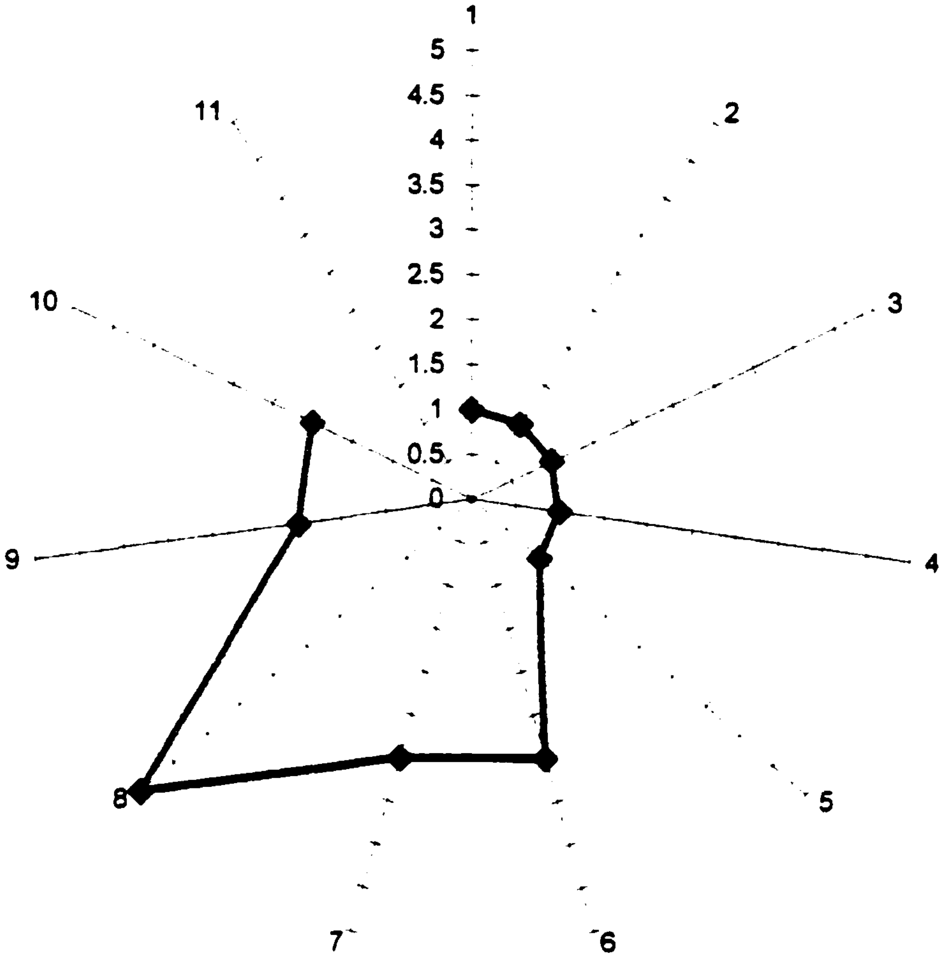
MRM



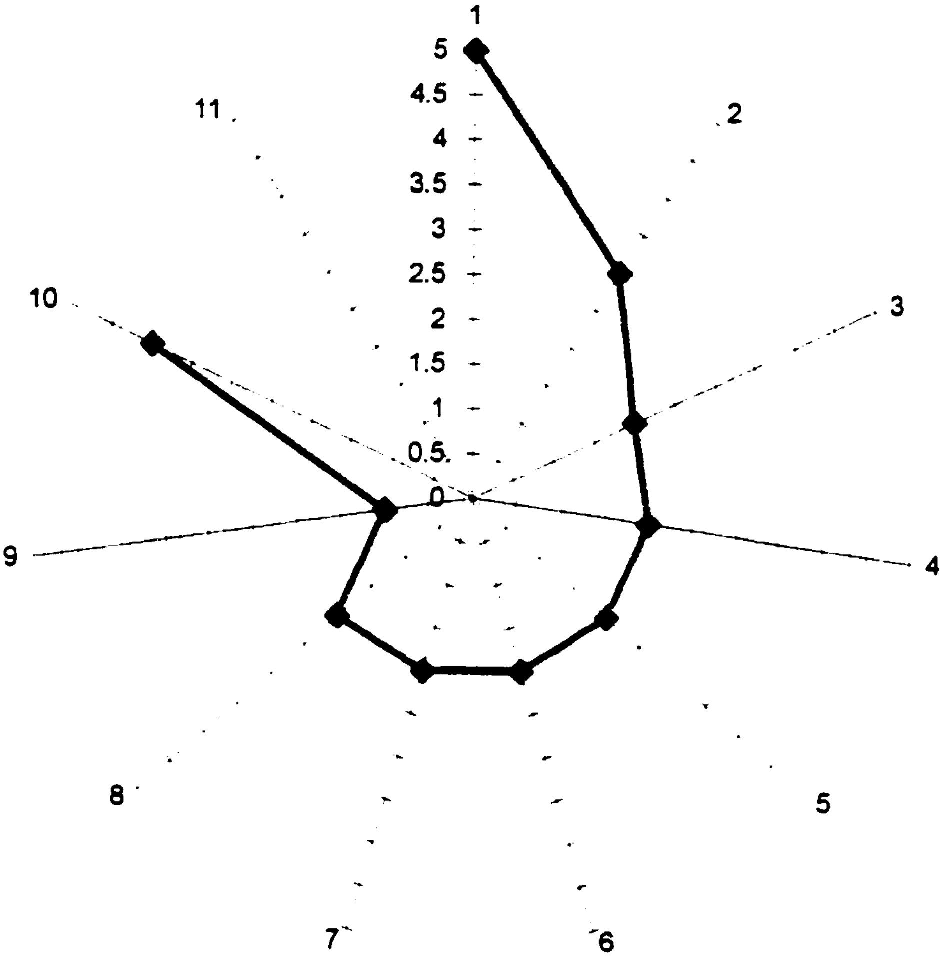
OMS



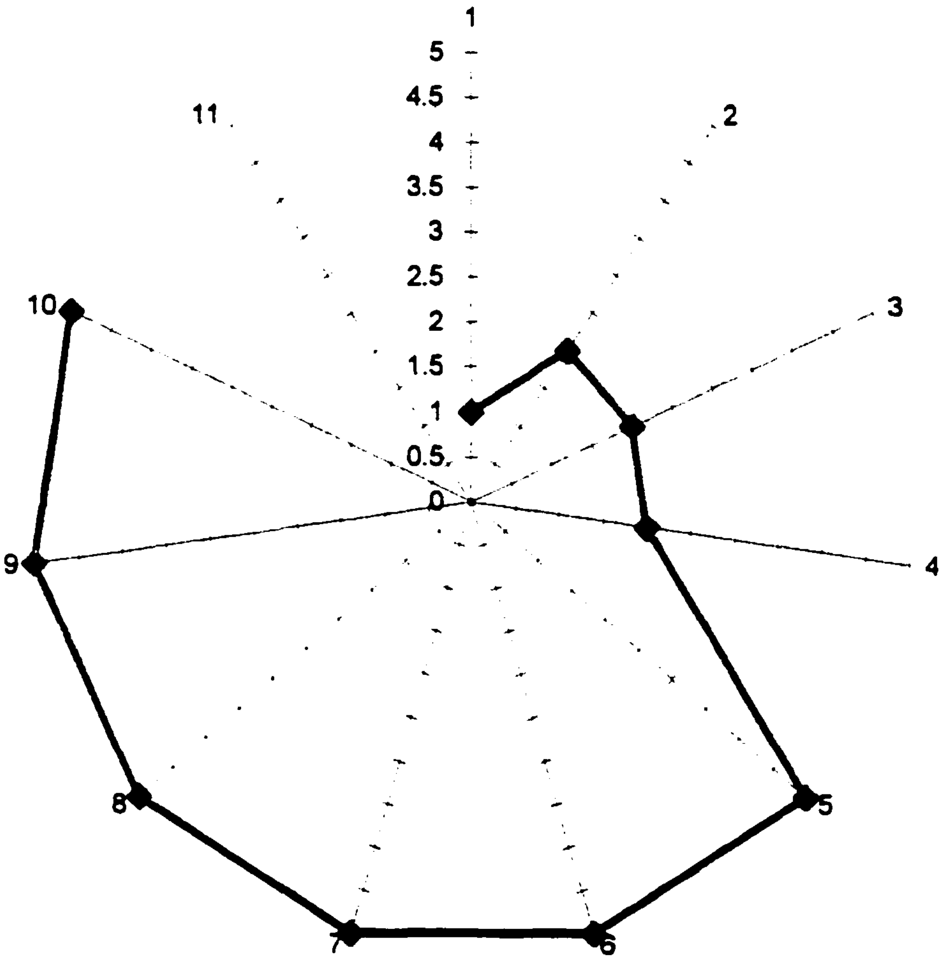
PCS



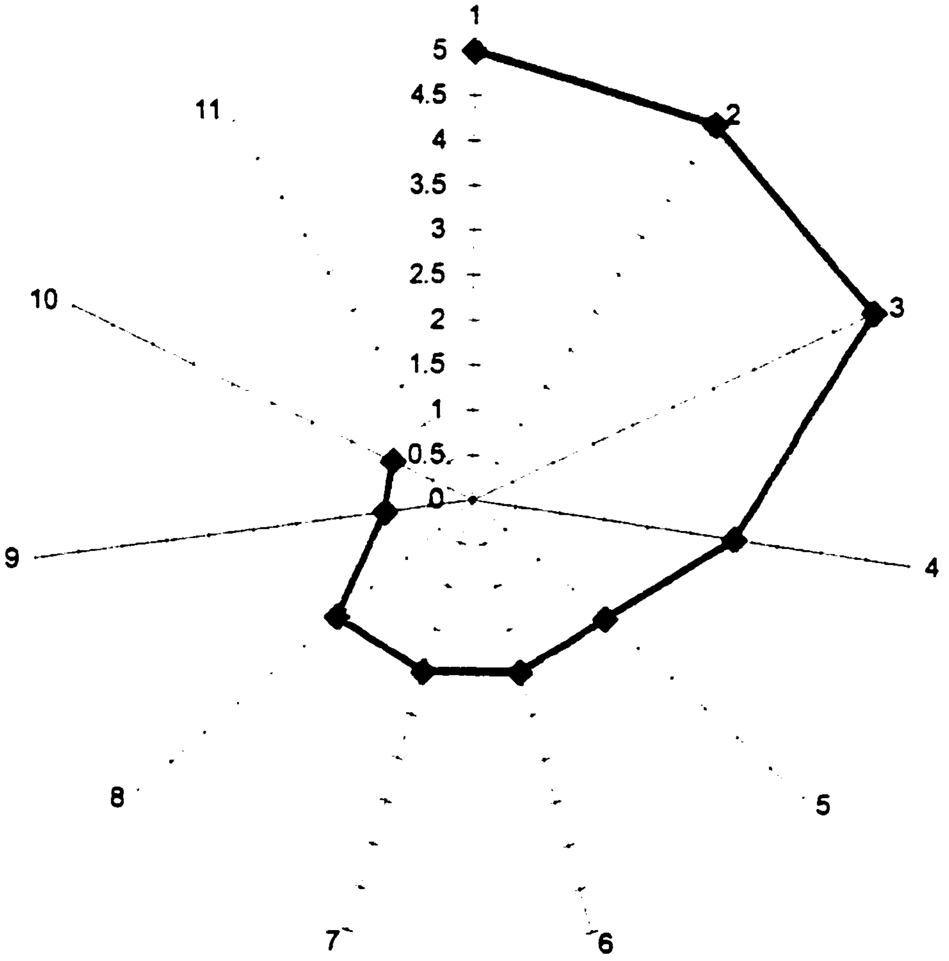
PMS



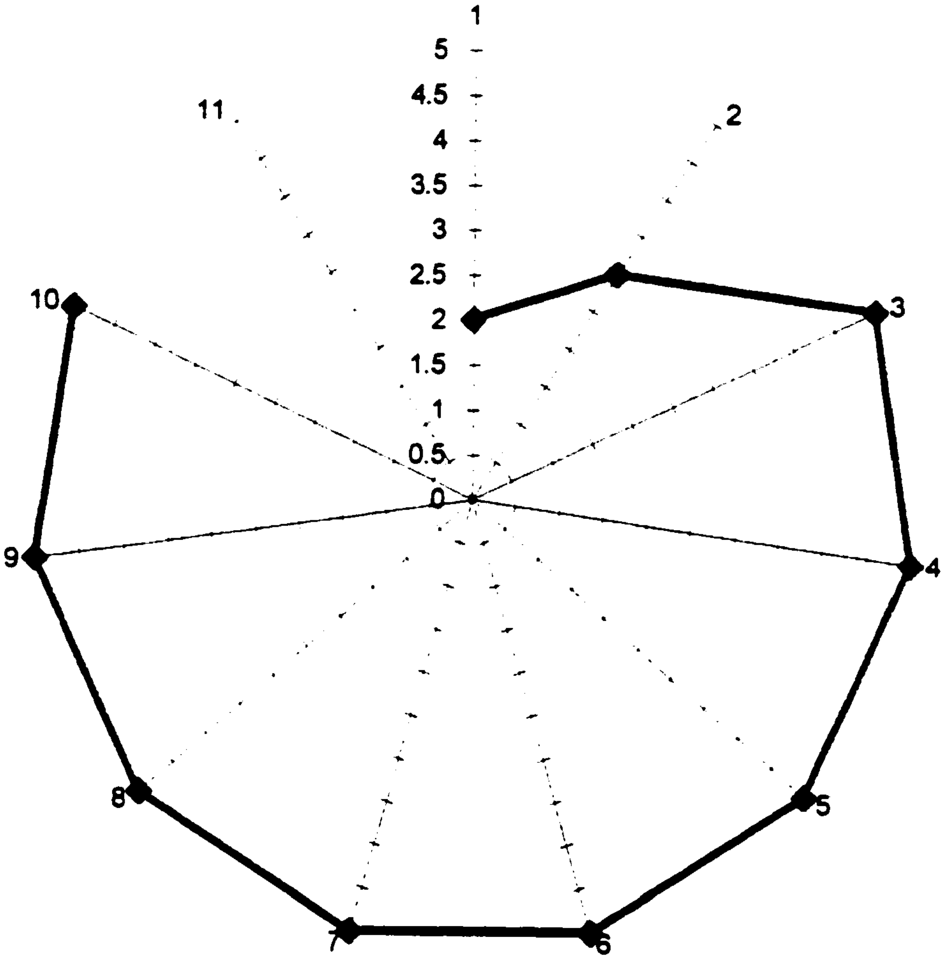
PPS



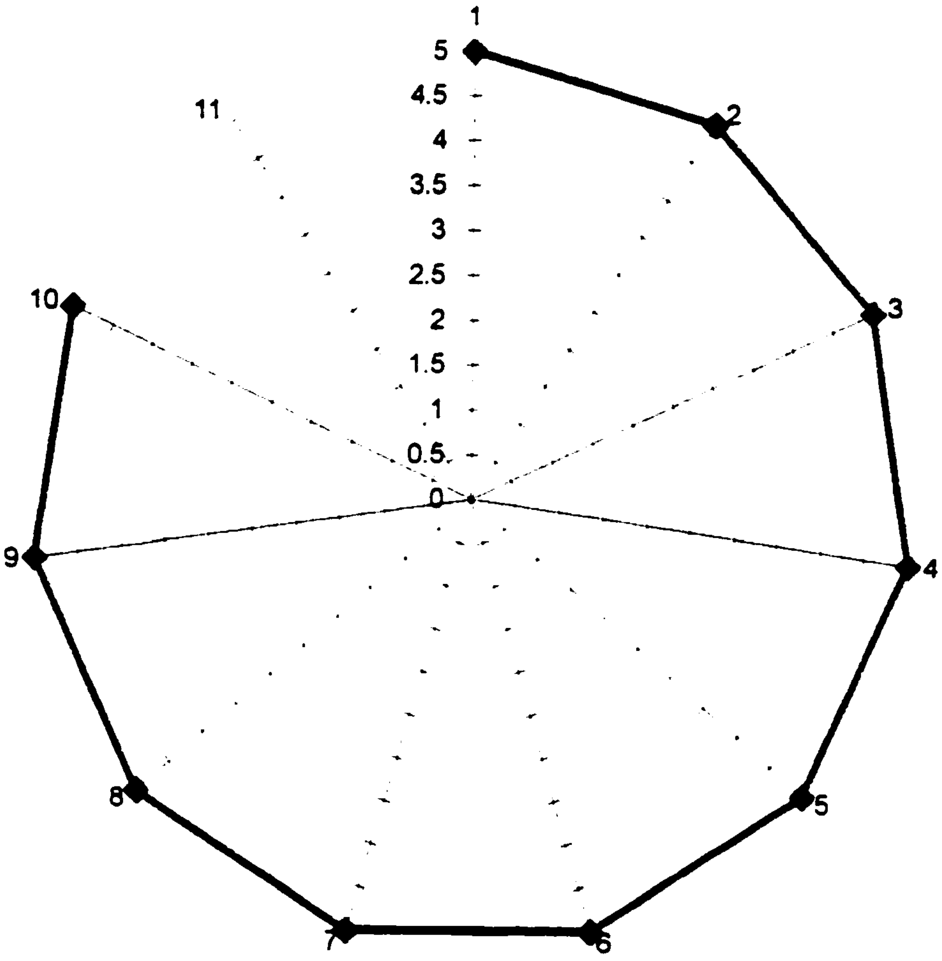
PRP

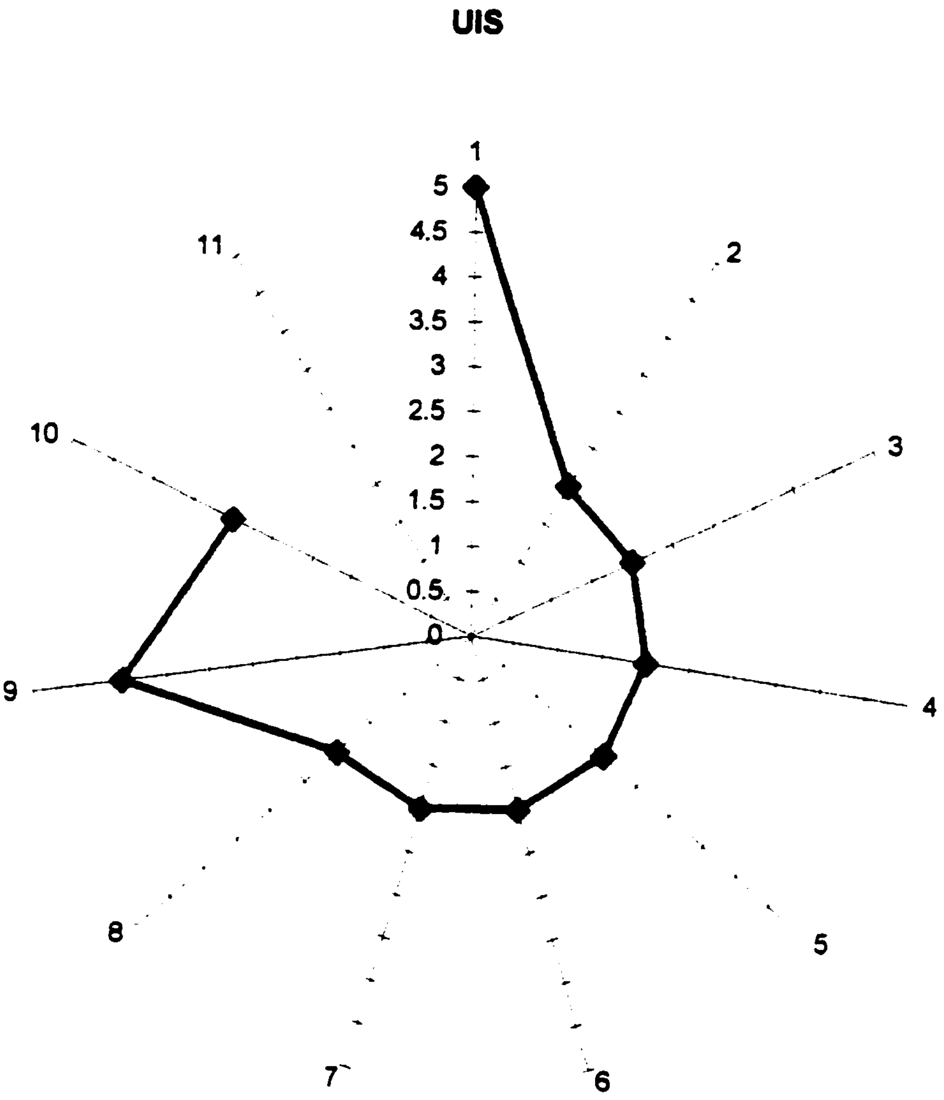


PRT



PYR



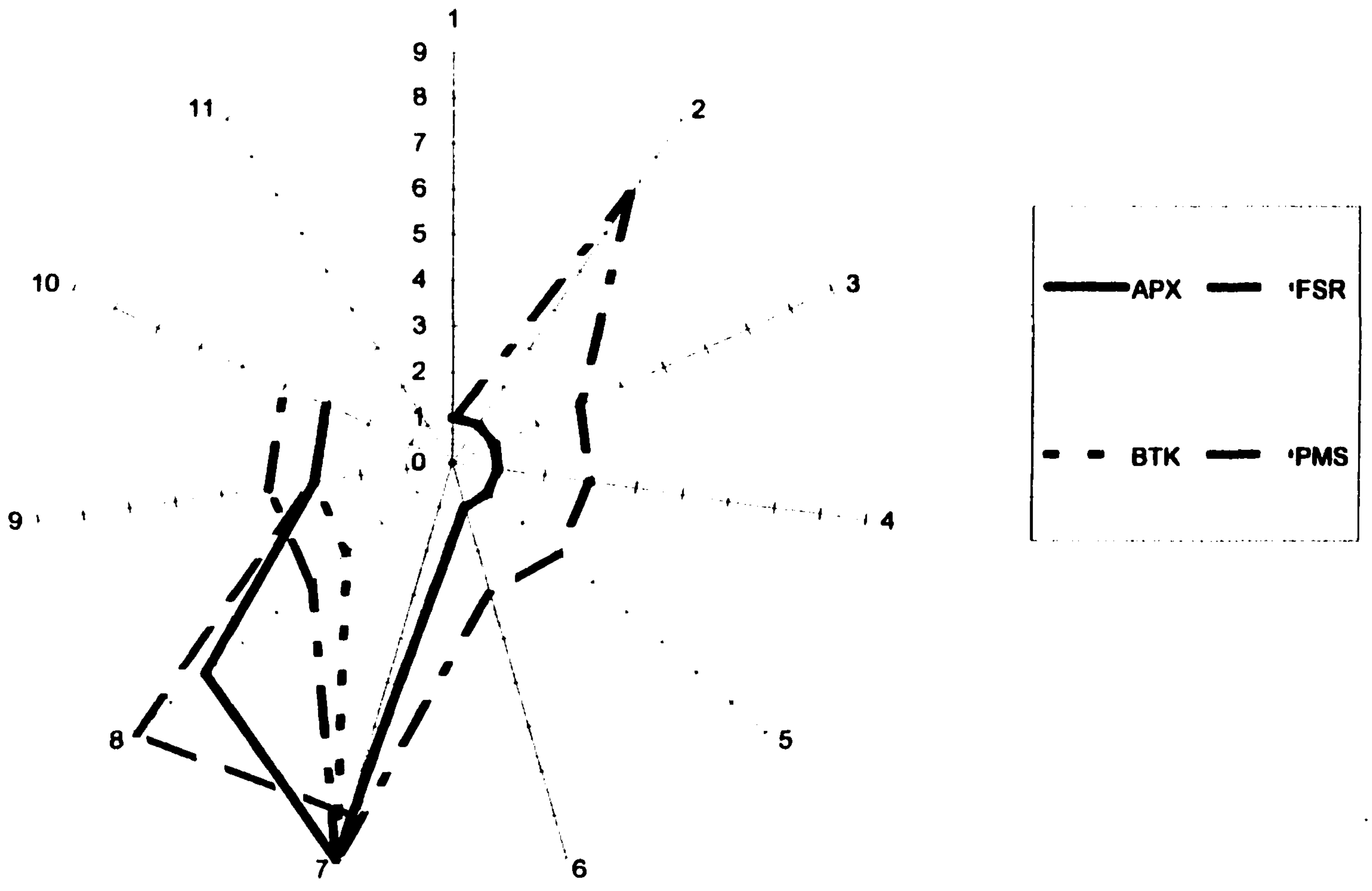

UIS
1
2
3
4
5
6
7
8
9
10
11
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0

APPENDIX D:

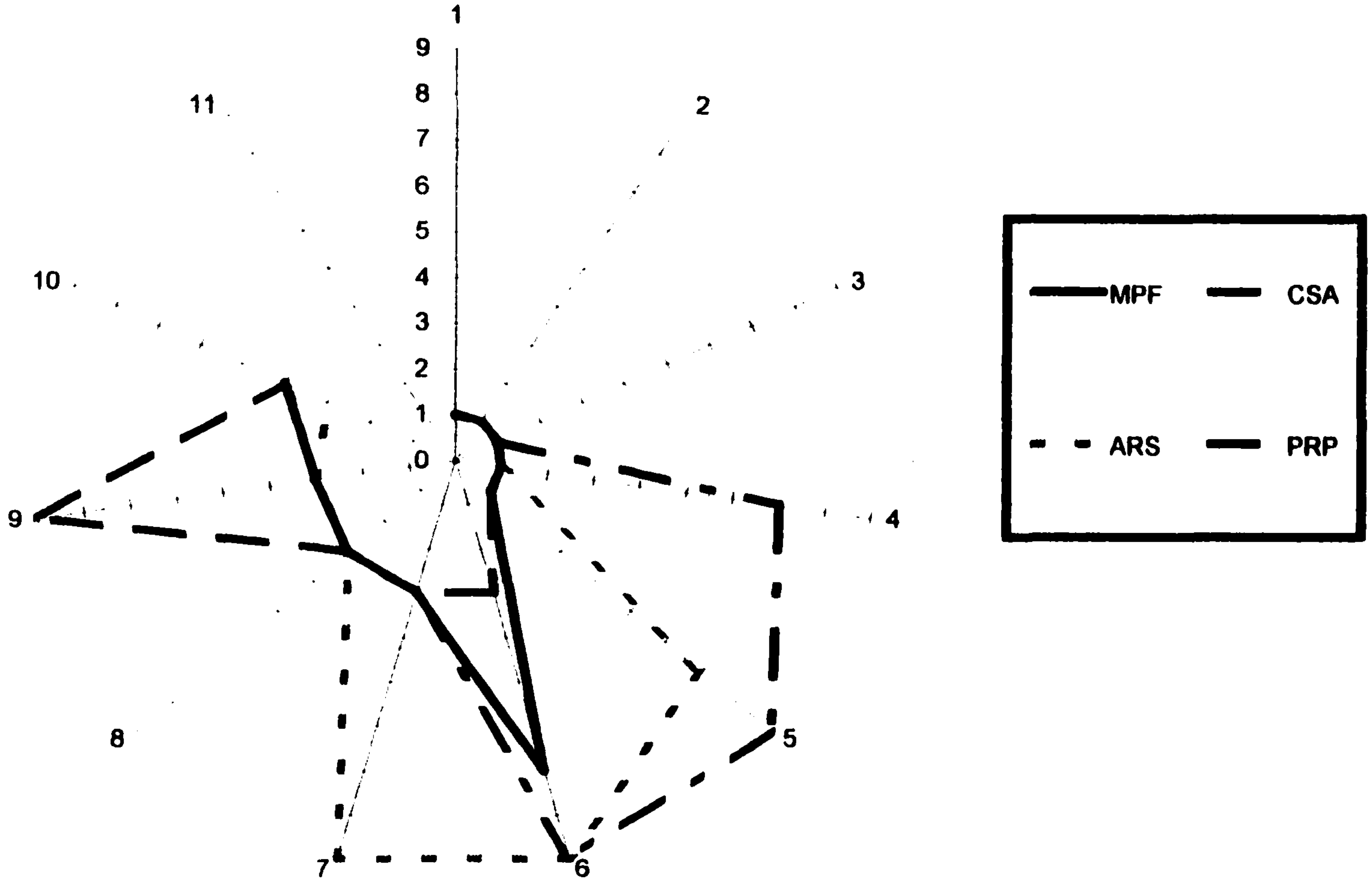
RADIAL GRAPH REPRESENTATIONS OF GAMMA ANALYSES

Using Periodicity/Amplitude/Deviation Classification of Software Volatility

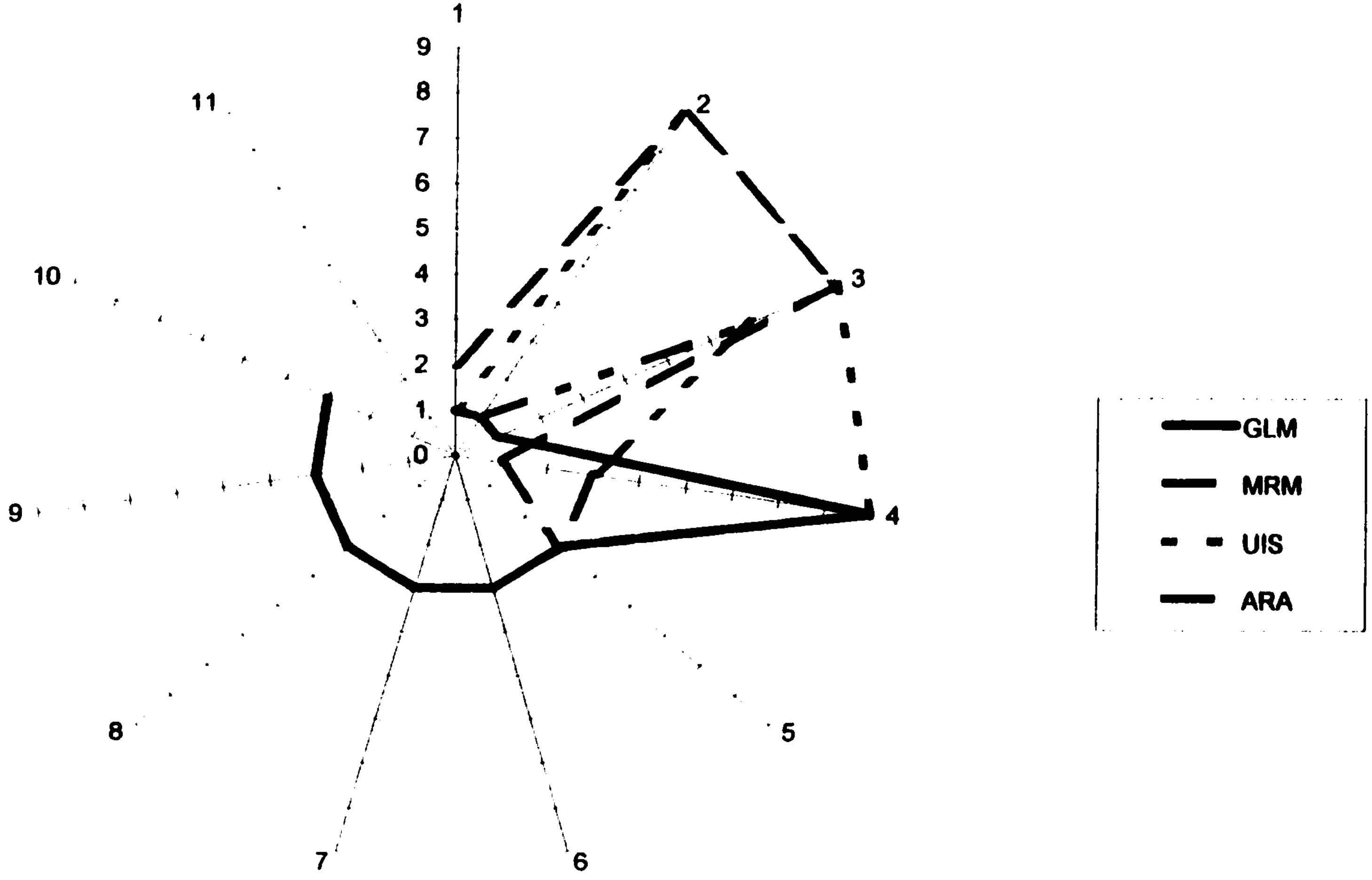
Group 1 Lifetime Volatility



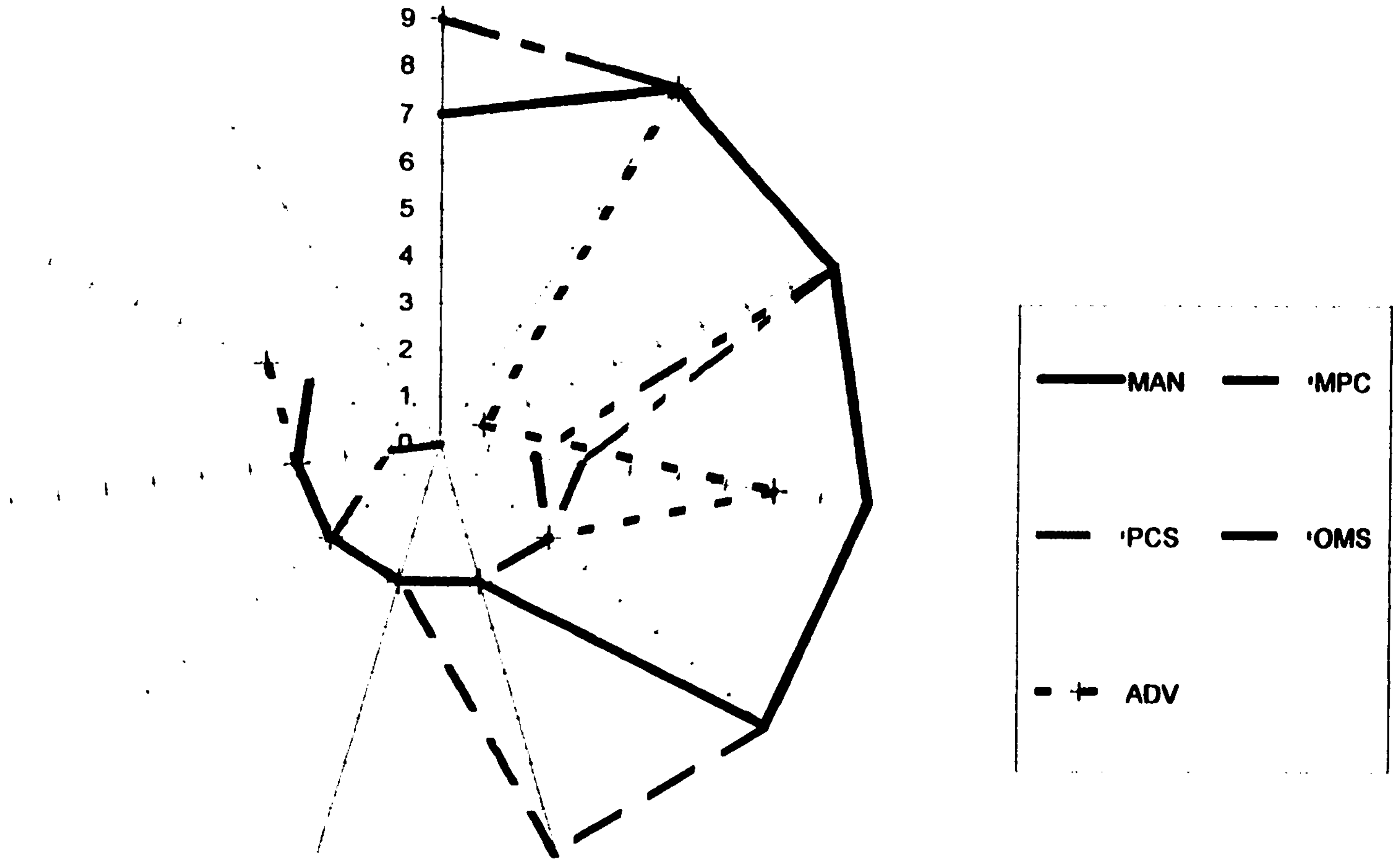
Group 2 Lifetime Volatility



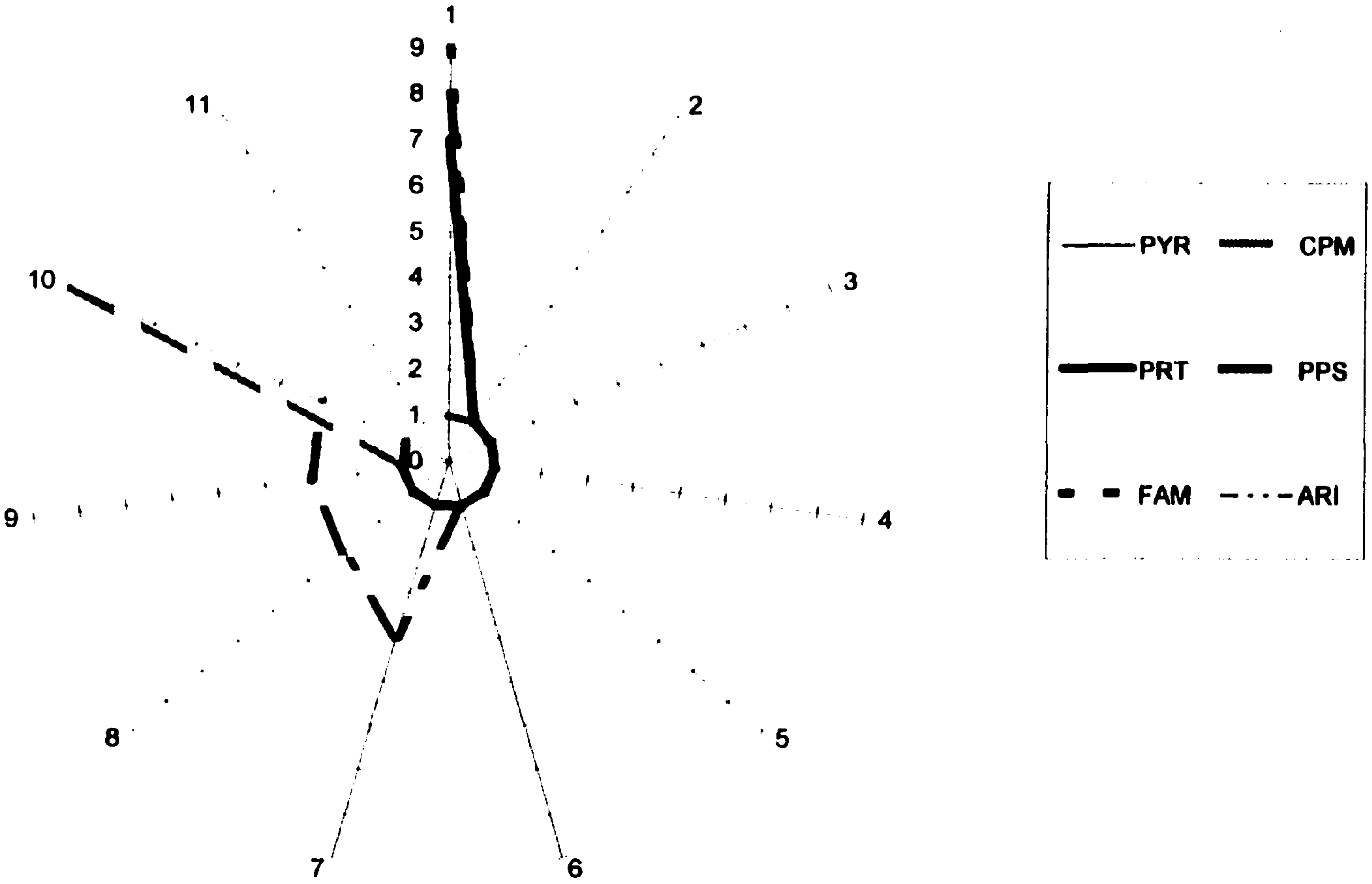
Group 3 Lifetime Volatility



Group 4 Lifetime Volatility



Group 5 Lifetime Volatility


PYR
CPM
PRT
PPS
FAM
ARI